

Hands-On Kubernetes on Azure

Second Edition

Automate management, scaling, and deployment of containerized applications



Packt

www.packt.com

Nills Franssens, Shivakumar Gopalakrishnan,
and Gunther Lenz

Hands-On Kubernetes on Azure

Second Edition

Automate management, scaling, and deployment of
containerized applications

Nills Franssens, Shivakumar Gopalakrishnan, and Gunther Lenz

Packt»

Hands-On Kubernetes on Azure - Second Edition

Copyright © 2020 Packt Publishing

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

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

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

Authors: Nills Franssens, Shivakumar Gopalakrishnan, and Gunther Lenz

Technical Reviewers: Peter De Tender and Suleyman Akbas

Managing Editors: Afzal Shaikh and Priyanka Sawant

Acquisitions Editor: Rahul Hande

Production Editor: Deepak Chavan

Editorial Board: Ben Renow-Clarke and Ian Hough

First Published: March 2019

Second Published: March 2020

Production Reference: 1190320

ISBN: 978-1-80020-967-1

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

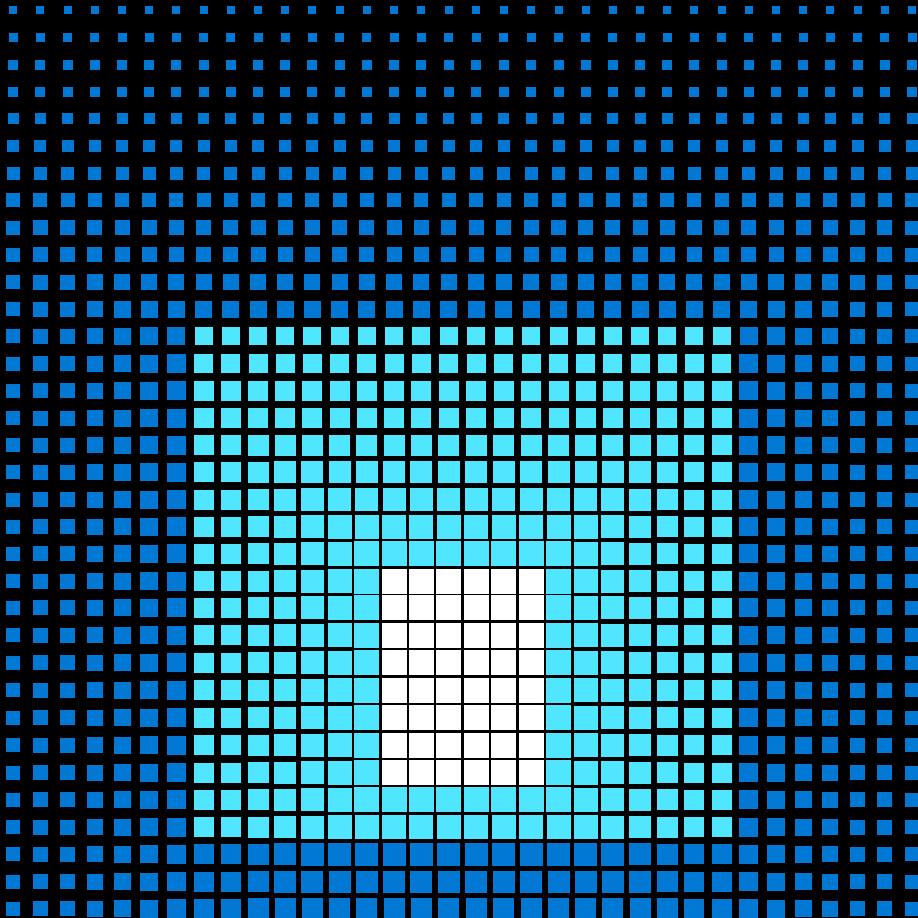
I dedicate this book to my parents. Without their support on everything from getting my first computer to encouraging me on whatever path I took, this book wouldn't have happened.

- Shivakumar Gopalakrishnan

To Okson and Hugo.

To everyone reading this.

- Gunther Lenz



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	i
Section 1: The Basics	1
Chapter 1: Introduction to Docker and Kubernetes	3
The software evolution that brought us here	5
Microservices	5
DevOps	8
Fundamentals of Docker containers	9
Docker images	10
Kubernetes as a container orchestration platform	14
Pods in Kubernetes	14
Deployments in Kubernetes	16
Services in Kubernetes	16
Azure Kubernetes Service	17
Summary	18
Chapter 2: Kubernetes on Azure (AKS)	21
Different ways to deploy an AKS cluster	22
Getting started with the Azure portal	23
Creating your first AKS cluster	23
A quick overview of your cluster in the Azure portal	30
Accessing your cluster using Azure Cloud Shell	32
Deploying your first demo application	35
Summary	41

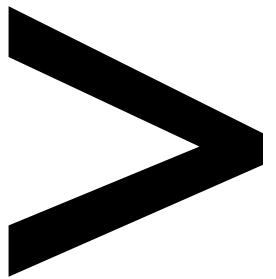
Section 2: Deploying on AKS	43
<hr/>	
Chapter 3: Application deployment on AKS	45
Deploying the sample guestbook application	47
Introducing the application	47
Deploying the Redis master	48
Redis master with a ConfigMap	54
Complete deployment of the sample guestbook application	60
Exposing the Redis master service	60
Deploying the Redis slaves	63
Deploying and exposing the front end	66
The guestbook application in action	73
Installing complex Kubernetes applications using Helm	74
Installing WordPress using Helm	75
Summary	82
<hr/>	
Chapter 4: Building scalable applications	85
Scaling your application	87
Implementing scaling of your application	87
Scaling the guestbook front-end component	90
Using the HPA	91
Scaling your cluster	95
Manually scaling your cluster	95
Scaling your cluster using the cluster autoscaler	97

Upgrading your application	100
Upgrading by changing YAML files	101
Upgrading an application using kubectl edit	105
Upgrading an application using kubectl patch	107
Upgrading applications using Helm	109
Summary	111
Chapter 5: Handling common failures in AKS	113
Handling node failures	114
Solving out-of-resource failures	120
Fixing storage mount issues	124
Starting the WordPress installation	125
Using persistent volumes to avoid data loss	125
Summary	138
Chapter 6: Securing your application with HTTPS and Azure AD	141
HTTPS support	143
Installing an Ingress controller	143
Adding an Ingress rule for the guestbook application	144
Getting a certificate from Let's Encrypt	147
Authentication versus authorization	159
Authentication and common authN providers	159
Deploying the oauth2_proxy proxy	160
Summary	170

Chapter 7: Monitoring the AKS cluster and the application	173
Commands for monitoring applications	174
The kubectl get command	174
The kubectl describe command	177
Debugging applications	181
Logs	187
Readiness and liveness probes	188
Building two web containers	189
Experimenting with liveness and readiness probes	192
Metrics reported by Kubernetes	196
Node status and consumption	196
Pod consumption	198
Metrics reported from Azure Monitor	200
AKS Insights	200
Summary	208
Section 3: Leveraging advanced Azure PaaS services	211
Chapter 8: Connecting an app to an Azure database	213
Setting up OSBA	214
The benefits of using a managed database service	214
What is OSBA?	215
Installing OSBA on the cluster	216
Deploying OSBA	217
Deploying WordPress	220
Securing MySQL	222
Connecting to the WordPress site	223

Exploring advanced database operations	224
Restoring from a backup	224
Disaster Recovery (DR) options	234
Reviewing audit logs	235
Summary	238
Chapter 9: Connecting to Azure Event Hubs	241
Deploying a set of microservices	242
Deploying the application using Helm	243
Using Azure Event Hubs	250
Creating the event hub	250
Modifying the Helm files	254
Summary	260
Chapter 10: Securing your AKS cluster	263
Role-based access control	264
Creating a new cluster with Azure AD integration	266
Creating users and groups in Azure AD	271
Configuring RBAC in AKS	275
Verifying RBAC	279
Setting up secrets management	283
Creating your own secrets	283
Creating the Docker registry key	288
Creating the TLS secret	289
Using your secrets	290
Secrets as environment variables	290
Secrets as files	291
Why secrets as files is the best method	293

Using secrets stored in Key Vault	295
Creating a Key Vault	295
Setting up Key Vault FlexVolume	299
Using Key Vault FlexVolume to mount a secret in a Pod	301
The Istio service mesh at your service	303
Describing the Istio service mesh	304
Installing Istio	305
Injecting Envoy as a sidecar automatically	306
Enforcing mutual TLS	307
Globally enabling mTLS	311
Summary	315
Chapter 11: Serverless functions	317
Multiple functions platforms	319
Setting up prerequisites	320
Azure Container Registry	321
Creating a development machine	322
Creating an HTTP-triggered Azure function	326
Creating a queue-triggered function	330
Creating a queue	330
Creating a queue-triggered function	333
Scale testing functions	338
Summary	340
Index	343



Preface

About

This section briefly introduces the authors, the coverage of this book, the technical skills you'll need to get started, and the hardware and software requirements required to complete all of the included activities and exercises.

About Hands-On Kubernetes on Azure, Second Edition

Kubernetes is the leading standard in container orchestration, used by start-ups and large enterprises alike. Microsoft is one of the largest contributors to the open source project, and it offers a managed service to run Kubernetes clusters at scale.

This book will walk you through what it takes to build and run applications on top of the [Azure Kubernetes Service \(AKS\)](#). It starts with an explanation of the fundamentals of Docker and Kubernetes, after which you will build a cluster and start deploying multiple applications. With the help of real-world examples, you'll learn how to deploy applications on top of AKS, implement authentication, monitor your applications, and integrate AKS with other Azure services such as databases, Event Hubs, and Functions.

By the end of this book, you'll have become proficient in running Kubernetes on Azure and leveraging the tools required for deployment.

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 senior cloud solution architect at Microsoft, he works with Microsoft's strategic customers on their cloud adoption. He has enabled multiple customers in their migration to Azure. One of these migrations was the migration and replatforming of a major public website to Kubernetes.

Outside of Kubernetes, Nills's areas of expertise are networking and storage in Azure.

He holds a master's degree in engineering from the University of Antwerp, Belgium.

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.

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*.

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.

Learning Objectives

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

- Understand the fundamentals of Docker and Kubernetes
- Set up an AKS cluster
- Deploy applications to AKS
- Monitor applications on AKS and handle common failures
- Set up authentication for applications on top of AKS
- Integrate AKS with Azure Database for MySQL
- Leverage Azure Event Hubs from an application in AKS
- Secure your cluster
- Deploy serverless functions to your cluster

Audience

If you're a cloud engineer, cloud solution provider, sysadmin, site reliability engineer, or developer who's interested in DevOps and is looking for an extensive guide to running Kubernetes in the Azure environment, then this book is for you.

Approach

This book provides a combination of practical and theoretical knowledge. It covers engaging real-world scenarios that demonstrate how Kubernetes-based applications run on the Azure platform. Each chapter is designed to enable you to apply everything you learn in a practical context. After chapter 1 and 2, each chapter is self-contained, and can be run independently from previous chapters.

Software Requirements

We also recommend that you have the following software installed 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

All the examples in the book have been designed to work using the Azure Cloud Shell. You won't have to install additional software on your machine.

Conventions

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

"The following code snippet will use the **kubectl** command line tool to create an application that is defined in the file **guestbook-all-in-one.yaml**."

Here's a sample block of code:

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

We'll use a backslash, \, to indicate that a line of code will span multiple lines in the book. You can either copy the backslash and continue on the next line or ignore the backslash and type the complete multi-line code on a single line. For example:

```
az aks nodepool update --disable-cluster-autoscaler \
-g rg-hansonaks --cluster-name handsonaks --name agentpool
```

On many occasions, we have used angled brackets, <>. You need to replace these with the actual parameter, and not use these brackets within the commands.

Download Resources

The code bundle for this book is also hosted on GitHub at <https://github.com/PacktPublishing/Hands-On-Kubernetes-on-Azure---Second-Edition>. You can find the YAML and other files used in this book, which are referred to at relevant instances.

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

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 Docker and Kubernetes. Then, we will explain how to create a Kubernetes cluster on Azure and deploy an example application.

When you finish this section, you will have a baseline foundational knowledge of Docker and Kubernetes and 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 Docker and Kubernetes*
- *Chapter 2, Kubernetes on Azure (AKS)*

1

Introduction to Docker and Kubernetes

Kubernetes has become the leading standard in container orchestration. Since its inception in 2014, it has gained tremendous popularity. It has been adopted by start-ups as well as major enterprises, and the major public cloud vendors all offer 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 standard way of creating and running software containers, often called Docker containers. A container itself 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 data center, to a cluster running in the public cloud.

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

Kubernetes takes Docker 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 will take care of scheduling the right container to run on the right machine, and it will take care of health monitoring and failover, as well as scaling your deployed application.

Docker 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. This is attracting 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 manages the complexity of putting together all the preceding services for you. 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 an engineer, 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. You will see how various processes fit nicely into Docker, and how Docker fits nicely into Kubernetes. Even though Kubernetes is technically a container runtime-agnostic platform, Docker is the most commonly used container technology and is used everywhere.

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, highly scalable applications on AKS. Over the next 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 Docker
- 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 Docker 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 Docker 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 Docker 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 part of the data the application required.

The client-server programming model has evolved toward truly 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 from 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 bounded 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.

This architectural approach is commonly adopted by applications run using Docker and Kubernetes. Docker is 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 the Docker 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 fact 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 they are independently deployable. There are multiple deployment strategies when it comes to microservices. The most popular are rolling upgrades and blue/green deployments.

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

With a blue/green deployment, you would deploy the new version of the service in isolation. Once the new version of the service is deployed and tested, you would 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 being **polyglot** – able 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#, while the machine learning algorithm can be developed in Python. This allows you to select the right language for the right service, and to have the developers 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 aforementioned disadvantages 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.

Often linked to microservices, but a separate transformation, 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 (in *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 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, or 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 need to 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 Docker and Kubernetes. After this introduction to microservices and DevOps, we'll continue this first chapter with the fundamentals of Docker and containers and then the fundamentals of Kubernetes.

Fundamentals of Docker containers

A form of container technology has existed in the Linux kernel since the 1970s. The technology powering today's containers, called cgroups, was introduced into the Linux kernel in 2006 by Google. The Docker company popularized the technology in 2013 by introducing an easy developer workflow. The company gave its name to the technology, so 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.

Docker as a technology is both a packaging format and a container runtime. We refer to packaging as an architecture 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.

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.

Docker images

Docker uses an image to start a new container. 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 **Azure Container Registry (ACR)**. When you, as a user, don't have an image locally on your PC, you will 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 by following these instructions:

```
#First we will pull an image
```

```
docker pull docker/whalesay
```

```
#We can then look at which images we have 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 1.1*:

```
#####
#           WARNING!!!!
# This is a sandbox environment. Using personal credentials
# is HIGHLY! discouraged. Any consequences of doing so are
# completely the user's responsibilites.
#
# The PWD team.
#####
[node2] (local) root@192.168.0.22 ~
$ docker pull docker/whalesay
Using default tag: latest
latest: Pulling from docker/whalesay
Image docker.io/docker/whalesay:latest uses outdated schema1 manifest format. Please
upgrade to a schema2 image for better future compatibility. More information at https://docs.docker.com/registry/spec/deprecated-schema-v1/
e190868d63f8: Pull complete
909cd34c6fd7: Pull complete
0b9bfabab7c1: Pull complete
a3ed95caeb02: Pull complete
00bf65475aba: Pull complete
c57b6bcc83e3: Pull complete
8978f6879e2f: Pull complete
8eed3712d2cf: Pull complete
Digest: sha256:178598e51a26abbc958b8a2e48825c90bc22e641de3d31e18aaf55f3258ba93b
Status: Downloaded newer image for docker/whalesay:latest
docker.io/docker/whalesay:latest
[node2] (local) root@192.168.0.22 ~
$ docker images
REPOSITORY          TAG      IMAGE ID      CREATED     SIZE
docker/whalesay    latest   6b362a9f73eb  4 years ago  247MB
[node2] (local) root@192.168.0.22 ~
$ docker run docker/whalesay cowsay boo

< boo >
-----
 \
 \
 \
      ##      .
      ## ## ##      ==
      ## ## ## ##      ==
      /"*****"__/_/ ==_
~~~ {~~ ~~~~ ~~ ~~~ ~ ~ ~ / === ~~~
      \____\ o      /_/
      \_\_\_\_/_/
```

Figure 1.1: Example of running Docker 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 we 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 our 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, we 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. An example of a Dockerfile is provided in the following code snippet, which we'll create in our Docker playground:

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

There are three lines in this Dockerfile. The first one will instruct Docker which image to use as a source image for this new image. The next step is a command that is run to add new functionality to our image. In this case, updating our `apt` repository and installing an application called `fortunes`. 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 our image, you need to execute the `docker build` command and point it to the Dockerfile you created. In building the Docker image, the process 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 our own image.

In order to create this Dockerfile, open up a text editor via the **vi Dockerfile** command. vi is an advanced text editor in 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-paste or type the three 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 our image. We will add a final bit to that command, namely adding a tag to our image so we can call it by a useful name. To build your image, you will use the **docker build -t smartwhale .** command (don't forget to add the final dot here).

You will now see Docker execute a number of steps – three in our case – in order to build our image. After your image is built, you can run your application. To run your container, you would run **docker run smartwhale**, and you should see an output similar to *Figure 1.2*. 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 1.2*:

```
[node2] (local) root@192.168.0.22 ~
$ docker run smartwhale

/ Lots of people drink from the wrong \
| bottle sometimes.
|
| -- Edith Keeler, "The City on the Edge
| of Forever",
|
\ stardate unknown

-----
\\
\\
##      .
## ## ##    ==
## ## ## ##   ===
/*****      _/ ===
~~~ {~~ ~~~~ ~~~ ~~~~ ~ ~ / ===- ~~~
 \_ \_ \_ \_ \_ /
```

Figure 1.2: Example of running a custom container

That concludes our overview and demo of Docker. In this section, you started with an existing container image and launched that on Docker Labs. Afterward, you took that a step further and built your own container image and started containers using your own 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 that host becomes unhealthy, and much more.

The current leading orchestration platform is Kubernetes (<https://kubernetes.io/>). Kubernetes was inspired by the Borg project in Google, 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 Docker containers you specified.

Note

Although Kubernetes supports multiple container runtimes, Docker is the most popular runtime.

Throughout the book, we will build multiple examples that run containers in Kubernetes, and you will learn more about the different objects in Kubernetes. In this introductory chapter, we'll introduce 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 block. A Pod is a group of one or more containers. This means a Pod contains 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.

When a Pod contains multiple containers, these containers share the same filesystem 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. 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 SSL offloading for that application as described in *Figure 1.3*. Your users would connect to your application using HTTPS, while the container in the middle converts HTTPS traffic to HTTP:

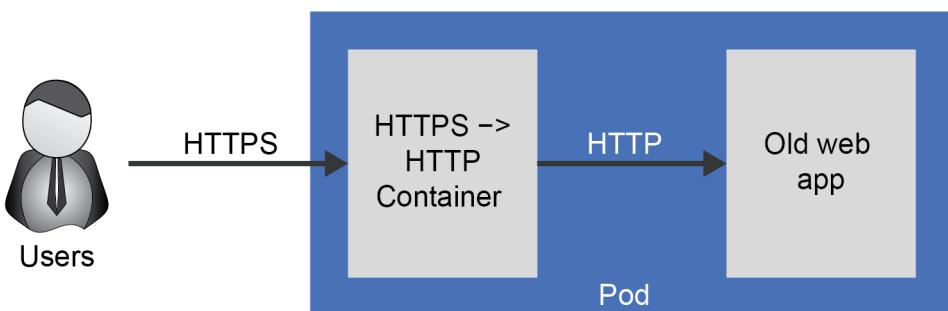


Figure 1.3: Example of a multi-container Pod that does HTTPS offloading

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 a 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 in a **StatefulSet**, which we'll touch on in *Chapter 3, Application deployment on AKS*, 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 covers, a Deployment creates a **ReplicaSet**, which in turn will create the Pod you requested. A **ReplicaSet** is another object in Kubernetes. The purpose of a **ReplicaSet** is to maintain a stable set of Pods running at any given time. If you perform updates to 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. If those are running correctly, then it will terminate a few old Pods and continue this loop until only new Pods are running.

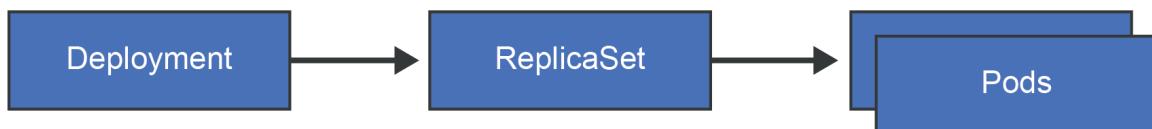


Figure 1.4: Relationship between Deployment, ReplicaSet, and Pods

Services in Kubernetes

A Service in Kubernetes is a network-level abstraction. This allows you to expose the multiple Pods you have in your Deployment under a single IP address and a single DNS name.

Each Pod in Kubernetes has its own private IP address. You could theoretically connect your applications using this private IP address. However, as mentioned before, Kubernetes Pods are ephemeral, meaning they can be terminated and moved, which would impact their IP address. By using a Service, you can connect your applications together using a single IP address. When a Pod moves from one node to another, the Service will ensure traffic is routed to the correct endpoint.

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

Azure Kubernetes Service

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 **virtual machine (VM)**. The master nodes contain the Kubernetes API and a database that contains the cluster state. The worker nodes are the VMs that run your actual workload.

AKS makes it a lot easier to create a cluster. When you create an AKS cluster, AKS sets up the Kubernetes master for you, free of charge. AKS will then create VMs in your subscription, and turn those VMs into worker nodes of your Kubernetes cluster in your network. You only pay for those VMs; you don't pay for the master.

Within AKS, Kubernetes Services are integrated with Azure Load Balancer and Kubernetes Ingresses are integrated with the application gateway. Azure Load Balancer is a layer-4 network load balancer Service; the 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 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 has the ability to easily scale your clusters, either making them bigger or smaller.

The service also comes with integrations that make operations easier. AKS clusters come pre-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 is also integrated into Azure Monitor for containers, which makes monitoring and troubleshooting your Deployments simpler.

Summary

In this chapter, we introduced the concepts of Docker and Kubernetes. We ran a number of containers, starting with an existing image and then using an image we built ourselves. After that demo, we explored three essential Kubernetes objects: the Pod, the Deployment, and the Service.

This provides the common context for the remaining chapters, where you will deploy Dockerized applications in Microsoft AKS. You will see how the AKS Platform as a Service (PaaS) 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, we will introduce the Azure portal and its components in the context of creating your first AKS cluster.

2

Kubernetes on Azure (AKS)

Installing and maintaining Kubernetes clusters correctly and securely is difficult. Thankfully, all the major cloud providers, such as Azure, 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. All of this will be accomplished from your browser.

The following topics will be covered in this chapter:

- Creating a new Azure free account
- Navigating the Azure portal
- Launching your first cluster
- Starting your first application

Let's start by looking at the different ways to create an AKS cluster, and then run our sample application.

Different ways to deploy an AKS cluster

This chapter will introduce the graphical way to deploy your AKS cluster. There are, however, multiple ways to create your AKS cluster:

- **Using the portal:** The portal gives you a graphical way of 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.
- **User Azure PowerShell:** Azure PowerShell is a set of PowerShell commands 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 infrastructure-as-code (IaC) language. They allow you to declaratively deploy your cluster. This allows 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 to deploy cloud resources, including AKS. Like ARM templates, Terraform also uses declarative templates for your cluster.

In this chapter, we will be creating our cluster using the Azure portal. If you are interested in deploying a cluster using either the CLI, ARM, or Terraform, the Azure documentation contains steps on how to use these tools to create your clusters at <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 need an Azure account. If you do not 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 **Azure Kubernetes Service (AKS)** cluster. By doing so, we are also going to familiarize ourselves with the Azure portal.

Creating your first AKS cluster

Enter the **aks** keyword in the search bar at the top of the Azure portal. Click on **Kubernetes services** under the **Services** category in the results:

Figure 2.1: Searching for AKS with the search bar

This will take you to the AKS blade 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:

The screenshot shows the 'Kubernetes services' blade in the Microsoft Azure portal. At the top, there's a search bar and several navigation icons. On the right, a user profile is shown with the email 'handsonaksdemo@outlook.com' and 'DEFAULT DIRECTORY'. Below the header, the title 'Kubernetes services' is displayed with a 'Default Directory' link. A red box highlights the '+ Add' button in the top-left corner of the main content area. The content area includes a toolbar with 'Edit columns', 'Refresh', 'Export to CSV', 'Assign tags', 'Feedback', and 'Leave preview'. Below the toolbar are filter options: 'Filter by name...', 'Subscription == all', 'Resource group == all', 'Type == all', 'Location == all', and a 'Add filter' button. A dropdown menu 'No grouping' is open. The main table has columns: 'Name ↑↓', 'Type ↑↓', 'Resource group ↑↓', 'Kubernetes version ↑↓', 'Location ↑↓', and 'Subscription ↑'. There are no records listed under 'Name'. In the center of the page is a grey icon of a cluster of three-dimensional boxes. Below the icon, the text 'No Kubernetes services to display' is shown. At the bottom, a note reads: 'Use Azure Kubernetes Service to create and manage Kubernetes clusters. Azure will handle cluster operations, including creating, scaling, and upgrading, freeing up developers to focus on their application. To get started, create a cluster with Azure Kubernetes Service.' followed by a 'Learn more' link. A large blue 'Create Kubernetes service' button is at the bottom.

Figure 2.2: Clicking on the "+ Add" button to start the cluster creation process

Note

There are a lot of options to configure when creating your 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 through the creation wizard to create your first AKS cluster. The first step here is to create a new resource group. Hit the **Create new** button, give your resource a name, and hit **OK**. If you want to follow the examples in this book, please give the resource group the name **rg-handsonaks**:

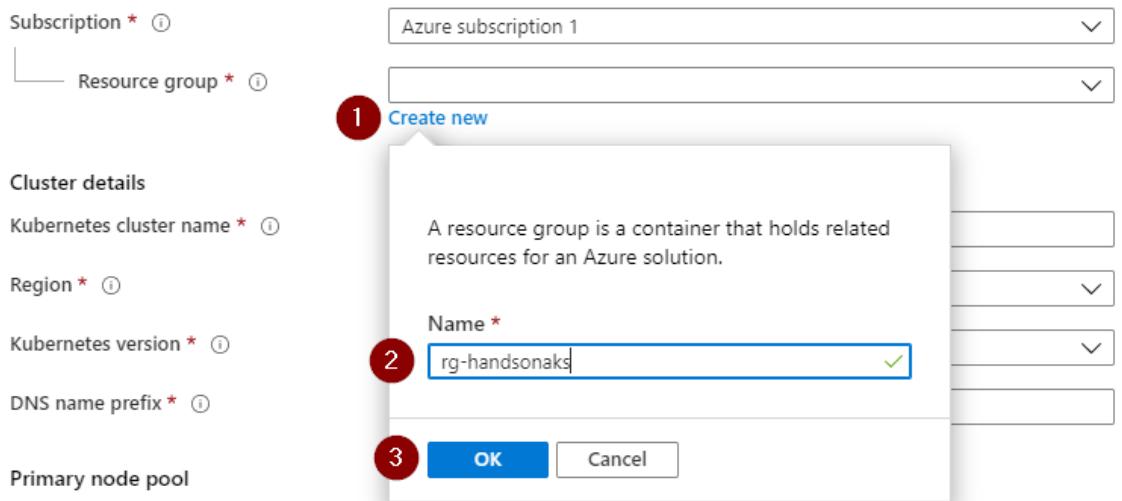


Figure 2.3: Creating a new resource group

Next up, we'll provide our cluster details. Give your cluster a name – if you want to follow the examples in the book, please call it **hansonaks**. 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. We will use Kubernetes version **1.15.7**, but don't worry if that version is not available for you. Kubernetes and AKS evolve very quickly, and new versions are introduced often. Next, you'll need to provide a DNS name prefix. This does not have to be unique since Azure will append this with random characters:

Cluster details	
Kubernetes cluster name *	hansonaks
Region *	(US) West US 2
Kubernetes version *	1.15.7
DNS name prefix *	hansonaks-hoa

Figure 2.4: Providing the cluster details

Next, we will need to change the node size and node count. To optimize our free budget for our sample cluster, we will use a **virtual machine (VM)** with one core without premium storage and deploy a two-node cluster. If you are not using the free trial, you could choose a more powerful VM size, although this is not required for the labs in this book.

Hit the **Change size** button below the machine size:



Figure 2.5: Clicking on the 'Change size' option to select a smaller machine

Remove the filter that looks for premium storage and look for **D1_v2**. Then change the slider for **Node count** to **2**:

Select a VM size									
Browse available virtual machine sizes and their features									
Search by VM size...		Clear all filters							
Size : Small (0-6) <input checked="" type="checkbox"/>		Generation : 2 selected <input type="checkbox"/>		Family : General purpose <input type="checkbox"/>		<input type="button" value="Add filter"/>			
Showing 17 of 186 VM sizes. Subscription: Azure subscription 1 Region: West US 2 Current size: Standard_DS1_v2									
VM Si...↑↓	Offering ↑↓	Family ↑↓	vCP...↑↓	RAM (...)↑↓	Data disks↑↓	Max IOPS ↑↓	Temporary stor...↑↓	Premium disk s...↑↓	Cost/month (es...)↑↓
A2_v2	Standard	General purpose	2	4	4	4x500	20	No	\$56.54
A2m_v2	Standard	General purpose	2	16	4	4x500	20	No	\$73.66
A4_v2	Standard	General purpose	4	8	8	8x500	40	No	\$118.30
A4m_v2	Standard	General purpose	4	32	8	8x500	40	No	\$154.75
B2ms	Standard	General purpose	2	8	4	1920	16	Yes	\$61.90
B2s	Standard	General purpose	2	4	4	1280	8	Yes	\$30.95
B4ms	Standard	General purpose	4	16	8	2880	32	Yes	\$123.50
D1_v2	Standard	General purpose	1	3.5	4	4x500	50	No	\$42.41
D2_v2	Standard	General purpose	2	7	8	8x500	100	No	\$84.82

Figure 2.6: Selecting D1_v2 as the machine size

This should make your cluster size look similar to that shown in Figure 2.7:

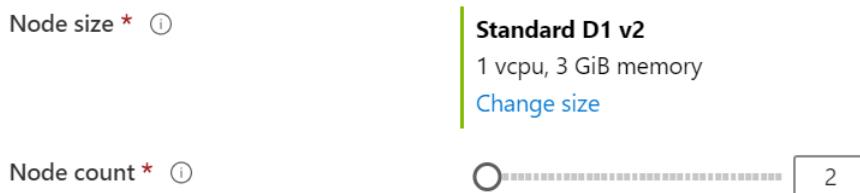


Figure 2.7: Updated Node size and Node count

Note

Your free account has a four-core limit that will be violated if we go with the defaults.

The final view of the first blade should look like *Figure 2.8*. There are a number of other configuration options that we will not change for our demo cluster. Since we are ready, hit the **Review + create** button now to do a final review and create your cluster:

Create Kubernetes cluster

Basics Scale Authentication Networking Monitoring Tags Review + create

Azure Kubernetes Service (AKS) manages your hosted Kubernetes environment, making it quick and easy to deploy and manage containerized applications without container orchestration expertise. It also eliminates the burden of ongoing operations and maintenance by provisioning, upgrading, and scaling resources on demand, without taking your applications offline. [Learn more about Azure Kubernetes Service](#)

Project details

Select a subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription * ⓘ Azure subscription 1

Resource group * ⓘ rg-hansonaks

[Create new](#)

Cluster details

Kubernetes cluster name * ⓘ handsonaks

Region * ⓘ (US) West US 2

Kubernetes version * ⓘ 1.15.7

DNS name prefix * ⓘ handsonaks-hoa

Primary node pool

The number and size of nodes in the primary node pool in your cluster. For production workloads, at least 3 nodes are recommended for resiliency. For development or test workloads, only one node is required. You will not be able to change the node size after cluster creation, but you will be able to change the number of nodes in your cluster after creation. If you would like additional node pools, you will need to enable the "X" feature on the "Scale" tab which will allow you to add more node pools after creating the cluster. [Learn more about node pools in Azure Kubernetes Service](#)

Node size * ⓘ **Standard D1 v2**
1 vcpu, 3 GiB memory
[Change size](#)

Node count * ⓘ 2

Review + create < Previous Next : Scale >

Figure 2.8: Setting the cluster configuration

On the final view, Azure will validate the configuration that was applied to your first cluster. If the validation passed, click on **Create**:

Create Kubernetes cluster

✓ Validation passed

Basics Scale Authentication Networking Monitoring Tags Review + create

Basics

Subscription	Azure subscription 1
Resource group	rg-handonaks
Region	(US) West US 2
Kubernetes cluster name	handsonaks
Kubernetes version	1.15.7
DNS name prefix	handsonaks-dns
Node count	2
Node size	Standard_D1_v2

Scale

Virtual nodes	Disabled
VM scale sets	Enabled

Authentication

Enable RBAC	Yes
-------------	-----

Networking

HTTP application routing	No
Load balancer	Standard
Network configuration	Basic

Monitoring

Enable container monitoring	Yes
Log Analytics workspace	(new) DefaultWorkspace-a4339399-5b72-463b-88f9-e67030102086-WUS2

Create < Previous Next > Download a template for automation

Figure 2.9: The final validation of your cluster configuration

Deploying your cluster should take roughly 15 minutes. Once the deployment is complete, you can check the deployment details as shown in *Figure 2.10*:

The screenshot shows the 'microsoft.aks-20200201113045 - Overview' page. At the top, there's a navigation bar with 'Home > microsoft.aks-20200201113045 - Overview'. Below it is a toolbar with 'Delete', 'Cancel', 'Redeploy', and 'Refresh' buttons. On the left, a sidebar has 'Overview' selected, along with 'Inputs', 'Outputs', and 'Template' options. The main content area displays a green checkmark icon and the message 'Your deployment is complete'. It provides deployment details: Deployment name: microsoft.aks-20200201113045, Subscription: Azure subscription 1, Start time: 2/1/2020, 11:43:54 AM, Correlation ID: ef4edebe-e3fa-4d68-a6ba-6c0a1cdb1dc3, and Resource group: handsonaks. A 'Deployment details (Download)' button is available. Below this, a table lists resources with status OK: ClusterMonitoringMetricP, handsonaks, SolutionDeployment-202C, and WorkspaceDeployment-2C. A 'Next steps' section and a 'Go to resource' button are also present.

Figure 2.10: Deployment details once the cluster is successfully deployed

If you get a quota limitation error, as shown in *Figure 2.11*, check the settings and try again. Make sure that you selected the **D1_v2** node size and only two nodes:

The screenshot shows the 'microsoft.aks-20181026 - Overview' page. A red error banner at the top says 'The resource operation completed with terminal provisioning state 'Failed''. The main content area displays a red exclamation mark icon and the message 'Your deployment failed'. It instructs the user to check the status of their deployment, manage resources, and go to their dashboard. Below this, it shows deployment details: Deployment name: microsoft.aks-20181026, Subscription: Free Trial, Resource group: handsonaks. A 'DEPLOYMENT DETAILS (Download)' button is available. At the bottom, a table lists resources with status Failed: myfirstakscluster, SolutionDeployment, and WorkspaceDeployment. To the right, an 'Errors' panel is open under the 'Raw Error' tab. It shows an error detail: 'The resource operation completed with terminal provisioning state 'Failed'. (Code: ResourceDeploymentFailure)'. It includes a link to a troubleshooting article: 'Provisioning of resource(s) for container service myfirstakscluster in resource group handsonaks failed. Message: Operation results in exceeding quota limits of Core. Maximum allowed: 4, Current in use: 0, Additional requested: 6. Please read more about quota increase at http://aka.ms/corequotaincrease.. Details: (Code: QuotaExceeded)'. There's also a 'WAS THIS HELPFUL?' button. Below the errors, there's a 'Troubleshooting Options' section with links to 'Common Azure deployment errors', 'Check Usage + Quota', and 'New Support Request'.

Figure 2.11: Retrying with a smaller cluster size due to a quota limit error

To move to the next section, in which we'll have a quick look at our cluster, hit the **Go to resource** button, which will take you to the AKS blade 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 should now see the overview of your cluster in the Azure portal:

The screenshot shows the AKS blade in the Azure portal for the 'handsonaks' Kubernetes service. The left sidebar contains navigation links for Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Settings (with Node pools 1, Upgrade 2, Scale, Networking, Dev Spaces, Deployment center (preview), Policies (preview), Properties, Locks, Export template), Monitoring (with Insights 3, Metrics, Logs, Workbooks), and Resource groups. The main content area displays cluster details: Resource group (rg-handsonaks), Status (Succeeded), Location (West US 2), Subscription (Azure subscription 1), and Node pools (1 node pools). It also shows API server address and HTTP application routing domain. Below the details are two cards: 'Monitor containers' (Get health and performance insights, Go to Azure Monitor insights) and 'View logs' (Search and analyze logs using ad-hoc queries, Go to Azure Monitor logs).

Figure 2.12: The AKS blade in the Azure portal

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

The first interesting option is the **Node pools** option. In the node pools view, you 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 server size, and you can also upgrade your node pools individually. In Figure 2.13, you can see the **Add node pool** option at the top left, and the options to **Scale** or **Upgrade** in the menu on the right:

Name	Provisioning state	Kubernetes version	OS type	Node count	Node size
agentpool	Succeeded	1.15.7	Linux	2	

...

- [Scale](#)
- [Upgrade](#)
- [Delete](#)

Figure 2.13: Adding, scaling, and upgrading node pools

The second interesting blade is the **Upgrade** blade. Here, you can instruct AKS to upgrade the management plane to a newer version. Typically, in a Kubernetes upgrade, you first upgrade the master plane, and then the individual node pools separately:

You can upgrade your cluster to a newer version of Kubernetes. This will upgrade the control plane components of your cluster. To upgrade your node pools, go to the 'Node pools' menu item instead.

[Learn more about upgrading your AKS cluster](#)
[View the Kubernetes changelog](#)

Kubernetes version

1.15.7 (current)
 ▼

Figure 2.14: Upgrading the Kubernetes version of the API sever using the Upgrade blade

The final interesting place to investigate is **Insights**. The **Insights** option provides you with monitoring of your cluster infrastructure and the workloads running on your cluster. Since our cluster is brand new, there isn't a lot of data to investigate. We will return here later in *Chapter 7, Monitoring the AKS cluster and the application*:

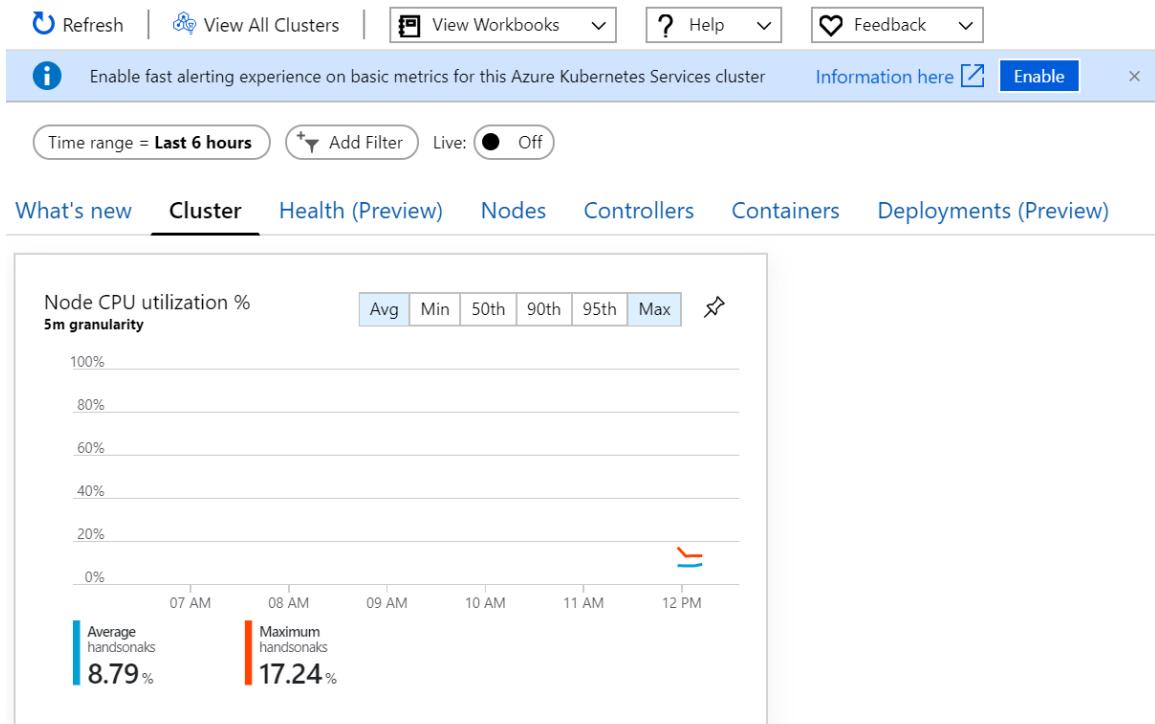


Figure 2.15: Displaying cluster utilization using the Insights blade

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 our 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 2.16, and click it:

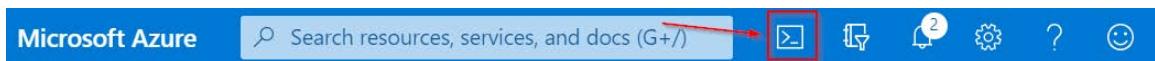


Figure 2.16: Clicking the Cloud Shell icon to open Azure Cloud Shell

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



X

Welcome to Azure Cloud Shell

Select Bash or PowerShell. You can change shells any time via the environment selector in the Cloud Shell toolbar. The most recently used environment will be the default for your next session.

A red rectangular box highlights the "Bash" button in the toolbar, while the "PowerShell" button is shown in blue.

Figure 2.17: 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.

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

A screenshot of the Azure Cloud Shell interface. The toolbar at the top shows "Bash" selected. A red box highlights the power button icon. The terminal window displays the following text:

```
Requesting a Cloud Shell. Succeeded.  
Connecting terminal...  
  
Welcome to Azure Cloud Shell  
  
Type "az" to use Azure CLI  
Type "help" to learn about Cloud Shell  
  
Warning: Failed to mount the Azure file share. Your cloud drive won't be available.
```

Figure 2.18: Hitting the restart button on getting a mount storage error

Click on the power button. It should restart, and you should see something similar to Figure 2.19:

A screenshot of the Azure Cloud Shell interface after a restart. The toolbar at the top shows "Bash" selected. The terminal window displays the following text:

```
Your cloud drive has been created in:  
  
Subscription Id: a4339399-5b72-463b-88f9-e67030102086  
Resource group: cloud-shell-storage-eastus  
Storage account: cs2a43393995b72x463bx88f  
File share: cs-handsontaksdemo-outlook-com-1003200099078f85  
  
Initializing your account for Cloud Shell...\\"  
Requesting a Cloud Shell. Succeeded.  
Connecting terminal...
```

Figure 2.19: Launching Cloud Shell successfully

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

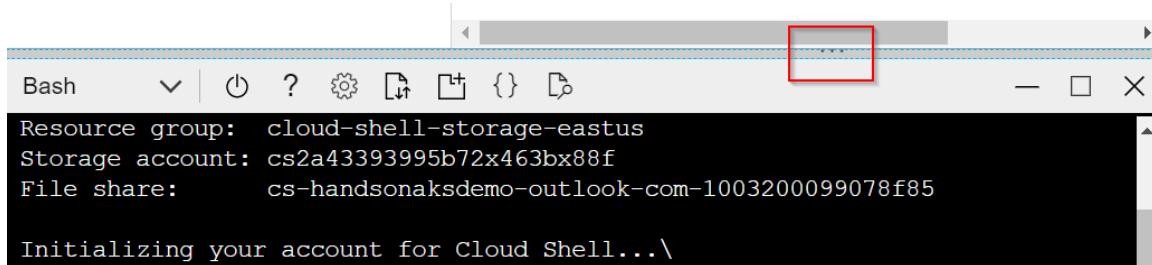


Figure 2.20: Using the divider to make the Cloud Shell larger or smaller

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-hansonaks --name handsonaks
```

To verify that you have access, type the following:

```
kubectl get nodes
```

You should see something like Figure 2.21:

NAME	STATUS	ROLES	AGE	VERSION
aks-agentpool-42828616-vmss000000	Ready	agent	7m26s	v1.15.7
aks-agentpool-42828616-vmss000001	Ready	agent	6m52s	v1.15.7

Figure 2.21: Output of the `kubectl get nodes` command

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

Deploying your first demo application

You are all connected. We are now going to launch our first application. 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 graphical code editor by typing the `code` command. We will be using the graphical editor in our examples but feel free to use the tool you feel most comfortable with.

For the purpose of this book, all the code examples are hosted on 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---  
Second-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/
```

We will use the code directly there for now. At this point in the book, we will not focus on what is in the `YAML` 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 they are built and how you could create your own.

We 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:

```
apiVersion: apps/v1  
kind: Deployment  
metadata:  
  name: azure-vote-back  
spec:  
  replicas: 1  
  selector:  
    matchLabels:  
      app: azure-vote-back
```

```
template:
  metadata:
    labels:
      app: azure-vote-back
  spec:
    containers:
      - name: azure-vote-back
        image: redis
        resources:
          requests:
            cpu: 100m
            memory: 128Mi
          limits:
            cpu: 250m
            memory: 256Mi
        ports:
          - containerPort: 6379
            name: redis
    ---
apiVersion: v1
kind: Service
metadata:
  name: azure-vote-back
spec:
  ports:
    - port: 6379
  selector:
    app: azure-vote-back
    ---
apiVersion: apps/v1
kind: Deployment
metadata:
```

```
name: azure-vote-front

spec:
  replicas: 1
  selector:
    matchLabels:
      app: azure-vote-front
  template:
    metadata:
      labels:
        app: azure-vote-front
    spec:
      containers:
        - name: azure-vote-front
          image: microsoft/azure-vote-front:v1
      resources:
        requests:
          cpu: 100m
          memory: 128Mi
        limits:
          cpu: 250m
          memory: 256Mi
      ports:
        - containerPort: 80
      env:
        - name: REDIS
          value: "azure-vote-back"
---
apiVersion: v1
kind: Service
metadata:
  name: azure-vote-front
spec:
```

```

type: LoadBalancer
ports:
- port: 80
selector:
  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 right-hand corner, and then **Save** to save the file:

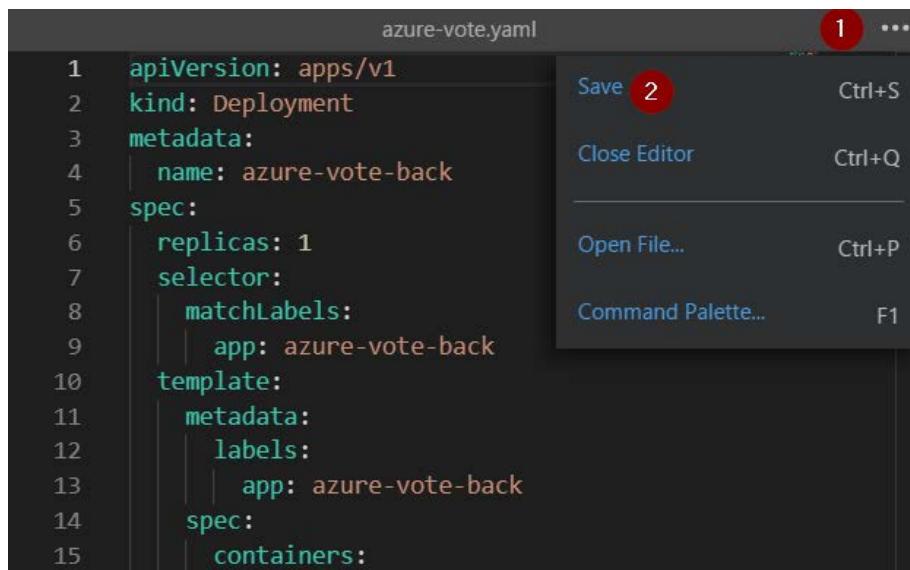


Figure 2.22: Hitting [...] to save the file

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 shown in *Figure 2.23* that 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 2.23: Output of the `kubectl create` command

You can check the progress by typing the following:

```
kubectl get pods
```

If you typed this quickly, you might have seen that a certain pod was still in the **ContainerCreating** process:

NAME	READY	STATUS	RESTARTS	AGE
azure-vote-back-5775d78ff5-9qjth	0/1	ContainerCreating	0	3s
azure-vote-front-559d85d4f7-km96f	1/1	Running	0	3s

Figure 2.24: 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** until the status of all pods is **Running**. Setting up all the pods takes some time and you can 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** (**command + C** on Mac).

In order to access our application publicly, we need to wait for one more thing. Now we want to know the public IP of the load balancer so that we can access it. If you remember from *Chapter 1, Introduction to Docker and Kubernetes*, a service in Kubernetes will create an Azure load balancer. This load balancer will get a public IP in our application so we 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** (**command + C** on Mac) to exit:

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
azure-vote-back	ClusterIP	10.0.251.78	<none>	6379/TCP	8s
azure-vote-front	LoadBalancer	10.0.27.185	<pending>	80:30035/TCP	7s
kubernetes	ClusterIP	10.0.0.1	<none>	443/TCP	4m3s
azure-vote-front	LoadBalancer	10.0.27.185	52.156.147.82	80:30035/TCP	96s

Figure 2.25: Change in the service IP from pending to the actual IP address

Note the external IP address, and type it in a browser. You should see the output shown in Figure 2.26:

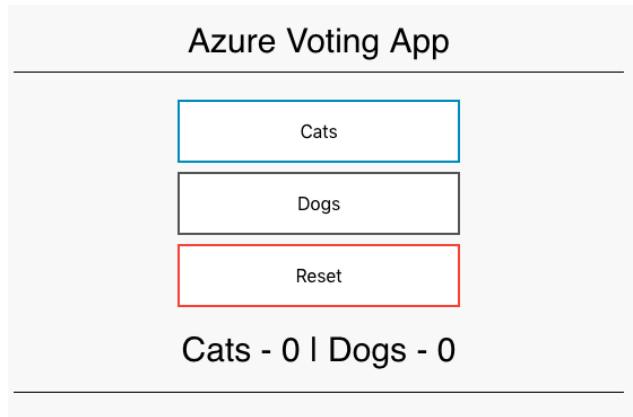


Figure 2.26: The actual application you just launched

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

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 it to the outside world, as well as providing storage for the services.

Before moving on to the next chapter, we will clean up our deployment. Since we created everything from a file, we 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 2.27: Cleaning up the deployment

In this section, we have connected to our AKS cluster using Cloud Shell, successfully launched and connected to a demo application, and finally, cleaned up the resources that were created.

Summary

Having completed this chapter, you are able to access and navigate the Azure portal to perform all the functions required to deploy an AKS cluster. You used the free trial on Azure to your advantage to learn the ins and outs of AKS and other Azure services. You launched your own AKS cluster with the ability to customize configurations if required using the Azure portal.

You also used Azure Cloud Shell without installing anything on your computer. This is important for all the upcoming sections, where you will be doing more than launching simple applications. Finally, you launched a publicly accessible service. The skeleton of this application is the same as for the complex applications that you will be launching in 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, we have covered the basics of Docker and Kubernetes and setup a Kubernetes cluster on Azure. In this section, we will cover how to deploy applications on top of that Kubernetes cluster.

Throughout this section, we will progressively build and deploy different applications on top of AKS. We 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 in 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 and Azure AD*
- *Chapter 7, Monitoring the AKS cluster and the application*

3

Application deployment on AKS

In this chapter, we 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 the 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 Deployment, ReplicaSet, 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. WordPress installation makes use of persistent storage in Kubernetes. 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
- 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

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 for scale. 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. Along the way, we will explain Kubernetes concepts such as deployments and ReplicaSets.

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. It's among the most popular container images according to <https://www.datadoghq.com/docker-adoption/>.

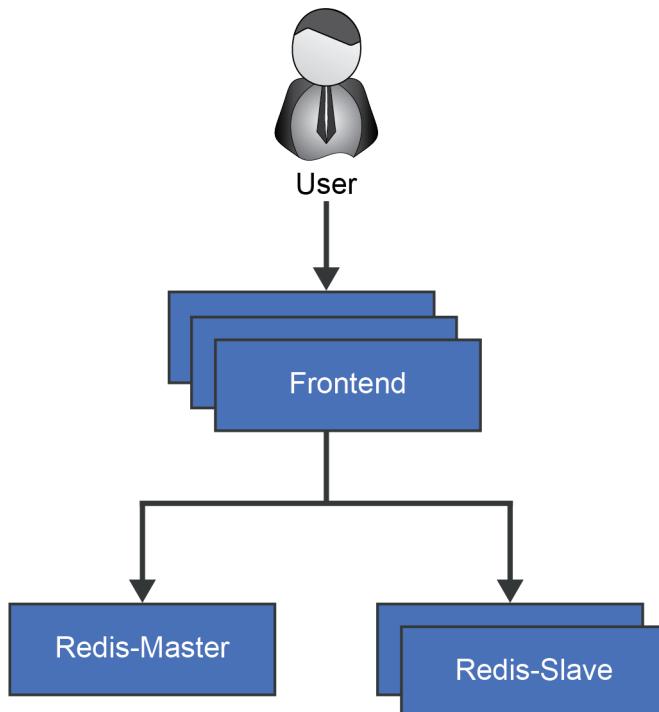


Figure 3.2: High-level overview of the guestbook application

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 Cloud Shell, as highlighted in *Figure 3.3*:

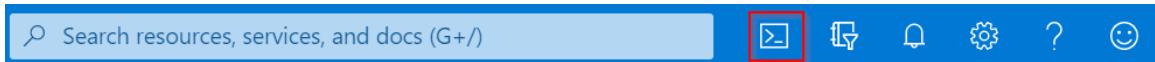


Figure 3.3: Opening the Cloud Shell

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

3. 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:

```
1 apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2  
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 # or just image: redis
24   resources:
25     requests:
26       cpu: 100m
27       memory: 100Mi
28   ports:
29     - containerPort: 6379
```

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

- **Line 2:** This states that we are creating a **Deployment**. As explained in *Chapter 1, Introduction to Docker 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:** 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:** Adds labels to the running instance so that it can be grouped and connected to other containers. We will discuss them later to see how they are used.
- **Line 22:** Gives this container 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 Docker 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 28-29:** These two lines indicate that the container is going to listen on port `6379`.
- **Lines 24-27:** Sets the `cpu/memory` resources requested for the container. 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 104857600 bytes, which is equal to ~105MB (<https://Kubernetes.io/docs/concepts/configuration/manage-compute-resources-container/>). You can also set CPU and memory limits 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 best practice in Kubernetes.

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 the output displayed in *Figure 3.4*:

user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter03\$ kubectl get all					
NAME	READY	STATUS	RESTARTS	AGE	
pod/redis-master-545d695785-5d96v	1/1	Running	0	19m	
NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
service/kubernetes	ClusterIP	10.0.0.1	<none>	443/TCP	3d2h
NAME	READY	UP-TO-DATE	AVAILABLE	AGE	
deployment.apps/redis-master	1/1	1	1	19m	
NAME	DESIRED	CURRENT	READY	AGE	
replicaset.apps/redis-master-545d695785	1	1	1	19m	

Figure 3.4: Output displaying the objects that were created by your deployment

You can see that we have a deployment named **redis-master**. It controls a ReplicaSet of **redis-master-<random id>**. On further examination, you will also find that the ReplicaSet is controlling a Pod, **redis- master-<replica set random id>-<random id>**. *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:  Tue, 04 Feb 2020 22:09:24 +0000
Labels:         app=redis
Annotations:    deployment.kubernetes.io/revision: 1
                kubectl.kubernetes.io/last-applied-configuration:
                  {"apiVersion":"apps/v1","kind":"Deployment","metadata":{"annotations":{},"labels":{"app":"redis"
}}, "name":"redis-master", "namespace":"defau...
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-545d695785 (1/1 replicas created)
Events:
  Type      Reason          Age      From            Message
  ----      ----   ----   -----
  Normal   ScalingReplicaSet 24m     deployment-controller  Scaled up replica set redis-master-545d695785 to 1
```

Figure 3.5: Output of describing 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, we will introduce a new concept called ConfigMaps and then recreate the Redis master. So, before proceeding, we need to clean up the current version, and we can do so by running the following command:

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

Executing this command will produce the following output:

```
deployment.extensions "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, we 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 configuration. It has a key-value pair for data that needs to be set on a container. A ConfigMap is used for non-secret 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, Securing your AKS cluster* of this book.

In this example, we are going to create a ConfigMap. In this ConfigMap, we will configure **redis-config** as the key and the value will be:

```
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

We will explore each one in detail.

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 it 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
```

3. You should get an output as follows:

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

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

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

5. 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: Output of describing 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:

```
apiVersion: v1
data:
  redis-config: |-  
    maxmemory 2mb  
    maxmemory-policy allkeys-lru
kind: ConfigMap
metadata:
  name: example-redis-config
  namespace: default
```

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

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

4. You should get an output as follows:

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

5. Next, run the following command:

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

6. This command returns the same output as the previous one:

```
Name:           example-redis-config
Namespace:      default
Labels:          <none>
Annotations:    <none>
Data
=====
redis-config:  
----  
maxmemory 2mb  
maxmemory-policy allkeys-lru
Events:         <none>
```

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

Note:

`kubectl get` has the useful option `-o`, 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:

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`, which has the necessary changes applied to it.

```
1 apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2
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
```

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:** Shows 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:** Gives the volume the name **config**. This name will be used within the context of this Pod.
- **Lines 41-42:** Declare 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.

2. Let's create this updated deployment:

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

3. This should output the following:

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

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

```
kubectl get pods
```

5. Then **exec** into the Pod and verify that the settings were applied:

```
kubectl exec -it redis-master-<pod-id> redis-cli
127.0.0.1:6379>; CONFIG GET maxmemory
    1) "maxmemory" 2) "2097152"
127.0.0.1:6379>; CONFIG GET maxmemory-policy
    "maxmemory-policy"
    "allkeys-lru" 127.0.0.1:6379>;exit
```

To summarize, you have just performed an important and tricky part of configuring cloud-native applications. 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.

Note

Connecting to a running container 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 we 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, we 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 for the back end and front end. 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 proxies traffic to the right Pods and does 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 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 proxied. 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.

We 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 3.7*:

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kubernetes	ClusterIP	10.0.0.1	<none>	443/TCP	3d22h
redis-master	ClusterIP	10.0.227.250	<none>	6379/TCP	7m36s

Figure 3.7: Output of the service that was created

You see that a new service, named **redis-master**, has been created. It has a cluster-wide IP of **10.0.227.250** (in your case, the IP will likely be different). Note that this IP will work only within the cluster (hence the **ClusterIP** type).

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 our case, that would be **redis-master.default.svc.cluster.local**. To see this in action, we'll do a name resolution on our **redis-master** VM. 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. 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 exit out of the Pod via the **exit** command, as shown in *Figure 3.8*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter03$ kubectl exec -it redis-master-79bc96f6cf-cxgsd bash
[ root@redis-master-79bc96f6cf-cxgsd:/data ]$ ping redis-master
PING redis-master.default.svc.cluster.local [10.0.227.250] 56(84) bytes of data.
^C
--- redis-master.default.svc.cluster.local ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms

[ root@redis-master-79bc96f6cf-cxgsd:/data ]$ exit
exit
command terminated with exit code 1
```

Figure 3.8: Using a ping command to view the FQDN of your service

In this section, you exposed the Redis master using a service. In the next section, you will deploy the Redis slaves.

Deploying the Redis slaves

Running a single back end on the cloud is not recommended. You can configure Redis in a master-slave setup. This means that you can have a master that will serve write traffic and multiple slaves 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-slave-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 3.9:

user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter03\$ kubectl get all					
NAME	READY	STATUS	RESTARTS	AGE	
pod/redis-master-79bc96f6cf-cxgsd	1/1	Running	0	19h	
pod/redis-slave-84548fdb-5qqjn	1/1	Running	0	118s	
pod/redis-slave-84548fdb-vdn4g	1/1	Running	0	118s	
NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
service/kubernetes	ClusterIP	10.0.0.1	<none>	443/TCP	3d23h
service/redis-master	ClusterIP	10.0.227.250	<none>	6379/TCP	54m
NAME	READY	UP-TO-DATE	AVAILABLE	AGE	
deployment.apps/redis-master	1/1	1	1	19h	
deployment.apps/redis-slave	2/2	2	2	118s	
NAME	DESIRED	CURRENT	READY	AGE	
replicaset.apps/redis-master-79bc96f6cf	1	1	1	19h	
replicaset.apps/redis-slave-84548fdb	2	2	2	118s	

Figure 3.9: Deploying the Redis slaves creates a number of new objects

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

```
1  apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2
2  kind: Deployment
3  metadata:
4    name: redis-slave
5    labels:
6      app: redis
7  spec:
8    selector:
9      matchLabels:
10       app: redis
11       role: slave
12       tier: backend
13   replicas: 2
14   template:
15     metadata:
16       labels:
17         app: redis
18         role: slave
19         tier: backend
20   spec:
21     containers:
22       - name: slave
23         image: gcr.io/google_samples/gb-redisslave:v1
24     resources:
25       requests:
26         cpu: 100m
27         memory: 100Mi
28     env:
29       - name: GET_HOSTS_FROM
30         value: dns
31         # Using 'GET_HOSTS_FROM=dns' requires your cluster to
32         # provide a dns service. As of Kubernetes 1.3, DNS is a
33         # built-in
34         # service launched automatically. However, if the cluster you
35         # are using
```

```

34      # does not have a built-in DNS service, you can instead
35      # access an environment variable to find the master
36      # service's host. To do so, comment out the 'value: dns' line
37      # above, and
38      # uncomment the line below:
39      # value: env
40      ports:
41          - 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 slave image.
 - **Lines 29-30:** Setting `GET_HOSTS_FROM` to `dns`. As you saw in the previous example, DNS resolves in the cluster.
4. Like the master service, you need to expose the slave service by running the following:

```
kubectl apply -f redis-slave-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:slave` label.

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

```
kubectl get service
```

This should give you the output shown in Figure 3.10:

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	3d23h
redis-master	ClusterIP	10.0.227.250	<none>	6379/TCP	62m
redis-slave	ClusterIP	10.0.227.158	<none>	6379/TCP	3s

Figure 3.10: Output displaying both a `redis-master` and a `redis-slave` 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 code:

```
kubectl get pods
```

This will display the output shown in *Figure 3.11*:

NAME	READY	STATUS	RESTARTS	AGE
frontend-678d98b8f7-48v5p	1/1	Running	0	63m
frontend-678d98b8f7-g5l5x	1/1	Running	0	63m
frontend-678d98b8f7-j5g5h	1/1	Running	0	63m
redis-master-79bc96f6cf-cxgsd	1/1	Running	0	20h
redis-slave-84548fdb-5qqjn	1/1	Running	0	77m
redis-slave-84548fdb-vdn4g	1/1	Running	0	77m

Figure 3.11: Output displaying the additional Pods running the front end

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 # for versions before 1.9.0 use apps/v1beta2 kind: Deployment
2 metadata:
3   name: frontend
4   labels:
5     app: guestbook
6   spec:
7     selector:
8       matchLabels:
9         app: guestbook
10        tier: frontend
```

```
11    replicas: 3
12    template:
13      metadata:
14        labels:
15          app: guestbook
16          tier: frontend
17    spec:
18      containers:
19        - name: php-redis
20          image: gcr.io/google-samples/gb-frontend:v4
21      resources:
22        requests:
23          cpu: 100m
24          memory: 100Mi
25      env:
26        - name: GET_HOSTS_FROM
27          value: dns
28          # Using GET_HOSTS_FROM=dns requires your cluster to
29          # provide a dns service. As of Kubernetes 1.3, DNS is a built-
30          # in
31          # service launched automatically. However, if the cluster you
32          # are using
33          # does not have a built-in DNS service, you can instead
34          # access an environment variable to find the master
35          # service's host. To do so, comment out the 'value: dns' line
above, and
36          # uncomment the line below:
37          # value: env
38      ports:
39        - 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 3.12*:

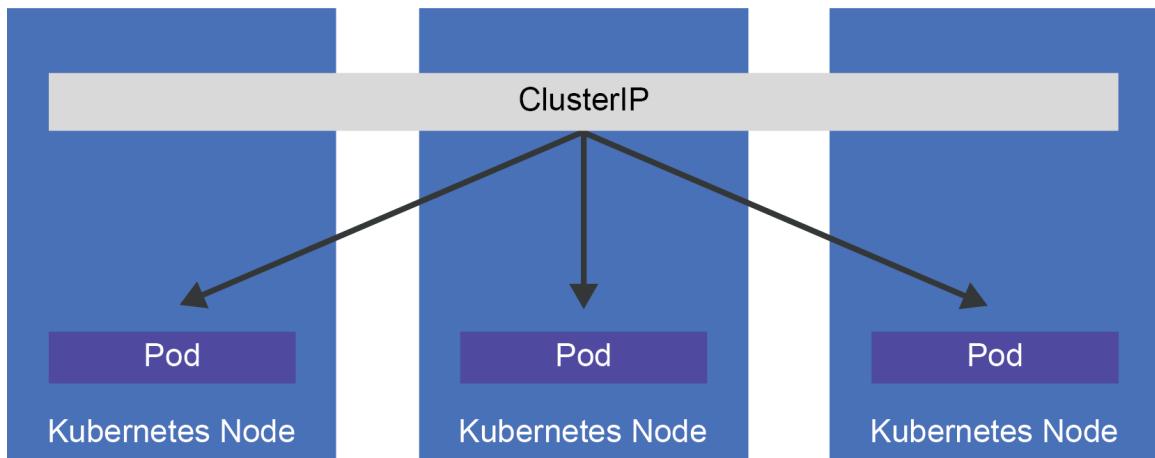


Figure 3.12: Kubernetes service of type ClusterIP

Another type of service is the type **NodePort**. This service would be exposed on a static port on each node as shown in *Figure 3.13*:

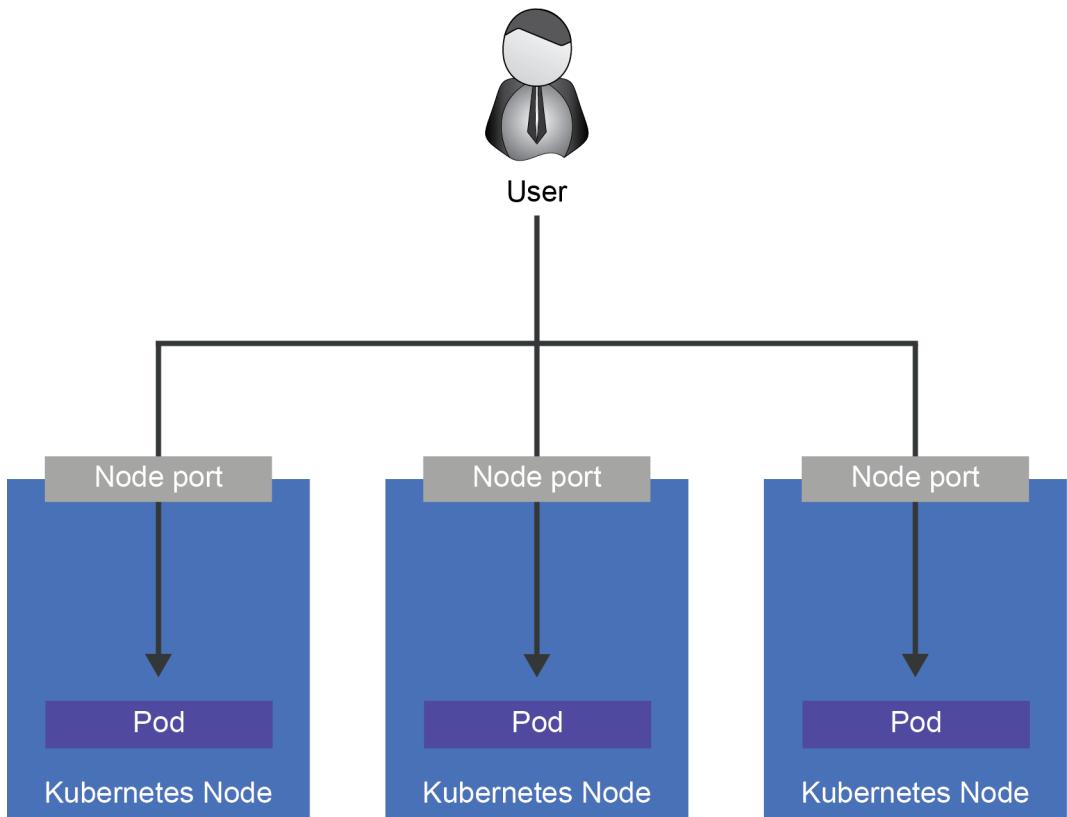


Figure 3.13: Kubernetes service of type NodePort

A final type – which we will use in our example – is the **LoadBalancer** type. This will create an Azure load balancer that will get a public IP that we can use to connect to, as shown in Figure 3.14:

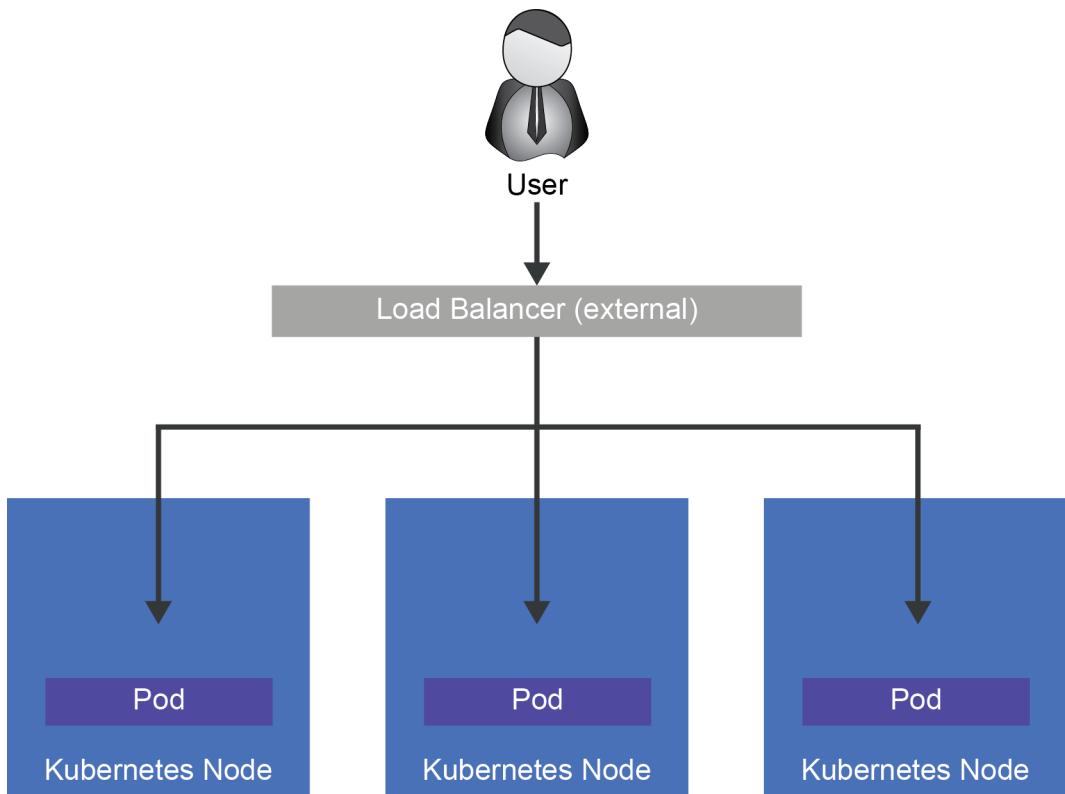


Figure 3.14: Kubernetes service of type LoadBalancer

The following code will help us to understand how a front-end 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    # comment or delete the following line if you want to use a
10   LoadBalancer
10   # type: NodePort # line commented out
```

```

11    # if your cluster supports it, uncomment the following to
12    # automatically create
13    type: LoadBalancer # line uncommented
14    ports:
15      - port: 80
16    selector:
17      app: guestbook
18      tier: frontend

```

- 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:

- 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.

- Run the following until there is a value in the **EXTERNAL-IP** column:

```
kubectl get service
```

This should display the output shown in *Figure 3.15*:

user@Azure:~\$ kubectl get service					
NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
frontend	LoadBalancer	10.0.32.90	51.143.53.78	80:31972/TCP	27m
kubernetes	ClusterIP	10.0.0.1	<none>	443/TCP	4d1h
redis-master	ClusterIP	10.0.227.250	<none>	6379/TCP	161m
redis-slave	ClusterIP	10.0.227.158	<none>	6379/TCP	99m

Figure 3.15: Output displaying a value for External IP after a while

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 3.16. The highlighted sections show you that there is a load balancing rule accepting traffic on port **80** and you have 2 public IP addresses:

The screenshot shows the Azure portal interface for a Kubernetes Load Balancer named 'kubernetes'. The left sidebar has options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Settings, Frontend IP configuration, and Backend pools. The main panel displays details such as Resource group (mc_handsontaks_handsontaks_westus2), Location (West US 2), Subscription (Azure subscription 1), Subscription ID (a4339399-5b72-463b-88f9-e67030102086), SKU (Standard), and Tags (Click here to add tags). The 'Load balancing rule' section shows a rule for port 80 with a specific IP address. The 'Public IP address' section indicates 2 public IP addresses.

Figure 3.16: Displaying the 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 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 a specific configuration to make outbound connections. This is why you see a second IP address configured.

We're finally ready to put our guestbook app into 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 3.17*:

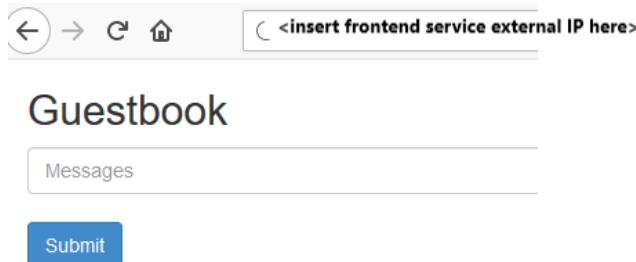


Figure 3.17: The guestbook application in action

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-slave  
kubectl delete service frontend redis-master redis-slave
```

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, we used static YAML files to deploy our 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**.

If you want a more thorough introduction to writing your own Helm Charts, you can refer to the following blog post by one of the authors of this book: <https://blog.nillsf.com/index.php/2019/11/23/writing-a-helm-v3-chart/>.

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 stable https://kubernetes-charts.storage.googleapis.com/
```

2. To install WordPress, we will run the following command:

```
helm install handsonakswp stable/wordpress
```

This execution will cause Helm to install the chart detailed at <https://github.com/helm/charts/tree/master/stable/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 our site that got created.

PersistentVolumeClaims

A 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/helm/charts/tree/master/stable/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 3.18:

user@Azure:~\$ kubectl get pods				
NAME	READY	STATUS	RESTARTS	AGE
handsonakswp-mariadb-0	1/1	Running	0	4m42s
handsonakswp-wordpress-68f5d6f857-kz4h7	1/1	Running	0	4m43s

Figure 3.18: Output displaying a predictable number for the MariaDB Pod, whereas a random name for the WordPress Pod

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 persistent volume. 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). Please refer to <https://Kubernetes.io/docs/concepts/storage/volumes/#persistentvolumeclaim> for more information.

StatefulSets require either a pre-provisioned volume or a dynamically provisioned volume handled by a **PersistentVolumeClaim (PVC)**. In our example, we are using a PVC. A PVC provides an abstraction over the underlying storage mechanism. Let's look at what the MariaDB Helm Chart did for us by running the following:

```
kubectl get statefulsets
```

This will show us something similar to Figure 3.19:

user@Azure:~\$ kubectl get statefulset		
NAME	READY	AGE
handsonakswp-mariadb	1/1	20m

Figure 3.19: Output displaying the StatefulSet that created the MariaDB Pods

Let's have a more in-depth look by exporting the YAML definition of our StatefulSet:

```
kubectl get statefulset -o yaml > mariadbss.yaml  
code 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
...
106        volumeMounts:
107          - mountPath: /bitnami/mariadb
108            name: data
...
128        volumeClaimTemplates:
129          - metadata:
130            creationTimestamp: null
131            labels:
132              app: mariadb
133              component: master
134              heritage: Helm
135              release: handsonakswp
136            name: data
137        spec:
138          accessModes:
139            - ReadWriteOnce
140          resources:
141            requests:
142              storage: 8Gi
143            volumeMode: Filesystem
...
```

Most of the elements of the preceding code have been covered earlier in the deployment. In the following block, 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.

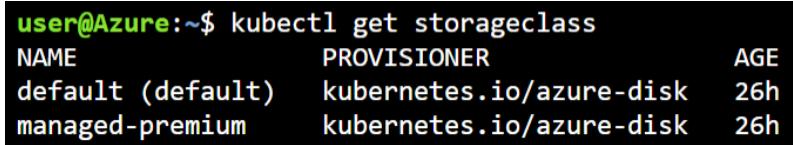
Let's discuss the different elements of the preceding code in detail:

- **Line 4:** This line indicates the **StatefulSet** declaration.
- **Lines 106-108:** Mount the volume defined as **data** and mount it under the **/bitnami/mariadb** path.
- **Lines 128-143:** Declare the PVC. Note specifically:
 - **Line 136:** This line gives it the name **data**, which is reused at line 108.
 - **Line 139:** Gives the access mode **ReadWriteOnce**, which will create block storage, which on Azure is a disk.
 - **Line 142:** Defines the size of the disk.

Based on the preceding information, Kubernetes dynamically requests and binds an 8Gi 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 we 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 3.20*:



A terminal window showing the output of the command 'kubectl get storageclass'. The output lists two storage classes: 'default' and 'managed-premium'. Both are provisioned by 'kubernetes.io/azure-disk' and are 26 hours old.

NAME	PROVISIONER	AGE
default (default)	kubernetes.io/azure-disk	26h
managed-premium	kubernetes.io/azure-disk	26h

Figure 3.20: Output displaying the 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 3.21*:

NAME	STATUS	VOLUME	CAPACITY	ACCESS MODES	STORAGECLASS	AGE
data-handsonakswp-mariadb-0	Bound	pvc-b27bed30-bb03-4196-b6c9-628ebb113796	8Gi	RWO	default	43m
data-wp-mariadb-0	Bound	pvc-11b39c3d-8759-42dd-9947-99bc07b15d33	8Gi	RWO	default	4d
handsonakswp-wordpress	Bound	pvc-ee1fccea3-9862-47cd-851d-9345242e19c3	10Gi	RWO	default	43m

Figure 3.21: 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:

The screenshot shows the Azure portal search interface. A search bar at the top contains the text "pvc-b27bed30-bb03-4196-b6c9-628ebb113796". Below the search bar are several sections: Services, Marketplace, Resources, Documentation, and Resource Groups. Under the Resources section, there is a card for "kubernetes-dynamic-pvc-b27bed30-bb03-4196-b6c9-... Disk". This card has a small icon of a green cloud with a blue disk, the long PVC name, and the word "Disk". Below the Resources section, a message says "No results were found." for Services, Marketplace, Documentation, and Resource Groups. At the bottom left, there is a note: "Searching all subscriptions. [Change](#)".

Figure 3.22: 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.

Checking the WordPress deployment

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

```
helm ls
```

This should return the output shown in *Figure 3.23*:

NAME	NAMESPACE	REVISION	UPDATED	STATUS	CHART	APP VERSION
handsonakswp	default	1	2020-02-05 21:23:08.335088153 +0000 UTC	deployed	wordpress-8.1.2	5.3.2

Figure 3.23: Helm shows that our WordPress application has been deployed

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 3.24*:

```
NAME: handsonakswp
LAST DEPLOYED: Wed Feb  5 21:23:08 2020
NAMESPACE: default
STATUS: deployed
REVISION: 1
NOTES:
1. Get the WordPress URL:

    NOTE: It may take a few minutes for the LoadBalancer IP to be available.
          Watch the status with: 'kubectl get svc --namespace default -w handsonakswp-wordpress'
    export SERVICE_IP=$(kubectl get svc --namespace default handsonakswp-wordpress --template "{{ range (index .status.loadBalancer.ingress 0) }} {{.}} {{ end }}")
    echo "WordPress URL: http://$SERVICE_IP/"
    echo "WordPress Admin URL: http://$SERVICE_IP/admin"

2. Login with the following credentials to see your blog

    echo Username: user
    echo Password: $(kubectl get secret --namespace default handsonakswp-wordpress -o jsonpath="{.data.wordpress-password}" | base64 --decode)
```

Figure 3.24: Getting more details about the application

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

```
kubectl get all
```

This will generate an output similar to Figure 3.25:

user@Azure:~\$ kubectl get all						
NAME		READY	STATUS	RESTARTS	AGE	
pod/handsonakswp-mariadb-0		1/1	Running	0	56m	
pod/handsonakswp-wordpress-68f5d6f857-kz4h7		1/1	Running	0	56m	
NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)		AGE
service/handsonakswp-mariadb	ClusterIP	10.0.162.122	<none>	3306/TCP		56m
service/handsonakswp-wordpress	LoadBalancer	10.0.33.40	52.149.54.61	80:32384/TCP,443:31739/TCP		56m
service/kubernetes	ClusterIP	10.0.0.1	<none>	443/TCP		4d2h
NAME		READY	UP-TO-DATE	AVAILABLE	AGE	
deployment.apps/handsonakswp-wordpress		1/1	1	1	56m	
NAME		DESIRED	CURRENT	READY	AGE	
replicaset.apps/handsonakswp-wordpress-68f5d6f857		1	1	1	56m	
NAME		READY	AGE			
statefulset.apps/handsonakswp-mariadb		1/1	56m			

Figure 3.25: Output displaying all the 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. The following screenshot is the resulting output:

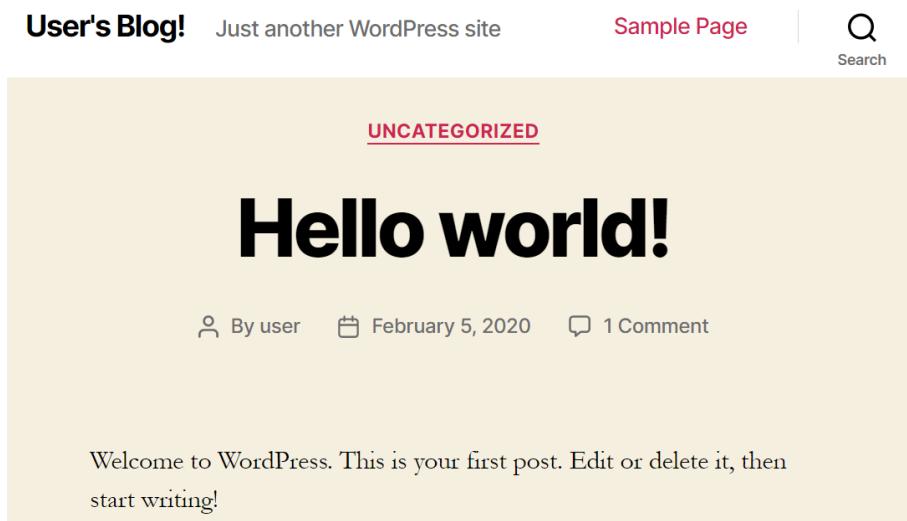


Figure 3.26: WordPress site being displayed on connection with the external IP

To make sure we 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, our PVCs won't be deleted. This ensures persistent data is kept. As we don't have any persistent data, we can safely delete those 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, we deployed two applications. We started the chapter by deploying the guestbook application. During that deployment, we looked into the details of Pods, ReplicaSets, and deployments. We also used dynamic configuration using ConfigMaps. Finally, we looked into how services are used to route traffic to the deployed applications.

The second application we deployed was a WordPress application. We deployed it via the Helm package manager. As part of this deployment, PVCs were used, and we explored how these were used in the system.

In the next chapter, we will look into scaling applications and the cluster itself. We will first look at the manual and automatic scaling of the application, and afterward, we'll look at the manual and automatic scaling of the cluster itself. Finally, we will explain different ways in which applications can be updated on Kubernetes.

4

Building scalable applications

When running an application, the ability to scale and upgrade your application is critical. **Scaling** is required to handle additional loads with your application, while upgrading is required to keep your application up to date and to be able 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 loads, you can scale the front end alone, while keeping the same number of back-end instances. You can increase or reduce the number/size of **Virtual Machines (VM)** required depending on your workload and peak demand hours. This chapter will cover both the scale dimensions in detail.

In this chapter, we will show you how to scale the sample guestbook application that we introduced in *Chapter 3, Application deployment on AKS*. We will first scale this application using manual commands, and afterward we'll 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 the [Azure Kubernetes Service \(AKS\)](#). After scaling the application itself, we will also scale the cluster. We'll first scale the cluster manually, and then use the cluster autoscaler to automatically scale the cluster. In addition, in this chapter, you will get a brief introduction to how you can upgrade applications running on top of AKS.

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

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

We will start this chapter by discussing the different dimensions when it comes to scaling applications on top of AKS.

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---Second-Edition/tree/SecondEdition
```

For this chapter, navigate to the **Chapter04** directory:

```
cd Chapter04
```

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 additional Pods to a deployment, also known as scaling out, you 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 the HPA. The HPA will watch metrics such as 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 either manually by changing the number of nodes, or you can use the cluster autoscaler to automatically scale out your cluster. The cluster autoscaler will watch 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.

Implementing scaling of 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:

1. Install 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, as 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-slave created
deployment.apps/redis-slave 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 svc
```

4. Figure 4.2 shows that none of the services have an external IP:

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
frontend	ClusterIP	10.0.19.106	<none>	80/TCP	2m52s
kubernetes	ClusterIP	10.0.0.1	<none>	443/TCP	4d4h
redis-master	ClusterIP	10.0.107.141	<none>	6379/TCP	2m52s
redis-slave	ClusterIP	10.0.212.71	<none>	6379/TCP	2m52s

Figure 4.2: Output displaying none of the services having a public IP

5. To test out our application, we will expose it publicly. For this, we will 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. Navigate to the line that now says **type: ClusterIP** (line 27) and change that to **type: LoadBalancer**, as shown in Figure 4.3. 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:

```
spec:
  clusterIP: 10.0.135.85
  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

- 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
```

- 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` (`command + C` on Mac):

user@Azure:~\$ kubectl get svc -w						
NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE	
frontend	LoadBalancer	10.0.19.106	<pending>	80:31807/TCP	18h	
kubernetes	ClusterIP	10.0.0.1	<none>	443/TCP	4d22h	
redis-master	ClusterIP	10.0.107.141	<none>	6379/TCP	18h	
redis-slave	ClusterIP	10.0.212.71	<none>	6379/TCP	18h	
frontend	LoadBalancer	10.0.19.106	51.143.111.108	80:31807/TCP	18h	

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

- 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. This will cause Kubernetes to add additional Pods to the deployment:

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

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 an output as shown in Figure 4.6:

NAME	READY	STATUS	RESTARTS	AGE
frontend-57d8c9fb45-86872	1/1	Running	0	3m
frontend-57d8c9fb45-cc77m	1/1	Running	0	73s
frontend-57d8c9fb45-fc4f9	1/1	Running	0	3m
frontend-57d8c9fb45-hq7kn	1/1	Running	0	3m
frontend-57d8c9fb45-k5g5h	1/1	Running	0	73s
frontend-57d8c9fb45-sgxhv	1/1	Running	0	73s
redis-master-545d695785-xtp7b	1/1	Running	0	3m
redis-slave-84548fdbca-lzv4j	1/1	Running	0	3m
redis-slave-84548fdbca-pmpf8	1/1	Running	0	3m

Figure 4.6: 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 an output as follows:

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE	NOMINATED NODE	READINESS GATES
frontend-57d8c9fb45-86872	1/1	Running	0	3m23s	10.244.0.98	aks-agentpool-39838025-vmss000000	<none>	<none>
frontend-57d8c9fb45-cc77m	1/1	Running	0	96s	10.244.0.99	aks-agentpool-39838025-vmss000000	<none>	<none>
frontend-57d8c9fb45-fc4f9	1/1	Running	0	3m23s	10.244.5.59	aks-agentpool-39838025-vmss000005	<none>	<none>
frontend-57d8c9fb45-hq7kn	1/1	Running	0	3m23s	10.244.4.52	aks-agentpool-39838025-vmss000004	<none>	<none>
frontend-57d8c9fb45-k5g5h	1/1	Running	0	96s	10.244.1.78	aks-agentpool-39838025-vmss000001	<none>	<none>
frontend-57d8c9fb45-sgxhv	1/1	Running	0	96s	10.244.5.60	aks-agentpool-39838025-vmss000005	<none>	<none>
redis-master-545d695785-xtp7b	1/1	Running	0	3m23s	10.244.4.54	aks-agentpool-39838025-vmss000004	<none>	<none>
redis-slave-84548fdbca-lzv4j	1/1	Running	0	3m23s	10.244.4.53	aks-agentpool-39838025-vmss000004	<none>	<none>
redis-slave-84548fdbca-pmpf8	1/1	Running	0	3m23s	10.244.0.97	aks-agentpool-39838025-vmss000000	<none>	<none>

Figure 4.7: 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.

Using the HPA

Scaling manually is useful when you're working on your cluster. 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)**.

The HPA monitors Kubernetes metrics at regular intervals and, based on rules you define, autoscales 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 1 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/v2beta1
2 kind: HorizontalPodAutoscaler
3 metadata:
4   name: frontend-scaler
5 spec:
6   scaleTargetRef:
7     apiVersion: extensions/v1beta1
8     kind: Deployment
9     name: frontend
10  minReplicas: 1
11  maxReplicas: 10
12  metrics:
13    - type: Resource
14      resource:
15        name: cpu
16        targetAverageUtilization: 25
```

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 12-16:** Here, we define the metric that Kubernetes will be monitoring, in order to do the scaling.

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 4.8*:

user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04\$ kubectl get hpa						
NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS	AGE
frontend-scaler	Deployment/frontend	<unknown>/25%	1	10	0	3s

Figure 4.8: 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 4.9*:

user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04\$ kubectl get hpa --watch						
NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS	AGE
frontend-scaler	Deployment/frontend	<unknown>/25%	1	10	0	7s
frontend-scaler	Deployment/frontend	10%/25%	1	10	1	16s

Figure 4.9: Once the target shows a percentage, the HPA is ready

4. We'll now go ahead and do two things: first, we will watch our Pods to see whether new Pods are created. Then, we will create a new shell, and create some load for our 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 button to open a new shell:



Figure 4.10: 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. We will generate some load for our application from this tab.

5. Next, we will use a program called **hey** to generate this load. **hey** is a tiny program that sends loads to a web application. We 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 our front end. This will generate CPU loads on our system, which will trigger the HPA to start scaling our 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 4.11:

NAME	READY	STATUS	RESTARTS	AGE
frontend-678d98b8f7-9qj4w	1/1	Running	0	6m51s
redis-master-545d695785-gzf9b	1/1	Running	0	20h
redis-slave-84548fdb-4k9b2	1/1	Running	0	20h
redis-slave-84548fdb-jp2xx	1/1	Running	0	20h
frontend-678d98b8f7-d8lwp	0/1	Pending	0	0s
frontend-678d98b8f7-cfwch	0/1	Pending	0	0s
frontend-678d98b8f7-x84v9	0/1	Pending	0	0s
frontend-678d98b8f7-d8lwp	0/1	Pending	0	0s
frontend-678d98b8f7-x84v9	0/1	Pending	0	0s
frontend-678d98b8f7-cfwch	0/1	Pending	0	0s
frontend-678d98b8f7-d8lwp	0/1	ContainerCreating	0	0s
frontend-678d98b8f7-cfwch	0/1	ContainerCreating	0	0s
frontend-678d98b8f7-x84v9	0/1	ContainerCreating	0	0s
frontend-678d98b8f7-cfwch	1/1	Running	0	2s
frontend-678d98b8f7-d8lwp	1/1	Running	0	3s

Figure 4.11: New Pods get started by the HPA

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

- Let's have a closer look at what our 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 4.12*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04$ kubectl describe hpa
Name:                      frontend-scaler
Namespace:                  default
Labels:                     <none>
Annotations:                <none>
CreationTimestamp:          Sun, 08 Mar 2020 22:21:22 +0000
Reference:                 Deployment/frontend
Metrics:                   resource cpu on pods  (as a percentage of request): 132% (13m) / 25% ①
Min replicas:              1
Max replicas:              10
Deployment pods:           10 current / 10 desired
Conditions:
  Type      Status  Reason
  ----      ----  -----
  AbleToScale  True   ScaleDownStabilized
  ScalingActive True   ValidMetricFound
  ScalingLimited True   TooManyReplicas ②
Events:
  Type     Reason        Age   From            Message
  ----     ----        --   --  -----
  Normal   SuccessfulRescale 99s  horizontal-pod-autoscaler  New size: 4; reason: cpu resource utilization (percentage of request) above target
  Normal   SuccessfulRescale 84s  horizontal-pod-autoscaler  New size: 8; reason: cpu resource utilization (percentage of request) above target
  Normal   SuccessfulRescale 69s  horizontal-pod-autoscaler  New size: 10; reason: cpu resource utilization (percentage of request) above target
```

Figure 4.12: Detailed view of the HPA

The annotations in *Figure 4.12* are explained as follows:

- This shows you the current CPU utilization (132%) versus the desired (25%). 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 we 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.
- 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 4.13:

NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS	AGE
frontend-scaler	Deployment/frontend	182%/25%	1	10	10	9m9s
frontend-scaler	Deployment/frontend	10%/25%	1	10	10	10m
frontend-scaler	Deployment/frontend	10%/25%	1	10	10	14m
frontend-scaler	Deployment/frontend	10%/25%	1	10	4	15m

Figure 4.13: Watching the HPA scale down

- 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, we first manually and then automatically scaled our application. However, the cluster resources were static; we ran this on a two-node cluster. In many cases, you might also run out of resources on the cluster itself. In the next section, we will deal with that issue, and explain how you can scale your AKS cluster yourself.

Scaling your cluster

In the previous section, we dealt with scaling the application running on top of a cluster. In this section, we'll explain how you can scale the actual cluster you are running. We will first discuss how you can manually scale your cluster. We'll start with scaling down our cluster to one node. Then, we'll configure the cluster autoscaler. The cluster autoscaler will monitor our cluster and will scale out when there are Pods that cannot be scheduled on our 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 via the command line.

In this section, we'll show you how you can manually scale your cluster by scaling the cluster 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 4.14*:

The screenshot shows the Azure portal interface for a Kubernetes service named "handsonaks". The left sidebar includes links for Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Settings, and Node pools (which is highlighted with a red circle labeled '1'). The main content area displays a table of node pools. One row is selected, showing "agentpool" with a provisioning status of "Succeeded", OS type "Linux", and a node count of "2" (which is also circled in red with '2'). A button to "Add node pool" and a "Refresh" button are visible at the top of the table.

Name	Provisioning...	Kubernetes ...	OS type	Node count	Node size
agentpool	Succeeded	1.15.7	Linux	2	Standard_D1... ***

Figure 4.14: Manually scaling the cluster

2. This will open a pop-up 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 4.15*:

The screenshot shows a "Scale" dialog box. At the top, it says "Scale method" with "Manual" selected (highlighted in purple). Below that is a "Node count" section with a slider set to "1", indicating "1 x Standard D1 v2 (1 vcpu, 3 GiB memory)". At the bottom, a "Total cluster capacity" table shows the following changes:

Cores	2 vCPUs	→	1 vCPU (new)
Memory	6 GiB	→	3 GiB (new)

Figure 4.15: Pop-up 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:

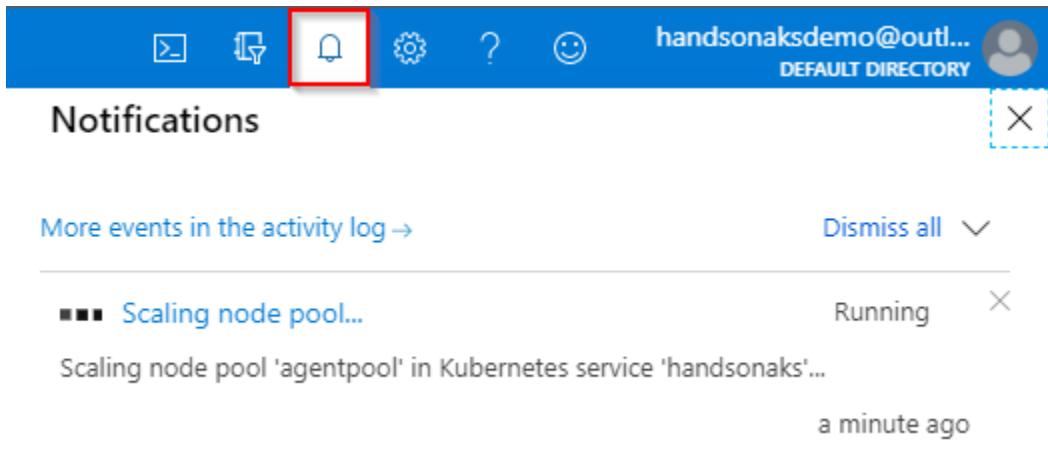


Figure 4.16: Cluster scaling can be followed using the notifications in the Azure portal

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

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

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

Scaling your cluster using the cluster autoscaler

In this section, we 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. We will first force our deployment to have Pods that cannot be scheduled, and then we will configure the cluster autoscaler to automatically scale our cluster.

To force our cluster to be out of resources, we will—manually—scale up the **redis-slave** deployment. To do this, use the following command:

```
kubectl scale deployment redis-slave --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 an output generated in *Figure 4.17*:

NAME	READY	STATUS	RESTARTS	AGE
frontend-57d8c9fb45-2vphh	1/1	Running	0	8s
frontend-57d8c9fb45-nvz7m	1/1	Running	0	8s
frontend-57d8c9fb45-p8425	1/1	Running	0	8s
redis-master-545d695785-55t86	1/1	Running	0	8s
redis-slave-84548fdb-6wmcp	0/1	Pending	0	4s
redis-slave-84548fdb-9svtl	0/1	Pending	0	4s
redis-slave-84548fdb Pg5cc	0/1	Pending	0	4s
redis-slave-84548fdb-prfbx	1/1	Running	0	8s
redis-slave-84548fdb-v85v2	0/1	Pending	0	8s

Figure 4.17: Four out of five Pods are pending, meaning they cannot be scheduled

As you can see, we now have four Pods in a **Pending** state. The **Pending** state in Kubernetes means that that Pod cannot be scheduled onto a node. In our case, this is due to the cluster being out of resources.

We will now configure the cluster autoscaler to automatically scale our cluster. As in the 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, we will use the CLI to enable the cluster autoscaler. The following command will configure the cluster autoscaler for our cluster:

```
az aks nodepool update --enable-cluster-autoscaler \
-g rg-handsonaks --cluster-name handsonaks \
--name agentpool --min-count 1 --max-count 3
```

This command configures the cluster autoscaler on the nodepool we have in our cluster. It configures it to have a minimum of one node and a maximum of three 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 your 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** (*command + C* on Mac). You should see an output similar to the one in *Figure 4.18*:

NAME	STATUS	ROLES	AGE	VERSION
aks-agentpool-42828616-vmss000000	Ready	agent	14d	v1.15.7
aks-agentpool-42828616-vmss000000	Ready	agent	14d	v1.15.7
aks-agentpool-42828616-vmss000000	Ready	agent	14d	v1.15.7
aks-agentpool-42828616-vmss000007	NotReady	agent	0s	v1.15.7
aks-agentpool-42828616-vmss000007	NotReady	agent	0s	v1.15.7
aks-agentpool-42828616-vmss000007	NotReady	agent	0s	v1.15.7
aks-agentpool-42828616-vmss000007	Ready	agent	0s	v1.15.7
aks-agentpool-42828616-vmss000007	Ready	agent	0s	v1.15.7
aks-agentpool-42828616-vmss000007	Ready	agent	16s	v1.15.7

Figure 4.18: The new node joins the cluster

The new node should ensure that our cluster has sufficient resources to schedule the scaled-out **redis-slave** 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:

NAME	READY	STATUS	RESTARTS	AGE
frontend-57d8c9fb45-2jssj	1/1	Running	0	15m
frontend-57d8c9fb45-8czcw	1/1	Running	0	15m
frontend-57d8c9fb45-qtc58	1/1	Running	0	15m
redis-master-545d695785-bnmvk	1/1	Running	0	15m
redis-slave-84548fdbcbmdj8	1/1	Running	0	13m
redis-slave-84548fdbcb2wk4	1/1	Running	0	13m
redis-slave-84548fdbcb8p4n	1/1	Running	0	5m40s
redis-slave-84548fdbcbnfvhm	1/1	Running	0	15m
redis-slave-84548fdbcbst9hx	1/1	Running	0	4m28s

Figure 4.19: All Pods are now in a Running state

We will now clean up the resources we created, disable the cluster autoscaler, and ensure that our cluster has two nodes for the next examples. 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 if the cluster already has two nodes. You can safely ignore this error.

In this section, we first manually scaled down our cluster, and then we used the cluster autoscaler to scale out our cluster. We first used the Azure portal to scale out the cluster manually, and then we also used the Azure CLI to configure the cluster autoscaler. In the next section, we will look into how we 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 fallbacks 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://services.google.com/fh/files/misc/state-of-devops-2019.pdf>) 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 we can make updates in a Kubernetes cluster. In this section, we will explore the following ways to update Kubernetes resources:

- Upgrading by changing YAML files
- Upgrading using `kubectl edit`
- Upgrading using `kubectl patch`
- Upgrading using Helm

The methods we will describe in the following section work great if you have stateless applications. If you have a state stored anywhere, make sure to back up that state before you try anything.

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

Upgrading by changing YAML files

In order to upgrade a Kubernetes service or deployment, we 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. We start with our guestbook application to demonstrate this example:

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

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

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

Uncomment line 108 in this file to set the type as **LoadBalancer** and save the file. as shown in *Figure 4.20*:

```

105 spec:
106   # if your cluster supports it, uncomment the following to automatically create
107   # an external load-balanced IP for the frontend service.
108   type: LoadBalancer
109   ports:
110     - port: 80
111     selector:
112       app: guestbook
113       tier: frontend

```

Figure 4.20: Changing this line 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
```

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

```
kubectl get svc
```

Give it a few minutes, and you should be shown the IP as displayed in *Figure 4.21*:

user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04\$ kubectl get svc						
NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE	
frontend	LoadBalancer	10.0.195.175	52.149.23.157	80:32664/TCP	90s	
kubernetes	ClusterIP	10.0.0.1	<none>	443/TCP	11h	
redis-master	ClusterIP	10.0.49.194	<none>	6379/TCP	91s	
redis-slave	ClusterIP	10.0.14.112	<none>	6379/TCP	91s	

Figure 4.21: Output displaying a public IP

5. We will now make another change. We'll downgrade the front-end image line on line 133 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
```

7. This will generate the following output:

```
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-slave unchanged
deployment.apps/redis-slave unchanged
service/frontend unchanged
deployment.apps/frontend configured
NAME          READY   STATUS        RESTARTS   AGE
frontend-55b5c47bb-2686k   0/1    ContainerCreating   0          1s
frontend-57d8c9fb45-4nctt   1/1    Running       0          26s
frontend-57d8c9fb45-9x47b   1/1    Running       0          26s
frontend-57d8c9fb45-ltrs7   1/1    Running       0          26s
redis-master-545d695785-mgjr9 1/1    Running       0          26s
redis-slave-84548fdb-rvvtx  1/1    Running       0          26s
redis-slave-84548fdb-v5664  1/1    Running       0          26s
frontend-55b5c47bb-2686k   1/1    Running       0          20s
frontend-57d8c9fb45-ltrs7   1/1    Terminating   0          45s
frontend-55b5c47bb-kx5zr    0/1    Pending       0          0s
frontend-55b5c47bb-kx5zr    0/1    Pending       0          0s
frontend-55b5c47bb-kx5zr    0/1    ContainerCreating   0          0s
frontend-57d8c9fb45-ltrs7   0/1    Terminating   0          46s
```

Figure 4.22: Pods from a new ReplicaSet are created

What you can see here is that the old version of the Pods (based on the old ReplicaSet) gets terminated, while the new version gets created.

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

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04$ kubectl get events | grep ReplicaSet
11m     Normal  ScalingReplicaSet  deployment/frontend  Scaled up replica set frontend-57d8c9fb45 to 3
116s    Normal  ScalingReplicaSet  deployment/frontend  Scaled up replica set frontend-57d8c9fb45 to 3
4m21s   Normal  ScalingReplicaSet  deployment/frontend  Scaled up replica set frontend-55b5c47bb to 1
3m26s   Normal  ScalingReplicaSet  deployment/frontend  Scaled down replica set frontend-57d8c9fb45 to 2
```

Figure 4.23: Monitoring Kubernetes events and filtering to only see ReplicaSet-related events

Note

In the preceding example, we 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 our case, we are sending the output of `kubectl get events` to the `grep` command. `grep` is a command in Linux that is used to filter text. In our case, we are using the `grep` command to only show us 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 an output as shown in *Figure 4.24*:

NAME	DESIRED	CURRENT	READY	AGE
frontend-55b5c47bb	0	0	0	6m19s
frontend-57d8c9fb45	3	3	3	10m
redis-master-545d695785	1	1	1	10m
redis-slave-84548fdbc	2	2	2	10m

Figure 4.24: Two different ReplicaSets

9. 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 an output as shown in *Figure 4.25*:

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

Figure 4.25: Deployment history of the application

10. Since Kubernetes keeps a history of our rollout, this also enables rollback. Let's do a rollback of our 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. We can verify this using the following command:

```
kubectl get replicaset
```

The resultant output is as shown in Figure 4.26:

NAME	DESIRED	CURRENT	READY	AGE
frontend-55b5c47bb	3	3	3	10m
frontend-57d8c9fb45	0	0	0	14m
redis-master-545d695785	1	1	1	14m
redis-slave-84548fdbc	2	2	2	14m

Figure 4.26: The old ReplicaSet now has three Pods, and the new one is scaled down to zero

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

- 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

We can make changes to our application running on top of Kubernetes by using **kubectl edit**. You used this previously in this chapter. 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 our guestbook application without a public load balancer and use **kubectl** to create the load balancer:

- You will start by deploying the guestbook application:

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

- To start the edit, execute the following command:

```
kubectl edit service frontend
```

3. 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 4.27*. 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:

```
spec:
  clusterIP: 10.0.135.85
  ports:
    - port: 80
      protocol: TCP
      targetPort: 80
  selector:
    app: guestbook
    tier: frontend
  sessionAffinity: None
  type: LoadBalancer
status:
  loadBalancer: {}
```

Figure 4.27: Changing this line to type: LoadBalancer

4. 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
```

5. 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** (**command + C** on Mac)

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, such as 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. We'll explore both approaches. First, in this example, we'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 typical YAML files. 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, and then it passes that to the **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:

```
Containers:  
php-redis:  
  Image:      gcr.io/google-samples/gb-frontend:v3  
  Port:       80/TCP  
  Host Port:  0/TCP
```

Figure 4.28: 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 back to v4:

1. 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"}]} }}}'
```

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

```
kubectl describe deployment frontend
```

This will display an output as shown in Figure 4.29:

```
Containers:  
php-redis:  
  Image:      gcr.io/google-samples/gb-frontend:v4  
  Port:       80/TCP  
  Host Port:  0/TCP
```

Figure 4.29: After another patch, we are running the new version again

Before we move on to the next example, let's remove the guestbook application from our 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, we will use Helm to upgrade our application.

Upgrading applications using Helm

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

1. Run the following command:

```
helm install wp stable/wordpress
```

We will force an update of the image of the **WordPress** container. Let's first check the version of the current image:

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

In our case, the image version is **10.3.21-debian-10-r0** as follows:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04$ kubectl describe statefulset wp-mariadb | grep Image
  Image: docker.io/bitnami/mariadb:10.3.21-debian-10-r0
```

Figure 4.30: 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. In our case, we will select the **10.3.22-debian-10-r9** tag to update our StatefulSet.

However, in order to update the MariaDB container image, we need to get the root password for the server and the password for the database. We can get these passwords in the following way:

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

This will generate an output as shown in Figure 4.31:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04$ kubectl get secret wp-mariadb -o yaml
apiVersion: v1
data:
mariadb-password: c2hkUDBpb1M3wA==
mariadb-root-password: eHVvU3p0WW51OQ==
kind: Secret
metadata:
creationTimestamp: "2020-03-08T22:52:56Z"
labels:
  app: mariadb
  chart: mariadb-7.3.12
  heritage: Helm
  release: wp
name: wp-mariadb
namespace: default
resourceVersion: "3182871"
selfLink: /api/v1/namespaces/default/secrets/wp-mariadb
uid: 71fd954f-46e8-49f0-9873-5563641eccb9
type: Opaque
```

Figure 4.31: 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.

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

```
helm upgrade wp stable/wordpress --set mariadb.image.tag=10.3.21-
debian-10-r1,mariadb.rootUser.password=<decoded password>,mariadb.
db.password=<decoded db password> && kubectl get pods -w
```

This will update the image of our MariaDB and make a new Pod start. 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 4.32:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04$ kubectl describe pod wp-mariadb-0 | grep Image
  Image:          docker.io/bitnami/mariadb:10.3.21-debian-10-r1
  Image ID:      docker-pullable://bitnami/mariadb@sha256:18f3fccf6c4b3f7da3414a41a9c29164755e12a420fee9d245dc5e4b19fef4fd
```

Figure 4.32: Showing the new image

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

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

Thus, we have upgraded our 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 so efficient.

Summary

This was a chapter with tons of information. Our goal was to show you how to scale deployments with Kubernetes. We did this by showing you how to create 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, we 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 our deployment based on load.

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

We finished the chapter by looking into different ways to upgrade a deployed application. First, we explored manually updating YAML files. Then, we delved into two additional **kubectl** commands (**edit** and **patch**) that can be used to make changes. Finally, we showed how Helm can be used to perform these upgrades.

In the next chapter, we will look at common failures you can see 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 failure modes 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, we are going to deploy the guestbook application and are going to bring a node down in our cluster and 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:

NAME	STATUS	ROLES	AGE	VERSION
aks-agentpool-39838025-vmss00000	Ready	agent	68s	v1.15.7
aks-agentpool-39838025-vmss00001	Ready	agent	45s	v1.15.7

Figure 5.1: Ensure you have two nodes running in your cluster

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

Name	Provisioning state	Kubernetes version	OS type	Node count	Node size	...
agentpool	Succeeded	1.15.7	Linux	2	Standard_D1_v2	...

Figure 5.2: Scaling the cluster

- As an example application in this chapter, we will work with 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
```

- We will give the service a public IP again, as we did in the previous chapters. To start the edit, execute the following command:

```
kubectl edit service frontend
```

- This will open a `vi` environment. Navigate to the line that now says `type: ClusterIP` (line 27) and change that to `type: LoadBalancer`. To make that change, hit the `I` button to enter insert mode, type your changes, then hit the `Esc` button, type `:wq!`, and hit `Enter` to save the changes.

- 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
```

- This will take a couple of minutes to show you the updated IP. *Figure 5.3* represents the service's public IP. Once you see the right public IP, you can exit the watch command by hitting Ctrl + C (command + C on Mac):

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
frontend	LoadBalancer	10.0.77.213	<pending>	80:31215/TCP	24s
kubernetes	ClusterIP	10.0.0.1	<none>	443/TCP	5m58s
redis-master	ClusterIP	10.0.167.40	<none>	6379/TCP	24s
redis-slave	ClusterIP	10.0.115.105	<none>	6379/TCP	24s
frontend	LoadBalancer	10.0.77.213	52.149.26.220	80:31215/TCP	33s

Figure 5.3: Get the service's public IP

- Go to **http://<EXTERNAL-IP>** as shown in *Figure 5.4*:

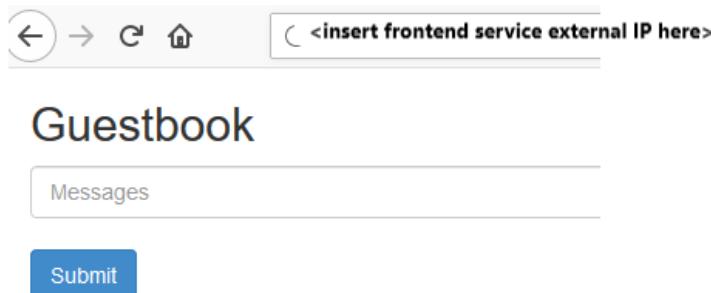


Figure 5.4: Ensure the application is running

- Let's see where the Pods are currently running using the following code:

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

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

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
frontend-57d8c9fb45-mhm5p	1/1	Running	0	2m3s	10.244.1.6	aks-agentpool-39838025-vmss000001
frontend-57d8c9fb45-vmspq	1/1	Running	0	2m3s	10.244.0.10	aks-agentpool-39838025-vmss000000
frontend-57d8c9fb45-zz6k2	1/1	Running	0	2m3s	10.244.1.7	aks-agentpool-39838025-vmss000001
redis-master-545d695785-zlgn5	1/1	Running	0	2m3s	10.244.1.5	aks-agentpool-39838025-vmss000001
redis-slave-84548fdb-44wkz	1/1	Running	0	2m3s	10.244.0.9	aks-agentpool-39838025-vmss000000
redis-slave-84548fdb-b47xn	1/1	Running	0	2m3s	10.244.1.4	aks-agentpool-39838025-vmss000001

Figure 5.5: Our Pods are spread between node 0 and node 1

This shows us that we have the workload spread between node 0 and node 1.

- In this example, we want to demonstrate how Kubernetes handles a node failure. To demonstrate this, we will shut down a node in the cluster. In this case, we are going for maximum damage, so let's shut down node 1 (you can choose whichever node you want – for illustration purposes, it doesn't really matter):

To shut down this node, look for **Virtual machine scale sets** back in our cluster in the Azure search bar, as shown in *Figure 5.6*:

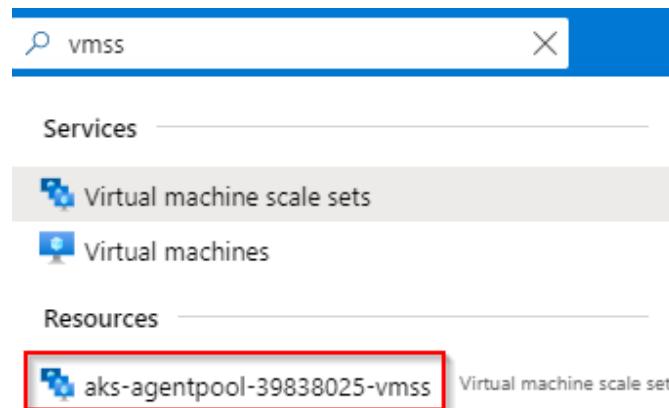


Figure 5.6: Looking for the VMSS hosting your cluster

After navigating to the blade for the scale set, go to the **Instances** view, select the instance you want to shut down, and then hit the **Deallocate** button, as shown in Figure 5.7:

Name	Status
aks-agentpool-39838025-vmss_0	Running
aks-agentpool-39838025-vmss_1	Running

Figure 5.7: Shutdown node 1

This will shut down our 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
```

- To verify whether your application can continue to run, you can optionally 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* (*command + C* on Mac). 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.

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 5.8:

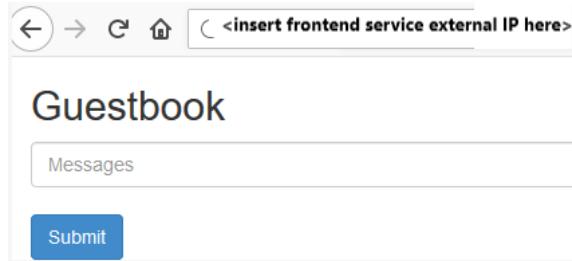


Figure 5.8: Writing a couple of messages in the guestbook

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.

- After a while, the output of watching the Pods should have shown additional output, showing you that the Pods got rescheduled on the healthy host, as shown in Figure 5.9:

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE	NOMINATED NODE	READINESS GATES
frontend-57d8c9fb45-mhm5p	1/1	Running	0	8m8s	10.244.1.6	aks-agentpool-39838025-vms000001	<none>	<none>
frontend-57d8c9fb45-vmspq	1/1	Running	0	8m8s	10.244.0.10	aks-agentpool-39838025-vms000000	<none>	<none>
frontend-57d8c9fb45-zz6k2	1/1	Running	0	8m8s	10.244.1.7	aks-agentpool-39838025-vms000001	<none>	<none>
redis-master-545d695785-zlgn5	1/1	Running	0	8m8s	10.244.1.5	aks-agentpool-39838025-vms000001	<none>	<none>
redis-slave-84548fdbca-4kwz	1/1	Running	0	8m8s	10.244.0.9	aks-agentpool-39838025-vms000000	<none>	<none>
redis-slave-84548fdbca-b47xn	1/1	Running	0	8m8s	10.244.1.4	aks-agentpool-39838025-vms000001	<none>	<none>
frontend-57d8c9fb45-mhm5p	1/1	Running	0	8m51s	10.244.1.6	aks-agentpool-39838025-vms000001	<none>	<none>
frontend-57d8c9fb45-zz6k2	1/1	Running	0	8m51s	10.244.1.7	aks-agentpool-39838025-vms000001	<none>	<none>
redis-master-545d695785-zlgn5	1/1	Running	0	8m52s	10.244.1.5	aks-agentpool-39838025-vms000001	<none>	<none>
redis-slave-84548fdbca-b47xn	1/1	Running	0	8m52s	10.244.1.4	aks-agentpool-39838025-vms000001	<none>	<none>
redis-slave-84548fdbca-b47xn	1/1	Terminating	0	13m	10.244.1.4	aks-agentpool-39838025-vms000001	<none>	<none>
frontend-57d8c9fb45-mhm5p	1/1	Terminating	0	13m	10.244.1.6	aks-agentpool-39838025-vms000001	<none>	<none>
frontend-57d8c9fb45-zz6k2	1/1	Terminating	0	13m	10.244.1.7	aks-agentpool-39838025-vms000001	<none>	<none>
redis-master-545d695785-zlgn5	1/1	Terminating	0	13m	10.244.1.5	aks-agentpool-39838025-vms000001	<none>	<none>
frontend-57d8c9fb45-2src	0/1	Pending	0	0s	<none>	<none>	<none>	<none>
redis-slave-84548fdbca-f6fg6	0/1	Pending	0	0s	<none>	<none>	<none>	<none>
redis-master-545d695785-4kn5m	0/1	Pending	0	0s	<none>	<none>	<none>	<none>
frontend-57d8c9fb45-2src	0/1	Pending	0	0s	<none>	aks-agentpool-39838025-vms000000	<none>	<none>
redis-slave-84548fdbca-f6fg6	0/1	Pending	0	0s	<none>	aks-agentpool-39838025-vms000000	<none>	<none>
redis-master-545d695785-4kn5m	0/1	Pending	0	0s	<none>	aks-agentpool-39838025-vms000000	<none>	<none>
frontend-57d8c9fb45-gbhxj	0/1	Pending	0	0s	<none>	<none>	<none>	<none>
redis-slave-84548fdbca-f6fg6	0/1	Terminating	0	0s	<none>	aks-agentpool-39838025-vms000000	<none>	<none>
redis-slave-84548fdbca-qzccr	0/1	Pending	0	0s	<none>	<none>	<none>	<none>
frontend-57d8c9fb45-gbhxj	0/1	Pending	0	0s	<none>	<none>	<none>	<none>
redis-slave-84548fdbca-qzccr	0/1	Pending	0	0s	<none>	aks-agentpool-39838025-vms000000	<none>	<none>
redis-slave-84548fdbca-qzccr	0/1	Pending	0	0s	<none>	<none>	<none>	<none>
frontend-57d8c9fb45-2src	0/1	ContainerCreating	0	0s	<none>	aks-agentpool-39838025-vms000000	<none>	<none>
redis-slave-84548fdbca-f6fg6	0/1	Terminating	0	0s	<none>	aks-agentpool-39838025-vms000000	<none>	<none>
redis-slave-84548fdbca-f6fg6	0/1	Terminating	0	0s	<none>	aks-agentpool-39838025-vms000000	<none>	<none>
redis-master-545d695785-4kn5m	0/1	ContainerCreating	0	0s	<none>	aks-agentpool-39838025-vms000000	<none>	<none>
frontend-57d8c9fb45-2src	1/1	Running	0	20s	10.244.0.12	aks-agentpool-39838025-vms000000	<none>	<none>

Figure 5.9: The Pods from the failed node getting recreated

What you see here is the following:

- One of the front end Pods running on host 1 got terminated as the host became unhealthy.
- A new front end Pod got created, on host 0. This went through the stages **Pending**, **ContainerCreating**, and then **Running**.

Note

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 1 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.

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.

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

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

This section explains that every Pod for the **redis-slave** deployment requires **100m** of a CPU core (100 milli or 10%) and 100MiB (Mebibyte) of memory. In our 1 CPU cluster (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-slave** deployment to 10 Pods:

```
kubectl scale deployment/redis-slave --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 follows:

NAME	READY	STATUS	RESTARTS	AGE
frontend-57d8c9fb45-2sqrc	1/1	Running	0	21m
frontend-57d8c9fb45-gbhxj	1/1	Running	0	21m
frontend-57d8c9fb45-mhm5p	1/1	Terminating	0	35m
frontend-57d8c9fb45-vmspq	1/1	Running	0	35m
frontend-57d8c9fb45-zz6k2	1/1	Terminating	0	35m
redis-master-545d695785-4kn5m	1/1	Running	0	21m
redis-master-545d695785-zlgn5	1/1	Terminating	0	35m
redis-slave-84548fdb-44wkz	1/1	Running	0	35m
redis-slave-84548fdb-5vkff	0/1	Pending	0	23s
redis-slave-84548fdb-64t78	0/1	Pending	0	23s
redis-slave-84548fdb-9grdp	0/1	Pending	0	23s
redis-slave-84548fdb-b47xn	1/1	Terminating	0	35m
redis-slave-84548fdb-bk266	0/1	Pending	0	23s
redis-slave-84548fdb-h98jl	0/1	Pending	0	23s
redis-slave-84548fdb-16dhf	0/1	Pending	0	23s
redis-slave-84548fdb-ptdsd	0/1	Pending	0	23s
redis-slave-84548fdb-qtb8r	0/1	Pending	0	23s
redis-slave-84548fdb-qzccr	0/1	Pending	0	21m

Figure 5.10: If the cluster is out of resources, Pods will go into the Pending state

Highlighted here is one of the Pods that are in the **Pending** state.

- We can get more information about these pending Pods using the following command:

```
kubectl describe pod redis-slave-<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	49s (x4 over 4m49s)	default-scheduler	0/2 nodes are available: 1 Insufficient cpu, 1 node(s) had taints that the pod didn't tolerate.

Figure 5.11: Kubernetes is unable to schedule this Pod

It explains two things to us:

- One of the nodes is out of CPU resources.
 - One of the nodes has a taint that the Pod didn't tolerate. This means that the node that is **NotReady** can't accept Pods.
- We can solve this capacity issue, by starting up node 1 as shown in *Figure 5.12*. This can be done in a way similar to the shutdown process:

The screenshot shows the Azure portal interface for managing a Virtual Machine Scale Set (VMSS). The main title is "aks-agentpool-39838025-vmss - Instances". On the left, there's a navigation menu with options like Overview, Activity log, Access control (IAM), Tags, and Diagnose and solve problems. The "Instances" tab is selected, indicated by a red circle with the number 1. In the center, there's a table listing virtual machine instances. The first instance, "aks-agentpool-39838025-vmss_0", is in a "Starting" state. The second instance, "aks-agentpool-39838025-vmss_1", is in a "Stopped (deallocated)" state and has a checked checkbox next to it, indicated by a red circle with the number 2. At the top right, there are buttons for Start, Restart, Deallocate, Reimage, and Delete, with a red circle labeled 3 above the Start button.

Figure 5.12: Start node 1 again

- It will take a couple of minutes for the other node to become available again in Kubernetes. If we re-execute the **describe** command on the previous Pod, we'll see an output like what's shown in Figure 5.13:

Events:				
Type	Reason	Age	From	Message
Warning	FailedScheduling	72s (x13 over 15m)	default-scheduler	0/2 nodes are available: 1 Insufficient cpu, 1 node(s) had taints that the pod didn't tolerate.
Normal	Scheduled	69s	default-scheduler	Successfully assigned default/redis-slave-84548fdb-c5tn9 to aks-agentpool-42828616-vms0000001
Normal	Pulled	64s	kubelet, aks-agentpool-42828616-vms0000001	Container image "gcr.io/google_samples/gb-redisslave:v1" already present on machine
Normal	Created	63s	kubelet, aks-agentpool-42828616-vms0000001	Created container slave
Normal	Started	62s	kubelet, aks-agentpool-42828616-vms0000001	Started container slave

Figure 5.13: When the node is available again, the other Pods get started on the new node

This shows that after node 1 became available, Kubernetes scheduled our Pod on that node, and then started the container.

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

Note

In *Chapter 4, Scaling your Application*, we discussed the cluster autoscaler. The cluster autoscaler will monitor out-of-resource errors and add new nodes to the cluster automatically.

Let's clean up by running the following **delete** command:

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

So far, we have discussed two failure modes for nodes in a Kubernetes cluster. First, we discussed how Kubernetes handles a node going offline and how the system reschedules Pods to a working node. After that, we saw how Kubernetes uses requests to schedule Pods on a node, and what happens when a cluster is out of resources. In the next section, we'll cover another failure mode in Kubernetes, namely what happens when Kubernetes moves Pods with PVCs attached.

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, we'll cover an example of how PVCs can be used to prevent data loss when Kubernetes moves a Pod to another node. We will show you a common error that occurs when Kubernetes moves Pods with PVCs attached, and we will show you how to fix this.

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

```
kubectl get all
```

This shows us just the one Kubernetes service, as shown in *Figure 5.14*:

user@Azure:~\$ kubectl get all					
NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
service/kubernetes	ClusterIP	10.0.0.1	<none>	443/TCP	5h28m

Figure 5.14: You should only have the 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 shown in *Figure 5.15*:

user@Azure:~\$ kubectl get nodes					
NAME	STATUS	ROLES	AGE	VERSION	
aks-agentpool-42828616-vmss000000	Ready	agent	6d2h	v1.15.7	
aks-agentpool-42828616-vmss000001	Ready	agent	6d2h	v1.15.7	

Figure 5.15: You should have two nodes available in your cluster

In the previous example, under the *Handling node failures* section, we saw that the messages stored in **redis-master** were 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, we'll 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:

Begin reinstallation by using the following command:

```
helm install wp stable/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 us Pods with a status of **Running** and with a ready status of **1/1** for both Pods, as shown in *Figure 5.16*:

NAME	READY	STATUS	RESTARTS	AGE
wp-mariadb-0	0/1	ContainerCreating	0	26s
wp-wordpress-6c8b7bcfdf-bxwxg	0/1	ContainerCreating	0	27s
wp-mariadb-0	0/1	Running	0	39s
wp-wordpress-6c8b7bcfdf-bxwxg	0/1	Running	0	56s
wp-mariadb-0	1/1	Running	0	72s
wp-wordpress-6c8b7bcfdf-bxwxg	1/1	Running	0	2m48s

Figure 5.16: All Pods will have the status of Running after a couple of minutes

In this section, we saw how to install the WordPress. Now, we will see how to avoid data loss using the 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. We explained PVs in more detail in *Chapter 3, Application deployment on AKS*. Let's explore the PVs created for our WordPress deployment:

1. In our case, run the following **describe nodes** command:

```
kubectl describe nodes
```

Scroll through the output until you see a section that is similar to *Figure 5.17*. In our case, both WordPress Pods are running on node 0:

Namespace	Name	CPU Requests	CPU Limits	Memory Requests	Memory Limits	AGE
default	wp-mariadb-0	0 (0%)	0 (0%)	0 (0%)	0 (0%)	4m7s
default	wp-wordpress-6c8b7bcfdf-bxwxg	300m (31%)	0 (0%)	512Mi (28%)	0 (0%)	4m8s
kube-system	kube-proxy-5566m	100m (10%)	0 (0%)	0 (0%)	0 (0%)	2d23h
kube-system	omsagent-9xw4h	75m (7%)	150m (15%)	225Mi (12%)	600Mi (33%)	2d23h
kube-system	omsagent-rs-76b774dc7f-z5cf2	150m (15%)	1 (106%)	250Mi (14%)	750Mi (42%)	4d2h

Figure 5.17: In our case, both WordPress Pods are running on node 0

Your Pod placement might vary.

2. The next thing we can check is the status of our PVCs:

```
kubectl get pvc
```

This will generate an output as shown in *Figure 5.18*:

NAME	STATUS	VOLUME	CAPACITY	ACCESS MODES
data-wp-mariadb-0	Bound	pvc-92724d07-7189-4818-9924-7ac52fc71968	8Gi	RWO
wp-wordpress	Bound	pvc-64fdc682-ecad-4b81-8726-d9b507ab07e6	10Gi	RWO

Figure 5.18: Two PVCs are created by the WordPress deployment

The following command shows the actual PV that is bound to the Pods:

```
kubectl describe pv
```

This will show you the details of both volumes. We will show you one of those in *Figure 5.19*:

```
Name: pvc-92724d07-7189-4818-9924-7ac52fc71968
Labels: <none>
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/data-wp-mariadb-0
Reclaim Policy: Delete
Access Modes: RWO
VolumeMode: Filesystem
Capacity: 8Gi
Node Affinity: <none>
Message:
Source:
  Type: AzureDisk (an Azure Data Disk mount on the host and bind mount to the pod)
  DiskName: kubernetes-dynamic-pvc-92724d07-7189-4818-9924-7ac52fc71968
  DiskURI: /subscriptions/a4339399-5b72-463b-88f9-e67030102086/resourceGroups/mc_handsonaks_handsonaks_westus2/providers/Microsoft.Compute/disks/kubernetes-dynamic-pvc-92724d07-7189-4818-9924-7ac52fc71968
    Kind: Managed
    FSType:
    CachingMode: ReadOnly
    ReadOnly: false
Events: <none>
```

Figure 5.19: The details of one of the PVCs

Here, we can see which Pod has claimed this volume and what the **DiskURI** is in Azure.

3. Verify that your site is actually working:

```
kubectl get service
```

This will show us the public IP of our WordPress site, as shown in *Figure 5.20*:

```
user@Azure:~$ kubectl get service
NAME         TYPE      CLUSTER-IP   EXTERNAL-IP
kubernetes   ClusterIP 10.0.0.1    <none>
wp-mariadb   ClusterIP 10.0.230.126 <none>
wp-wordpress LoadBalancer 10.0.164.136 52.137.102.242
```

Figure 5.20: Get the service's public IP

4. If you remember from *Chapter 3, Application deployment of AKS*, Helm showed us the commands we 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 5.21*:

```
user@Azure:~$ helm status wp
NAME: wp
LAST DEPLOYED: Fri Feb 7 22:00:17 2020
NAMESPACE: default
STATUS: deployed
REVISION: 1
NOTES:
1. Get the WordPress URL:

NOTE: It may take a few minutes for the LoadBalancer IP to be available.
      Watch the status with: 'kubectl get svc --namespace default -w wp-wordpress'
      export SERVICE_IP=$(kubectl get svc --namespace default wp-wordpress --template="{{ range (ind
ex .status.loadBalancer.ingress 0) }}{{.}}{{ end }}")
      echo "WordPress URL: http://$SERVICE_IP/"
      echo "WordPress Admin URL: http://$SERVICE_IP/admin"

2. Login with the following credentials to see your blog

echo Username: user
echo Password: $(kubectl get secret --namespace default wp-wordpress -o jsonpath=".data.wordpress-password" | base64 --decode)
user@Azure:~$ echo Username: user
Username: user
user@Azure:~$ echo Password: $(kubectl get secret --namespace default wp-wordpress -o jsonpath=".data.wordpress-password" | base64 -d)
Password: lcsUSJTk8e
```

Figure 5.21: Getting the username and password for the WordPress application

Note

You might notice that the commands we use in the book are slightly different than the ones that **helm** returned. The command that **helm** returned for the password didn't work for us, and we provided you with the working commands.

We 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 5.22:

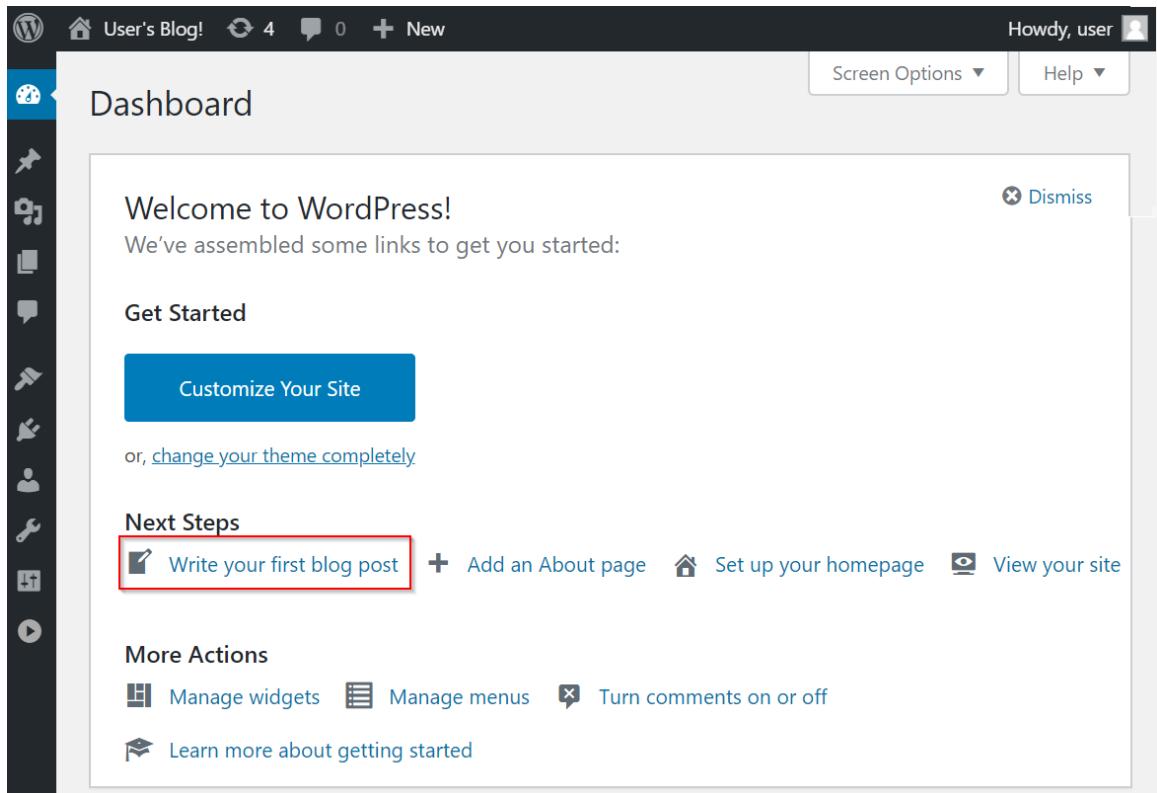


Figure 5.22: Writing your first blog post

Type a little text now and hit the **Publish** button, as shown in Figure 5.23. The text itself isn't important; we are writing this to verify that data is indeed persisted to disk:

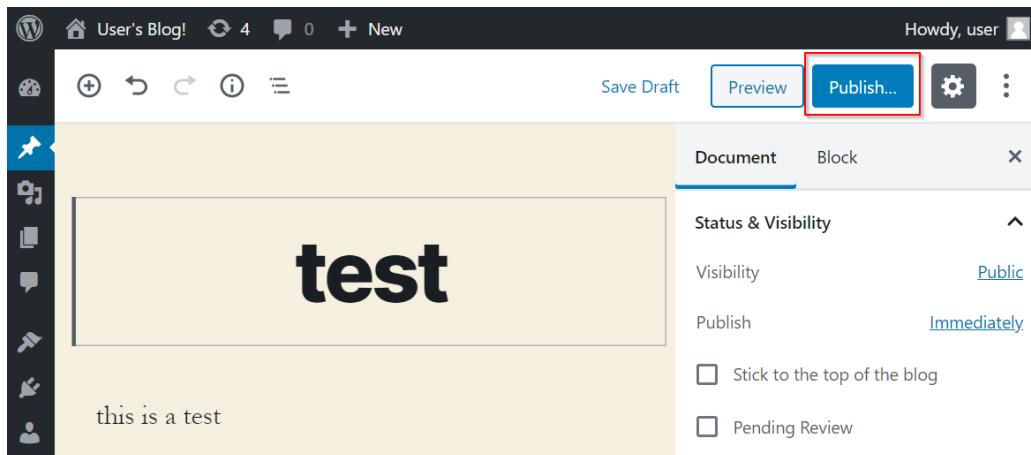


Figure 5.23: Publishing a post with random text

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 5.23. We will verify whether this post survives a reboot in the next section.

Handling Pod failure with PVC involvement

The first test we'll do with our PVC is to kill the Pods and verify whether the data has indeed persisted. To do this, let's do two things:

1. **Watch our Pods in our application.** To do this, we'll use our 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, we'll create a new Cloud Shell window by clicking on the icon shown in Figure 5.24:



Figure 24: Opening a new Cloud Shell instance

Once you open a new Cloud Shell, execute the following command:

```
kubectl delete pod --all
```

If you follow along with the `watch` command, you should see an output like what's shown in *Figure 5.25*:

NAME	READY	STATUS	RESTARTS	AGE
wp-mariadb-0	1/1	Running	0	23m
wp-wordpress-6c8b7bcfdf-bxwxg	1/1	Running	0	23m
wp-mariadb-0	1/1	Terminating	0	25m
wp-wordpress-6c8b7bcfdf-bxwxg	1/1	Terminating	0	25m
wp-wordpress-6c8b7bcfdf-6w7kp	0/1	Pending	0	0s
wp-wordpress-6c8b7bcfdf-6w7kp	0/1	Pending	0	0s
wp-wordpress-6c8b7bcfdf-6w7kp	0/1	ContainerCreating	0	0s
wp-mariadb-0	0/1	Terminating	0	25m
wp-wordpress-6c8b7bcfdf-bxwxg	0/1	Terminating	0	25m
wp-mariadb-0	0/1	Terminating	0	25m
wp-mariadb-0	0/1	Terminating	0	25m
wp-mariadb-0	0/1	Pending	0	0s
wp-mariadb-0	0/1	Pending	0	0s
wp-wordpress-6c8b7bcfdf-bxwxg	0/1	Terminating	0	25m
wp-wordpress-6c8b7bcfdf-bxwxg	0/1	Terminating	0	25m
wp-mariadb-0	0/1	ContainerCreating	0	1s
wp-mariadb-0	0/1	Running	0	21s
wp-mariadb-0	1/1	Running	0	57s
wp-wordpress-6c8b7bcfdf-6w7kp	0/1	Running	0	113s
wp-wordpress-6c8b7bcfdf-6w7kp	1/1	Running	0	3m10s

Figure 5.25: After deleting the Pods, Kubernetes will automatically recreate both Pods

As you can see, 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 container itself.

Figure 5.26 shows that the blog post was persisted even though the Pod was recreated:

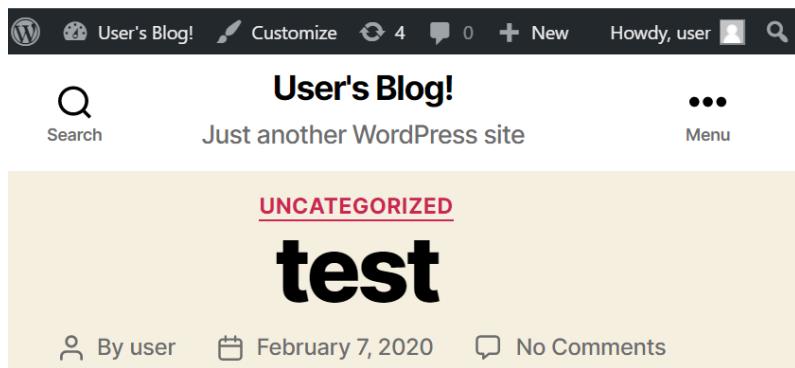


Figure 5.26: Your data is persisted, and your blog post is still there

One final interesting data point to watch here is the Kubernetes event stream. If you run the following command, you can see events related to volumes:

```
kubectl get events | grep -i volume
```

This will generate an output as shown in Figure 5.27:

```
user@Azure:~$ kubectl get events | grep -i volume
33m      Normal   ProvisioningSucceeded   persistentvolumeclaim/data-wp-mariadb-0
33m      Warning  FailedScheduling        pod/wp-mariadb-0
33m      Normal   SuccessfulAttachVolume  pod/wp-mariadb-0
8m30s    Warning  FailedAttachVolume    pod/wp-wordpress-6c8b7bcfdf-6w7kp
7m46s    Normal   SuccessfulAttachVolume  pod/wp-wordpress-6c8b7bcfdf-6w7kp
33m      Warning  FailedScheduling        pod/wp-wordpress-6c8b7bcfdf-bxwxg
32m      Normal   SuccessfulAttachVolume  pod/wp-wordpress-6c8b7bcfdf-bxwxg
33m      Normal   ProvisioningSucceeded   persistentvolumeclaim/wp-wordpress
```

Figure 5.27: Kubernetes dealt with a FailedAttachVolume error

This shows you the events related to volumes. There are two interesting messages to explain in more detail: **FailedAttachVolume** and **SuccessfulAttachVolume**.

This shows us how Kubernetes handles volumes that have a read-write-once configuration. Since the characteristic is to only read and write from a single Pod, Kubernetes will only mount the volume to the new Pod when it is successfully unmounted from the current Pod. Hence, initially, when the new Pod was scheduled, it showed the **FailedAttachVolume** message as the volume was still attached to the Pod that was being deleted. Afterward, the Pod could successfully mount the volume and showed this with the message **SuccessfulAttachVolume**.

In this section, we've learned how PVCs can help when Pods get recreated on the same node. In the next section, we'll see how PVCs are used when a node has a failure.

Handling node failure with PVC involvement

In the previous example, we saw how Kubernetes can handle Pod failures when those Pods have a PV attached. In this example, we'll look at how Kubernetes handles node failures when a volume is attached:

1. Let's first check which node is hosting our application, using the following command:

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

We found that, in our cluster, node 1 was hosting MariaDB, and node 0 was hosting the WordPress site, as shown in *Figure 5.28*:

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
wp-mariadb-0	1/1	Running	0	13m	10.244.1.93	aks-agentpool-42828616-vmss000001
wp-wordpress-6c8b7bcfdf-6w7kp	1/1	Running	0	13m	10.244.0.93	aks-agentpool-42828616-vmss000000

Figure 5.28: We have two Pods running in our deployment – one on node 1, one on node 0

2. We are going to introduce a failure and stop the node that could cause the most damage by shutting down node 0 on the Azure portal. We'll do this the same way as in the earlier example. First, look for the scale set backing our cluster, as shown in *Figure 5.29*:

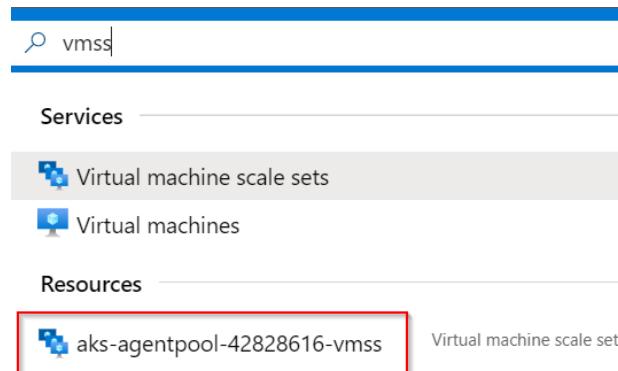


Figure 5.29: Finding the scale set backing our cluster

3. Then shut down the node as shown in Figure 5.30:

Name	Status
<input checked="" type="checkbox"/> aks-agentpool-42828616-vmss_0	Running
<input type="checkbox"/> aks-agentpool-42828616-vmss_1	Running

Figure 5.30: Shutting down the node

4. After this action, we'll once again watch our 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 our failed node. We can see that happening in Figure 5.31:

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE	NOMINATED NODE	READINESS	GATES
wp-mariadb-0	1/1	Running	0	23m	10.244.1.93	aks-agentpool-42828616-vmss000001	<none>	<none>	
wp-wordpress-6c8b7bcfdf-6w7kp	1/1	Running	0	23m	10.244.0.93	aks-agentpool-42828616-vmss000000	<none>	<none>	
wp-wordpress-6c8b7bcfdf-6w7kp	1/1	Terminating	0	25m	10.244.0.93	aks-agentpool-42828616-vmss000000	<none>	<none>	
wp-wordpress-6c8b7bcfdf-84vjq	0/1	Pending	0	0s	<none>	<none>	<none>	<none>	
wp-wordpress-6c8b7bcfdf-84vjq	0/1	Pending	0	0s	<none>	aks-agentpool-42828616-vmss000001	<none>	<none>	
wp-wordpress-6c8b7bcfdf-84vjq	0/1	Terminating	0	0s	<none>	aks-agentpool-42828616-vmss000001	<none>	<none>	
wp-wordpress-6c8b7bcfdf-k78t1	0/1	Pending	0	0s	<none>	<none>	<none>	<none>	
wp-wordpress-6c8b7bcfdf-k78t1	0/1	Pending	0	0s	<none>	<none>	<none>	<none>	
wp-wordpress-6c8b7bcfdf-84vjq	0/1	Terminating	0	1s	<none>	aks-agentpool-42828616-vmss000001	<none>	<none>	
wp-wordpress-6c8b7bcfdf-84vjq	0/1	Terminating	0	1s	<none>	aks-agentpool-42828616-vmss000001	<none>	<none>	
wp-wordpress-6c8b7bcfdf-84vjq	0/1	Terminating	0	2s	<none>	aks-agentpool-42828616-vmss000001	<none>	<none>	
wp-wordpress-6c8b7bcfdf-84vjq	0/1	Terminating	0	2s	<none>	aks-agentpool-42828616-vmss000001	<none>	<none>	

Figure 5.31: A Pod in a pending state

5. We are seeing a new issue here. Our new Pod is stuck in a **Pending** state and has not been assigned to a new node. Let's figure out what is happening here. First, we'll **describe** our 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
Warning	FailedScheduling	45s (x5 over 116s)	default-scheduler	0/2 nodes are available: 2 Insufficient cpu.

Figure 5.32: Output displaying the Pod in a pending state

- This shows us that we don't have sufficient CPU resources in our cluster to host the new Pod. We can fix this by using the `kubectl edit deploy/...` command to fix any insufficient CPU/memory errors. We'll change the CPU requests from 300 to 3 for our example to continue:

```
kubectl edit deploy wp-wordpress
```

This brings us to a `vi` environment. We can quickly find the section referring to the CPU by typing the following:

```
/cpu <enter>
```

Once there, move the cursor over the two zeros and hit the `x` key twice to delete the zeros. Finally, type `:wq!` to save our changes so that we can continue our example.

- This will result in a new ReplicaSet being created with a new Pod. We can get the name of the new Pod by entering the following command:

```
kubectl get pods
```

Look for the Pod with the status `ContainerCreating`, as follows:

NAME	READY	STATUS	RESTARTS	AGE
wp-mariadb-0	1/1	Running	0	4h1m
wp-wordpress-544cd695b9-924fj	0/1	ContainerCreating	0	3h32m
wp-wordpress-6c8b7bcfdf-6w7kp	1/1	Terminating	0	4h1m
wp-wordpress-6c8b7bcfdf-k78t1	0/1	Pending	0	3h36m

Figure 5.33: The new Pod is stuck in a ContainerCreating state

- Let's have a look at the details for that Pod with the **describe** command:

```
kubectl describe pod wp-wordpress-<pod-id>
```

In the **Events** section of this **describe** output, you can see an error message as follows:

Events:			
Type	Reason	Age	From
Warning	FailedMount	48s (x94 over 3h31m)	kublet, aks-agentpool-42828616-vmss000001
			Unable to mount volumes for pod "wp-wordpress-544cd695b9-924fj_default(95b374aa-d9a3-4169-a238-3dba0e4b6a02)": timeout expired waiting for volumes to attach or mount for pod "default"/"wp-wordpress-544cd695b9-924fj". list of unmounted volumes=[wordpress-data]. list of unattached volumes=[wordpress-data default-token-qf5xz]

Figure 5.34: The new Pod has a new error message, describing volume mounting issues

- This tells us that the volume that our new Pod wants to mount is still mounted to the Pod that is stuck in a **Terminating** state. We can solve this by manually detaching the disk from the node we shut down and forcefully removing the 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/>.

- To do that, we'll need the scale set's name and the resource group name. To find those, look for the scale set in the portal as shown in Figure 5.35:

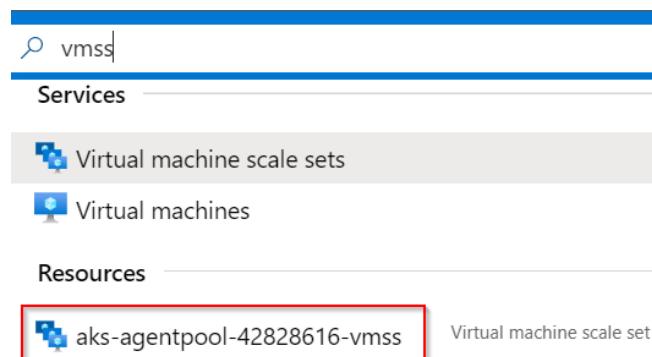


Figure 5.35: Look for the scale set backing your cluster

11. In the scale set view, copy and paste the scale set name and the resource group. Edit the following command to detach the disk from your failed node and then run this command in Cloud Shell:

```
az vmss disk detach --lun 0 --vmss-name <vmss-name> -g <rgname>
--instance-id 0
```

12. This will detach the disk from node 0. The second step required here is the forceful removal of the Pod from the cluster, while it is stuck in the terminating state:

```
kubectl delete pod wordpress-wp-<pod-id> --grace-period=0 --force
```

13. This will bring our new Pod to a healthy state. It will take a couple of minutes for the system to pick up the changes and then mount and schedule 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
Warning	FailedMount	6m22s (x110 over 4h13m)	kubelet, aks-agentpool-42828616-vmss000001	Unable to mount volumes for pod "wp-wordpress-544cd695b9-924fj_default(95b374aa-d9a3-4169-a238-3dba0e4b6a02)": timeout expired waiting for volumes to attach or mount for pod "default"/"wp-wordpress-544cd695b9-924fj". list of unmounted volumes=[wordpress-data]. list of unattached volumes=[wordpress-data default-token-qf5xz]
Normal	SuccessfulAttachVolume	5m13s	attachdetach-controller	AttachVolume.Attach succeeded for volume "pvc-64fdc682-ecad-4b81-8726-d9b507ab07e6"
Normal	Pulled	3m9s	kubelet, aks-agentpool-42828616-vmss000001	Container image "docker.io/bitnami/wordpress:5.3.2-debian-10-r0" already present on machine

Figure 5.36: Our new Pod is now attaching the volume and pulling the container image

14. This shows us that the new Pod successfully got the volume attached and that the container image got pulled. Let's verify that the Pod is actually running:

```
kubectl get pods
```

This will show the Pods running as shown in Figure 5.37:

user@Azure:~\$ kubectl get pods				
NAME	READY	STATUS	RESTARTS	AGE
wp-mariadb-0	1/1	Running	0	4h46m
wp-wordpress-544cd695b9-924fj	1/1	Running	0	4h17m

Figure 5.37: Both Pods are successfully running

And this should make your WordPress website available again.

Before continuing, let's clean up our deployment using the following command:

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

Let's also restart the node that was shut down, as shown in *Figure 5.38*:

Name	Status
<input checked="" type="checkbox"/> aks-agentpool-39838025-vmss_0	<input type="radio"/> Stopped (deallocated)
<input type="checkbox"/> aks-agentpool-39838025-vmss_1	<input checked="" type="radio"/> Running

Figure 5.38: Starting node 1 again

In this section, we covered how you can recover from a node failure when PVCs aren't mounted to new Pods. We needed to manually unmount the disk and then forcefully delete the Pod that was stuck in a **Terminating** state.

Summary

In this chapter, you learned about common Kubernetes failure modes and how you can recover from these. We started this chapter with an example on 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 shut down 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 unmounting the disk from the node and 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 and Azure AD

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 then renew them periodically. While that is still possible today, the **Let's Encrypt** service and helpers in Kubernetes make it very easy to set 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, we will introduce a new object called an **Ingress**, and we will use a Kubernetes add-on called **cert-manager**. An Ingress is an object within Kubernetes that manages external access to services. Ingresses are 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. An Ingress can be configured to handle HTTPS traffic. It can also be configured to route traffic to different back-end services based on the host name, 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, we will see how to set up the Ingress and **cert-manager** to interface with Let's Encrypt.

Also in this chapter, we will explore different approaches to authentication for the guestbook app. We will look at the **oauth2_proxy** reverse proxy to add authentication to the sample guest app using Azure **Active Directory (AD)**. The **oauth2_proxy** is a reverse proxy that will forward authentication requests to a configuration authentication platform. You will learn how to easily secure apps that have no built-in authentication. The authentication scheme can be extended to use GitHub, Google, GitLab, LinkedIn, or Facebook.

The following topics will be covered in this chapter:

- Setting up an Ingress in front of a service
- Adding TLS support to an Ingress
- Authentication and common authentication providers
- Authentication versus authorization
- Deploying the **oauth2_proxy** sidecar

Let's start with setting up the Ingress.

HTTPS support

Obtaining TLS certificates traditionally has been an expensive and manual business. If you wanted to do it cheaply, you could self-sign your certificates, but browsers would complain when opening up your site and identify it as not trusted. The Let's Encrypt service changes all that. Let's Encrypt is a free, automated, and open CA, run for the public's benefit. It gives people the digital certificates they need in order to enable HTTPS (SSL/TLS) for websites, for free, in the most user-friendly way.

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.

Installing an Ingress controller

With the Ingress object, Kubernetes provides a clean way of securely exposing your services. It provides an SSL endpoint and name-based routing, meaning different DNS names can be routed to different back-end services.

If you want to create an Ingress object in your cluster, you first need to set up an **Ingress controller**. The Ingress controller will manage the state of the Ingresses you have deployed in your cluster. There are multiple options when selecting an Ingress controller. For a full list of options, please refer to <https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/>. When running AKS, the two most common options are either using an NGINX-based Ingress controller, or an Ingress controller based on the Azure application gateway. For our example, we'll use the NGINX version.

Let's go ahead and install the NGINX version of the Ingress controller by performing the following steps:

1. To follow along, run this sample in the Bash version of Cloud Shell.
2. Type in the following command to begin installation:

```
helm repo add stable https://kubernetes-charts.storage.googleapis.com/  
helm install ingress stable/nginx-ingress
```

This will set up the Ingress controller for our cluster. This will additionally create a public IP that we will use to access the Ingress controller.

3. Let's connect to the Ingress controller. To get the exposed IP of the **ingress-controller** service, enter this command:

```
kubectl get service
```

You should see an entry for the Ingress controller as shown in *Figure 6.1*:

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP
ingress-nginx-ingress-controller	LoadBalancer	10.0.239.116	40.91.124.21
ingress-nginx-ingress-default-backend	ClusterIP	10.0.163.116	<none>
kubernetes	ClusterIP	10.0.0.1	<none>

Figure 6.1: Getting the IP of the Ingress controller

You can browse to the web page by entering `http://<EXTERNAL-IP>` in the browser:



Figure 6.2: An Ingress showing its default back end

This shows you two things:

4. There is no back-end application, just a default application.
5. The website is served over HTTP, not HTTPS (hence the **Not secure** warning).

In the next two sections, we'll solve both of these issues. We'll first create an Ingress rule for our guestbook application, and then we'll add HTTPS support via Let's Encrypt.

Adding an Ingress rule for the guestbook application

Let's start by relaunching our guestbook application. To launch the guestbook application, type in the following command:

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

This will create our trusted guestbook application. You should see the objects being created as shown in *Figure 6.3*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter05$ kubectl create -f guestbook-all-in-one.yaml
service/redis-master created
deployment.apps/redis-master created
service/redis-slave created
deployment.apps/redis-slave created
service/frontend created
deployment.apps/frontend created
```

Figure 6.3: Creating the guestbook application

We 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: extensions/v1beta1
2  kind: Ingress
3  metadata:
4    name: simple-frontend-ingress
5  spec:
6    rules:
7      - http:
8        paths:
9          - path: /
10         backend:
11           serviceName: frontend
12           servicePort: 80
```

Let's have a look at what we define in this YAML file:

- **Line 2:** Here, we define the fact that we are creating an Ingress object.
- **Lines 5-12:** These lines define the configuration of the Ingress. Pay specific attention to:
 - **Line 9:** Here, we 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 10-12:** These lines define the actual service this traffic should be pointed to.

We can use the following command to create this Ingress:

```
kubectl apply -f simple-frontend-ingress.yaml
```

If you now go to **https://<EXTERNAL-IP>/**, you should get an output as shown in Figure 6.4:

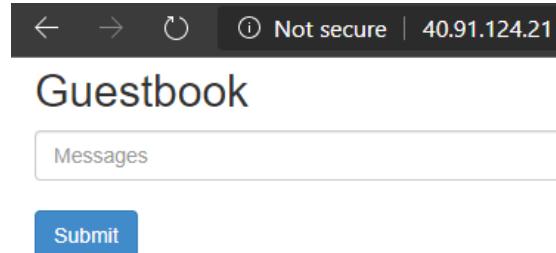


Figure 6.4: Accessing the guestbook application via the Ingress

Please pay attention to the following: we didn't have to publicly expose the front-end service as we have done in the preceding chapters. We have added the Ingress as the exposed service, and the front-end service remains private to the cluster:

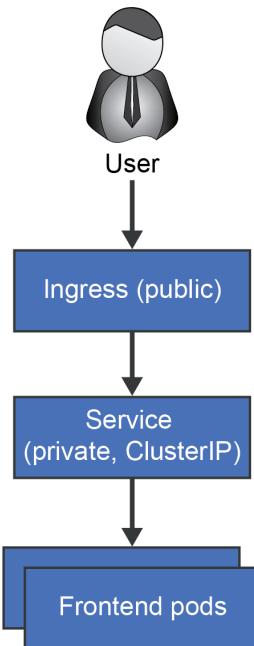


Figure 6.5: Flowchart displaying publicly accessible Ingress

You can verify this by running the following command:

```
kubectl get svc
```

This should only show you one public service:

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP
frontend	ClusterIP	10.0.192.144	<none>
ingress-nginx-ingress-controller	LoadBalancer	10.0.239.116	40.91.124.21
ingress-nginx-ingress-default-backend	ClusterIP	10.0.163.116	<none>
kubernetes	ClusterIP	10.0.0.1	<none>
redis-master	ClusterIP	10.0.18.196	<none>
redis-slave	ClusterIP	10.0.123.247	<none>

Figure 6.6: Output displaying only the Ingress having a public IP

In this section, you have launched an instance of the guestbook application. You exposed it publicly by creating an Ingress. Only the Ingress was publicly accessible. In the next section, we'll see how to get a certification from Let's Encrypt.

Getting a certificate from Let's Encrypt

In this section, we will add HTTPS support to our application. To do this, we need a TLS certificate. We will be using the **cert-manager** Kubernetes add-on to request a certificate from Let's Encrypt. There are a couple of steps involved. The process of adding HTTPS to our application involves the following steps:

1. Install **cert-manager**, which interfaces with the Let's Encrypt API to request a certificate for the domain name you specify.
2. Map the **Azure Fully Qualified Domain Name (FQDN)** to the NGINX Ingress public IP. An FQDN is the complete DNS name of a service, sometimes referred to as the DNS record, such as **www.google.com**. A TLS certificate is issued for an FQDN, which is why we need to map an FQDN for our Ingress.
3. Install the certificate issuer, which will get the certificate from Let's Encrypt.
4. Create an SSL certificate for a given FQDN.
5. Secure the front-end service section by creating an Ingress to the service with the certificate created in step 4. In our example, we will not be executing this step. We will, however, reconfigure our Ingress to automatically pick up the certificate created in step 4.

Let's start with the first step; installing **cert-manager** in our cluster.

Installing cert-manager

The first step in getting a TLS certificate is to install **cert-manager** in your cluster. **cert-manager** (<https://github.com/jetstack/cert-manager>) automates the management and issuance of TLS certificates from various issuing sources. It is an open-source solution managed by the company **Jetstack**. Renewing certificates and ensuring that they are updated periodically is all managed by **cert-manager**, which is a Kubernetes add-on.

The following commands install **cert-manager** in your cluster:

```
kubectl create ns cert-manager  
helm repo add jetstack https://charts.jetstack.io  
helm install cert-manager --namespace cert-manager jetstack/cert-manager
```

These commands do a couple of things in your cluster:

1. Create a new **namespace**. Namespaces are used within Kubernetes to isolate workloads from each other.
2. Add a new repository to Helm to grab charts from.
3. Install the **cert-manager** Helm chart.

Now that you have installed **cert-manager**, we can move on to the next step: mapping an FQDN to the Ingress.

Mapping the Azure FQDN to the NGINX Ingress public IP

The next step in the process of getting a TLS certificate is adding an FQDN to your IP address. Let's Encrypt requires a publicly available DNS entry to verify ownership of the DNS entry *before* it issues a certificate. This ensures that you cannot hijack someone else's site. We have to map the public domain name given to us by Azure to the external IP we get from Azure Load Balancer in order to prove ownership.

The following steps will help us link a DNS entry to our public IP:

1. Let's go ahead and link a DNS name to our public IP address. First, make sure you get your IP address from your Ingress service:

```
kubectl get service
```

Note down the IP of the Ingress service. In the Azure Search bar, now look for **public ip**:

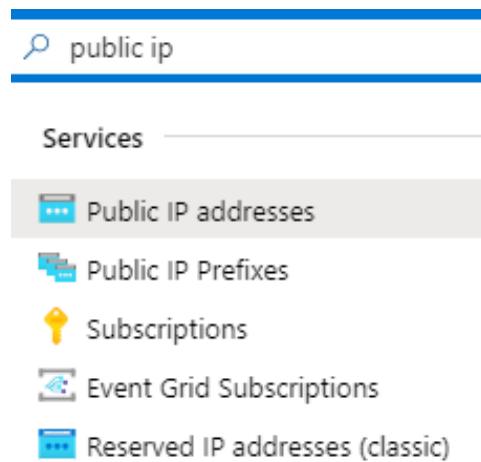


Figure 6.7: Searching for public ip in the Azure search bar

- Once there, you should see a number of public IP addresses. To find our public IP address, you can show an extra column here that would show you the actual IP address. Hit the **Edit columns** button to add the extra column:



Figure 6.8: Clicking on Edit columns to add an extra column

3. In the column selector, pick **IP address** and hit the arrow pointing right, as shown in *Figure 6.9*:

The screenshot shows the 'Edit columns' interface for 'Public IP addresses'. It consists of two main panes: 'Available columns' on the left and 'Selected columns' on the right.

Available columns pane:

- Header: Available columns
- Text: Select columns below to display in your grid.
- Text: Learn how to [create your own columns with tags](#).
- Filter input field
- Properties dropdown
- List of columns:
 - Assignment
 - Associated to
 - DNS name
 - Idle timeout
 - IP address** (highlighted with a red circle labeled '1')
 - Kind (info icon)
 - Location ID (info icon)
 - Resource group ID (info icon)
 - Resource ID (info icon)
 - Resource type (info icon)
 - Subscription ID
 - Tags
 - Type (info icon)
 - Zones

Selected columns pane:

- Header: Selected columns
- Text: Drag the column names below to reorder how they will appear above your grid.
- List of columns:
 - Resource group (highlighted with a grey background)
 - Location (info icon)
 - Subscription
- Buttons:
 - (highlighted with a red circle labeled '2')
 - ←

Figure 6.9: Adding the IP address to the selected columns

4. Hit **Apply** to show the actual IP addresses. When you see yours, click on it. In the blade for your IP address, enter the **Configuration** view. Then, enter a *unique* DNS name and hit **Save**:

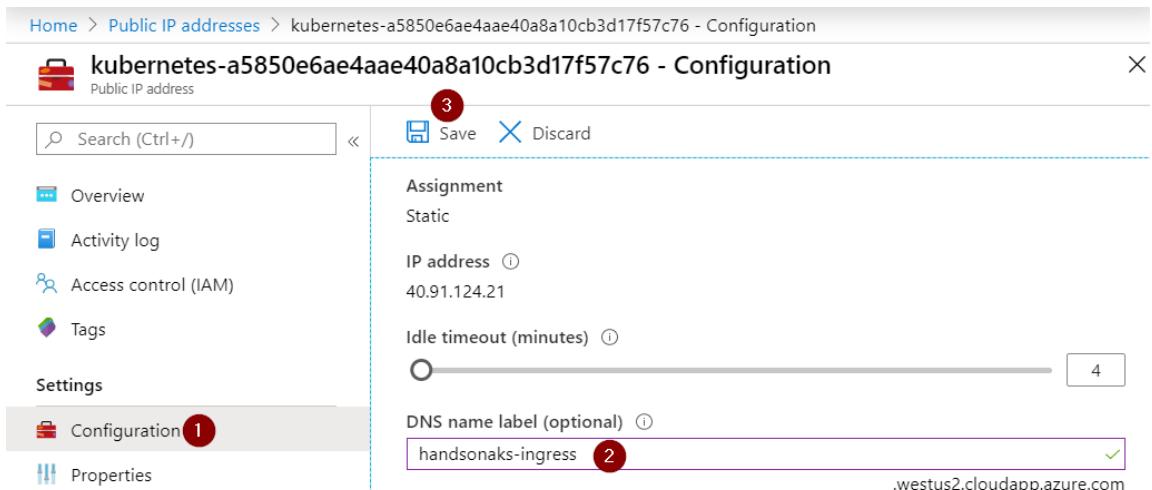


Figure 6.10: Adding a unique DNS name label and saving the configuration

You now have a DNS record linked to your public IP address. Next, you will install the certificate issuer in your cluster.

Installing the certificate issuer

In this section, we 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 we are building tests, we'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`, which has the following contents:

```

1  apiVersion: cert-manager.io/v1alpha2
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-mailaddress>
9          privateKeySecretRef:
10             name: letsencrypt-staging
11         solvers:
12             - http01:
13                 ingress:
14                     class: nginx
```

Now, let's look at what we have defined here:

- **Lines 1-2:** Here, we have used the **CustomResourceDefinition (CRD)** that we installed earlier. A CRD is a way to extend the Kubernetes API server to create custom resources, like a certificate issuer. In this case specifically, we point to the **cert-manager** API CRD we injected into the Kubernetes API and create an **Issuer** object.
- **Lines 6-10:** We have provided the configuration for Let's Encrypt in these lines and pointed to the staging server.
- **Lines 11-14:** This is additional configuration for the ACME client to certify domain ownership.

With the certificate issuer installed, we can now move to the next step: creating the TLS certificate.

Creating the TLS certificate and securing our service

In this section, we 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 controller, or you can configure your Ingress controller so **cert-manager** automatically creates the certificate. In this example, we'll show you the second way, by editing our 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: extensions/v1beta1
2  kind: Ingress
3  metadata:
4    name: simple-frontend-ingress
5  annotations:
6    cert-manager.io/issuer: "letsencrypt-staging"
7  spec:
8    tls:
9      - hosts:
10        - <your DNS prefix>.<your azure region>.cloudapp.azure.com
11        secretName: frontend-tls
12    rules:
13      - host: <your DNS prefix>.<your Azure location>.cloudapp.azure.com
14        http:
15          paths:
16            - path: /
17              backend:
18                serviceName: frontend
19                servicePort: 80
```

We have made the following changes to the original Ingress:

- **Line 6:** We have added an annotation to our Ingress that points to a certificate issuer.
- **Lines 10 and 13:** The domain name for our Ingress has been added here. This is required because Let's Encrypt only issues certificates for domains.
- **Line 11:** This is the name of the secret that will be created to store our certificate.

You can update the Ingress we 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 our Ingress to use that certificate. While we are waiting for that, let's have a look at the intermediate resources that **cert-manager** created on our behalf.

First off, **cert-manager** created a **certificate** object for us. We can look at the status of that object using:

```
kubectl get certificate
```

This command will generate an output as shown in Figure 6.11:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter06$ kubectl get certificate
NAME      READY   SECRET     AGE
frontend-tls  False  frontend-tls  3s
```

Figure 6.11: Output displaying the status of the certificate

As you can see, our certificate isn't ready yet. There is another object that **cert-manager** created to actually get the certificate. This object is **certificaterequest**. We can get its status by using the following command:

```
kubectl get certificaterequest
```

This will generate the output shown in Figure 6.12:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter06$ kubectl get certificaterequest
NAME      READY   AGE
frontend-tls-1688671169  False  8s
```

Figure 6.12: Output displaying the status of the certificaterequest object as False

We can also get more details about the request by issuing a **describe** command against the **certificaterequest** object:

```
kubectl describe certificaterequest
```

While we're waiting for our certificate to be issued, the status will look similar to *Figure 6.13*:

```
Status:  
  Conditions:  
    Last Transition Time: 2020-03-01T01:32:32Z  
    Message: Waiting on certificate issuance from order default/frontend-tls-1688671169-4040173943: "pending"  
    Reason: Pending  
    Status: False  
    Type: Ready  
Events:  
  Type Reason Age From Message  
  ---- ---- - - -  
  Normal OrderCreated 13s cert-manager Created Order resource default/frontend-tls-1688671169-4040173943
```

Figure 6.13: Output providing more details on the certificaterequest object

If we give this a couple of additional seconds, the **describe** command should return a successful certificate creation message, as shown in *Figure 6.14*:

```
Conditions:  
  Last Transition Time: 2020-03-01T01:32:58Z  
  Message: Certificate fetched from issuer successfully  
  Reason: Issued  
  Status: True  
  Type: Ready  
Events:  
  Type Reason Age From Message  
  ---- ---- - - -  
  Normal OrderCreated 44s cert-manager Created Order resource default/frontend-tls-1688671169-4040173943  
  Normal CertificateIssued 18s cert-manager Certificate fetched from issuer successfully
```

Figure 6.14: Output displaying the issued certificate

This should now enable our front-end Ingress to be served over HTTPS. Let's try this out in a browser by browsing to the DNS name you created in the section mapping an FQDN. This will indicate an error in the browser, showing you that the certificate isn't valid as shown in *Figure 6.15*. This is to be expected since we are using the Let's Encrypt staging service:

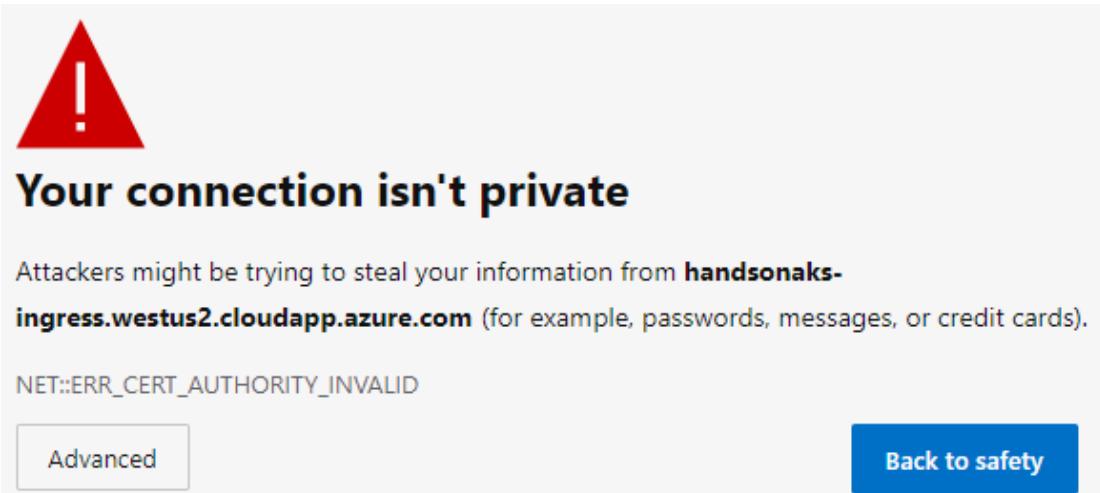


Figure 6.15: Using the Let's Encrypt staging server, our certificate isn't trusted by default

You can browse to your application by clicking **Advanced** and selecting **Continue**.

Since we were able to complete our test with the staging certificate, we can now move on to production.

Switching from staging to production

In this section, we 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`). Don't forget to change your email address in the file:

```
1  apiVersion: cert-manager.io/v1alpha2
2  kind: Issuer
3  metadata:
4    name: letsencrypt-prod
5  spec:
```

```
6   acme:
7     server: https://acme-v02.api.letsencrypt.org/directory
8     email: <your e-mail>
9     privateKeySecretRef:
10       name: letsencrypt-staging
11   solvers:
12     - http01:
13       ingress:
14         class: nginx
```

And then replace the reference to the issuer in the `ingress-with-tls.yaml` file to `letsencrypt-prod`, like this (provided in the `ingress-with-tls-prod.yaml` file):

```
1  apiVersion: extensions/v1beta1
2  kind: Ingress
3  metadata:
4    name: simple-frontend-ingress
5  annotations:
6    cert-manager.io/issuer: "letsencrypt-prod"
7  spec:
8    tls:
9      - hosts:
10        - <your dns prefix>.<your azure region>.cloudapp.azure.com
11        secretName: frontend-tls
12    rules:
13      - host: <your dns prefix>.<your azure region>.cloudapp.azure.com
14        http:
15          paths:
16            - path: /
17              backend:
18                serviceName: frontend
19                servicePort: 80
```

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 item in the browser, you should see that your connection is secure and uses a valid certificate:

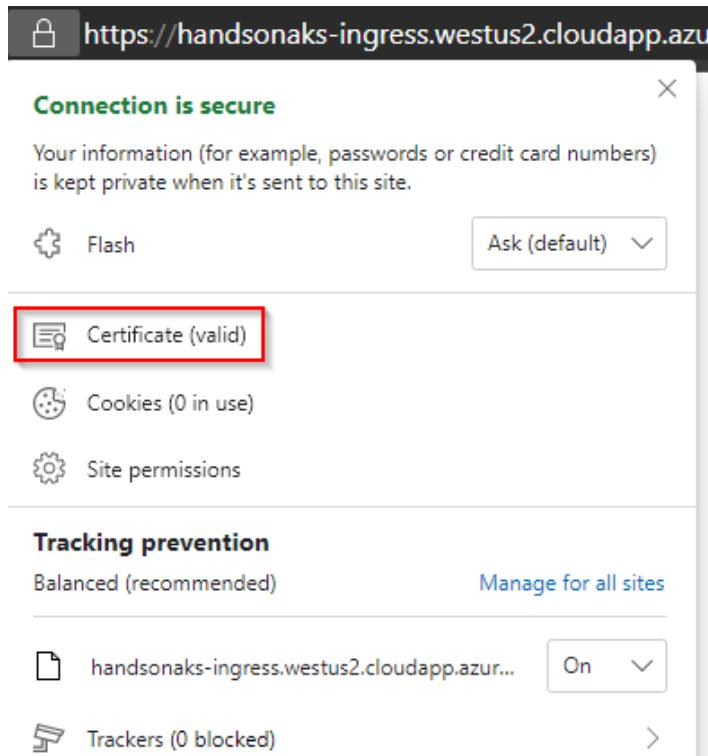


Figure 6.16: The web page displaying a valid certificate

In this section, we have introduced two new concepts: first, we added an Ingress to our guestbook application. We did this by first setting up an Ingress controller on our cluster and then setting up the Ingress for our guestbook application. An Ingress allows advanced routing and HTTPS offloading for applications on top of Kubernetes. After that, we added HTTPS to our guestbook application. We didn't have to change the source code of the application itself; we were able to add HTTPS support by configuring our Ingress.

In the next section, we'll add yet another security layer. We will add authentication to our application. Before diving into this, let's first discuss a common misunderstanding about authentication and authorization.

Authentication versus authorization

Authentication (AuthN) is very often mixed up with **authorization (AuthZ)**.

Authentication deals with identity (who are you?) and, in general, requires a trusted identity provider. Multiple providers exist, such as Azure AD, Okta, or GitHub, and even social media platforms such as Facebook, Google, or Twitter can be used as a provider. Authorization deals with permissions (what are you trying to do?) and is very implementation-specific in terms of what application resources need to be protected.

It generally takes multiple attempts to understand the difference, and even then you can still get confused between the two. The source of confusion is that in some cases, the authentication provider and the authorization provider are the same. For instance, in our WordPress example, WordPress provides the authentication (it has the username and password) and authorization (it stores the users under admin or user roles, for example).

However, in most cases, the authentication system and the authorization system are different. We'll use a practical example of this in *Chapter 10, Securing your AKS cluster*. In that chapter, we will use Azure AD as an authentication source, while using Kubernetes RBAC as the authorization source.

Authentication and common authN providers

Our guestbook application is open to all and lets anyone with a public IP access the service. The image by itself has no authentication. A common problem is wanting to apply additional functionality separately from the application implementation. This can be done by introducing a proxy that will serve the authentication traffic, rather than introducing the authentication logic in the main application.

With recent hacks, it has proven to be difficult to set up and maintain a secure authentication system by yourself. To help their customers build secure applications, many companies allow you to use their authentication service to verify a user's identity. Authentication is provided as a service by those providers with support for OAuth. Here are some of the well-known providers:

- **Azure** (https://github.com/pusher/oauth2_proxy#azure-auth-provider)
- **Facebook** (https://github.com/pusher/oauth2_proxy#facebook-auth-provider)
- **GitHub** (https://github.com/pusher/oauth2_proxy#github-auth-provider)
- **GitLab** (https://github.com/pusher/oauth2_proxy#gitlab-auth-provider)
- **Google** (https://github.com/pusher/oauth2_proxy#google-auth-provider)
- **LinkedIn** (https://github.com/pusher/oauth2_proxy#linkedin-auth-provider)

In the following sections, we will use a proxy implementation, `oauth2_proxy`, to implement authentication for our guestbook example.

Deploying the `oauth2_proxy` proxy

Let's start with cleaning up the previously deployed Ingress. We'll keep the certificate issuer deployed in the cluster. We can clean up the Ingress using:

```
kubectl delete -f ingress-with-tls-prod.yaml
```

We are going to implement `oauth2_proxy` from Pusher (https://github.com/pusher/oauth2_proxy). Implement the following steps to configure `oauth2_proxy` to use Azure AD as an authentication system.

First, register an app with Azure AD. Open the Azure AD blade in the portal by searching for `azure active directory` in the search bar:

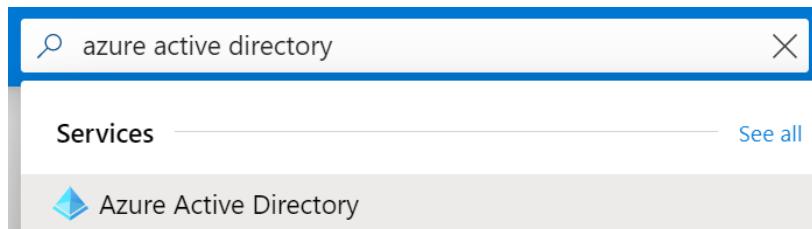


Figure 6.17: Searching for Azure Active Directory in the Azure search bar

Then, go to **App registrations** and click **New registration**:

The screenshot shows the 'Default Directory - App registrations' page in the Azure portal. On the left, there's a sidebar with links like Overview, Getting started, Diagnose and solve problems, Manage (with sub-links for Users, Groups, Organizational relationships, Roles and administrators, Enterprise applications, and Devices), and App registrations (marked with a red circle containing '1'). The main area has a search bar and navigation tabs for All applications, Owned applications, and Applications from personal account. A prominent red box highlights the '+ New registration' button at the top of the list of existing applications. Another red circle with '2' is placed above the 'Endpoints' tab.

Display name	Application (client) ID	Created On	Certificates & secrets
handsonaksSP-20200201114145	9fd72c79-1848-4b0b-a6c...	2/1/2020	-
testSP-20200207185429	03cb3fbc-6b55-4e80-ad8...	2/7/2020	-

Figure 6.18: Creating a new application registration

Then, provide a name for the application and hit **Create**:

The screenshot shows the 'Register an application' form. At the top, it says 'Home > Default Directory - App registrations > Register an application'. The main section starts with a required 'Name' field, which contains 'hands-on-aks-oauth'. Below it is a 'Supported account types' section with three options: 'Accounts in this organizational directory only (Default Directory only - Single tenant)' (selected), 'Accounts in any organizational directory (Any Azure AD directory - Multitenant)', and 'Accounts in any organizational directory (Any Azure AD directory - Multitenant) and personal Microsoft accounts (e.g. Skype, Xbox)'. There's also a 'Help me choose...' link. The next section is 'Redirect URI (optional)', with a note about returning the authentication response to a URI. It includes a dropdown set to 'Web' and a text input field with 'e.g. https://myapp.com/auth'.

Figure 6.19: Providing a name for the application

Next, create a client ID secret by performing the following steps:

1. Select **Certificates & secrets** and go to **New client secret**. Give the secret a description and hit **Add**:

The screenshot shows the Azure portal interface for managing app registrations. On the left, a sidebar lists various management options like Overview, Quickstart, Manage (Branding, Authentication, Certificates & secrets, Token configuration (preview), API permissions, Expose an API, Owners, Roles and administrators (Preview), Manifest, Support + Troubleshooting, Troubleshooting, and New support request). The 'Certificates & secrets' option is selected and highlighted with a red circle containing the number '1'. In the main content area, a sub-menu titled 'hands-on-aks-oauth - Certificates & secrets' is open, showing the 'Add a client secret' form. The 'Description' field contains 'oauth2 proxy' (marked with a red circle '3'). The 'Expires' section has 'In 1 year' selected (radio button marked with a red circle '4'). At the bottom of the form are 'Add' and 'Cancel' buttons. Below the form, a table titled 'Client secrets' is shown with one entry: 'No client secrets have been created for this application.' A 'New client secret' button is visible at the top of the table (marked with a red circle '2').

Figure 6.20: Creating a new client secret

2. Click on the copy icon and save the secret in a safe place:

Description	Expires	Value	
oauth2 proxy	2/9/2021	Oo0ZKKKRXB?@3gLpX5K...	 

Figure 6.21: Copying the client secret

3. Next, we will need to configure a redirect URL. This is the URL that Azure AD will call back once the user has been authenticated. To configure this, head over to **Authentication** in the Azure AD blade, click on **Add a platform**, and select **Web**, as shown in Figure 6.22:

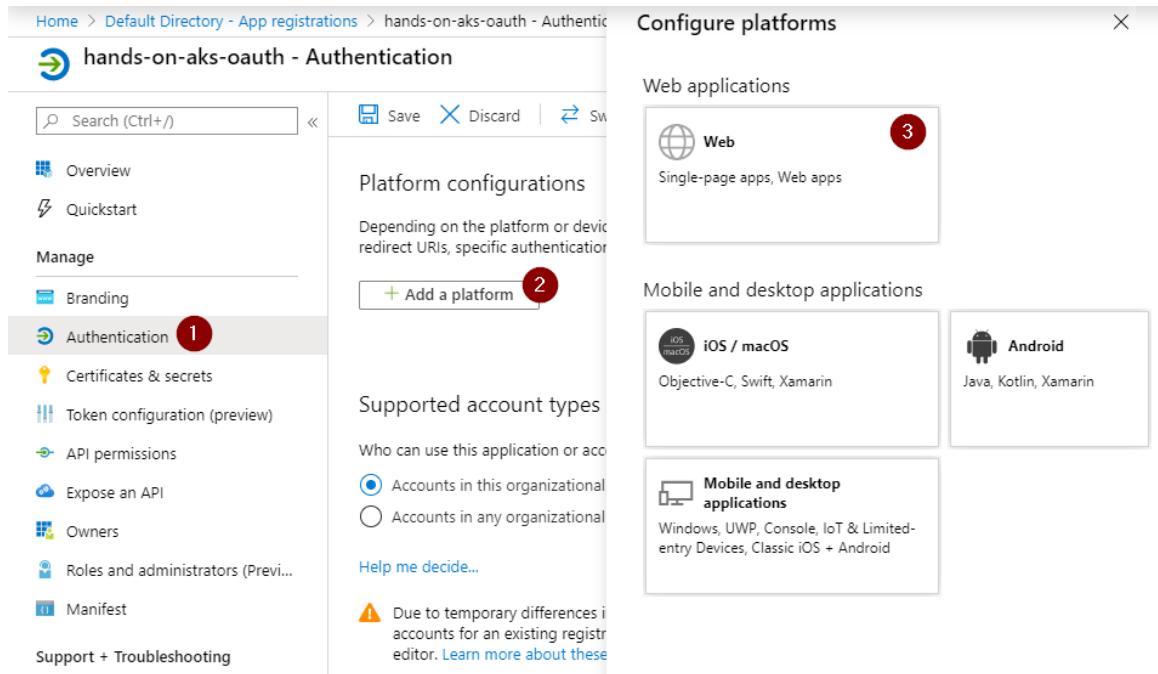


Figure 6.22: Providing a redirect URL

Once there, you can enter the following URL:

`https://<your dns prefix>.<your azure region>.cloudapp.azure.com/oauth2/callback` and hit **Configure**.

4. Then, go back to the **Overview** blade and save the **Application** and the **Directory ID**:

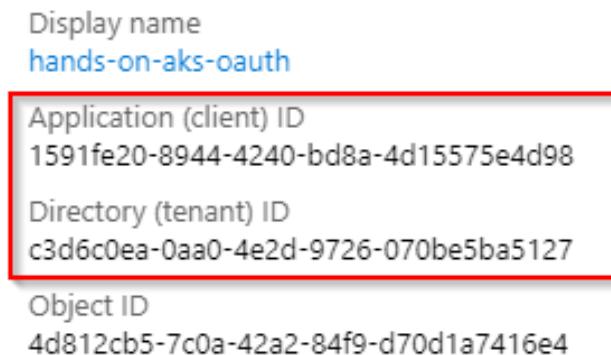


Figure 6.23: Copying the Application ID and the Directory ID

After creating the client ID secret, setting the redirect URL, and copying the application and directory IDs, we need to create the following three objects in Kubernetes to get **oauth2_proxy** working on our cluster and perform a final step to link OAuth to our existing Ingress:

1. First, we need to create a deployment for **oauth2_proxy**.
2. Then, we need to expose this as a service.
3. After that, we will create a new Ingress for **oauth2**.
4. And finally, we will reconfigure our current Ingress to send unauthenticated requests to **oauth2_proxy**.

We will execute all three steps while showing the YAML files as follows:

1. Let's start with the first item – creating the deployment. The deployment can be found in the source as the **oauth2_deployment.yaml** file:

```
1  apiVersion: extensions/v1beta1
2  kind: Deployment
3  metadata:
4    name: oauth2-proxy
5  spec:
6    replicas: 1
7    selector:
8      matchLabels:
9        app: oauth2-proxy
10   template:
11     metadata:
12       labels:
13         app: oauth2-proxy
14   spec:
15     containers:
16       - env:
17         - name: OAUTH2_PROXY_PROVIDER
18           value: azure
19         - name: OAUTH2_PROXY_AZURE_TENANT
20           value: <paste in directory ID>
21         - name: OAUTH2_PROXY_CLIENT_ID
22           value: <paste in application ID>
23         - name: OAUTH2_PROXY_CLIENT_SECRET
24           value: <paste in client secret>
25         - name: OAUTH2_PROXY_COOKIE_SECRET
26           value: somethingveryrandom
27         - name: OAUTH2_PROXY_HTTP_ADDRESS
28           value: "0.0.0.0:4180"
29         - name: OAUTH2_PROXY_UPSTREAM
30           value: "https://<your DNS prefix>.<your azure region>.
cloudapp.azure.com/"
31           - name: OAUTH2_PROXY_EMAIL_DOMAINS
32             value: '*'
33   image: quay.io/pusher/oauth2_proxy:latest
34   imagePullPolicy: IfNotPresent
35   name: oauth2-proxy
36   ports:
37     - containerPort: 4180
38       protocol: TCP
```

This deployment has a couple of interesting lines to discuss. We discussed the other lines in previous examples in *Chapter 3, Application deployment on AKS*. We'll focus on the following lines here:

Lines 17-18: These lines tell the **oauth2** proxy to redirect to Azure AD.

Lines 19-32: This is the **oauth2** configuration.

Line 33: This line points to the correct container image. This is the first time we have used an image that is not hosted on Docker Hub. **Quay** is an alternative container repository, hosted by RedHat.

Create the deployment using the following command:

```
kubectl create -f oauth2_deployment.yaml
```

2. Next, **oauth2** needs to be exposed as a service so that the Ingress can talk to it by creating the following service (**oauth2_service.yaml**):

```
1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: oauth2-proxy
5    namespace: default
6  spec:
7    ports:
8      - name: http
9        port: 4180
10       protocol: TCP
11       targetPort: 4180
12     selector:
13       app: oauth2-proxy
```

Create this service using:

```
kubectl create oauth2_service.yaml
```

3. Next, we will create an Ingress so that any URL that goes to `handsonaks-ingress-<yourname>. <your azure region>.cloudapp.azure.com/oauth` will be redirected to the `oauth2-proxy` service. The same Let's Encrypt certificate issuer is used here (the `oauth2_ingress.yaml` file in the source code for this chapter):

```
1  apiVersion: extensions/v1beta1
2  kind: Ingress
3  metadata:
4    name: oauth2-proxy-ingress
5  annotations:
6    kubernetes.io/ingress.class: nginx
7    cert-manager.io/issuer: "letsencrypt-prod"
8  spec:
9    tls:
10   - hosts:
11     - <your DNS prefix>.<your azure region>.cloudapp.azure.com
12     secretName: tls-secret
13   rules:
14   - host: <your DNS prefix>.<your azure region>.cloudapp.azure.com
15     http:
16       paths:
17         - path: /oauth2
18           backend:
19             serviceName: oauth2-proxy
20             servicePort: 4180
```

There is one interesting line to point out in this Ingress. **Line 17** introduced a new path to our Ingress. As mentioned earlier in this chapter, the same Ingress can have multiple paths being directed to different back-end services. This is what we are configuring here.

Create this Ingress by using the following command:

```
kubectl create -f oauth2_ingress.yaml
```

4. Finally, we will link the `oauth2` proxy to the front-end service by creating an Ingress that configures `nginx` so that authentication is checked using the paths in `auth-url` and `auth-signin`. If the request is not authenticated, traffic is sent to the `oauth2_proxy`. If it is successfully authenticated, the traffic is redirected to the back-end service (in our case, it is the front-end service).

The following code performs the redirection once authentication is successful (`frontend-oauth2-ingress.yaml` in the GitHub repository):

```
1 apiVersion: extensions/v1beta1
2 kind: Ingress
3 metadata:
4   name: frontend-oauth2-ingress
5   annotations:
6     kubernetes.io/ingress.class: nginx
7     nginx.ingress.kubernetes.io/auth-url: "http://oauth2-proxy.default.svc.cluster.local:4180/oauth2/auth"
8     nginx.ingress.kubernetes.io/auth-signin: "http://<your DNS prefix>.<your azure region>.cloudapp.azure.com/oauth2/start"
9   spec:
10  rules:
11    - host: <your DNS prefix>.<your azure region>.cloudapp.azure.com
12      http:
13        paths:
14          - path: /
15            backend:
16              serviceName: frontend
17              servicePort: 80
```

There are a couple of interesting things to point in this Ingress configuration. The other lines are common with the other Ingresses we have created throughout this chapter:

Line 5: As mentioned previously, the Ingress object can be backed by multiple technologies (such as NGINX or Application Gateway). The Ingress object has a syntax to configure basic tasks, such as `hosts` and `paths`, but it doesn't have a configuration for authentication redirects, for example. Annotations are used by multiple Ingress providers to pass detailed configuration data to the Ingress provider in the back end.

Lines 7-8: This configures our Ingress to send non-authenticated requests to these URLs.

Create this Ingress using the following command:

```
kubectl create -f frontend-oauth2-ingress.yaml
```

We are now done with configuration. You can now log in with your existing Microsoft account to the service at <https://handsonaks-ingress-<yourname>.cloudapp.azure.net/>. To make sure you get the authentication redirect, please make sure to use a new browser window or a private window. You should get automatically redirected to Azure AD's sign in page.

Note

oauth2-proxy supports multiple authentication providers, such as GitHub and Google. Only the **oauth2-proxy** deployment's YAML has to be changed with the right service to change the authentication provider. Please see the relevant details at https://github.com/pusher/oauth2_proxy#oauth-provider-configuration.

Now that everything has been deployed, let's clean up the resources we created in our cluster:

```
kubectl delete -f guestbook-all-in-one.yaml  
kubectl delete -f frontend-oauth2-ingress.yaml  
kubectl delete -f oauth2_ingress.yaml  
kubectl delete -f oauth2_deployment.yaml  
kubectl delete -f oauth2_service.yaml  
kubectl delete ns cert-manager  
helm delete ingress
```

In this section, we have added Azure AD authentication to your application. We did this by adding the **oauth2_proxy** to our cluster, and we then reconfigured the existing Ingress to redirect unauthenticated requests to **oauth2_proxy**.

Summary

In this chapter, we added HTTPS security and identity control to our guestbook application without actually changing the source code. We started by getting the Kubernetes Ingress objects set up in our cluster. Then, we installed a certificate manager that interfaces with the Let's Encrypt API to request a certificate for the domain name we subsequently specified. We leveraged a certificate issuer to get the certificate from Let's Encrypt. We then reconfigured our Ingress to request a certificate from this issuer in the cluster.

Then, we jumped into authentication and authorization and showed you how to leverage Azure AD as an authentication provider for the guestbook application. You learned how to secure your applications on an enterprise scale. By integrating with Azure AD, you can enable any application to link to an organization's AD.

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 you will learn 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 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 **Azure Monitor** is used, and how it is integrated within the Azure portal, as well as how to set up alerts for critical events on your AKS cluster. You will see how you can use Azure Monitor to monitor the status of the cluster itself, the Pods on the cluster and 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 to provide 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 applications errors

When monitoring 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. 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>
```

We'll describe each of these commands in detail in this chapter.

Before we begin, we are going to have a clean start with our guestbook example. Recreate the guestbook example again using the following command:

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

While the **create** command is running, we will watch its progress in the following sections.

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. We have already run this command in the previous chapters to verify that our application was ready to be used. 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-57d8c9fb45-c6qtm	1/1	Running	0	55s	
pod/frontend-57d8c9fb45-dvgqc	1/1	Running	0	55s	
pod/frontend-57d8c9fb45-npx92	1/1	Running	0	55s	
pod/redis-master-545d695785-rnb8j	1/1	Running	0	55s	
pod/redis-slave-84548fdbca-hjzs5	1/1	Running	0	55s	
pod/redis-slave-84548fdbca-xsd56	1/1	Running	0	55s	
NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
service/frontend	ClusterIP	10.0.210.64	<none>	80/TCP	55s
service/kubernetes	ClusterIP	10.0.0.1	<none>	443/TCP	75s
service/redis-master	ClusterIP	10.0.143.119	<none>	6379/TCP	55s
service/redis-slave	ClusterIP	10.0.36.12	<none>	6379/TCP	55s
NAME	READY	UP-TO-DATE	AVAILABLE	AGE	
deployment.apps/frontend	3/3	3	3	55s	
deployment.apps/redis-master	1/1	1	1	55s	
deployment.apps/redis-slave	2/2	2	2	55s	
NAME	DESIRED	CURRENT	READY	AGE	
replicaset.apps/frontend-57d8c9fb45	3	3	3	55s	
replicaset.apps/redis-master-545d695785	1	1	1	55s	
replicaset.apps/redis-slave-84548fdbca	2	2	2	55s	

Figure 7.1: All the resources running in the default namespace

- Let's focus our attention on the Pods in our Deployment. We can get the status of the Pods with the following command:

```
kubectl get pods
```

You will see that, now, 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-57d8c9fb45-c6qtm	1/1	Running	0	81s	
frontend-57d8c9fb45-dvgqc	1/1	Running	0	81s	
frontend-57d8c9fb45-npx92	1/1	Running	0	81s	
redis-master-545d695785-rnb8j	1/1	Running	0	81s	
redis-slave-84548fdbca-hjzs5	1/1	Running	0	81s	
redis-slave-84548fdbca-xsd56	1/1	Running	0	81s	

Figure 7.2: All the Pods in your namespace

The first column indicates the Pod name, for example, **frontend-57d8c9fb45-c6qtm**. 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. We have a dedicated section called *Readiness and liveness probes* later in this chapter.

The third column indicates the status, for example, **Pending**, or **ContainerCreating**, or **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.

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:

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE	NOMINATED NODE	READINESS	GATES
frontend-57d8c9fb45-c6qtm	1/1	Running	0	109s	10.244.5.52	aks-agentpool-39838025-vmss000005	<none>	<none>	
frontend-57d8c9fb45-dvgqc	1/1	Running	0	109s	10.244.1.72	aks-agentpool-39838025-vmss000001	<none>	<none>	
frontend-57d8c9fb45-npx92	1/1	Running	0	109s	10.244.0.85	aks-agentpool-39838025-vmss000000	<none>	<none>	
redis-master-545d95785-rnb8j	1/1	Running	0	109s	10.244.0.87	aks-agentpool-39838025-vmss000000	<none>	<none>	
redis-slave-84548fdbcb-hjzs5	1/1	Running	0	109s	10.244.4.42	aks-agentpool-39838025-vmss000004	<none>	<none>	
redis-slave-84548fdbcb-xsd56	1/1	Running	0	109s	10.244.0.86	aks-agentpool-39838025-vmss000000	<none>	<none>	

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 is the node where that higher-priority Pod will start once the lower-priority Pods gracefully terminate. 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. 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. All the commands work only if the event was fired within the past hour.

If everything goes well, you should have an output similar to *Figure 7.4*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter07$ kubectl get events
LAST SEEN   TYPE    REASON   OBJECT   MESSAGE
3m48s       Normal   Scheduled  pod/frontend-57d8c9fb45-c6qtm   Successfully assigned default/frontend-57d8c9fb45-c6qtm to aks-a
ss0000005
3m47s       Normal   Pulling   pod/frontend-57d8c9fb45-c6qtm   Pulling image "gcr.io/google-samples/gb-frontend:v4"
2m59s       Normal   Pulled   pod/frontend-57d8c9fb45-c6qtm   Successfully pulled image "gcr.io/google-samples/gb-frontend:v4"
2m56s       Normal   Created   pod/frontend-57d8c9fb45-c6qtm   Created container php-redis
2m56s       Normal   Started   pod/frontend-57d8c9fb45-c6qtm   Started container php-redis
```

Figure 7.4: Getting the events shows all events from the past hour

As you can see in the output, the general states for a Pod are **Scheduled** | **Pulling** | **Pulled** | **Created** | **Started**. 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 get events** command lists all the events for the entire namespace. 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 **podname**. 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 7.5*, which will be explained in detail later:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter07$ kubectl describe pod frontend-57d8c9fb45-c6qtm
Name:           frontend-57d8c9fb45-c6qtm
Namespace:      default
Priority:      0
Node:          aks-agentpool-39838025-vmss000005/10.240.0.7
Start Time:    Wed, 04 Mar 2020 02:53:55 +0000
Labels:         app=guestbook
                pod-template-hash=57d8c9fb45
                tier=frontend
Annotations:   <none>
Status:        Running
IP:            10.244.5.52
IPs:           <none>
Controlled By: ReplicaSet/frontend-57d8c9fb45
Containers:
  php-redis:
    Container ID:   docker://942f795eb6fc82d0cc4dce4e767586cd30a3e67cb68097c74e94986cd2cd027d
    Image:          gcr.io/google-samples/gb-frontend:v4
    Image ID:       docker-pullable://gcr.io/google-samples/gb-frontend@sha256:d44e7d7491a537f822e7fe86154
    Port:          80/TCP
    Host Port:     0/TCP
    State:         Running
      Started:    Wed, 04 Mar 2020 02:54:47 +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-w2j4n (ro)
Conditions:
  Type      Status
  Initialized  True
  Ready      True
  ContainersReady  True
  PodScheduled  True
Volumes:
  default-token-w2j4n:
    Type:      Secret (a volume populated by a Secret)
    SecretName: default-token-w2j4n
    Optional:   false
  QoS Class:  Burstable
  Node-Selectors: <none>
  Tolerations: node.kubernetes.io/not-ready:NoExecute for 300s
                node.kubernetes.io/unreachable:NoExecute for 300s
Events:
  Type  Reason  Age   From               Message
  ----  -----  --   --   -----
  Normal Scheduled  5m49s  default-scheduler  Successfully assigned default/bronze-frontend-57d8c9fb45-c6qtm
  Normal Pulling   5m48s  kubelet, aks-agentpool-39838025-vmss000005  Pulling image "gcr.io/google-samples/gb-frontend:v4"
  Normal Pulled    5m   kubelet, aks-agentpool-39838025-vmss000005  Successfully pulled image "gcr.io/google-samples/gb-frontend:v4"
  Normal Created   4m57s  kubelet, aks-agentpool-39838025-vmss000005  Created container php-redis
  Normal Started   4m57s  kubelet, aks-agentpool-39838025-vmss000005  Started container php-redis
```

Figure 7.5: 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-57d8c9fb45-c6qtm**. As mentioned in *Chapter 1, Introduction to Docker 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 5 chars>**.

Figure 7.6 shows the relationship between a Deployment, a ReplicaSet, and Pods:



Figure 7.6: Relationship between a Deployment, a ReplicaSet, and Pods

The namespace under which the Pod runs is **default**. So far, we have just been using the **default** namespace, appropriately named **default**. In the next chapters, we will see how namespaces help us to isolate Pods.

Another section that is important from the preceding output is the node section:

```
Node: aks-agentpool-39838025-vmss000005/10.240.0.7
```

The node section lets us 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. Having this information is essential to perform advanced debugging.

The following is the time the Pod was initially scheduled:

```
Start Time: Wed, 04 Mar 2020 02:53:55 +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), a Pod will be restarted.

Connections between resources are made using **Labels**, as shown here:

```
Labels:app=guestbook
pod-template-hash=57d8c9fb45
tier=frontend
```

This is how connections such as **Service | Deployment | ReplicaSet | Pod** 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. 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:10.244.5.52

As mentioned in previous chapters, when building out your application, the Pods can be moved to different nodes and get a different IP. However, when debugging application issues, having a direct IP for a Pod can help in troubleshooting. Instead of connecting to your application through a Service object, you can connect directly from one Pod to another 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

Host Port:0/TCP

Environment:

GET_HOSTS_FROM dns

In this case, we 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 were to have two Pods running a web server on the same node, those could both 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. The port that needs to be configured is also fixed so that scripts can be written simply without the logic of figuring out which port actually got allocated for the Pod.

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. In the next section, we'll focus on debugging applications.

Debugging applications

Now that we have a basic understanding of how to monitor Deployments, we can start seeing how we can debug issues with Deployments.

In this section, we will introduce common errors and determine how to debug and fix them.

If you have not implemented the guestbook application already, run the following command:

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

After a period of time, the Services should be up and running.

Image pull errors

In this section, we 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.

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:

1. Type **/gb-frontend** and hit the Enter button to have your cursor brought to the image definition.
2. Hit the I key to go into insert mode. Delete **v4** and type **v_non_existent**.
3. Now, close the editor by first hitting the Esc key, then type **:wq!** and hit Enter.

Running the following command lists all the Pods in the current namespace:

```
kubectl get pods
```

The preceding command should indicate errors, as shown in *Figure 7.7*:

NAME	READY	STATUS	RESTARTS	AGE
frontend-57d8c9fb45-c6qtm	1/1	Running	0	10m
frontend-57d8c9fb45-dvgqc	1/1	Running	0	10m
frontend-57d8c9fb45-npx92	1/1	Running	0	10m
frontend-f65bf449d-j1jbw	0/1	ImagePullBackOff	0	3s

Figure 7.7: One of the Pods has the status of ImagePullBackOff

Run the following command to get the full error details:

```
kubectl describe pods/<failed pod name>
```

A sample error output is shown in *Figure 7.8*. The key error line is highlighted in red:

Events:			
Type	Reason	Age	From
Normal	Scheduled	60s	default-scheduler
Normal	Pulling	18s (x3 over 59s)	kubelet, aks-agentpool-39838025-vmss000005
Warning	Failed	17s (x3 over 59s)	kubelet, aks-agentpool-39838025-vmss000005
Warning	Failed	17s (x3 over 59s)	kubelet, aks-agentpool-39838025-vmss000005
Normal	BackOff	5s (x4 over 58s)	kubelet, aks-agentpool-39838025-vmss000005
Warning	Failed	5s (x4 over 58s)	kubelet, aks-agentpool-39838025-vmss000005

Figure 7.8: Using describe shows more details on the error

So, 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.

Let's fix the error by setting the image tag back to **v4**:

1. First, type the following command in Cloud Shell to edit the Deployment:

```
kubectl edit deployment/frontend
```

2. Type **/gb-frontend** and hit **<enter>** to have your cursor brought to the image definition.
3. Hit the **I** button to go into insert mode. Delete **v_non_existent**, and type **v4**.
4. Now, close the editor by first hitting the **Esc** key, then type **:wq!** and hit Enter.

The Deployment should get automatically fixed. You can verify it by getting the events for the Pods again.

Note

Because Kubernetes did a rolling update, the frontend 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. In the next section, we'll explore an error within the application itself.

Application errors

We will now see how to debug an application error. The errors in this section will be self-induced, similar to in the last section. The method for debugging the issue is the same as the one we used to debug errors on running applications.

In order to test our failure, we'll have to make the **frontend** Service publicly accessible:

1. To start, we'll edit the **frontend** Service:

```
kubectl edit service frontend
```

2. Type **/ClusterIP** and hit Enter to bring your cursor to the type field (line 27).
3. Hit the I button to go into insert mode. Delete the **ClusterIP** and type **LoadBalancer**.
4. Now, close the editor by first hitting the Esc key, then type **:wq!** and hit Enter. This will create a public IP for our frontend Service.
5. We can get this IP using the following command:

```
kubectl get service
```

6. Let's connect to the Service, by pasting its public IP in a browser. Create a couple of entries:

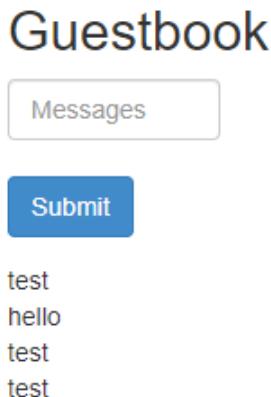


Figure 7.9: Make a couple of entries in the guestbook application

Note

Most errors come from misconfiguration, where they can be fixed by editing the specification. Errors in the application code itself require a new image to be built and used.

You now have an instance of the guestbook application running. To improve the experience with the example, we will scale down the frontend so there is only a single replica running.

Scaling down the frontend

In Chapter 3, *Application deployment on AKS*, you learned how the deployment of the frontend 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, we would need to make changes in all three of them.

To make this example easier, scale the **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, we are going to make the **Submit** button fail to work. We 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 preferable way is to create a new container image and update your Deployment.

We 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 us a shell that we can run our commands on. The following command launches a Bash terminal on the Pod:

```
kubectl exec -it <frontend-pod-name> bash
```

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. Now, use **vim** to open the **guestbook.php** file:

```
vim guestbook.php
```

Add the following line after line 18. Remember, to insert 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 then 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 7.10*:

```
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 7.10: The updated code that introduced an error and additional logging

We are introducing an error where reading messages would work, but not writing them. We do this by asking the frontend 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:

Guestbook

Messages

Submit

test
hello
test
test

Figure 7.11: The entries from earlier are still present

Now, create a new message by typing a message and hitting the **Submit** button:

Guestbook

Messages

Submit

test
hello
test
test
new message

Figure 7.12: A new message was created

Submitting a new message makes it appear in our application. If we did not know any better, we would have thought the entry was written safely. However, if you refresh your browser, you will see that the message is no longer there.

If you have network debugging tools turned on in your browser, you can catch the error response from the server.

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:

Guestbook

Messages

Submit

test
hello
test
test

Figure 7.13: 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.*

Logs

The first step is to get the logs. Let's exit out of our front-end Pod for now and get the logs for this Pod:

```
exit
kubectl logs <frontend-pod-name>
```

You will see entries such as those seen in *Figure 7.14*:

```
10.244.0.1 - - [17/Mar/2020:02:43:23 +0000] "GET /guestbook.php?cmd=set&key
=messages&value=test,hello,test,test HTTP/1.1" 200 254 "http://51.143.59.1
15/" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/81.0.4044.62 Safari/537.36 Edg/81.0.416.31"
hostname at the beginning of 'set' command localhost
10.244.0.1 - - [17/Mar/2020:02:44:41 +0000] "GET /guestbook.php?cmd=set&key
=messages&value=test,hello,test,test,new%20message HTTP/1.1" 200 1401 "htt
p://51.143.59.115/" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/5
37.36 (KHTML, like Gecko) Chrome/81.0.4044.62 Safari/537.36 Edg/81.0.416.3
1"
```

Figure 7.14: The new message shows up as part of the application logs

Hence, you know that the error is somewhere when writing to the database in the **set** section of the code.

You will see this entry:

```
hostname at the beginning of 'set' command localhost
```

So we 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 we 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.

There are two options to fix this bug we introduced now: we can either navigate into the Pod and make the code changes, or we can ask Kubernetes to give us a healthy new Pod. It is not recommended to make manual changes to Pods, so we will use the second approach. Let's fix this bug by deleting our faulty Pod:

```
kubectl delete pod <podname>
```

As we have a ReplicaSet that controls our Pods, we should immediately get a new Pod that has started from the correct image. Try to connect to the guestbook again and verify that messages are persisted across browser refreshes again.

The following points summarize some of the common errors and methods to fix these errors:

- Errors can come in many shapes and forms.
- Most of the errors encountered by the deployment team are configuration issues.
- Logs are your friends.
- Using `kubectl exec` on a container is a useful debugging tool.
- Note that broadly allowing `kubectl exec` is a serious security risk, as it pretty much lets the Kubernetes operator do what they want in the Pods they have access to.
- Anything printed to `stdout` and `stderr` shows up in the logs (independent of the application/language/logging framework).

We introduced an application error to the guestbook application and were able to leverage Kubernetes logs to pinpoint the issue in the code. In the next section, we will look into a powerful mechanism in Kubernetes called *Readiness and liveness probes*.

Readiness and liveness probes

We touched upon readiness probes briefly in the previous section. In this section, we'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 unready Pods. This is useful if your application has to go through some configuration before it becomes available, or if your application could become overloaded but recover from the additional 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. We 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, we'll use a couple of web pages that we'll use to connect to our readiness and liveness probes. 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, we'll mount these files to our Kubernetes Deployments. We'll turn each of these into a **configmap** that we connect to our Pods. 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, we can go ahead and create our 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  
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;
46 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 be restarted.
- **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 our container starts. Instead of simply running the nginx server, we copy the index and ready files in the right location, then start nginx, and then use a sleep command (so our 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, we will also create a Service that routes traffic to both Deployments (`webservice.yaml`):

```
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
```

We can create that Service using the following:

```
kubectl create -f webservice.yaml
```

We now have our application up and running. In the next section, we'll introduce some failures to verify the behavior of liveness and readiness probes.

Experimenting with liveness and readiness probes

In the previous section, we explained the functionality of the liveness and readiness probes and created a sample application. In this section, we will introduce errors in our application and verify the behavior of the liveness and readiness probes. We will see how a failure of the readiness probe will cause the Pod to remain running but no longer accept traffic. After that, we 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 we have a simple application up and running, we can experiment with the behavior of the liveness and readiness probes. To start, let's get our 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**:

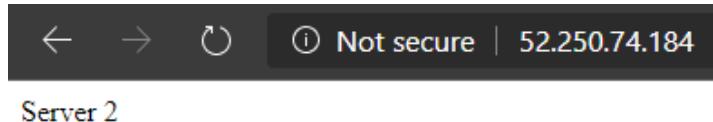


Figure 7.15: Our application is returning traffic from server 2

During our tests, we'll use a small script called `testWeb.sh` to connect to our web page 50 times, so we can monitor a good distribution of results between server 1 and 2. We'll first need to make that script executable, and then we can run that script while our 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$ 
server 1:
 24      48     216
server 2:
 26      52     234
```

Figure 7.16: 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, we will `exec` into the container and 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 7.17:

NAME	READY	STATUS	RESTARTS	AGE
server1-d6c4c9f77-qzkgn	1/1	Running	0	86s
server2-748b56796c-1x6mk	1/1	Running	0	5m58s
server1-d6c4c9f77-qzkgn	0/1	Running	0	93s

Figure 7.17: 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:

```
./testWeb.sh <external-ip>
```

In our case, traffic is indeed redirected to server 2:

server 1:	0	0	0
server 2:	50	100	450

Figure 7.18: All traffic is now served by server 2

We 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 our Pod to a healthy state, and should again split traffic equally:

```
./testWeb.sh <external-ip>
```

This will show an output similar to Figure 7.19:

server 1:	25	50	225
server 2:	25	50	225

Figure 7.19: 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. We have verified this by causing a readiness probe in our example application to fail. In the next section, we'll explore the impact of a failing liveness probe.

A failing liveness probe causes container restart

We can repeat the previous process as well with the liveness probe. When the liveness probe fails, we are expecting Kubernetes to go ahead and restart our Pod. Let's try this by deleting the health file:

```
kubectl exec <server1 pod name> rm /usr/share/nginx/html/healthy.html
```

Let's see what this does with our Pod:

```
kubectl get pods -w
```

We should see that the Pod restarts within a couple of seconds:

user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter07\$ kubectl get pods -w				
NAME	READY	STATUS	RESTARTS	AGE
server1-d6c4c9f77-qzkgn	1/1	Running	0	3m40s
server2-748b56796c-lx6mk	1/1	Running	0	8m12s
server1-d6c4c9f77-qzkgn	0/1	Running	1	4m18s
server1-d6c4c9f77-qzkgn	1/1	Running	1	4m21s

Figure 7.20: A failing liveness probe will cause the Pod to be restarted

As you can see in Figure 7.20, the Pod was successfully restarted, with limited impact. We can inspect what was going on in the Pod by running a **describe** command:

```
kubectl describe pod <server1 pod name>
```

The preceding command will give you an output similar to Figure 7.21:

Events:						
Type	Reason	Age	From	Message		
Normal	Scheduled	8m24s	default-scheduler			Successfully assigned default/server1-d6c4c9f77-qzkgn to aks-agentpool-39838025-vmss000005
Normal	Pulling	8m23s	kubelet, aks-agentpool-39838025-vmss000005	Pulling image "nginx"		
Normal	Pulled	8m14s	kubelet, aks-agentpool-39838025-vmss000005	Successfully pulled image "nginx"		
Normal	Created	8m8s	kubelet, aks-agentpool-39838025-vmss000005	Created container nginx-1		
Normal	Started	8m8s	kubelet, aks-agentpool-39838025-vmss000005	Started container nginx-1		
Warning	Unhealthy	5m57s (x21 over 6m57s)	kubelet, aks-agentpool-39838025-vmss000005	Readiness probe failed: HTTP probe failed with statuscode: 404		
Warning	Unhealthy	53s (x4 over 4m44s)	kubelet, aks-agentpool-39838025-vmss000005	Liveness probe failed: HTTP probe failed with statuscode: 404		

Figure 7.21: More details on the Pod showing how the liveness probe failed

In the **describe** command, we can clearly see that the Pod failed the liveness probe. After four failures, the container was killed and restarted.

This concludes our 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 suffer less load. A liveness probe is used to restart your Pod if there is an actual failure in your Pod.

Let's also make sure to clean up the Deployments we just created:

```
kubectl delete deployment server1 server2  
kubectl delete service web
```

Liveness and readiness probes are useful to ensure only healthy Pods will receive traffic in your cluster. In the next section, we 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, we'll first use a number of `kubectl` commands to get these metrics. Afterward, we'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:

NAME	STATUS	ROLES	AGE	VERSION
aks-agentpool-39838025-vmss00000	Ready	agent	12d	v1.15.7
aks-agentpool-39838025-vmss00001	Ready	agent	12d	v1.15.7

Figure 7.22: 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 7.23:

NAME	STATUS	ROLES	AGE	VERSION	INTERNAL-IP	EXTERNAL-IP	OS-IMAGE	KERNEL-VERSION	CONTAINER-RUNTIME
aks-agentpool-39838025-vmss000000	Ready	agent	12d	v1.15.7	10.240.0.4	<none>	Ubuntu 16.04.6 LTS	4.15.0-1066-azure	docker://3.0.8
aks-agentpool-39838025-vmss000001	Ready	agent	12d	v1.15.7	10.240.0.5	<none>	Ubuntu 16.04.6 LTS	4.15.0-1066-azure	docker://3.0.8

Figure 7.23: Using `-o wide` adds more details about our 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:

NAME	CPU(cores)	CPU%	MEMORY(bytes)	MEMORY%
aks-agentpool-39838025-vmss000000	83m	8%	1026Mi	57%
aks-agentpool-39838025-vmss000001	202m	21%	1353Mi	76%

Figure 7.24: 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:

```
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:

Non-terminated Pods:		(7 in total)					
Namespace	Name		CPU Requests	CPU Limits	Memory Requests	Memory Limits	AGE
default	frontend-57d8c9fb45-4dkfp		10m (1%)	0 (0%)	10Mi (0%)	0 (0%)	19m
default	frontend-f65bf449d-dn1xz		10m (1%)	0 (0%)	10Mi (0%)	0 (0%)	51m
default	redis-master-545d695785-zwbnf		100m (10%)	0 (0%)	100Mi (5%)	0 (0%)	101m
default	redis-slave-84548fdb-919kp		100m (10%)	0 (0%)	100Mi (5%)	0 (0%)	101m
kube-system	kube-proxy-1m9gq		100m (10%)	0 (0%)	0 (0%)	0 (0%)	3d6h
kube-system	omsagent-9xw4h		75m (7%)	150m (15%)	225Mi (12%)	600Mi (33%)	6d6h
kube-system	omsagent-rs-76b774dc7f-z5cf2		150m (15%)	1 (106%)	250Mi (14%)	750Mi (42%)	7d9h

Allocated resources:				
(Total limits may be over 100 percent, i.e., overcommitted.)				
Resource	Requests	Limits		
cpu	545m (57%)	1150m (122%)		
memory	695Mi (39%)	1350Mi (76%)		
ephemeral-storage	0 (0%)	0 (0%)		
attachable-volumes-azure-disk	0	0		

Figure 7.25: Describing the nodes shows details on requests and limits

You now know where you can find information about the utilization of your nodes. In the next section, we 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, we will explore how we can use **kubectl** to get information about the CPU and memory utilization of Pods.

Let's start by exploring how we can see the requests and limits for a Pod that we currently have running:

1. For this example, we 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 7.26:

NAME	READY	STATUS	RESTARTS	AGE
coredns-698c77c5d7-5m6qb	1/1	Running	0	12d
coredns-698c77c5d7-zpfgh	1/1	Running	0	12d
coredns-autoscaler-79b778686c-m94hk	1/1	Running	0	12d
kube-proxy-4j529	1/1	Running	0	30h
kube-proxy-8cl1p	1/1	Running	0	30h
kube-proxy-dgn52	1/1	Running	0	30h
kube-proxy-j6jph	1/1	Running	0	30h
kubernetes-dashboard-74d8c675bc-qw5gr	1/1	Running	0	12d
metrics-server-69df9f75bf-fqsms	1/1	Running	2	12d
omsagent-hx59s	1/1	Running	1	12d
omsagent-n7qrz	1/1	Running	0	8d
omsagent-rs-76dfb8b464-565c5	1/1	Running	1	12d
omsagent-xvcrz	1/1	Running	0	9d
omsagent-z2fhr	1/1	Running	1	12d
tunnelfront-68d6d94ccf-qdk29	1/1	Running	0	12d

Figure 7.26: The Pods running in the kube-system namespace

- 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 7.27*:

```

Limits:
  memory: 170Mi
Requests:
  cpu:     100m
  memory: 70Mi

```

Figure 7.27: Limits and requests for the CoreDNS Pod

This shows us that this Pod has a memory limit of **170Mi**, no CPU limit, and has a request for 100m CPU (which means 0.1 CPU) and **70Mi** of memory.

Requests and limits are used to perform capacity management in a cluster. We can also get the actual CPU and memory consumption of a Pod by running the following command:

```
kubectl top pods -n kube-system
```

This should show you the output similar to *Figure 7.28*:

NAME	CPU(cores)	MEMORY(bytes)
coredns-698c77c5d7-5m6qb	2m	19Mi
coredns-698c77c5d7-zpfgh	3m	16Mi
coredns-autoscaler-79b778686c-m94hk	1m	7Mi
kube-proxy-4j529	3m	18Mi
kube-proxy-8cllp	1m	26Mi
kube-proxy-dgn52	3m	23Mi
kube-proxy-j6jph	1m	20Mi
kubernetes-dashboard-74d8c675bc-qw5gr	1m	17Mi
metrics-server-69df9f75bf-fqsms	1m	23Mi
omsagent-hx59s	4m	114Mi
omsagent-n7qrz	7m	117Mi
omsagent-rs-76dfb8b464-565c5	6m	123Mi
omsagent-xvcrz	6m	143Mi
omsagent-z2fhr	4m	156Mi
tunnelfront-68d6d94ccf-qdk29	68m	67Mi

Figure 7.28: 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, we can see that the **coredns** Pods are using **2m** and **3m** CPU and are using **21Mi** and **17Mi** of memory.

In this section, we have been using the **kubectl** command to get an insight into the resource utilization of the nodes and Pods in our cluster. This is useful information, but it is limited to that specific point in time. In the next section, we'll use Azure Monitor to get more detailed information on the cluster and the applications on top of the cluster.

Metrics reported from Azure Monitor

The Azure portal shows many of the metrics that you would like to see combined with authorization, as only personnel with access to the portal can see these metrics.

AKS Insights

The **Insights** section of the AKS blade 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.

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 then stores 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 blade. We'll start with the cluster metrics.

Cluster metrics

Insights shows the cluster metrics. Figure 7.29 shows the CPU utilization and the memory utilization of all the nodes in the cluster:

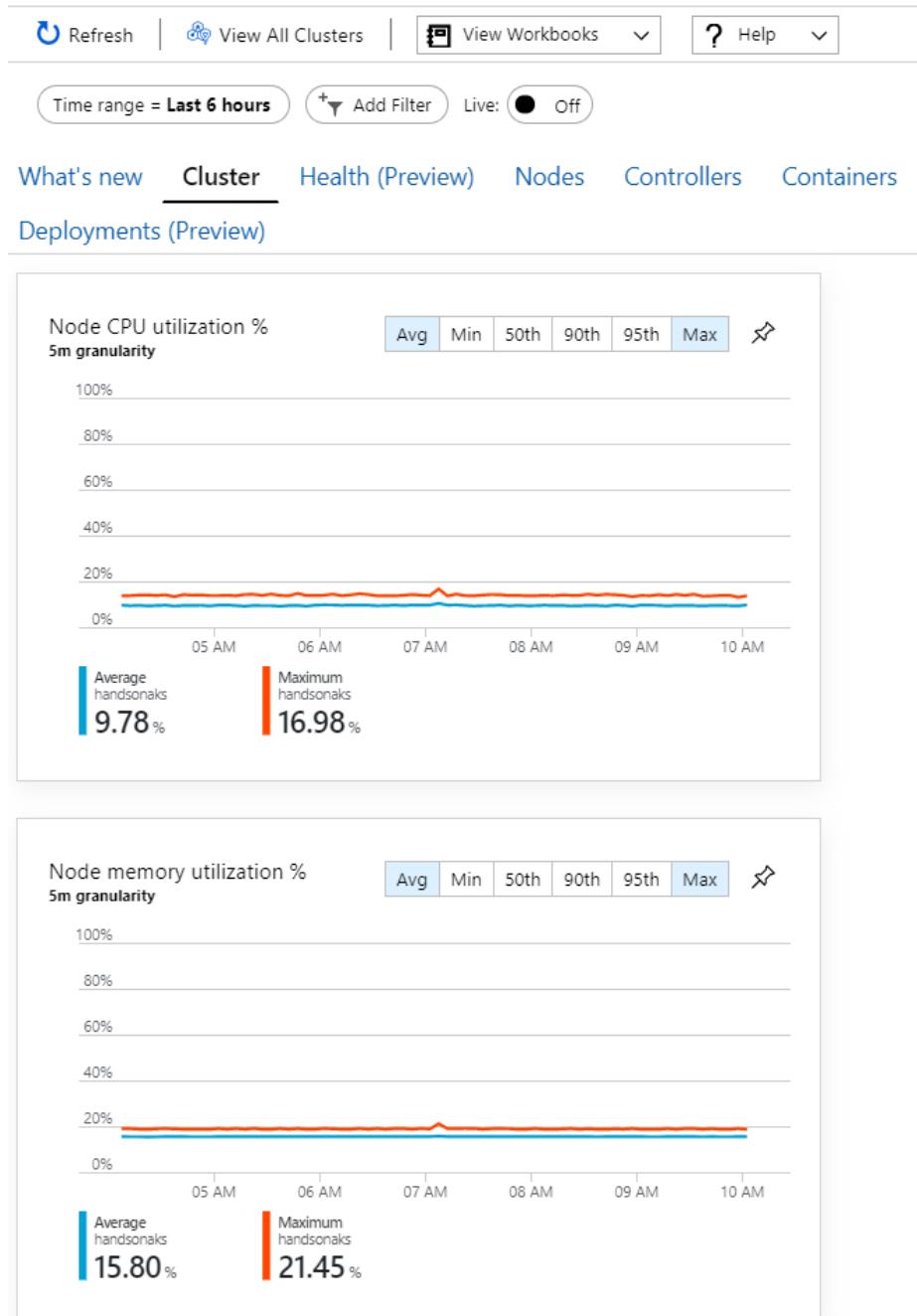


Figure 7.29: The Cluster tab shows CPU and memory utilization for the cluster

The cluster metrics also show the node count and the number of active Pods. The node count is very important, as you can track whether you have any nodes that are in a **Not Ready** state:



Figure 7.30: 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, we'll explore the **Health** tab.

Using the Health tab

The **Health** tab was in preview at the time of writing this book. This tab shows you a view of your cluster health. To show you this status, Azure monitors and checks the required infrastructure components as well as node health:

The screenshot shows the Azure Monitor interface with the 'Health (Preview)' tab selected. At the top, there are navigation links: Refresh, View All Clusters, View Workbooks, and Help. Below that are filter options: Time range = Last 6 hours and Add Filter. The main content area has tabs: What's new, Cluster, Health (Preview), Nodes, Controllers, Containers, and Deployments (Preview). The 'Health (Preview)' tab is active. On the left, under 'Current state', it says 'Healthy'. It also shows 'Last recalculated' as a minute ago on February 23, 2020, and 'Last state change' as 3 days ago on February 20, 2020. A table below lists 'HEALTH ASPECT' and 'STATE'. The first row, 'Kubernetes infrastructure', is highlighted in blue and labeled 'Healthy'. The second row, 'Nodes', is also labeled 'Healthy'. On the right, a detailed list of healthy components is shown, each preceded by a green checkmark icon:

- ▶ ✓ Kubernetes infrastructure
- ▶ ✓ Kubernetes API server
- ▶ ✓ coredns (ReplicaSet)
- ▶ ✓ coredns-autoscaler (ReplicaSet)
- ▶ ✓ kube-proxy (DaemonSet)
- ▶ ✓ kubernetes-dashboard (ReplicaSet)
- ▶ ✓ metrics-server (ReplicaSet)
- ▶ ✓ omsagent (DaemonSet)
- ▶ ✓ omsagent-rs (ReplicaSet)
- ▶ ✓ tunnelfront (ReplicaSet)

Figure 7.31: The Health tab shows the overall health of the cluster

The **Health** tab is useful for tracking the overall health of your cluster. The next tab we'll explore 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 we can see in *Figure 7.32*:

The screenshot shows the 'Nodes' tab selected in the top navigation bar. A search bar and a metric selector ('CPU Usage (millicores)') are at the top. Below is a table with columns: NAME, STATUS, 95TH %, CONTAINERS, UPTIME, CONTROLLER, and TREND 95TH % (1 BAR = 15M). The table lists 14 items under the 'aks-agentpool-428286...' node. To the right, a detailed view for the first node is expanded, showing its configuration and event logs.

NAME	STATUS	95TH %	CONTAINERS	UPTIME	CONTROLLER	TREND 95TH % (1 BAR = 15M)
aks-agentpool-428286...	Ok	9%	95 mc	14	-	
Other Processes	-	0%	0 mc	-	-	
tunnelfront-8fcfc...	Ok	8%	79 mc	1	3 days	tunnelfront-8fcfc...
tunnel-front	Ok	8%	79 mc	1	3 days	tunnelfront-8fcfc...
omsagent-2wfz8	Ok	0.5%	5 mc	1	3 days	omsagent
omsagent	Ok	0.5%	5 mc	1	3 days	omsagent
kube-proxy-vv9ss	Ok	0.3%	3 mc	1	3 days	kube-proxy
kube-proxy	Ok	0.3%	3 mc	1	3 days	kube-proxy
coredns-698c77c...	Ok	0.3%	3 mc	1	3 days	coredns-698c77c5...
coredns	Ok	0.3%	3 mc	1	3 days	coredns-698c77c5...
coredns-698c77c...	Ok	0.3%	3 mc	1	3 days	coredns-698c77c5...
coredns	Ok	0.3%	3 mc	1	3 days	coredns-698c77c5...

Figure 7.32: Detailed metrics of the nodes in the Nodes view pane

If you want even more details, you can click through and get Kubernetes event logs from your nodes as well:

The expanded view for the 'aks-agentpool-428286...' node shows a summary card with a 'View live data (preview)' button. Below it are two dropdown menus: 'View in analytics' and 'View Kubernetes event logs'.

Figure 7.33: 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 our case, we can see that our nodes have been rebooted a couple of times:

The screenshot shows the Azure Log Analytics interface. At the top, it displays the workspace name: DefaultWorkspace-a4339399-5b72-463b-88f9-e67030102086-WUS2. Below the header, there's a search bar labeled "New Query 1*" and a "Run" button. To the right of the run button are links for "Sample queries", "Query explorer", and "Export". The main area shows a query script in the editor:

```
let startTime = datetime('2020-02-01T22:15:00.000Z');
let endTime = datetime('2020-02-11T04:28:17.889Z');
let EmptyKubeEvents_CLTable = datatable(TimeGenerated: datetime, Name_s: string, ObjectKind_s: string, KubeEvent_type: string)
| where TimeGenerated >= startTime and TimeGenerated < endTime
| project ObjectKind_s, ClusterId = ClusterId_s, Name = Name_s, KubeEvent_type =
let EmptyKubeEventsTable = datatable(TimeGenerated: datetime, Name: string, ObjectKind: string, KubeEvent_type: string)
let KubeEventsTable = union isfuzzy = true EmptyKubeEvents_CLTable, EmptyKubeEventsTable, KubeEvents
| where TimeGenerated >= startTime and TimeGenerated < endTime
| where ClusterId == '/subscriptions/a4339399-5b72-463b-88f9-e67030102086/resourceGroups/handsoneaks/providers/Microsoft.ContainerInstance/containerGroups/aks-agentpool/providers/Microsoft.ContainerInstance/containerPools/aks-agentpool'
| where ObjectKind ~ 'Node'
```

Below the query results, there are two tabs: "Table" and "Chart". The "Table" tab is selected, showing a list of log entries with columns: TimeGenerated [UTC], Name, ObjectKind, KubeEvent_type, Reason, and Message. The data shows three records of nodes being rebooted:

TimeGenerated [UTC]	Name	ObjectKind	KubeEvent_type	Reason	Message
18/2020, 3:57:17.000 AM	aks-agentpool-42828616-vmss000000	Node		Rebooted	Node aks-agentpool
14/2020, 10:10:06.000 PM	aks-agentpool-42828616-vmss000000	Node		Rebooted	Node aks-agentpool
12/2020, 7:56:00.000 PM	aks-agentpool-42828616-vmss000000	Node		Rebooted	Node aks-agentpool

Figure 7.34: Log Analytics showing the logs for the nodes

Controllers

The **Controllers** view shows you details on all the controllers (that is, ReplicaSet, DaemonSet, and so on) on your cluster and the Pods running in them. This shows you a controller-centric view of running containers. For instance, you can find the frontend ReplicaSet and see all the Pods and containers running in it, as shown in Figure 7.35:

NAME	STATUS
▶ 🏡 frontend-57d8c9fb45 (ReplicaSet)	3 ✓
▶ 🏡 frontend-57d8c9fb45-pk9hc	✓ Ok
▶ 🏡 php-redis	✓ Ok
▶ 🏡 frontend-57d8c9fb45-x52ff	✓ Ok
▶ 🏡 php-redis	✓ Ok
▶ 🏡 frontend-57d8c9fb45-w5ktp	✓ Ok
▶ 🏡 php-redis	✓ Ok

Figure 7.35: The Controller tab shows us all the containers running in a ReplicaSet

The next tab is the **Containers** tab, which will show us 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 7.36*:

The screenshot shows the Azure Monitor interface with the 'Containers' tab selected. At the top, there are navigation links: 'What's new', 'Cluster', 'Health (Preview)', 'Nodes', 'Controllers', 'Containers' (which is underlined), and 'Deployments (Preview)'. Below these are filter options: a search bar containing 'php', a dropdown for 'Metric' set to 'CPU Usage (millicores)', and buttons for 'Min', 'Avg', '50th', '90th', '95th', and 'Max'. The main area displays a table of container metrics. The columns are: NAME, STATUS, 95TH %, 95TH, POD, NODE, RESTARTS, and UPTIME. There are seven rows, each representing a 'php-redis' container. The first row is marked as 'Ok', while the others are 'Unknown' (Unk). The 'NODE' column shows they are all running on the 'aks-agentpool-428...' node. The 'RESTARTS' column shows 0 for all containers, and the 'UPTIME' column shows '36 mins' for the first container and question marks for the others.

NAME	STATUS	95TH %	95TH	POD	NODE	RESTARTS	UPTIME
php-redis	Ok	0%	0.1 mc	frontend-57d8c9fb...	aks-agentpool-428...	0	36 mins
php-redis	Unk	-	-	frontend-57d8c9fb...	aks-agentpool-428...	0	?
php-redis	Unk	-	-	frontend-57d8c9fb...	aks-agentpool-428...	0	?
php-redis	Unk	-	-	frontend-57d8c9fb...	aks-agentpool-428...	0	?
php-redis	Unk	-	-	frontend-57d8c9fb...	aks-agentpool-428...	0	?
php-redis	Unk	-	-	frontend-57d8c9fb...	aks-agentpool-428...	0	?
php-redis	Unk	-	-	frontend-57d8c9fb...	aks-agentpool-428...	0	?

Figure 7.36: The Containers tab shows us all the individual containers

Note

You might notice a couple of containers with an **Unknown** state. This is to be expected in our situation. Our time range in Azure Monitor is set to the last 6 hours, and in the past 6 hours, we have created and deleted a number of Pods. They no longer exist, but Azure Monitor knows of their existence and even kept logs for them.

We can get access to our container's logs from this view:

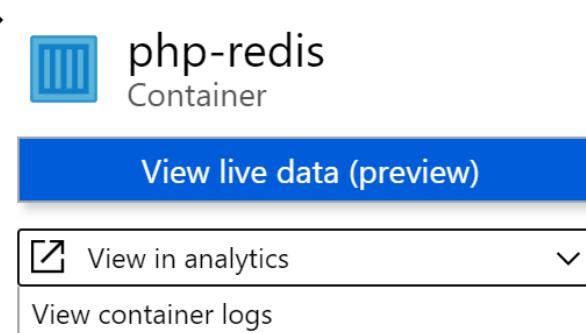


Figure 7.37: Access the container's logs

This shows us all the logs that Kubernetes logged from our application. We accessed these logs manually earlier in this chapter. Using this approach can be a lot more productive, as we can edit the log queries and correlate logs from different Pods and applications in a single view:

```

Logs
DefaultWorkspace-a4339399-5b72-483b-88f9-e67030102086-WUS2
New Query 1+ Run Time range : Set in query Save Copy link + New alert rule Export Pin to dashboard Prettify query
Tables Filter <> Search Group by: Solution Filters: not selected
Favorites You can add favorites by clicking on the star icon
ContainerInsights LogManagement
Completed
Table Chart Columns v
Drag a column header and drop it here to group by that column
TimeGenerated (UTC) LogEntrySource LogEntry Computer Image
> 2/11/2020, 4:24:54.557 AM stdout 10.244.1.1 - - [Tue/Feb/2020 04:24:54 +0000] "GET / HTTP/1.1" 400 0 "-" "-" aks-agentpool-42828616-vms0000001
> 2/11/2020, 3:56:35.104 AM stdout [Tue Feb 11 03:56:35.104274 2020] [mpm_preforknotice] [pid 1] AH00163: Apache/2.4.10 (Debian) PHP/5.6.20 config...
> 2/11/2020, 3:56:35.104 AM stdout AH00558: apache2: Could not reliably determine the server's fully qualified domain name, using 10.244.1.147. Set the ...
> 2/11/2020, 3:56:35.049 AM stdout AH00558: apache2: Could not reliably determine the server's fully qualified domain name, using 10.244.1.147. Set the ...
> 2/11/2020, 3:56:35.028 AM stderr AH00558: apache2: Could not reliably determine the server's fully qualified domain name, using 10.244.1.147. Set the ...

```

Figure 7.38: 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 this view:

» ▲ Environment Variables

Environment Variables	Value
GET_HOSTS_FROM	dns
KUBERNETES_PORT_443_TCP_PROTO	tcp
REDIS_SLAVE_SERVICE_PORT	6379
KUBERNETES_PORT	tcp://10.0.0.1:443
KUBERNETES_PORT_443_TCP	tcp://10.0.0.1:443
FRONTEND_SERVICE_HOST	10.0.102.101

Figure 7.39: The environment variables set for the container

And that concludes this section. Let's make sure to clean up our Deployments so we can continue with a clean guestbook in the next chapter:

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

In this section, we explored monitoring applications running on top of Kubernetes. We used the AKS **Insights** tab in the Azure portal to get a detailed view of our cluster and the containers running on our cluster.

Summary

We started the chapter by showing how to use different **kubectl** commands to monitor an application. Then, we showed how the logs Kubernetes creates can be used to debug that application. The logs contain all the information that is written to **stdout** and **stderr**. Lastly, we explained the use of Azure Monitor to show the AKS metrics and environment variables, as well as logs with log filtering. We also showed how to debug application and cluster issues by using **kubectl** and Azure Monitor monitoring.

In the next chapter, we will learn how to connect an AKS cluster to Azure PaaS services. We will specifically focus on how you can connect an AKS cluster to a MySQL database managed by Azure.

Section 3: Leveraging advanced Azure PaaS services

Up to this point in the book, we have run multiple applications on top of AKS. The applications were always self-contained, meaning the full application was able to be run in its entirety on top of AKS. There are certain advantages to 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 of 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, we will introduce 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, Event Hubs, and Azure Functions.

This section contains the following chapters:

- *Chapter 8, Connecting an app to an Azure database*
- *Chapter 9, Connecting to Azure Event Hubs*
- *Chapter 10, Securing your AKS cluster*
- *Chapter 11, Serverless Functions*

We will start this section with *Chapter 8, Connecting an app to an Azure database*, in which we will connect an application to a managed Azure database.

8

Connecting an app to an Azure database

In previous chapters, we stored the state of our application in our 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 versus running **StatefulSets** on Kubernetes itself. To create this hosted and managed database, we will make use of **Open Service Broker for Azure (OSBA)**. OSBA is a way to create Azure resources, such as a managed MySQL database, from within a Kubernetes cluster. In this chapter, we will explain more details about the OSBA project and we will set up and configure OSBA on our cluster.

We will then make use of OSBA to create a MySQL database in Azure. We 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.

In addition, we will show you aspects of security, backup, [**disaster recovery \(DR\)**](#), authorization, and audit logging. The independent scaling of the database and the cluster will also be explored. We will break down the discussion of this chapter into the following topics:

- Setting up OSBA
- Extending our app to connect to an Azure database
- Exploring advanced database operations
- Reviewing audit logs

Let's start by setting up OSBA on our cluster.

Setting up OSBA

In this section, we will set up OSBA on our cluster. OSBA will allow us to create a MySQL database without leaving the Kubernetes cluster. We will start this section by explaining the benefits of using a hosted database versus running StatefulSets on Kubernetes itself.

The benefits of using a managed database service

All the examples that we have gone through so far have been self-contained, that is, everything ran inside the Kubernetes cluster. Almost any production application has 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, disaster recovery, 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 and can optionally very easily be configured for disaster recovery.

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, we will explore a way that Kubernetes can be used to create these databases on Azure.

What is OSBA?

In this section, we will explore what OSBA is.

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. To aid in this effort, they have released Helm charts that use these managed services as a backend (<https://github.com/Azure/helm-charts>).

A key part of the architecture that allows you to create Azure resources from within Kubernetes is OSBA (<https://osba.sh/>). OSBA is an **Open Service Broker (OSB)** implementation for Azure. The OSB API is a spec that defines a common language for platform providers that cloud-native applications can use to manage cloud services without lock-in.

The OSB API itself isn't Azure or Kubernetes specific. It is an industry effort to simplify the provisioning of resources through a standardized API. It allows you to connect to third-party services in a standardized way.

When using the OSB API with Kubernetes, an extension called the **Service Catalog** is run on the cluster. The Service Catalog will listen to the Kubernetes API for requests and will translate them to the OSB API to interface with the platform provider. This means that when you make a request for a database, the Kubernetes API will send that request to the Service Catalog, which in turn will use the OSB API to interface with the platform. Figure 8.1 illustrates this logical flow:

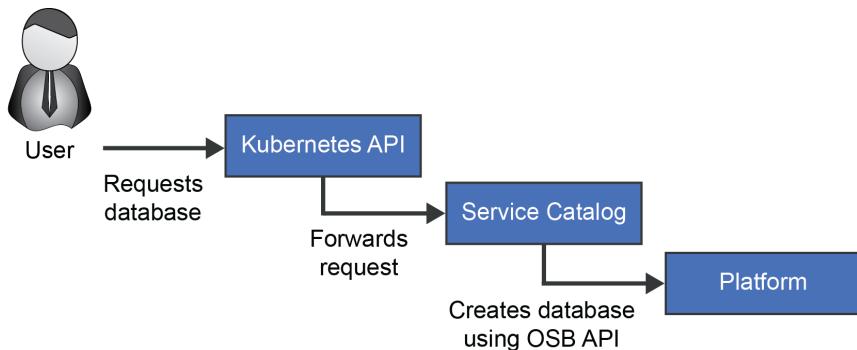


Figure 8.1: Logical flow of requesting a database using OSB on a Kubernetes cluster

OSBA is an implementation of OSB for multiple Azure services. It allows a user to use the OSB API to create any of 14 supported Azure services. One of those services is Azure Database for MySQL. This means that you can define a MySQL database on Azure through OSBA without having to use the Azure portal.

In the next section, we will focus on how to install OSBA on our cluster.

Installing OSBA on the cluster

We will install OSBA on our cluster. There are two elements to this installation. First, we will install the Service Catalog extension on our cluster. After that, we can install OSBA on the cluster.

As we will be installing multiple components on our cluster, our two-node cluster won't suffice for this example. Let's proactively scale our AKS cluster to three nodes so we don't run into any issues during this example:

```
az aks scale -n <clustername> -g <cluster resource group> -c 3
```

This scaling will take a couple of minutes. When the cluster is scaled out to three nodes, we can start with deploying the Service Catalog on the cluster.

Deploying the Service Catalog on the cluster

The Service Catalog provides the catalog servers that are required for the OSB. To deploy the Service Catalog on the cluster, follow these steps:

1. Let's deploy the Service Catalog by running the following commands:

```
kubectl create namespace catalog
helm repo add svc-cat https://svc-catalog-charts.storage.googleapis.com
helm install catalog svc-cat/catalog --namespace catalog
```

2. Wait until the Service Catalog is deployed. You can check this by running the following command:

```
kubectl get all -n catalog
```

3. Verify that both Pods in the deployment are **Running** and fully ready:

```
user@Azure:~$ kubectl get all -n catalog
NAME                                         READY   STATUS    RESTARTS   AGE
pod/catalog-catalog-apiserver-7c47dfdddb-r5zmf   2/2     Running   0          3m1s
pod/catalog-catalog-controller-manager-855bfbdfc4-sxjjb   1/1     Running   0          3m1s

NAME                           TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)           AGE
service/catalog-catalog-apiserver   NodePort    10.0.123.56   <none>        443:30443/TCP   3m1s
service/catalog-catalog-controller-manager   ClusterIP  10.0.65.73   <none>        443/TCP         3m1s

NAME                               READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/catalog-catalog-apiserver   1/1     1           1           3m1s
deployment.apps/catalog-catalog-controller-manager   1/1     1           1           3m1s

NAME                               DESIRED   CURRENT   READY   AGE
replicaset.apps/catalog-catalog-apiserver-7c47dfdddb   1         1         1         3m1s
replicaset.apps/catalog-catalog-controller-manager-855bfbdfc4   1         1         1         3m1s
```

Figure 8.2: Successful Service Catalog deployment

- To interface with the service broker, we need to install another CLI tool, namely **svcat**. We can do that with the following command:

```
curl -sL0 https://download.svcat.sh/cli/latest/linux/amd64/svcat
chmod +x ./svcat
./svcat version --client
```

We now have a Service Catalog configured on top of our cluster. Now, we can move on and install OSBA on the cluster.

Deploying OSBA

In this section, we will deploy the actual OSBA on our cluster. For this setup, we need to obtain the subscription ID, tenant ID, client ID, and secrets for OSBA to launch Azure services on our behalf:

- Run the following command to obtain the required lists:

```
az account list
```

The output will be as shown in *Figure 8.3*:

```
user@Azure:~$ az account list
[{"id": "a4339399-5b72-463b-88f9-e67030102086", "name": "Azure subscription 1", "state": "Enabled", "tenantId": "c3d6c0ea-0aa0-4e2d-9726-070be5ba5127", "user": {"cloudShellID": true, "name": "live.com#handsonaksdemo@outlook.com", "type": "user"}}
```

Figure 8.3: Output displaying the required list – subscription ID and the tenant ID

- Copy your **subscription ID** along with the **tenant ID** and save it in an environment variable:

```
export AZURE_SUBSCRIPTION_ID=<SubscriptionId>
export AZURE_TENANT_ID=<Tenant>"
```

3. Create a service principal with RBAC enabled so that it can launch Azure services. If you are sharing your subscription with somebody else, make sure that the name of the service principal is unique in your directory:

```
az ad sp create-for-rbac --name osba-quickstart -o table
```

This will generate an output as shown in *Figure 8.4*:

AppId	DisplayName	Name	Password	Tenant
a2ae431e-d09a-4c5c-bcfc-5429e244c512	osba-quickstart	http://osba-quickstart	[REDACTED]	c3d6c0ea-0aa0-4e2d-9726-070be5ba5127

Figure 8.4: Output displaying service principal credentials

Note

For the previous step to complete successfully, you need to have the owner role on your Azure subscription.

4. Save the values from the command output in the environment variable:

```
export AZURE_CLIENT_ID=<AppId>
export AZURE_CLIENT_SECRET=<Password>
```

5. Now, we can deploy OSBA as follows:

```
kubectl create namespace osba
helm repo add azure https://kubernetescharts.blob.core.windows.net/azure
helm install osba azure/open-service-broker-azure \
--namespace osba \
--set azure.subscriptionId=$AZURE_SUBSCRIPTION_ID \
--set azure.tenantId=$AZURE_TENANT_ID \
--set azure.clientId=$AZURE_CLIENT_ID \
--set azure.clientSecret=$AZURE_CLIENT_SECRET
```

To verify that everything got deployed correctly, you can run the following command:

```
kubectl get all -n osba
```

Wait until both Pods are in the **Running** state. If one of the Pods is in the **Error** state, you don't have to be concerned. The OSBA Pods will automatically restart and should reach a healthy state. In our case, one of the Pods restarted three times, as shown in Figure 8.5:

user@Azure:~\$ kubectl get all -n osba					
NAME		READY	STATUS	RESTARTS	AGE
pod/osba-open-service-broker-azure-84bfbb4567-xl88s		1/1	Running	3	92s
pod/osba-redis-98fd45846-h8vh		1/1	Running	0	92s
NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
service/osba-open-service-broker-azure	ClusterIP	10.0.29.71	<none>	443/TCP	92s
service/osba-redis	ClusterIP	10.0.79.224	<none>	6379/TCP	92s
NAME		READY	UP-TO-DATE	AVAILABLE	AGE
deployment.apps/osba-open-service-broker-azure		1/1	1	1	92s
deployment.apps/osba-redis		1/1	1	1	92s
NAME		DESIRED	CURRENT	READY	AGE
replicaset.apps/osba-open-service-broker-azure-84bfbb4567		1	1	1	92s
replicaset.apps/osba-redis-98fd45846		1	1	1	92s

Figure 8.5: Output displaying OSBA Pods in the Running status

- To verify that our deployment was completely successful, we can use the **svcat** utility we downloaded in the previous section:

```
./svcat get brokers
```

This should show you your Azure broker:

user@Azure:~\$./svcat get brokers					
NAME	NAMESPACE	URL	STATUS		
osba		https://osba-open-service-broker-azure.osba.svc.cluster.local	Ready		

Figure 8.6: Output displaying the Azure broker running in the cluster

- You can additionally also verify all the services you can deploy via the OSBA driver:

```
./svcat get classes
```

This will show you a list of services that can be created using OSBA, as shown in Figure 8.7:

NAME	NAMESPACE	DESCRIPTION
azure-rediscache		Azure Redis Cache (Preview)
azure-storage-blob-storage-account		Specialized Azure storage account for storing block blobs and append blobs
azure-postgresql-10-database		Azure Database for PostgreSQL 10-- database only
azure-postgresql-9-6-database		Azure Database for PostgreSQL 9.6-- database only
azure-sql-12-0-database		Azure SQL 12.0-- database only
azure-mysql-5-7-dbms		Azure Database for MySQL 5.7-- DBMS only
azure-postgresql-10		Azure Database for PostgreSQL 10-- DBMS and single database
azure-cosmosdb-table-account		Azure Cosmos DB Database Account (Table API)

Figure 8.7: A (cropped) list of the services that can be created using OSBA

In this section, we set up the Service Catalog and OSBA on our cluster. This means we can now create managed services by Azure from our cluster. We will use this capability in the next section, when we deploy WordPress using an Azure-managed database.

Deploying WordPress

The following are the steps to deploy WordPress:

1. Run the following command to install WordPress:

```
kubectl create ns wordpress
helm install wp azure/wordpress --namespace wordpress --set replicaCount=1
--set externalDatabase.azure.location=<your Azure region>
```

2. To verify the status of the WordPress Pod, run the following command:

```
kubectl get pods -n wordpress
```

This should show the status of a single WordPress Pod as displayed in Figure 8.8. In our previous WordPress examples, we always had two Pods running, but we were able to offload the database functionality to Azure here:

NAME	READY	STATUS	RESTARTS	AGE
wp-wordpress-6d4dc574f-478ff	0/1	ContainerCreating	0	12s

Figure 8.8: Output displaying only one WordPress Pod and no database on our cluster

- While the WordPress Pod is being created, we can check on the status of the database as well. We can use two tools to get this status, either **svcat** or **kubectl**:

```
./svcat get instances -n wordpress
```

This will generate the output shown in *Figure 8.9*:

NAME	NAMESPACE	CLASS	PLAN	STATUS
wp-wordpress-mysql-instance	wordpress	azure-mysql-5-7	general-purpose	Provisioning

Figure 8.9: Output displaying the use of svcat to get our MySQL instance

We can get a similar result by using **kubectl**:

```
kubectl get serviceinstances -n wordpress
```

This will generate an output as shown in *Figure 8.10*:

NAME	CLASS	PLAN	STATUS	AGE
wp-wordpress-mysql-instance	ClusterServiceClass/azure-mysql-5-7	general-purpose	Provisioning	3m13s

Figure 8.10: Output displaying the use of kubectl to get our MySQL instance

As you can see, the outputs of each method are similar.

- Give the deployment a couple of minutes to complete. First, the database needs to be fully provisioned, and afterward, the WordPress Pod needs to enter the **Running** state. To verify everything is running correctly, check the status of the WordPress Pod and ensure it is **Running**:

```
kubectl get pods -n wordpress
```

This will generate an output as shown in *Figure 8.11*:

NAME	READY	STATUS	RESTARTS	AGE
wp-wordpress-6d4dc574f-478ff	1/1	Running	0	75m

Figure 8.11: Output displaying the status of WordPress Pod

We have now deployed WordPress using an Azure-managed database. However, the connectivity to our database, by default, is open to the internet. We will change this in the following section.

Securing MySQL

Although many steps are automated for us, this doesn't mean our MySQL database is production-ready. For instance, the network settings for MySQL Server have a default rule that allows traffic from everywhere. We will change this to a more secure service endpoint rule, also called **VNet rules**.

A service endpoint in Azure is a security connection between the network (also called a VNet) you use for your deployment and the service it connects to. In the case of AKS and MySQL, this would make a secure connection between the VNet that AKS is deployed into and the MySQL service.

We will configure our MySQL database to use a service endpoint in this section:

1. To make this change, look for **mysql** in the Azure search bar:

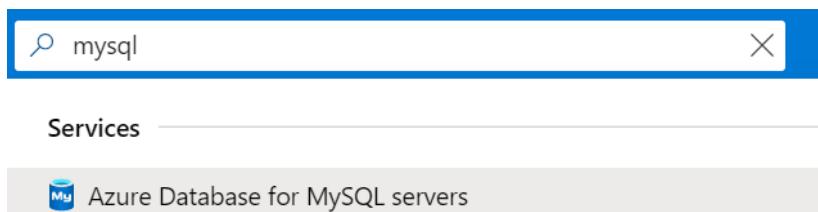


Figure 8.12: Searching for MySQL in the Azure search bar

2. In the MySQL blade, go to **Connection security** in the left-hand navigation:

The screenshot shows the 'Connection security' blade for an Azure MySQL server. The left sidebar has a 'Connection security' option selected. The main area displays the 'Firewall rules' section, which includes a note about public-facing IP addresses and a toggle switch for 'Allow access to Azure services' set to 'ON'. Below this is a table for 'VNET rules' with one entry: 'AllowAll' with 'Start IP address' 0.0.0.0 and 'End IP address' 255.255.255.255. There are also buttons for '+ Adding existing virtual network' and '+ Create new virtual network'.

Figure 8.13: Clicking on Connection security

3. There is a default rule that allows a connection to the database from any IP address. You can add the AKS VNet to the **VNet rules** section and delete the **AllowAll 0.0.0.0** rule, as shown in *Figure 8.14*:

The screenshot shows the 'Connection security' page for an Azure Database for MySQL server named 'Oddaa0c1-a07b-4ec1-82b4-94ecb3714494'. The left sidebar includes links for Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Settings (selected), Connection security (selected), Connection strings, Server parameters, Replication, Active Directory admin, and Pricing tier. The main content area has tabs for Save, Discard, and Add client IP. Under 'Firewall rules', there is a note about public-facing IP addresses and a switch for 'Allow access to Azure services' which is set to OFF. A table lists a single rule named 'AllowAll' with 'Start IP address' 0.0.0.0 and 'End IP address' 0.0.0.0. This row has a 'Delete' button highlighted with a dashed blue border. Below this is a 'VNet rules' section with a table:

VNet rules		+ Adding existing virtual network		+ Create new virtual network			
Rule name	Virtual n...	Subnet	Address ...	Endpoint...	Resource ...	Subscription ID	St...
AKSOnly	aks-vnet...	aks-su...	10.240.0....	Enabled	MC_hands...	a4339399-5b72-...	Re... ***

Figure 8.14: Adding your AKS VNet to the VNet rules section and deleting the AllowAll rule

We have reduced the attack surface tremendously by performing this simple change. We can now connect to our WordPress site.

Connecting to the WordPress site

You can verify that your blog site is available and running by using **EXTERNAL_IP**, which is obtained by running the following command:

```
kubectl get service -n wordpress
```

This will generate an output as shown *Figure 8.15*:

user@Azure:~\$ kubectl get service -n wordpress			
NAME	TYPE	CLUSTER-IP	EXTERNAL-IP
wp-wordpress	LoadBalancer	10.0.243.122	52.250.73.173

Figure 8.15: Output displaying the external IP of the service

Then, open a web browser and go to http://<EXTERNAL_IP>/. You should see your brand new blog:

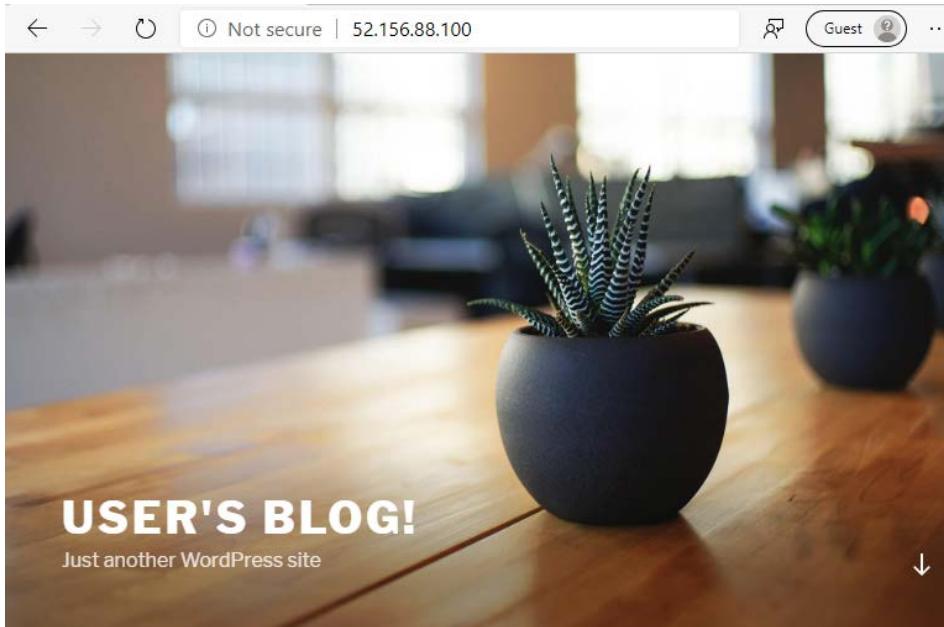


Figure 8.16: The final look of the WordPress blog

In this section, we have launched a WordPress site that is backed by an Azure-managed database. We have also secured it by modifying the firewall. In the next section, we will go through the advantages of letting Azure manage your database.

Exploring advanced database operations

Running your database as a managed service on top of Azure has many advantages. In this section, we'll explore those benefits. We will explore restoring from a backup, how you can set up disaster recovery, and how you can access audit logs to verify who made changes to your database.

We'll begin by restoring our database from a backup.

Restoring from a backup

When you run a database within your Kubernetes cluster, **high availability (HA)**, backups, and DR are your responsibilities. Let's take some time to explain the differences between those three concepts:

- **HA:** HA refers to local redundancy in a service to ensure that the service remains available in the event that a single component fails. This means setting up multiple replicas of a service and coordinating the state between them. In a database context, this means setting up a database cluster.

The Azure Database for MySQL service comes with HA built in. It offers, at the time of writing, an availability SLA of 99.99% per month.

- **Backups:** Backups refer to making historical copies of your data. Backups are useful when something unforeseen happens to your data, such as accidental data deletion or data being overwritten. If you run your own database, you need to set up `cron` jobs to take backups and store them separately.

Azure Database for MySQL handles backups automatically, without additional configuration. The service takes a backup every 5 minutes and makes it possible for you to restore to any point in time. Backups are retained by default for 7 days, with optional configuration making it possible to keep backups for up to 25 days.

- **DR:** DR refers to the ability of a system to recover from a disaster. This typically refers to the ability to recover from a full regional outage. If you run your own database, this would involve setting up a secondary database in a second region and replicating data to that database.

In the case of Azure Database for MySQL, it is easy to configure DR. The service makes it possible to set up a secondary managed database and replicate data from your primary region to the secondary region.

Note

You can refer to <https://docs.microsoft.com/azure/mysql/concepts-backup> to find up-to-date information on the backup frequency, replication, and restore options.

The terms HA, backup, and DR often get confused with each other. It is important to use the right terminology and for you to understand the difference between the three concepts. In this section, we'll focus on backups, and we'll perform a restore from our WordPress database. To demonstrate that the restore operation will restore user data, we will first create a blog post.

Creating a blog post on WordPress

We will create a blog post to demonstrate that the restore operation will capture new data that we generate on the database. To be able to make this post, we need the admin credentials for our site. We will first get those credentials and then make a new post:

1. To get the admin credentials, use the following command:

```
echo Password: $(kubectl get secret --namespace wordpress \
    wp-wordpress -o jsonpath=".data.wordpress-password" | \
    base64 --decode)
```

This will show you the password to connect to your admin website:

```
user@Azure:~$ echo Password: $(kubectl get secret --namespace wordpress \
>    wp-wordpress -o jsonpath=".data.wordpress-password" | \
>    base64 --decode)
```

```
Password: 9uuETWYErz
```

Figure 8.17: Getting the admin credentials

2. Now browse to `http://<EXTERNAL IP>/admin` to open the admin page for the WordPress site. Use the username `user` and the password from the previous step to log in.
3. Once connected, select the **Write your first blog post** link:

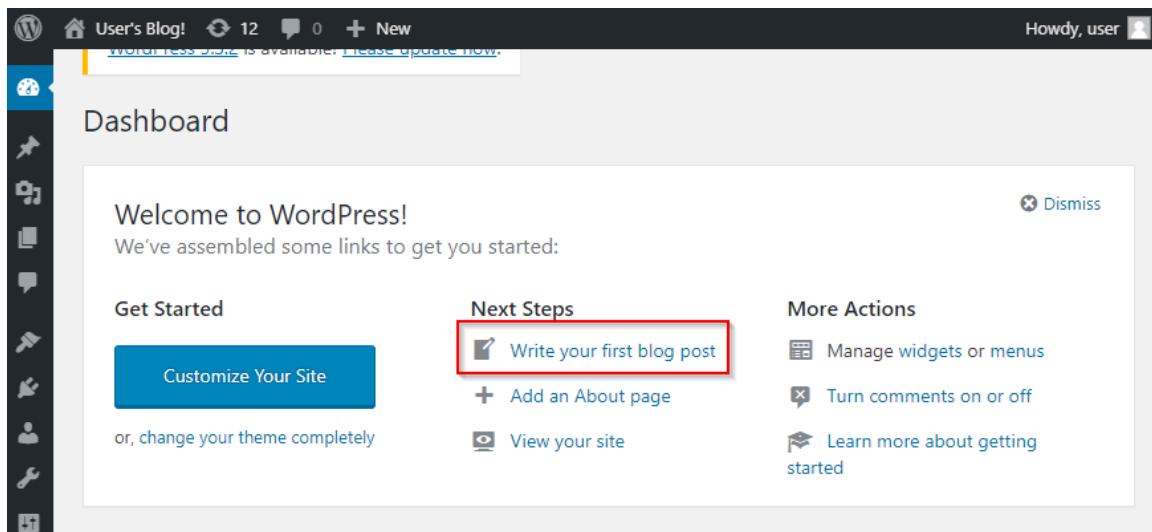


Figure 8.18: Clicking the link to write a post

4. Create a blog post. The content isn't important. Once you are happy with your blog post, select the **Publish** button to save and publish the blog post:

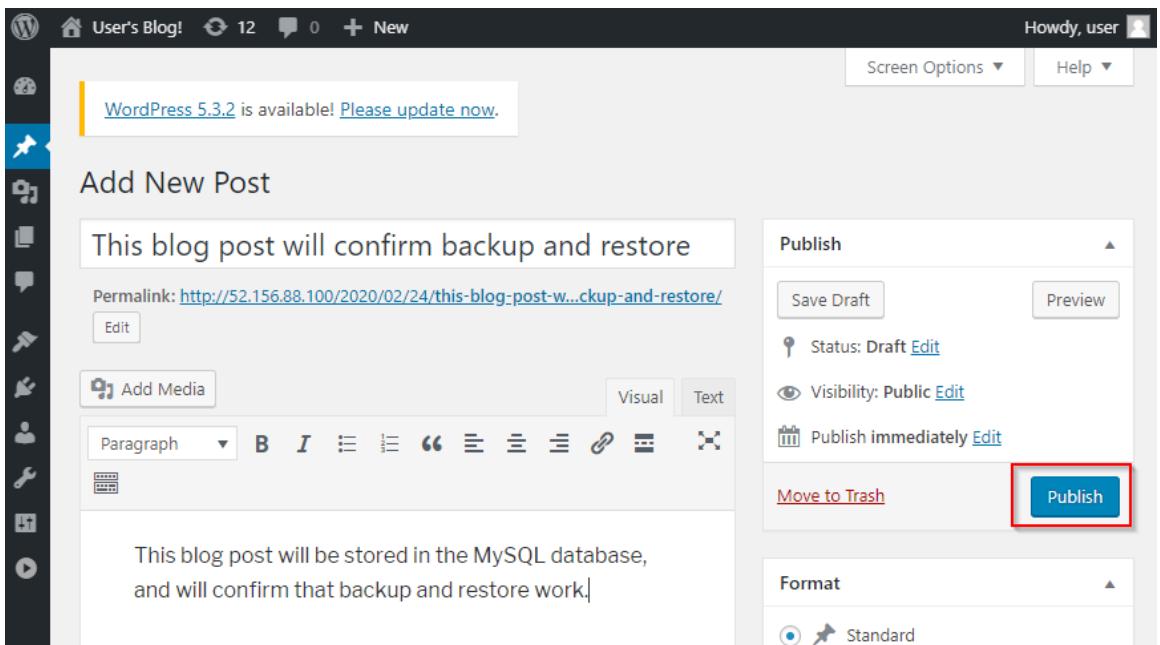


Figure 8.19: Creating a sample blog post and clicking the Publish button to save it

5. You can now connect to `http://<EXTERNAL IP>` to see your blog post:

The screenshot shows a published blog post titled 'POSTS' on 'FEBRUARY 24, 2020'. The content of the post is 'This blog post will confirm backup and restore'. Below the post, a note states: 'This blog post will be stored in the MySQL database, and will confirm that backup and restore work.' The entire note is displayed in a monospace font.

Figure 8.20: Note displaying the successful status of the blog post

Now that we have a blog post saved, please wait for at least 5 minutes. Azure takes a backup of the MySQL database every 5 minutes, and we want to make sure that our new data has been backed up. Once those 5 minutes have passed, we can continue to the next step and perform the actual restore.

Performing a restore

We now have actual content on our blog and in the database. Let's assume that during an update, the database was corrupted, and so we want to do a point-in-time restore:

1. To start the restore operation, click on **Restore** in the MySQL blade in the Azure portal:

The screenshot shows the Azure MySQL server blade for the database 'Oddaa0c1-a07b-4ec1-82b4-94ecb3714494'. The 'Restore' button in the top navigation bar is highlighted with a red box. The main pane displays various configuration details such as resource group, status, location, subscription, and tags.

Setting	Value
Resource group (change)	wordpress
Status	Available
Location	West US 2
Subscription (change)	Azure subscription 1
Subscription ID	a433999-5b72-463b-88f9-e67030102086
Tags (change)	heritage : open-service-broker-azure

Figure 8.21: Clicking the Restore button to initiate the restore process

2. Then, you need to choose the point in time from which you want to perform the restore. This point in time can be the current time. Give the restored database a name, which must be unique as shown in *Figure 8.22*. Finally, click **OK**. After approximately 5 to 10 minutes, the MySQL service should be restored:

The screenshot shows the 'Restore' dialog box for the MySQL server 'Oddaa0c1-a07b-4ec1-82b4-94ecb3714494'. The 'Restore point (UTC)' is set to '02/12/2020 2:44:49 AM'. The 'Restore to new server' field contains 'mysql-wordpress-restored' and has a green checkmark. Other fields include 'Location' (US) West US 2 and 'Pricing tier' General Purpose (2 vCores, 10 GB storage).

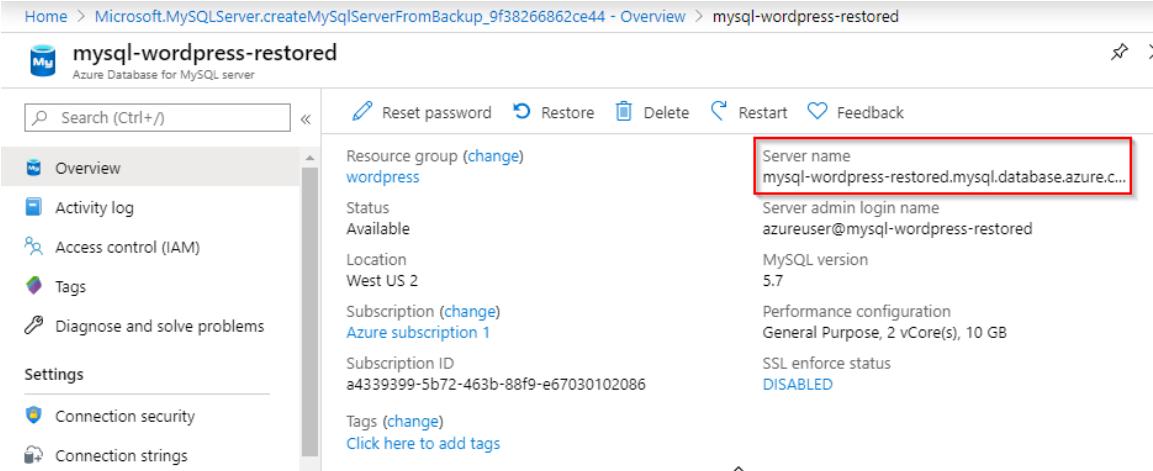
Figure 8.22: Selecting the point in time to restore and clicking the OK button

In this section, we restored our MySQL database. When the restore operation is complete, one step remains, which is to connect WordPress to the restored database.

Connecting WordPress to the restored database

The restore operation created a new instance of the database. To make our WordPress installation connect to the restored database, we need to modify the Kubernetes deployment files. Ideally, you will modify the Helm values file and perform a Helm upgrade; however, that is beyond the scope of this book. The following steps will help you connect WordPress to the restored database:

1. From the Azure portal, note down the **Server name**, as shown in Figure 8.23:



The screenshot shows the Azure portal's "Overview" page for a MySQL server named "mysql-wordpress-restored". The left sidebar lists navigation options like Overview, Activity log, Access control (IAM), Tags, and Settings. Under Settings, there are sections for Connection security and Connection strings. The main content area displays various server details. A red box highlights the "Server name" field, which contains the value "mysql-wordpress-restored.mysql.database.azure.c...". Other visible details include the Resource group ("wordpress"), Status ("Available"), Location ("West US 2"), Subscription ("Azure subscription 1"), Subscription ID ("a4339399-5b72-463b-88f9-e67030102086"), and Tags ("Click here to add tags").

Figure 8.23: Displaying the full name of the restored database

2. Also, modify **Connection security**, as we did before, to allow the cluster to talk to the restored database. Remove the Allow All rule and add a VNet rule to the network of your AKS cluster. The result is shown in *Figure 8.24*:

The screenshot shows the 'Connection security' blade for the 'mysql-wordpress-restored' database. On the left, there's a sidebar with navigation links like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Settings, Connection security (which is selected and highlighted in grey), Connection strings, Server parameters, Replication, Active Directory admin, and Pricing tier. The main content area has tabs for Save, Discard, and Add client IP. Under 'Firewall rules', it says 'Allow access to Azure services' is OFF. There's a table for firewall rules with columns for Rule name, Start IP address, and End IP address. One row is listed: Rule name 'AKSOnly', Start IP address '10.240.0.0', and End IP address '10.240.0.255'. Below this table, it says 'No firewall rules configured.' Under 'VNET rules', there's a table with columns for Rule name, Virtual n..., Subnet, Address ..., Endpoint..., Resource ..., Subscription ID, and St... One row is listed: Rule name 'AKSOnly', Virtual n... 'aks-vnet...', Subnet 'aks-su...', Address ... '10.240.0....', Endpoint... 'Enabled', Resource ... 'MC_hands...', Subscription ID 'a4339399-5b72...', and St... '***'.

Figure 8.24: Editing the connection security for the restored database

3. Next, we need to connect our WordPress Pod to the new database. Let's point out how that happens. To get that info, run the following command:

```
kubectl describe deploy wp -n wordpress
```

You can see that the values to connect to the database are obtained from a secret, as shown in *Figure 8.25*:

```
Environment:
MARIADB_HOST: <set to the key 'host' in secret 'wp-wordpress-mysql-secret'> Optional: false
MARIADB_PORT_NUMBER: <set to the key 'port' in secret 'wp-wordpress-mysql-secret'> Optional: false
WORDPRESS_DATABASE_NAME: <set to the key 'database' in secret 'wp-wordpress-mysql-secret'> Optional: false
WORDPRESS_DATABASE_USER: <set to the key 'username' in secret 'wp-wordpress-mysql-secret'> Optional: false
WORDPRESS_DATABASE_PASSWORD: <set to the key 'password' in secret 'wp-wordpress-mysql-secret'> Optional: false
```

Figure 8.25: Displaying the environment variables for the WordPress Pod

During the setup of WordPress, the installer will save this configuration in a file: `/bitname/wordpress/wp-config.php`. In the next steps, we'll first edit the secret and then reconfigure `wp-config.php`.

4. To set secrets, we need the **base64** value. Obtain the **base64** value of the server name by running the following command:

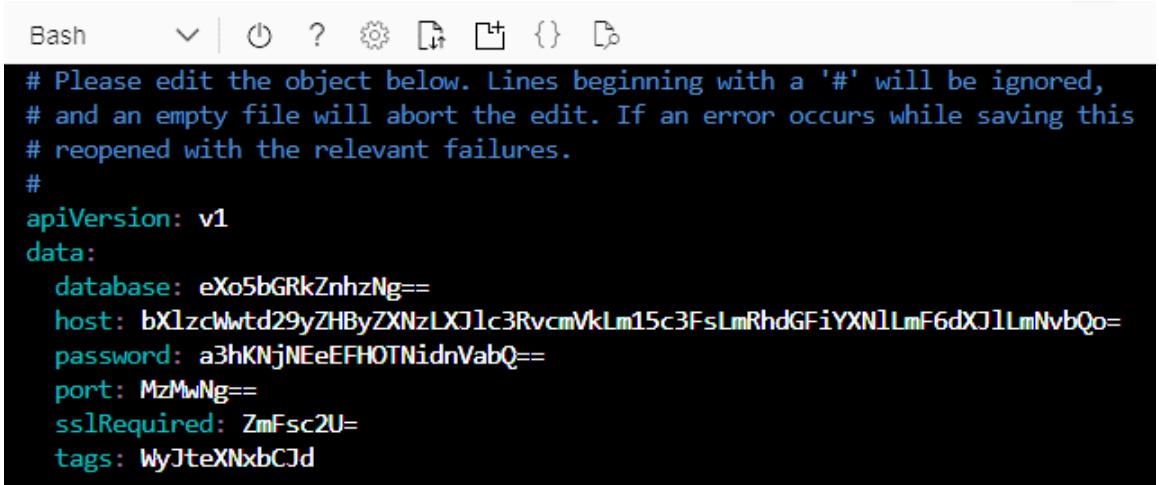
```
echo <restored db server name> | base64
```

Note the **base64** value.

5. Now, we'll go ahead and edit the hostname in the secret. To do this, we'll use the **edit** command:

```
kubectl edit secret wp-wordpress-mysql-secret -n wordpress
```

This will open up a **vi** editor. Navigate to the line that contains **host** and press the **I** button. Delete the current value for the host and paste in the new **base64** encoded value. Then hit Esc, type **:wq!**, and hit Enter. Your secret should look as shown in *Figure 8.26*:



```
Bash      ▾ | ⌁ ? ⚙ ⌂ ⌂ {} ⌂

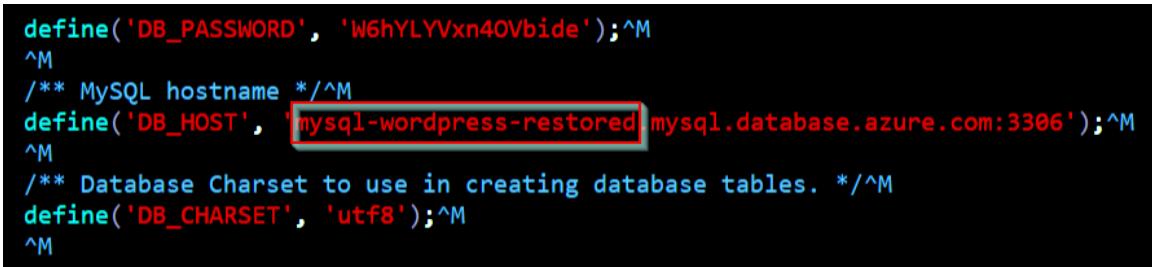
# Please edit the object below. Lines beginning with a '#' will be ignored,
# and an empty file will abort the edit. If an error occurs while saving this
# reopened with the relevant failures.
#
apiVersion: v1
data:
  database: eXo5bGRkZnhzNg==
  host: bXlzcWtd29yZHByZXNzLXJlc3RvcmlkLm15c3FsLmRhGFiYXN1LmF6dXJ1LmNvbQo=
  password: a3hKNjNEeEFHOTNidnVabQ==
  port: MzMwNg==
  sslRequired: ZmFsc2U=
  tags: WyJteXNxbCJd
```

Figure 8.26: Editing the host line to contain the base64 encoded value of the new server name

6. Next, we'll need to change this in the `wp-config.php` file as well. To do this, let's `exec` into the current WordPress container and change that value:

```
kubectl exec -it <wordpress pod name> -n wordpress sh  
apt update  
apt install vim -y  
vim /bitnami/wordpress/wp-config.php
```

This will again open a `vi` editor. Navigate to line 32, which contains the `DB_HOST` config line. Hit `I` to enter insert mode, and delete the current value and replace that with the restored database's name as shown in *Figure 8.27*. Then hit `Esc`, type `:wq!`, and hit `Enter`. Make sure to paste in the real value, not the `base64` encoded one:



```
define('DB_PASSWORD', 'W6hYLYVxn40Vbide');^M  
^M  
/** MySQL hostname */^M  
define('DB_HOST', 'mysql-wordpress-restored.mysql.database.azure.com:3306');^M  
^M  
/** Database Charset to use in creating database tables. */^M  
define('DB_CHARSET', 'utf8');^M  
^M
```

Figure 8.27: Changing the name of the database to the restored database

Then, exit out of the Pod with the following command:

```
exit
```

Even though we have now reset the secret value and the config file, this doesn't mean that our server will automatically pick up the new value. We'll have to restart our Pod now to ensure the config is read in again.

7. There are many ways to do this, and we are going to delete the existing Pod. Once this Pod is deleted, our `ReplicaSet` controller will pick up on this and create a new Pod. To delete the Pod, use the following command:

```
kubectl delete pod <wordpress pod name> -n wordpress
```

8. After a couple of seconds, you should see a new Pod being created. It will take between 5 and 10 minutes for that new Pod to come online. Once it is online, you can watch the container logs from that Pod and verify that you are indeed connected to the new database:

```
kubectl logs <new wordpress pod> -n wordpress
```

This should contain a line as shown in Figure 8.28:

```
wordpre INFO WordPress has been already initialized, restoring...
mysql-c INFO Trying to connect to MySQL server
mysql-c INFO Found MySQL server listening at mysql-wordpress-restored.mysql.database.azure.com:3306
mysql-c INFO MySQL server listening and working at mysql-wordpress-restored.mysql.database.azure.com:3306
wordpre INFO Upgrading WordPress Database ...
```

Figure 8.28: Logs displaying the WordPress Pod connected to the restored database

This shows us that we are now connected to our restored database. We can confirm that the actual content was restored. You can connect to the WordPress site itself by browsing to <http://<EXTERNAL IP>>:

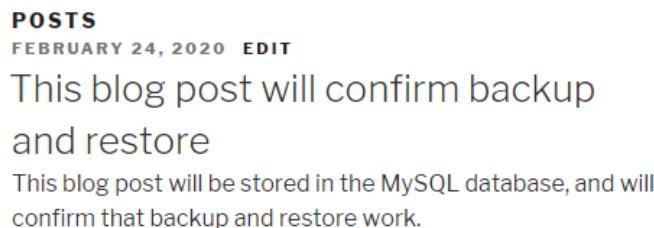


Figure 8.29: Note displaying the successful restoration of the blog post

In this section, we explored the backup and restore capabilities of Azure Database for MySQL. We published a blog post and restored the database where this blog post was stored. We connected our WordPress instance to the restored database and were able to verify that the blog post was successfully restored.

Performing backups is just one of the capabilities of Azure Database for MySQL. In the next section, we will explore the DR capabilities of the service.

Disaster Recovery (DR) options

Depending on your application requirements and DR needs, you can add replicas to your MySQL server. Replicas can be created either in the same region to improve read performance or in a secondary region.

If you are preparing for a DR scenario, you will need to set up a replica in a secondary region. This will protect you from regional Azure outages. When you set this up, Azure will asynchronously replicate data from the master server to the replica server you set up. While replication is ongoing, the replica server can be used to read from, but cannot be used to write to. If a disaster occurs, meaning an Azure region has a regional outage, you would need to stop replication to turn the replica server into a server capable of serving both read and write requests.

It is very straightforward to create a replica in a new region. Although setting up and testing replication is out of the scope of this book, we will show how this can be set up. To configure replication, you need to open the **Replication** tab in the MySQL blade, as shown in *Figure 8.30*:

The screenshot shows the Azure portal interface for managing a MySQL server. On the left, there's a navigation sidebar with links like Home, Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Settings, Connection security, Connection strings, Server parameters, and Replication. The Replication link is highlighted. The main content area is titled '1efaf0ac-10ca-49d9-bdb2-74ec4f7af240 - Replication' and 'Azure Database for MySQL server'. It features a toolbar with 'Add Replica', 'Delete Replica', and 'Stop Replication' buttons. Below this, there are two sections: 'Master' and 'Replicas'. The Master section shows one entry: '1efaf0ac-10ca-49d9-bd...' with a 'General Purpose, 2 vCore(s), 10 GB' pricing tier. The Replicas section shows 'No results'. To the right, a modal window titled 'MySQL server' is open, showing configuration details: 'Server name' is 'mysql-replica-eastus2', 'Location' is '(US) East US 2', and 'Pricing tier' is 'General Purpose, 2 vCore(s), 10 GB'. The monthly cost is listed as '157.77 USD'.

Figure 8.30: Creating a replica via the Azure portal

Note

A full list of backup, restore, and replication options are documented at <https://docs.microsoft.com/azure/mysql/concepts-backup> and <https://docs.microsoft.com/azure/mysql/concepts-read-replicas>.

In this section, we described the capability of Azure Database for MySQL to replicate to a secondary region. This replica can be used to build a DR strategy for your database. In the next section, we will cover how the activity log can be used to audit who has made changes to your server.

Reviewing audit logs

A database contains business-critical data. You will want to have a logging system in place that can show you who has made changes to your database.

When you run the database on the Kubernetes cluster, it is difficult to get audit logs if something goes wrong. You need a robust way of dynamically setting the audit level depending on the scenario. You also have to ensure that the logs are shipped outside the cluster.

The Azure Database for MySQL service solves the preceding issues by providing a robust auditing mechanism via the Azure portal. The service has two different views into logs:

- **Activity logs:** The activity logs show you all the changes that happened to the Azure object of your database. Azure records all create, update, and delete transactions against Azure resources, and keeps those logs for 90 days. In the case of MySQL, this means all changes to the size, to the backup and replication settings, and so on. These logs are useful to determine who made changes to your database.
- **Server logs:** The server logs include logs from the actual data in your database. MySQL has multiple logs available that can be configured. It is typically recommended to turn on audit logging to verify who has accessed your database and to turn on slow query monitoring to track any queries that are running slowly.

Let's have a look at both logs:

1. To access the activity logs, open the MySQL database blade in the Azure portal. In the left-hand navigation, look for **Activity log**. This will open the activity log view, as in Figure 8.31:

The screenshot shows the Azure Activity Log interface for a MySQL server. The left sidebar includes links for Overview, Activity log (which is selected), Access control (IAM), Tags, Diagnose and solve problems, Settings (Connection security, Connection strings, Server parameters, Replication, Active Directory admin, Pricing tier, Properties, Locks, Export template), and Logs. The main area displays a table of 20 items with columns: Operation name, Status, Time, Time stamp, Subscription, and Event initiated by. Most entries are for 'Update MySQL Server Create' with various status codes like Succeeded, Started, Accepted, or Pending.

Operation name	Status	Time	Time stamp	Subscription	Event initiated by
Update MySQL Server Create	Succeeded	7 min ago	Tue Feb 11 ...	Azure subscription 1	osba-quickstart
Update MySQL Server Create	Started	18 min ago	Tue Feb 11 ...	Azure subscription 1	osba-quickstart
Update MySQL Server Create	Started	18 min ago	Tue Feb 11 ...	Azure subscription 1	osba-quickstart
Update MySQL Server Create	Accepted	17 min ago	Tue Feb 11 ...	Azure subscription 1	osba-quickstart
Update MySQL Server Create	Accepted	17 min ago	Tue Feb 11 ...	Azure subscription 1	osba-quickstart
Update MySQL Server Create	Succeeded	11 min ago	Tue Feb 11 ...	Azure subscription 1	osba-quickstart
Update MySQL Server Create	Succeeded	11 min ago	Tue Feb 11 ...	Azure subscription 1	osba-quickstart
Update Firewall Rule Create	Started	10 min ago	Tue Feb 11 ...	Azure subscription 1	osba-quickstart
Update Firewall Rule Create	Started	10 min ago	Tue Feb 11 ...	Azure subscription 1	osba-quickstart
Update Firewall Rule Create	Accepted	10 min ago	Tue Feb 11 ...	Azure subscription 1	osba-quickstart

Figure 8.31: Azure activity logs showing the actions taken against the Azure database

The Activity log provides very valuable information in retracing the activities that have been performed. You should find events in the activity log that point to the changes you made earlier to the connection security settings.

2. Server logs can be obtained by looking for **Server logs** in the left-hand navigation. Server logs aren't turned on by default, as can be seen in Figure 8.32:

The screenshot shows the Azure Server logs interface for a MySQL server. The left sidebar includes links for Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Settings (Connection security, Connection strings, Server parameters, Replication, Active Directory admin, Pricing tier, Properties, Locks, Export template), and Logs (which is selected). The main area features a search bar with 'server', a button to enable logs, a search for log files, a dropdown menu, and a section indicating 'No server log found'.

Figure 8.32: Navigation displaying no sever logs by default

3. Let's turn on the server logs. We will enable the audit log and monitoring for performance issues by enabling the `log_slow...` statements and `slow_query_log`, as shown in *Figure 8.33*:

Parameter name	Value
audit_log_enabled	ON
audit_log_events	CONNECTION
audit_log_exclude_users	azure_superuser
log_bin_trust_function_creators	OFF
log_output	FILE
log_slow_admin_statements	ON
log_slow_slave_statements	ON
long_query_time	10
slow_query_log	ON

Figure 8.33: Enabling the audit log and slow query logging

Once you have turned on these logs, it will take a couple of minutes for the actual logs to show up. After a couple of minutes, you should see the logs in the **Server logs** tab in the Azure portal as shown in *Figure 8.34*:

Name
mysql-slow-888b1ec7-766f-41a6-80ba-743fba88f37b.log

Figure 8.34: Displaying the Server logs in the Azure portal

Let's make sure to again clean up after our deployment and scale our cluster back down to two nodes. Scaling down to two nodes will make sure you save costs on your Azure subscription:

```
helm delete wp -n wordpress  
helm delete osba -n osba  
helm delete catalog -n catalog  
kubectl delete ns wordpress osba catalog  
az aks scale -n <clustername> -g <cluster resource group> -c 2
```

In this section, we covered the two types of logs that Azure generates for a MySQL database. We looked into the activity log to see which actions have been taken against the Azure database, and we turned on server logs to get insights into what happened inside the database.

Summary

This chapter focused on working with a WordPress sample solution that leverages a MySQL database as a data store. We started by showing you how to set up the cluster to connect the MySQL database by installing Open Service Broker for Azure. We then showed you how to set up a MySQL database and drastically minimize the attack surface by changing the default configuration to not allow public access to the database. Then, we discussed how to restore the database from a backup and how to leverage the solution for DR. Finally, we discussed how to configure the audit logs for troubleshooting.

In the next chapter, you will learn how to implement microservices on AKS, including by using Event Hubs for loosely coupled integration between applications.

9

Connecting to Azure Event Hubs

Event-based integration is a key pattern for implementing **microservices**. The idea of a microservices architecture is to decompose a monolithic application into a smaller set of services. Events are commonly used to coordinate between these different services. When you think about an event, it can be one of many things. Financial transactions can be an event, as well as IoT sensor data, web page clicks and views, and much more.

A piece of software that is commonly used to handle these types of events is Apache Kafka (Kafka for short). **Kafka** was originally developed by LinkedIn, and later donated to the Apache Software Foundation. It is a popular open-source streaming platform. A streaming platform is a platform that has three core capabilities: publishing and subscribing a stream of messages (similar to a queue), storing these streams in a durable fashion, and processing these streams as they occur.

Azure has a similar offer to Apache Kafka, called Azure Event Hubs. **Event Hubs** is a managed service that offers real-time data ingestion. It is simple to set up and use, and can scale dynamically. Event hub is also integrated with other Azure services, such as stream analytics, functions, and databricks. These prebuilt integrations make it easier for you to build applications that consume events from event hubs.

Event hubs also provide a Kafka endpoint. This means that you can configure your existing Kafka-based applications and point them to Event Hubs instead. The benefit of using Event Hubs for your Kafka applications is that you no longer have to manage your Kafka cluster because you consume it as a managed service.

In this chapter, you will learn how to implement microservices on AKS and use Event Hubs for loosely coupled integration between applications. You will deploy an application that uses Kafka to send events, and you will replace your own Kafka cluster with Azure Event Hubs. As you will learn in this chapter, event-based integration is one of the key differentiators between monolithic and microservice-based applications.

- Deploying a set of microservices
- Using Azure Event Hubs

We will start this chapter by deploying a set of microservices that builds a social network.

Deploying a set of microservices

In this section, we will be deploying a set of microservices from a demo application called social network. The application is composed of two main microservices: **users** and **friends**. The users service stores all the users in its own data store. A user is represented by an ID, and their first and last names. The friends service stores the user's friends. A friend relationship links the user IDs of both friends, and also has its own ID.

The events of adding a user/adding a friend are sent to a message queue. This application uses Kafka as the message queue to store events related to users, friends, and recommendations.

This queue is consumed by a recommendation service. This service is backed by a **Neo4j** database that can then be used to query relationships between users. Neo4j is a popular graph database platform. A graph database is different from a typical relational database such as MySQL. A graph database is a database that is focused on storing the relationship between different elements. You can query a graph database with questions, such as *give me the common friends of user X and user Y*.

In terms of data flow, you can create users and friendship relationships. Creating a user or a friendship relationship will generate a message on the message queue, which will result in the data being populated in the Neo4j database. The application doesn't have a web interface. You will mainly work with the application using the command line, although we can connect to the Neo4j database to verify that data was populated in the database.

In the following section, you will learn to do the following:

- Use Helm to deploy a sample microservice-based application.
- Test the service by sending events and watch objects being created and updated.

Let's start by deploying the application.

Deploying the application using Helm

In this section, we will deploy the demo application using Helm. This will deploy the full application using a local Kafka instance. Once the application is deployed, we will generate a small social network and verify that we were able to create the social network.

1. This example has a lot of resource requirements. To meet them, scale your cluster to four nodes:

```
az aks nodepool scale --node-count 4 -g rg-handsonaks \
--cluster-name handsonaks --name agentpool
```

2. The code of this example has been included in the GitHub repo for this book. You can find the code in the **social-network** folder under **Chapter09**. Navigate to this folder:

```
cd Chapter09/social-network
```

3. To run Kafka, we also need to run **ZooKeeper**. ZooKeeper is another open-source software project by the Apache foundation. It provides naming, configuration management, synchronization, and the ability to group services. We will use Kafka and ZooKeeper Helm charts from **bitnami**, so let's add the required Helm repository:

```
helm repo add bitnami https://charts.bitnami.com
helm repo add incubator https://kubernetes-charts-incubator.storage.googleapis.com
```

This will generate the output shown in *Figure 9.1*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter09/social-network$  
"bitnami" has been added to your repositories  
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter09/social-network$  
"incubator" has been added to your repositories
```

Figure 9.1: Adding the Helm repositories

4. Let's update the dependencies to make the dependent charts available:

```
helm dep update deployment/helm/social-network  
helm dep update deployment/helm/friend-service  
helm dep update deployment/helm/user-service  
helm dep update deployment/helm/recommendation-service
```

This will show you something similar to *Figure 9.2* four times:

```
Hang tight while we grab the latest from your chart repositories...  
...Successfully got an update from the "svc-cat" chart repository  
...Successfully got an update from the "incubator" chart repository  
...Successfully got an update from the "azure" chart repository  
...Successfully got an update from the "jetstack" chart repository  
...Successfully got an update from the "bitnami" chart repository  
...Successfully got an update from the "stable" chart repository  
Update Complete. Happy Helming!✿  
Saving 1 charts  
Downloading neo4j from repo https://kubernetes-charts.storage.googleapis.com/  
Deleting outdated charts
```

Figure 9.2: Updating dependencies

Note

During this example, you might see a warning similar to the following: **walk.go:74: found symbolic link in path:**. This is a warning that can safely be ignored.

5. Next, create a new **namespace** for this application:

```
kubectl create namespace social-network
```

This will generate an output as follows:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter09/social-network
$ kubectl create namespace social-network
namespace/social-network created
```

Figure 9.3: Creating a new namespace

- Now, go ahead and deploy the application:

```
helm install social-network --namespace social-network \
--set fullNameOverride=social-network \
--set edge-service.service.type=LoadBalancer \
deployment/helm/social-network
```

- Check the status of the Pods in the deployment using the following command:

```
kubectl get pods -w -n social-network
```

As you can see in *Figure 9.4*, it takes about 5 minutes until all of the Pods are up and running:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter09/social-network$ kubectl get pods -n social-network
NAME                                         READY   STATUS    RESTARTS   AGE
edge-service-6d5df96cc-z4n6l                  1/1     Running   0          8m4s
friend-db-0                                    1/1     Running   0          8m4s
friend-service-58b8899dc9-9sxqh                1/1     Running   0          8m4s
kafka-0                                         1/1     Running   1          8m4s
kafka-1                                         1/1     Running   0          6m26s
kafka-2                                         1/1     Running   0          5m48s
recommendation-service-5d58967485-tt9jm        1/1     Running   0          8m4s
social-network-grafana-67cc6fd496-fgdth        1/1     Running   0          8m4s
social-network-neo4j-core-0                     1/1     Running   0          8m4s
social-network-prometheus-kube-state-metrics-6fcdf54df4-v6gfh 1/1     Running   0          8m4s
social-network-prometheus-node-exporter-hqfgm   1/1     Running   0          8m4s
social-network-prometheus-node-exporter-kppg5   1/1     Running   0          8m4s
social-network-prometheus-node-exporter-tbprx   1/1     Running   0          8m4s
social-network-prometheus-node-exporter-xxfp4   1/1     Running   0          8m4s
social-network-prometheus-server-75b59cf698-cj2pw 2/2     Running   0          8m4s
social-network-zookeeper-0                      1/1     Running   0          8m4s
social-network-zookeeper-1                      1/1     Running   0          7m34s
social-network-zookeeper-2                      1/1     Running   0          7m1s
user-db-0                                       1/1     Running   0          8m4s
user-service-645ddfb488-4klsb                   1/1     Running   0          8m4s
```

Figure 9.4: Output displaying all of the Pods with a Running status

- When the application has been successfully deployed, you can connect to the edge service. To get its IP, use the following command:

```
kubectl get service -n social-network
```

This command will generate an output as follows:

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)
edge-service	LoadBalancer	10.0.34.235	52.148.151.233	9000:30321/TCP

Figure 9.5: Getting the edge service external IP

- You can do two tests to verify that the application is working correctly. Test 1 is to connect the edge service on port **9000** in a browser. Figure 9.6 displays the Whitelabel Error Page, showing that the application is running:



Figure 9.6: The Whitelabel Error Page, showing that the application is running

- The second test to verify that the application is running, is to actually generate a small social network. This will verify that all services are working correctly. You can create this network using the following command:

```
bash ./deployment/sbin/generate-serial.sh <external-ip>:9000
```

This command will generate a lot of output. The output will start with the elements shown in *Figure 9.7*:

```
Using edge-service URI: 52.148.167.86:9000
--> Wake up user service... "UP"
--> Wake up friend service... "UP"
--> Wake up recommendation service... "UP"
===== Create users
{
  "firstName": "Andrew",
  "lastName": "Rutherford",
  "createdAt": "2020-02-25T04:22:02.473+0000",
  "lastModified": "2020-02-25T04:22:02.473+0000",
  "id": 1
}
```

Figure 9.7: Initial output when creating a new social network

11. It will take about a minute to generate a 15-person network. To verify that the network has been created successfully, browse to `http://<external-ip>:9000/user/v1/users/1` in your web browser. This should show you a small JSON object that represents a user in the social network, as shown in *Figure 9.8*:

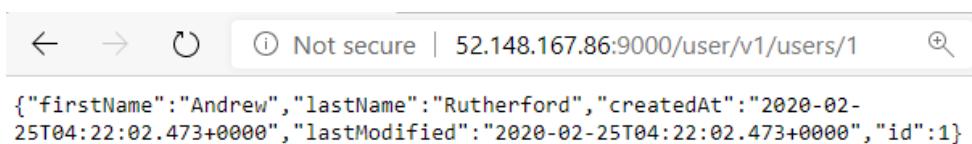


Figure 9.8: Successful creation of the user in the user service

12. As a final validation, you can connect to the Neo4j database and visualize the social network you created. To be able to connect to Neo4j, you need to first expose it as a service. Use the `neo4j-service.yaml` file in the social network folder to expose it:

```
kubectl create -f neo4j-service.yaml -n social-network
```

Then, get the service's public IP address. This can take about a minute to be available:

```
kubectl get service neo4j-service -n social-network
```

The preceding command will generate the output as follows:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter09/social-network$ kubectl get service neo4j-service -n social-network
NAME          TYPE        CLUSTER-IP   EXTERNAL-IP      PORT(S)        AGE
neo4j-service  LoadBalancer  10.0.25.62  52.148.167.132  7474:32085/TCP,7687:31597/TCP  58s
```

Figure 9.9: Output displaying the external IP address of the Neo4j service

Note

Please note that the external IP of the Neo4j service can differ from the external IP of the edge service.

13. Use your browser to connect to `http://<external-ip>:7474`. This will open a login screen. Use the following information to log in:

- **Connect URL:** `bolt://<external-ip>:7687`
- **Username:** `neo4j`
- **Password:** `neo4j`

Your connection information should look similar to *Figure 9.10.*:

```
$ :server connect
```

**Connect to
Neo4j**
Database access requires
an authenticated
connection.

Connect URL

Username

Password

Connect

Figure 9.10: Logging in to the Neo4j browser

14. Once you're connected to the Neo4j browser, you can see the actual social network. Click on the **Database Information** icon, and then click on **User**. This will generate a query that will display the social network you just created. This will look similar to Figure 9.11:

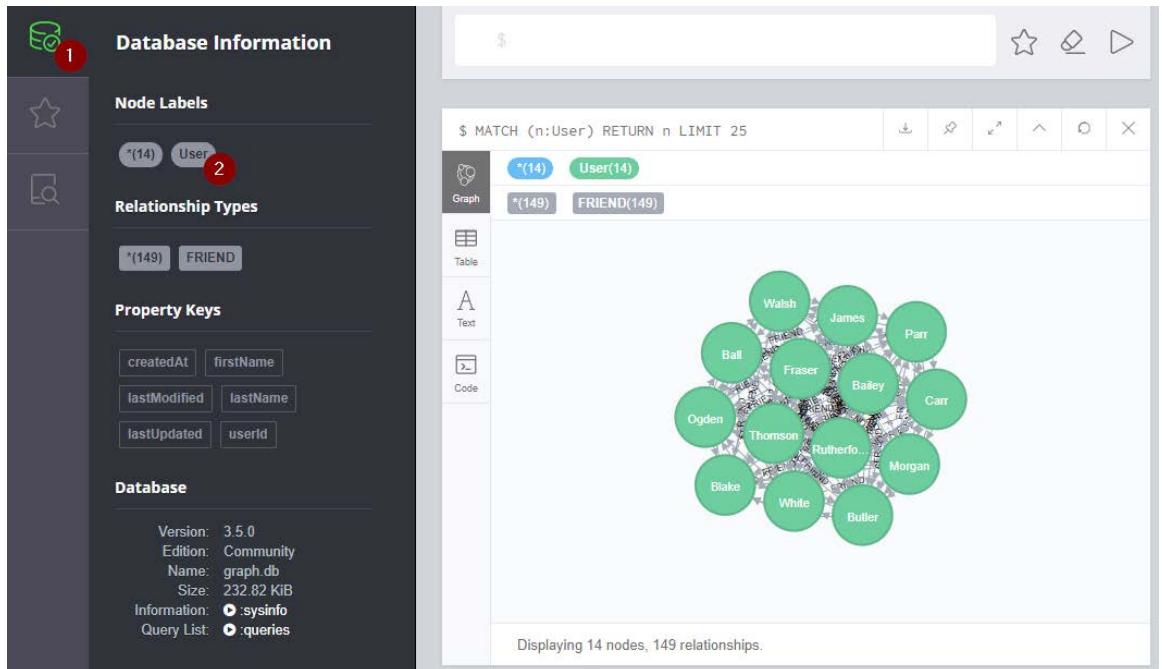


Figure 9.11: A view of the social network you just created

In the current example, we have set up the end-to-end application, using Kafka running on our Kubernetes cluster as a message queue. Let's remove that example before we move on the next section. To delete the local deployment, use the following commands:

```
helm delete social-network -n social-network
kubectl delete pvc -n social-network --all
kubectl delete pv --all
```

In the next section, we will move away from storing events in the cluster and store them in Azure Event Hubs. By leveraging native Kafka support on Azure Event Hubs and switching to using a more production-ready event store, we will see that the process is straightforward.

Using Azure Event Hubs

Running Kafka by yourself on a cluster is possible but can be hard to run for production usage. In this section, we will transfer the responsibility of maintaining a Kafka cluster to Azure Event Hubs. Event Hubs is a fully managed, real-time data ingestion service. It has native support for the Kafka protocol, so, with minor modifications, we can update our application from using a local Kafka instance to the scalable Azure Event Hubs instance.

In the following sections, we will do the following:

- Create the event hub via the portal and gather the required details to connect our microservice-based application
- Modify the Helm chart to use the newly created event hub

Let's start by creating the event hub.

Creating the event hub

In this section, we will create the Azure event hub. We will use this event hub later to stream the new messages to. Perform the following steps to create the event hub:

1. To create the event hub on the Azure portal, search for **event hub**, as shown in *Figure 9.12*:

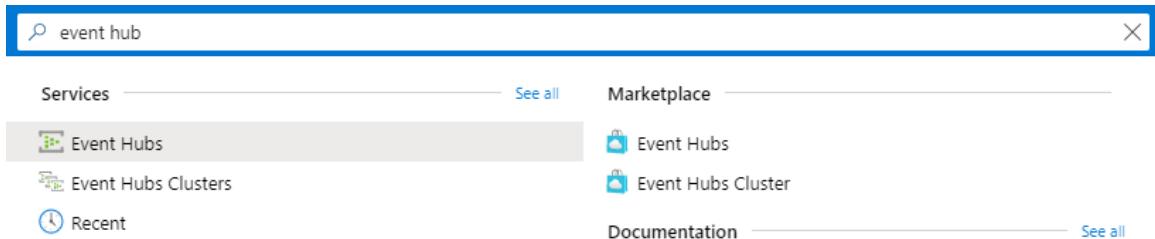


Figure 9.12: Looking for Event Hubs in the search bar

2. Click on **Event Hubs**.

3. On the **Event Hubs** tab, click on **Add**, as displayed in *Figure 9.13*:

The screenshot shows the Azure Event Hubs dashboard. At the top, there's a breadcrumb navigation: Dashboard > Event Hubs. Below that is the title 'Event Hubs' with the Microsoft logo. At the bottom of the header, there are three buttons: '+ Add' (which is highlighted with a red box), 'Edit columns', and 'Refresh'. The main area below the header is currently empty, showing a light gray background.

Figure 9.13: Adding a new event hub

4. Fill in the following details:

- **Name:** This name should be globally unique. Consider adding your initials to the name.
- **Pricing tier:** Select the Standard pricing tier. The Basic pricing tier does not support Kafka.
- **Makes this namespace zone redundant:** Disabled.
- **Azure Subscription:** Select the same subscription as the one hosting your Kubernetes cluster.
- **Resource Group:** Select the resource group we created for the cluster, `rg-hands-on-aks` in our case.
- **Location:** Select the same location as your cluster. In our case, this is `West US 2`.
- **Throughput Unit:** 1 unit will suffice for this test.
- **Auto-inflate:** Disabled.

This should give you the create view similar to *Figure 9.14*:

 **Create Namespace**

Name *****

Pricing tier ([View full pricing details](#)) *****

Enable Kafka (i)

Make this namespace zone redundant (i)

Subscription *****

Resource group *****
 [Create new](#)

Location *****

Throughput Units *****

Enable Auto-Inflate (i)

Figure 9.14: Your event hub creation should look like this

5. Hit the **Create** button at the bottom of the wizard to create your event hub.
6. Once the event hub is created, select it, as shown in *Figure 9.15*:

The screenshot shows the 'Event Hubs' blade in the Azure portal. At the top, there's a breadcrumb navigation: Home > Event Hubs. Below that is a title 'Event Hubs' and a 'Default Directory'. A toolbar with buttons for 'Add', 'Edit columns', 'Refresh', 'Assign tags', and 'Delete' is visible. The main area is titled 'Subscriptions: Azure subscription 1'. It contains three dropdown filters: 'Filter by name...', 'All resource groups', and 'All locations'. Below these, a message says '1 of 1 items selected'. A table lists one item:

<input type="checkbox"/> Name ↑↓	Status	Tier	Location ↑↓
<input type="checkbox"/> handsonakseh	✓ Active	Standard	West US 2

Figure 9.15: Clicking on the event hub name once it is created

7. Click on **Shared access policies**, select **RootManageSharedAccessKey**, and copy the **Connection string-primary key**, as shown in *Figure 9.16*:

The screenshot shows the 'Shared access policies' blade for the 'handsonakseh' event hub. On the left, a sidebar has 'Shared access policies' highlighted with a red circle labeled '1'. The main area shows a table of shared access policies:

Policy	
RootManageSharedAccessKey 2	<input checked="" type="checkbox"/> Manage <input type="checkbox"/> Send <input type="checkbox"/> Listen

To the right, a detailed view for the 'RootManageSharedAccessKey' policy is shown:

- Primary key:** 3Xpu5e8CXZ2YTJn9dpMjXTlpY8Mw3qKmx1c+86T9... (with a copy icon)
- Secondary key:** nGL+FDNNWZXgGpgQdRPvCB4Da+zFN9LrYaQ+z... (with a copy icon)
- Connection string-primary key:** Endpoint=sb://handsonakseh.servicebus.windows.n... (with a copy icon) 3

Figure 9.16: Copying the primary connection string

Using the Azure portal, we have created an event hub that can store and process our events as they are generated. We needed to gather the connection strings so that we can hook up our microservice-based application. In the next section, we will redeploy our social network and configure it to connect to Event Hubs. To be able to deploy our social network, we will have to make a couple of changes to the Helm charts, in order to point our application to Event Hubs rather than to Kafka.

Modifying the Helm files

We are going to switch the microservice deployment from using the local Kafka instance to using the Azure-hosted, Kafka-compatible Event Hubs instance. To make this change, we will modify the Helm charts to use Event Hubs rather than Kafka:

1. Modify the **values.yaml** file for the social network deployment to disable Kafka in the cluster and include the connection details to your event hub:

```
code deployment/helm/social-network/values.yaml
```

Make sure to change the following values:

Lines 5, 18, 26, and 34: Change this to **enabled: false**.

Lines 20, 28, and 36: Change this to your event hub's name.

Lines 21, 29, and 37: Change this to your event hub's connection string:

```
1  nameOverride: social-network
2  fullNameOverride: social-network
3
4  kafka:
5    enabled: true
6    nameOverride: kafka
7    fullnameOverride: kafka
8    persistence:
9      enabled: false
10   resources:
11     requests:
12       memory: 325Mi
13
14
15 friend-service:
```

```
16   fullnameOverride: friend-service
17   kafka:
18     enabled: true
19   eventhub:
20     name: <event hub name>
21     connection: "<event hub connection string>"  
22
23 recommendation-service:
24   fullnameOverride: recommendation-service
25   kafka:
26     enabled: true
27   eventhub:
28     name: <event hub name>
29     connection: "<event hub connection string>"  
30
31 user-service:
32   fullnameOverride: user-service
33   kafka:
34     enabled: true
35   eventhub:
36     name: <event hub name>
37     connection: "<event hub connection string>"
```

Note

For our demo, we are storing the connection string in a Helm values file. This is not a best practice. For a production use case, you should store those values as a secret and reference them in your deployment. We will explore this in *Chapter 10, Securing your AKS cluster*.

2. Run the deployment as follows:

```
helm install social-network deployment/helm/social-network/ -n social-
network --set edge-service.service.type=LoadBalancer
```

- Wait for all the Pods to be up. You can verify that all Pods are up and running with the following command:

```
kubectl get pods -n social-network
```

This will generate the following output:

NAME	READY	STATUS	RESTARTS	AGE
edge-service-6d5df96cc-gjhhn	1/1	Running	0	9m56s
friend-db-0	1/1	Running	0	9m56s
friend-service-6784f86b44-p2lfr	1/1	Running	0	9m56s
recommendation-service-6b4d4c8484-hpxv9	1/1	Running	0	9m56s
social-network-grafana-67c6cf496-9bbx6	1/1	Running	0	4m43s
social-network-neo4j-core-0	1/1	Running	0	9m56s
social-network-prometheus-kube-state-metrics-6fcfdf54df4-dpw2f	1/1	Running	0	9m56s
social-network-prometheus-node-exporter-64mr5	1/1	Running	0	9m57s
social-network-prometheus-node-exporter-g5q5f	1/1	Running	0	9m57s
social-network-prometheus-node-exporter-ww4w	1/1	Running	0	9m57s
social-network-prometheus-node-exporter-zgk7s	1/1	Running	0	9m57s
social-network-prometheus-server-75b59cf698-vdkng	2/2	Running	0	9m56s
user-db-0	1/1	Running	0	9m56s
user-service-78f5d9bf85-8n9lf	1/1	Running	0	4m39s

Figure 9.17: Output displaying Running status for all the Pods

- To verify that you are connected to Event Hubs, and not to local Kafka, you can check the event hubs in the portal, and check the different topics. You should see a friend and a user topic, as shown in *Figure 9.18*:

Figure 9.18: Showing the two topics created in your event hub

5. Keep watching the Pods. When all the Pods are up and running, get the external IP of the edge service. You can get that IP by using the following command:

```
kubectl get svc -n social-network
```

6. Then, run the following command to verify the creation of the actual social network:

```
bash ./deployment/sbin/generate-serial.sh <external-ip>:9000
```

This will again create a social network with 15 users, but will now use Event Hubs to send all the user-, friend-, and recommendation-related events.

7. You can see that activity on the Azure portal. The Azure portal creates detailed monitoring graphs for Event Hubs. To access those, click on the **friend** event hub, as shown in *Figure 9.19*:

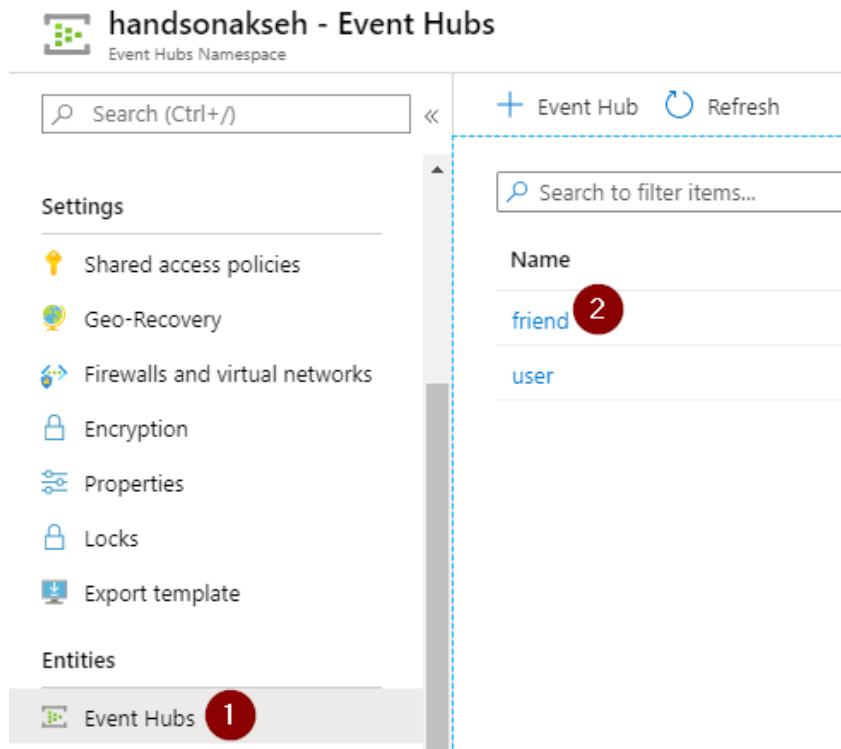


Figure 9.19: Clicking on the friend topic to get more metrics

In Figure 9.20, you can see that there are three graphs that the Azure portal provides you with out of the box: the number of requests, the number of messages, and the total throughput:

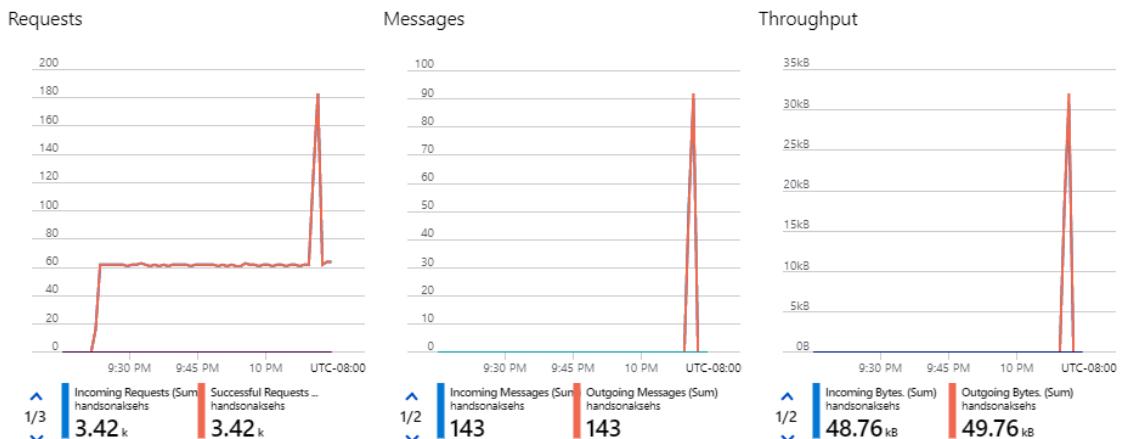


Figure 9.20: Display of the high-level graphs by default per topic

You can drill down further in the individual graphs. For example, click on the messages graph. This will bring you to an interactive graph editor in Azure monitor. You can see on a minute-by-minute basis how many messages were coming in and going out of your event hub, as shown in Figure 9.21:

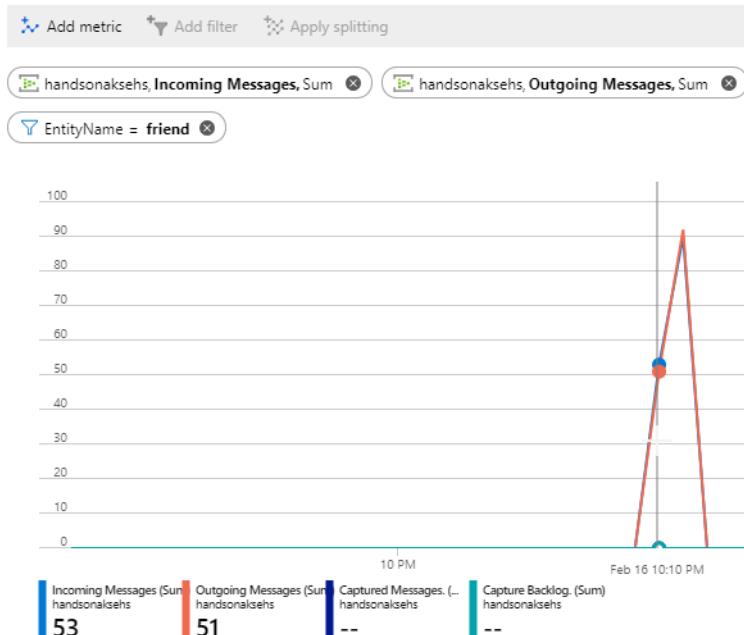


Figure 9.21: Clicking on the graph to get more details

Let's make sure to clean up the deployment we just created and scale our cluster back down:

```
helm delete social-network -n social-network
kubectl delete pvc -n social-network --all
kubectl delete pv --all
kubectl delete service neo4j-service -n social-network
az aks nodepool scale --node-count 2 -g rg-handsonaks \
--cluster-name handsonaks --name agentpool
```

You can also delete the event hub in the Azure portal. To delete the Event Hub, go to the **Overview** page of the event hub and select the **Delete** button, as shown in *Figure 9.22*. You will be asked to repeat the event hub's name to ensure that you don't delete it accidentally:

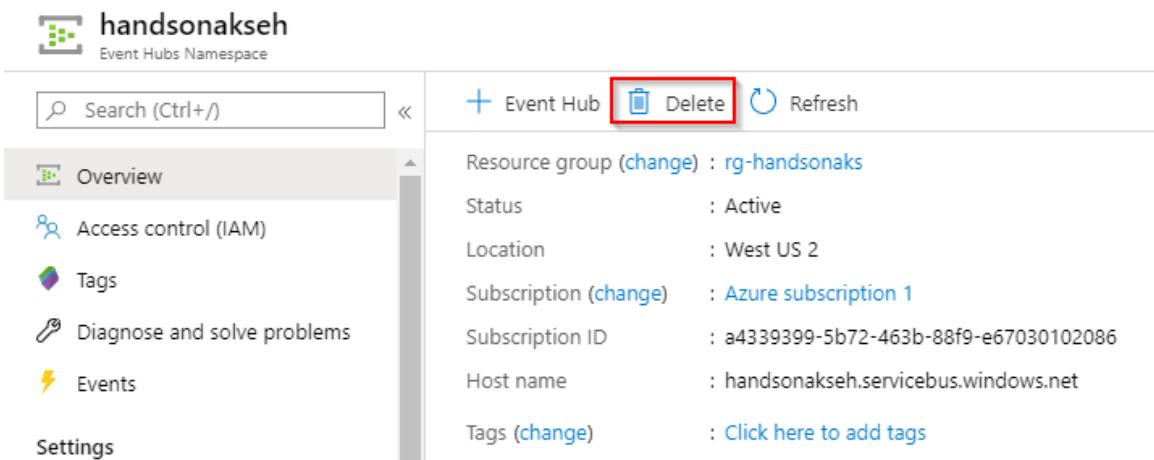


Figure 9.22: Clicking the delete button to delete your event hub

This concludes our example of using Event Hubs with the Azure Kubernetes Service. In this example, we reconfigured a Kafka application to use Azure Event Hubs instead.

Summary

In this chapter, we deployed a microservices-based application that connects to Kafka. We used Helm to deploy this sample application. We were able to test the application by sending events to a locally created Kafka cluster and watching objects being created and updated.

Finally, we covered storing events in Azure Event Hubs using Kafka support, and we were able to gather the required details to connect our microservice-based application and modify the Helm chart.

The next chapter will cover cluster security. We will cover RBAC security, secret management, and network security using Istio.

10

Securing your AKS cluster

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.

In this chapter, we will explore Kubernetes security in more depth. You will be introduced to the concept of **role-based access control (RBAC)** in Kubernetes. After that, you will learn about secrets and how to use them. You will first create secrets in Kubernetes itself, and afterward create a Key Vault to store secrets more securely. You'll finish this chapter with a brief introduction to service mesh concepts, and you'll be given a practical example to follow.

The following topics will be covered briefly in this chapter:

- Role-based access control
- Setting up secrets management
- Using secrets stored in Key Vault
- The Istio service mesh at your service

Note

To complete the example about RBAC, you need access to an Azure AD instance, with global administrator permissions.

Let's start this chapter with RBAC.

Role-based access control

In production systems, you need to allow different users different levels of access to certain resources; this is known as **role-based access control (RBAC)**. This section will take you through how to configure RBAC in AKS, and how to assign different roles with different rights. The benefits of establishing RBAC are that it not only acts as a guardrail against the accidental deletion of critical resources but also that it is an important security feature that limits full access to the cluster to roles that really need it. On an RBAC-enabled cluster, users will be able to observe that they can modify only those resources to which they have access.

Up till now, using Cloud Shell, we have been acting as root, which allowed us 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 log into any computer system. This prevents both access to secure data and unintentional downtime through the deletion of key resources. Anywhere between 22% and 29% (<https://blog.storagecraft.com/data-loss-statistics-infographic/>) of data loss is attributed to human error. You don't want to be part of that statistic.

Kubernetes developers realized this was a problem and added RBAC 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 has a default of 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, 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 notion to understand is that when interfacing with AKS, there are two layers of RBAC: Azure RBAC and Kubernetes RBAC. 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.

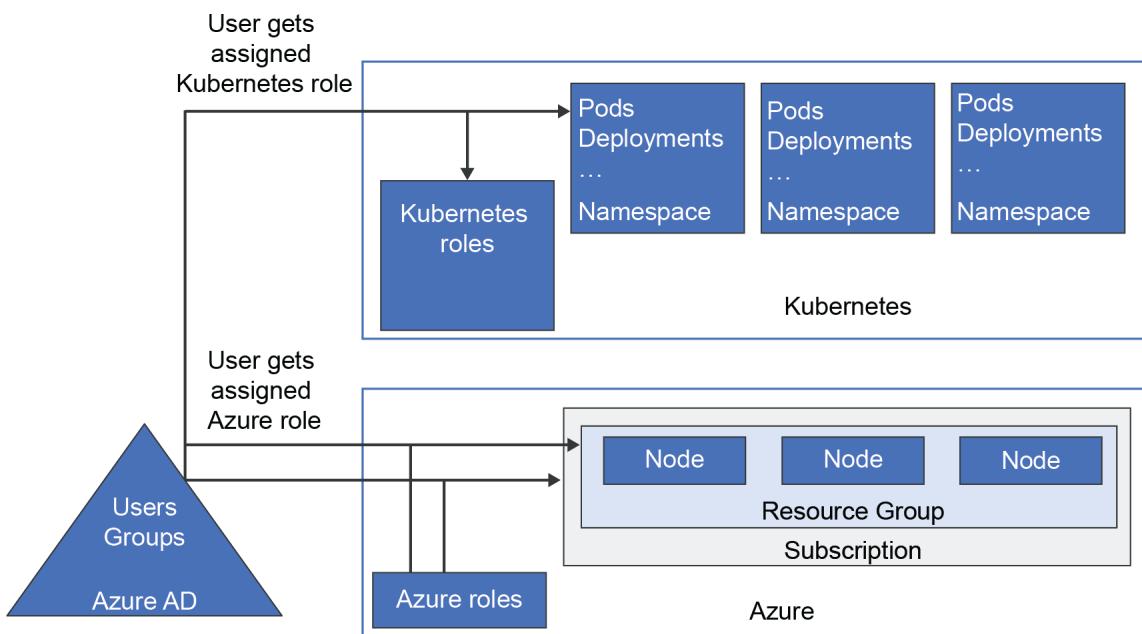


Figure 10.1: Two different RBAC planes, Azure and Kubernetes

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 this example, we will create a new cluster integrated with Azure AD. Let's start our exploration of RBAC by creating a new user and a new group in Azure AD.

Creating a new cluster with Azure AD integration

In this section, we will create a new cluster that is integrated with Azure AD. This is required so we can reference users in Azure AD in the next steps. All these steps will be executed in Cloud Shell. We have provided the steps as well in a file called **cluster-aad.sh**. If you prefer to execute the script, please change the variables in the first four lines to reflect your preferences. Let's go ahead and perform the steps:

1. We will start by scaling down our current cluster to one node:

```
az aks nodepool scale --cluster-name handsonaks \
-g rg-handsonaks --name agentpool--node-count 1
```

2. Then, we will set some variables that we'll use in the script:

```
EXISTINGAKSNAME="handsonaks"
NEWAKSNAME="handsonaks-aad"
RGNAME="rg-handsonaks"
LOCATION="westus2"
TENANTID=$(az account show --query tenantId -o tsv)
```

3. We will now get the existing service principal from our AKS cluster. We will reuse this service principal to grant permissions to the new cluster to access our Azure subscription:

```
# Get SP from existing cluster and create new password
RBACSP=$(azaks show -n $EXISTINGAKSNAME -g $RGNAME \
--query servicePrincipalProfile.clientId -o tsv)
RBACSPASSWD=$(openssl rand -base64 32)
az ad sp credential reset --name $RBACSP \
--password $RBACSPASSWD --append
```

4. Next, we will create a new Azure AD application. This Azure AD application will be used to get the Azure AD group memberships of a user:

```
serverApplicationId=$(az ad app create \
    --display-name "${NEWAKSNAME}Server" \
    --identifier-uris "https://${NEWAKSNAME}Server" \
    --query appId -o tsv)
```

5. In the next step, we will update the application, create a service principal, and get the secret from the service principal:

```
az ad app update --id $serverApplicationId --set groupMembershipClaims=All

az ad sp create --id $serverApplicationId

serverApplicationSecret=$(az ad sp credential reset \
    --name $serverApplicationId \
    --credential-description "AKSPassword" \
    --query password -o tsv)
```

6. Then we will give this service principal permission to access directory data in Azure AD:

```
az ad app permission add \
--id $serverApplicationId \
    --api 00000003-0000-0000-c000-000000000000 \
    --api-permissions e1fe6dd8-ba31-4d61-89e7-88639da4683d=Scope \
    06da0dbc-49e2-44d2-8312-53f166ab848a=Scope \
    7ab1d382-f21e-4acd-a863-ba3e13f7da61=Role
az ad app permission grant --id $serverApplicationId\
    --api 00000003-0000-0000-c000-000000000000
```

7. There is one manual step here that takes us to the Azure portal. We need to grant admin consent to the application. To achieve this, look for *Azure Active Directory* in the Azure search bar:

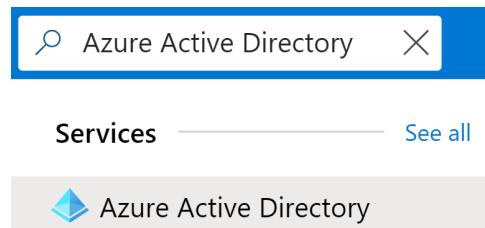


Figure 10.2: Looking for Azure Active Directory in the search bar

8. Then, select **App registrations** in the left-hand menu:

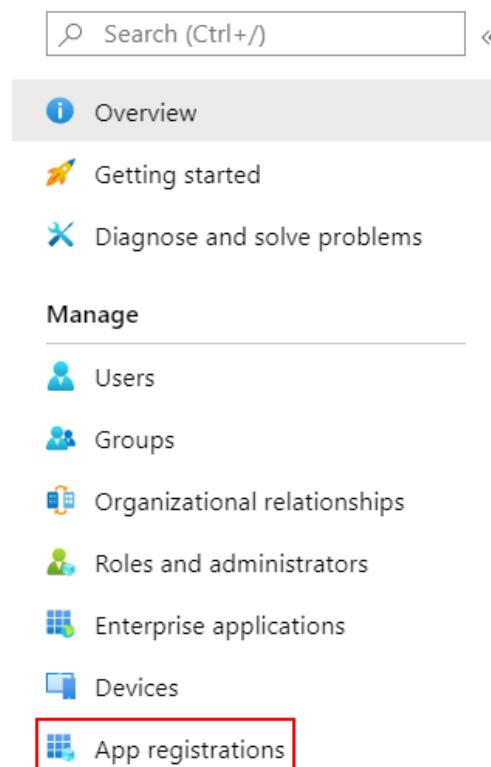


Figure 10.3: Selecting App registrations

9. In the **App registrations** blade, go to **All applications**, look for <clustername>Server, and select that application:

The screenshot shows the 'Default Directory - App registrations' blade in the Azure portal. The left sidebar includes 'Manage' sections for 'Users', 'Groups', and 'Organizational relationships'. The main area has a search bar and navigation links for 'New registration', 'Endpoints', 'Troubleshooting', 'App registrations (Legacy)', and 'Got feedback?'. A welcome message states: 'Welcome to the new and improved App registrations (now Generally Available). See what's new and learn more on how it's changed.' Below this, there are three tabs: 'All applications' (which is selected and highlighted with a red circle labeled '1'), 'Owned applications', and 'Applications from personal account'. A search bar contains the text 'handsonaksaadserver' (highlighted with a red circle labeled '2'). The results table shows one entry: 'Display name' is 'handsonaksaadServer', 'Application (client) ID' is 'ca7d95e0-3f21-4e14-8a72-656c5deb...', and 'Created On' is '2/17/2020'. A small red circle labeled '3' is positioned next to the first column of the table.

Figure 10.4: Looking for the app registration we created using the script earlier

10. In the view of that app, click on **API permissions**, and click on **Grant admin consent for Default Directory** (this name might depend on your Azure AD name):

The screenshot shows the 'handsonaksaadServer - API permissions' blade. The left sidebar includes 'Overview', 'Quickstart', 'Manage' sections for 'Branding', 'Authentication', 'Certificates & secrets', 'Token configuration (preview)', and 'API permissions' (which is selected and highlighted with a red circle labeled '1'). The main area has a search bar and a 'Refresh' button. A section titled 'Configured permissions' explains that applications are authorized to call APIs when granted permissions by users/admins. It includes a link to 'Learn more about permissions and consent'. Below this, there are two buttons: '+ Add a permission' and 'Grant admin consent for Default Directory' (which is highlighted with a red circle labeled '2'). The table below lists configured permissions:

API / Permissions name	Type	Description
Microsoft Graph (3)		
Directory.Read.All	Delegated	Read directory data
Directory.Read.All	Application	Read directory data

Figure 10.5: Granting the admin consent

In the following prompt, select **Yes** to grant those permissions.

Note

It can take about a minute for the **Grant admin consent** button to become active. If it is not active yet, please wait a minute and try again. You will need admin rights in Azure AD to be able to grant this consent.

11. Next, we'll create another service principal and grant it permissions as well. This service principal will take the authentication request from the user and will verify their credentials and permissions:

```
clientApplicationId=$(az ad app create \
    --display-name "${NEWAKSNAME}Client" \
    --native-app \
    --reply-urls "https://${NEWAKSNAME}Client" \
    --query appId -o tsv)

az ad sp create --id $clientApplicationId
oAuthPermissionId=$(az ad app show --id $serverApplicationId\
    --query "oauth2Permissions[0].id" -o tsv)
az ad app permission add --id $clientApplicationId \
    --api$serverApplicationId --api-permissions \
    $oAuthPermissionId=Scope
az ad app permission grant --id $clientApplicationId\
    --api $serverApplicationId
```

12. Then, as a final step, we can create the new cluster:

```
azaks create \
    --resource-group $RGNAME \
    --name $NEWAKSNAME \
    --location $LOCATION
    --node-count 2 \
    --node-vm-size Standard_D1_v2 \
    --generate-ssh-keys \
    --aad-server-app-id $serverApplicationId \
    --aad-server-app-secret $serverApplicationSecret \
    --aad-client-app-id $clientApplicationId \
    --aad-tenant-id $TENANTID \
    --service-principal $RBACSP \
    --client-secret $RBACSPPASSWD
```

In this section, we have created a new AKS cluster that is integrated with Azure AD for RBAC. It takes about 5 to 10 minutes to create a new cluster. While the new cluster is being created, you can continue to the next section and create a new user and group in Azure AD.

Creating users and groups in Azure AD

In this section, we will create a new user and a new group in Azure AD. We will use them later on in the chapter to assign them permissions to our AKS cluster.

Note

You need the User Administrator role in Azure AD to be able to create users and groups.

1. To start with, look for *azure active directory* in the Azure search bar:

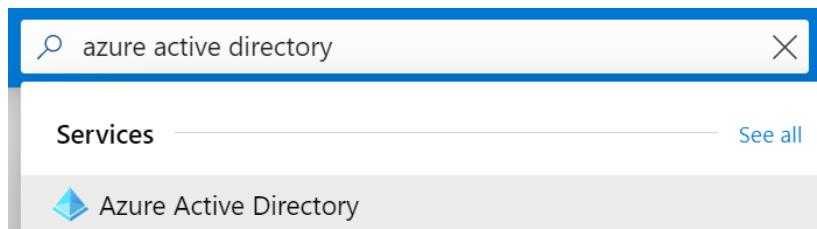


Figure 10.6: Looking for Azure Active Directory in the search bar

2. Click on **Users** on the left-hand side. Then select **New user** to create a new user:

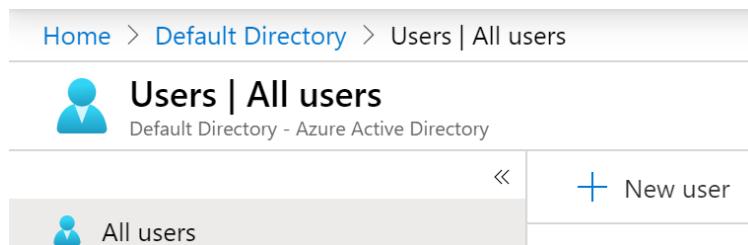


Figure 10.7: Clicking on New user to create a new user

3. Provide the information about the user, including the username. Make sure to note down the password, as this will be required to log in:

Home > Default Directory > Users - All users > New user

New user

Default Directory

Got feedback?

Create user

Create a new user in your organization.
This user will have a user name like
`alice@handsonaksdemooutlook.onmicrosoft.com`
[I want to create users in bulk](#)

Invite user

Invite a new guest user to collaborate with
your organization. The user will be emailed
an invitation they can accept in order to
begin collaborating.
[I want to invite guest users in bulk](#)

[Help me decide](#)

Identity

User name * (i)

✓ @ 🔗

The domain name I need isn't shown here

Name * (i)

✓

First name

Last name

Password

Auto-generate password

Let me create the password

Initial password

🔗

Create

Figure 10.8: Providing the user details (make sure to note down the password)

- Once the user is created, go back to the Azure AD blade and select **Groups**. Then click the **New group** button to create a new group:

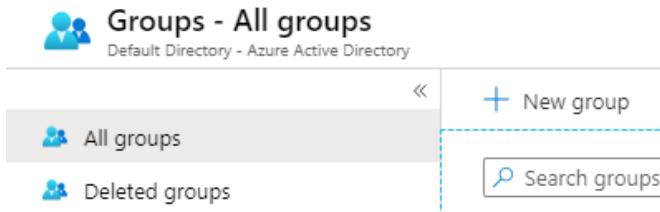


Figure 10.9: Clicking on New group to create a new group

- Create a new security group. Call the group **kubernetes-admins**, and add **Tim** as a member of the group. Then hit the **Create** button at the bottom:

Figure 10.10: Adding the group type, group name and group description

- We have now created a new user and a new group. As a final step, we'll make that user a cluster owner in AKS so they can use the Azure CLI to get access to the cluster. To do that, look for your cluster in the Azure search bar:

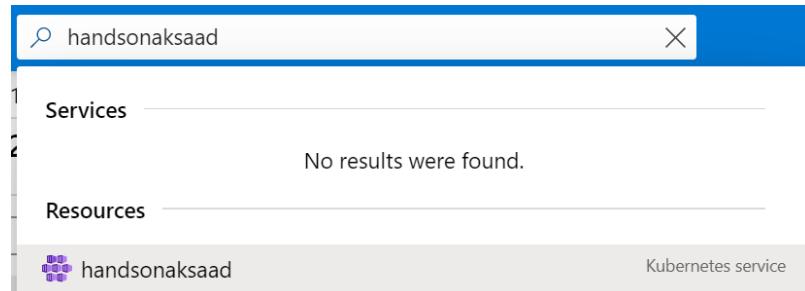


Figure 10.11: Looking for your cluster in the Azure search bar

- In the cluster blade, click on **Access control (IAM)** and then click on the **Add** button to add a new role assignment. Select the **Azure Kubernetes Service Cluster User Role** and assign that to the new user you just created:

The screenshot shows two windows side-by-side. On the left is the 'Access control (IAM)' blade for a Kubernetes service named 'handsonaksaad'. The 'Access control (IAM)' link is highlighted with a red circle labeled '1'. The 'Check access' tab is selected. On the right is the 'Add role assignment' dialog. It shows the 'Role' dropdown set to 'Azure Kubernetes Service Cluster User Role' (highlighted with a red circle labeled '3'), the 'Assign access to' dropdown set to 'Azure AD user, group, or service principal' (highlighted with a red circle labeled '2'), and a search bar containing 'tim'. Below it, a user card for 'Tim' (tim@handsonaksdemooutlook.onmicrosoft.com) is shown, also highlighted with a red circle labeled '4'.

Figure 10.12: Assigning the cluster user role to the new user you created

- As we will also be using Cloud Shell for our new user, we will give them contributor access to the Cloud Shell storage account. First, look for *storage* in the Azure search bar:

The screenshot shows the Azure search bar with the word 'storage' typed into it. Below the search bar, the 'Services' section is visible, with 'Storage accounts' highlighted.

Figure 10.13: Looking for storage accounts in the Azure search bar

- Select the resource group in which this storage account was created by Cloud Shell:

The screenshot shows the 'Storage accounts' blade. The search bar at the top contains 'storage'. Below it, a table lists storage accounts. One account, 'cloud-shell-storage-e...', is highlighted with a red box. The table columns include Name, Type, Kind, Resource group, and Location.

Name	Type	Kind	Resource group	Location
cloud-shell-storage-e...	Storage account	StorageV2	cloud-shell-storage-e...	East US

Figure 10.14: 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:

The screenshot shows the Azure portal interface. On the left, there's a sidebar with options like Overview, Activity log, Access control (IAM) (marked with a red circle 1), Tags, Events, Settings, and Quickstart. The main area shows the 'Access control (IAM)' blade for a resource group named 'cloud-shell-storage-eastus'. At the top, there's a search bar and a 'Check access' button (marked with a red circle 2). Below it are tabs for 'Role assignments' and 'Deny assignments'. The 'Role assignments' tab is selected. To its right is the 'Add role assignment' dialog. In this dialog, the 'Role' dropdown is set to 'Storage Account Contributor' (marked with a red circle 3). The 'Assign access to' dropdown is set to 'Azure AD user, group, or service principal'. A search bar contains the name 'tim' (marked with a red circle 4). Below the search bar, a result for 'Tim' is listed, with the email 'tim@handsonaksdemoutlook.onmicrosoft.com'.

Figure 10.15: Giving the newly created user Storage Account Contributor access

This has concluded the creation of a new user and group and giving that user access to AKS. In the next section, we will configure RBAC for that user and group.

Configuring RBAC in AKS

To demonstrate RBAC in AKS, we will create two namespaces and deploy the Azure Vote application in each namespace. We will give our group cluster-wide read-only access to Pods, and we will give the user the ability to delete Pods in only one namespace. Practically, we will need to create the following objects in Kubernetes:

- **ClusterRole** to give read-only access
- **ClusterRoleBinding** to grant our group the access to this role
- **Role** to give delete permissions in the **delete-access** namespace
- **RoleBinding** to grant our user access to this role

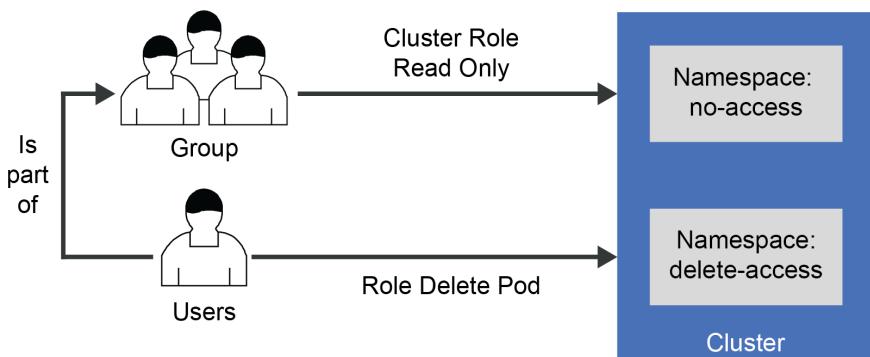


Figure 10.16: The Group getting read-only access to the whole cluster, and the User getting delete permissions to the delete-access namespace

Let's set up the different roles on our cluster:

1. To start our example, we will need to retrieve the ID of the group. The following commands will retrieve the group ID:

```
az ad group show -g 'kubernetes-admins' --query objectId -o tsv
```

This will show your group ID. Note this down because we'll need it in the next steps:

```
tim@Azure:~$ az ad group show -g 'kubernetes-admins' --query objectId -o tsv
eecb806a-35c4-4a0d-a8dc-4cb983c5371b
```

Figure 10.17: Getting the group ID

2. As we created a new cluster for this example, we will get the credentials to log in to this cluster. We will use the admin credentials to do the initial setup:

```
az aks get-credentials -n handsonaksad -g rg-handsonaks --admin
```

3. In Kubernetes, we will create two namespaces for this example:

```
kubectl create ns no-access
kubectl create ns delete-access
```

4. We will 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
```

5. Next, we will create the **ClusterRole** file. 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**

Line 4: Gives a name to our **ClusterRole**

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 this **ClusterRole** using the following command:

```
kubectl create -f clusterRole.yaml
```

6. The next step is to create a ClusterRoleBinding. The binding links the role to a user. 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:

Line2: Defines that we are creating a **ClusterRoleBinding**

Line 4: Gives a name to our **ClusterRoleBinding**

Lines 5-8: Refer to the **ClusterRole** we created in the previous step

Lines 9-12: Refer to our group in Azure AD

We can create this **ClusterRoleBinding** using the following command:

```
kubectl create -f clusterRoleBinding.yaml
```

7. Next, we'll create the **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** file from earlier. There are two meaningful differences:

Line 2: Defines we are creating a **Role**, and not a **ClusterRole**

Line 5: Defines in which namespace this **Role** is created

We can create this **Role** using the following command:

```
kubectl create -f role.yaml
```

8. Finally, we will create the **RoleBinding** that links our user to the namespaced 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 file from earlier. There are a couple of meaningful differences:

Line2: Defines the creation of a **RoleBinding** and not a **ClusterRoleBinding**

Line 5: Defines in which namespace this **RoleBinding** is created

Line 7: Refers to a regular **Role** and note a **ClusterRole**

Lines 11-13: Define our user instead of a group

We can create this **RoleBinding** using the following command:

```
kubectl create -f roleBinding.yaml
```

This has concluded the requirements for RBAC. We have created two roles and set up two RoleBindings. In the next section, we will explore the impact of RBAC by logging in to the cluster as our user.

Verifying RBAC

To verify that RBAC works as expected, we will log 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 log 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:

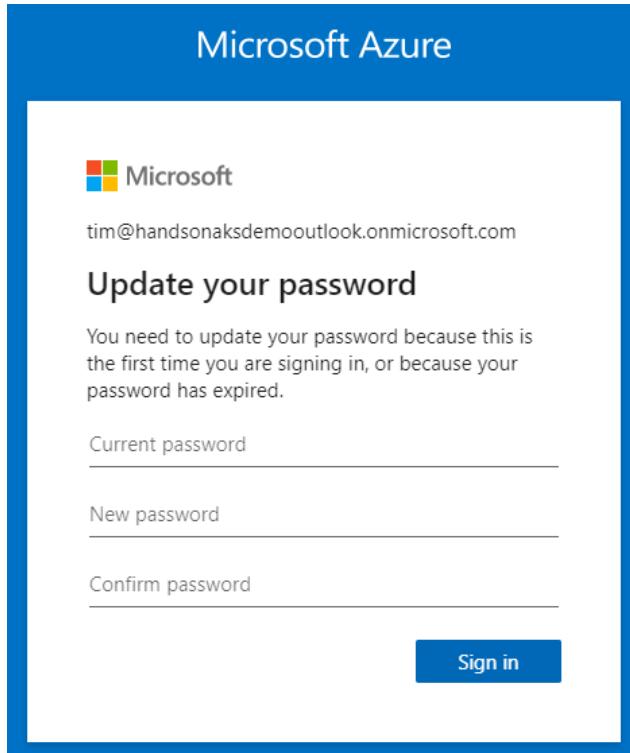


Figure 10.18: You will be asked to change your password

Once we have changed your password, we can start testing the different RBAC roles:

1. We will start our experiment by setting up Cloud Shell for the new user. Launch Cloud Shell and select Bash:

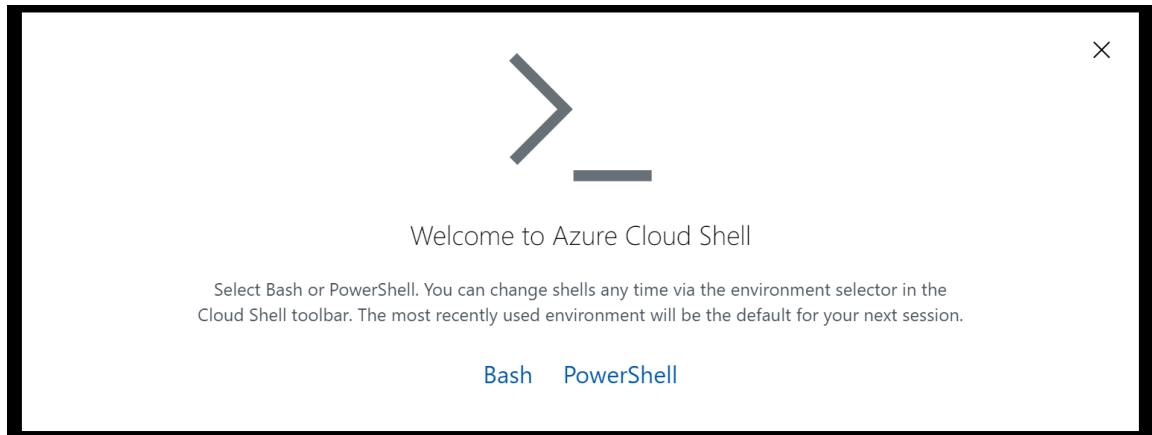


Figure 10.19: Selecting Bash as Cloud Shell

2. In the next view, select **Show advanced settings**:

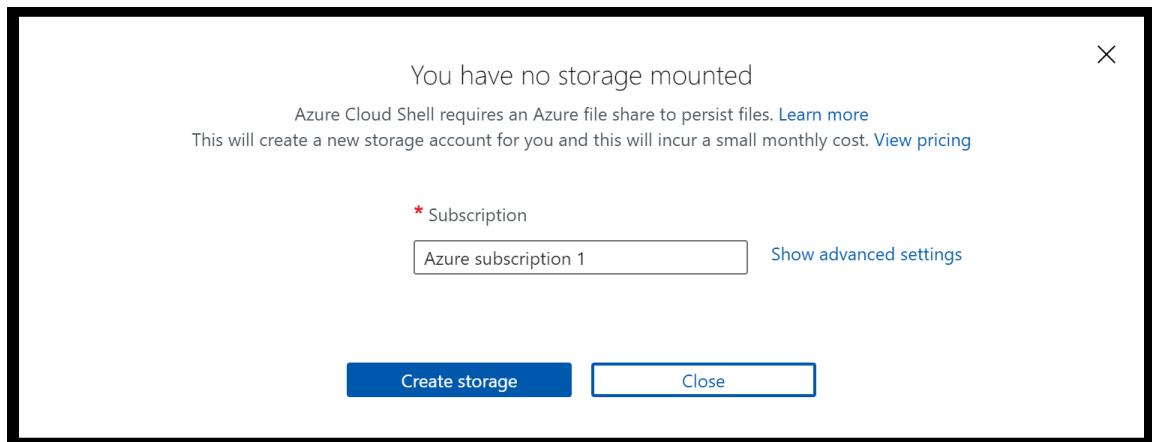


Figure 10.20: Selecting Show advanced settings

3. Then, point Cloud Shell to the existing storage account and create a new file share:

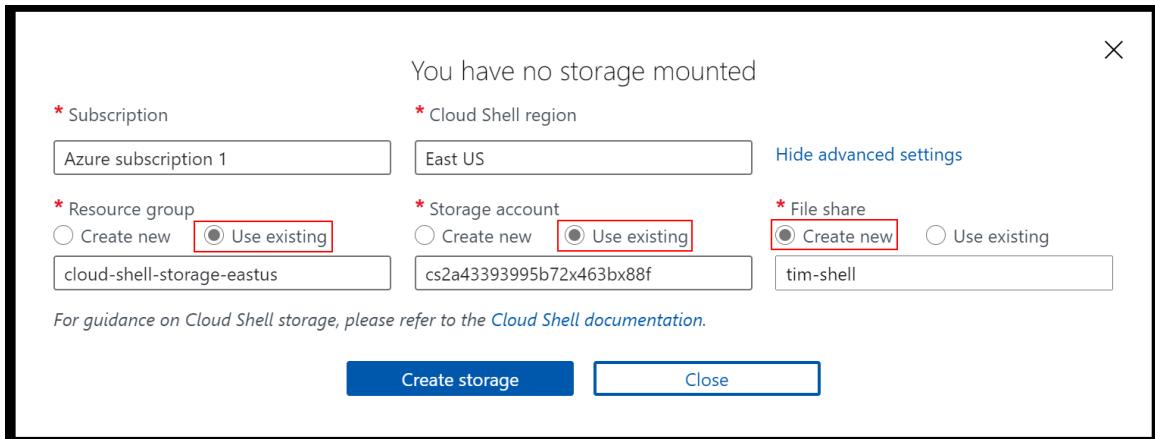


Figure 10.21: Point to the existing storage account and create a new file share

4. Once Cloud Shell is available, let's get the credentials to connect to our AKS cluster:

```
az aks get-credentials -n handsonaksaad -g rg-handsonaks
```

5. Then, we'll 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 log in again. Browse to <https://microsoft.com/devicelogin> and provide the code Cloud Shell showed you. Make sure you log in here with your new user's credentials:

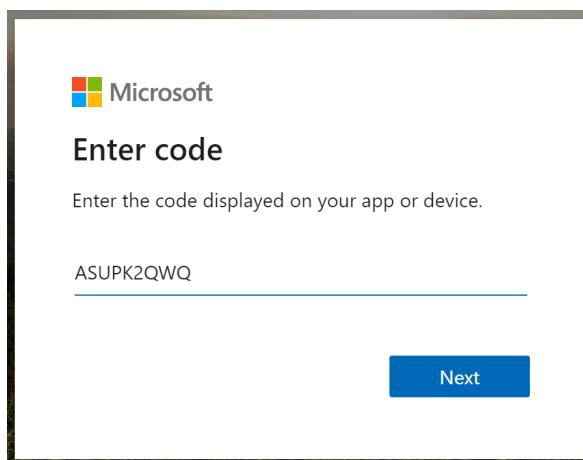


Figure 10.22: Copy-paste the code Cloud Shell showed you in the prompt

After you have logged 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/devicelogin and
enter the code ASUPK2QWQ2 to authenticate.
Error from server (Forbidden): nodes is forbidden: User "tim@handsonaksdemoutlook.on
microsoft.com" connect list resource "nodes" in API group "" at cluster scope
```

Figure 10.23: The prompt asking you to log in and the Forbidden message

- Now we can verify that our user has access to view Pods in all namespaces and that the user has the permissions 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** configured for the user's group:

```
tim@Azure:~$ kubectl get pods -n no-access
NAME                  READY   STATUS    RESTARTS   AGE
azure-vote-back-55dff664bb-tpd5v  1/1     Running   0          7m18s
azure-vote-front-d47b48c84-ng9j6  1/1     Running   0          7m18s
tim@Azure:~$ kubectl get pods -n delete-access
NAME                  READY   STATUS    RESTARTS   AGE
azure-vote-back-55dff664bb-nb655  1/1     Running   0          7m22s
azure-vote-front-d47b48c84-w65fv  1/1     Running   0          7m22s
```

Figure 10.24: Our user has access to view Pods in both namespaces

- 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 can be seen in Figure 10.25:

```
tim@Azure:~$ kubectl delete pod --all -n no-access
Error from server (Forbidden): pods "azure-vote-back-55dff664bb-tpd5v" is forbidden: User "tim@handsonaksdemoutlook.onmicrosoft.com" cannot delete resource "pods" in API group "" in the namespace "no-access"
Error from server (Forbidden): pods "azure-vote-front-d47b48c84-ng9j6" is forbidden: User "tim@handsonaksdemoutlook.onmicrosoft.com" cannot delete resource "pods" in API group "" in the namespace "no-access"
tim@Azure:~$ kubectl delete pod --all -n delete-access
pod "azure-vote-back-55dff664bb-r9nk5" deleted
pod "azure-vote-front-d47b48c84-c21m8" deleted
```

Figure 10.25: Deletes are denied in the no-access namespace and allowed in the delete-access namespace

In this section, we have set up a new cluster integrated with Azure AD and verified the correct configuration of RBAC with Azure AD identities. Let's clean up the resources we have created in this section, get the credentials for our existing cluster, and scale our regular cluster back to two nodes:

```
az aks delete -n handsonaksaad -g rg-handsonaks  
az aks get-credentials -n handsonaks -g rg-handsonaks  
az aks nodepool scale --cluster-name handsonaks \  
-g rg-handsonaks --name agentpool --node-count 2
```

In the next section, we will continue down the path of Kubernetes security, this time investigating Kubernetes secrets.

Setting up secrets management

All production applications require some secret information to function. Kubernetes has a pluggable Secrets back end to manage these secrets. Kubernetes also provides multiple ways of using the secrets in your Deployment. The ability to manage secrets and properly use the Secrets back end will make your services resistant to attacks.

We have used secrets in some of our Deployments in previous chapters. Mostly, we passed the secrets as a string in some kind of variable, or Helm took care of creating the secrets for us. In Kubernetes, secrets are a resource, just like Pods and ReplicaSets. Secrets are always tied to a specific namespace. Secrets have to be created in all the namespaces where you want to use them. In this section, we'll learn how to create, decode, and use our own secrets. We will start by using the built-in secrets from Kubernetes, and finish by leveraging Azure Key Vault to store secrets.

Creating your own 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 three types of secrets:

- **Generic secrets:** These can be created using literal values.
- **Docker-registry credentials:** These are used to pull images from a private registry.
- **TLS certificates:** These are used to store SSL certificates.

We'll begin by using the file method of creating secrets.

Creating secrets from files

Let's say that you need to store a URL and a secret token for accessing an API. To achieve this, you'll need to follow these steps:

1. Store the URL in `apiurl.txt`, as follows:

```
echo https://my-secret-url-location.topsecret.com \
> secreturl.txt
```

2. Store the token in another file, as follows:

```
echo 'superSecretToken' > secrettoken.txt
```

3. Let Kubernetes create the secrets from the files, as follows:

```
kubectl create secret generic myapi-url-token \
--from-file=./secreturl.txt --from-file=./secrettoken.txt
```

In this command, we specify the secret type as `generic`.

The command should return the following output:

```
secret/myapi-url-token created
```

4. We 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 10.26:

NAME	TYPE	DATA	AGE
default-token-qf5xz	kubernetes.io/service-account-token	3	27d
frontend-tls	kubernetes.io/tls	3	19d
letsencrypt-staging	Opaque	1	19d
myapi-url-token	Opaque	2	5s
tls-secret	kubernetes.io/tls	3	19d

Figure 10.26: The output of `kubectl get secrets`

Here, you will see the secret we just created, and any other secrets that are still present in the `default` namespace. Our 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 Docker registry or TLS secrets, which have a schema that will be verified as having the required details.

- For more details about the secrets, you can also run the `describe` command:

```
kubectl describe secrets/myapi-url-token
```

You will get an output similar to *Figure 10.27*:

```
Name:      myapi-url-token
Namespace: default
Labels:    <none>
Annotations: <none>

Type:  Opaque

Data
=====
secrettoken.txt: 17 bytes
secreturl.txt: 45 bytes
```

Figure 10.27: Output of describing the secret

As you can see, neither of the preceding commands displayed the actual secret values.

- To get the secrets, run the following command:

```
kubectl get -o yaml secrets/myapi-url-token
```

You will get an output similar to *Figure 10.28*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter10$ kubectl get -o yaml secrets/myapi-url-token
apiVersion: v1
data:
  secrettoken.txt: c3VwZXJtZWNyZXRUb2tlbgo=
  secreturl.txt: aHR0cHM6Ly9teS1zZWNyZXQtdXJsLWxvY2F0aW9uLnRvcHNlY3JldC5jb20K
kind: Secret
metadata:
  creationTimestamp: "2020-02-29T05:29:54Z"
  name: myapi-url-token
  namespace: default
  resourceVersion: "3499956"
  selfLink: /api/v1/namespaces/default/secrets/myapi-url-token
  uid: 2643714a-12fa-42fb-becb-f237feddd104
type: Opaque
```

Figure 10.28: Using the -o yaml switch in kubectl get secret gets us the encoded value of the secret

The data is stored as key-value pairs, with the file name 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 scrambles 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, run the following command:

```
echo 'c3VwZXJTZWNyZXRUb2t1bgo=' | base64 -d
```

You will get the value that was originally entered:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter10$ echo 'c3VwZXJTZWNyZXRUb2t1bgo=' | base64 -d  
superSecretToken
```

Figure 10.29: Base64-encoded secrets can easily be decoded

8. Similarly, for the **url** value, you can run the following command:

```
echo 'aHR0cHM6Ly9teS1zZWNyZXQtdXJsLWxvY2F0aW9uLnRvcHN1Y3JldC5jb20K' |  
base64 -d
```

You will get the originally entered **url** value, as shown in Figure 10.30:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter10$ echo 'aHR0cHM6Ly9teS1zZWNyZXQtdXJsLWxvY2F0aW9uLnRvcHN1Y3JldC5jb20K' | base64 -d  
https://my-secret-url-location.topsecret.com
```

Figure 10.30: The encoded URL can also easily be decoded

In this section, we were able to encode the URL with a secret token and get the actual secret values back using files. Let's move on and explore the second method – creating secrets from YAML or JSON definitions.

Creating secrets manually using YAML files

We will create the same secrets using YAML files by following these steps:

1. First, we need to encode the secrets to **base64**, as follows:

```
echo 'superSecretToken' | base64
```

You will get the following value:

```
c3VwZXJTZWNyZXRUb2t1bgo=
```

You might notice that this is the same value that was present when we got the **yaml** definition of the secret in the previous section.

2. Similarly, for the **url** value, we 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. We 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 we are creating a secret.

Line 5: This specifies that we 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 our secret.

4. Now we can create the secrets in the same way as any other Kubernetes resource by using the **create** command:

```
kubectl create -f myfirstsecret.yaml
```

5. We can verify whether our secret was successfully created using this:

```
kubectl get secrets
```

This will show us an output similar to *Figure 10.31*:

NAME	TYPE	DATA	AGE
default-token-qf5xz	kubernetes.io/service-account-token	3	27d
frontend-tls	kubernetes.io/tls	3	19d
letsencrypt-staging	Opaque	1	19d
myapi-url-token	Opaque	2	95s
myapiurltoken-yaml	Opaque	2	6s
tls-secret	kubernetes.io/tls	3	19d

Figure 10.31: Our secret was successfully created from a YAML file

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 we described in the previous section.

Creating generic secrets using literals

The third method of creating secrets is by using the `literal` method, which means you pass the value on the command line. To do this, run the following command:

```
kubectl create secret generic myapiurltoken-literal \
--from-literal=token='superSecretToken' \
--from-literal=url=https://my-secret-url-location.topsecret.com
```

We can verify that the secret was created by running the following command:

```
kubectl get secrets
```

This will give us a similar output to Figure 10.32:

NAME	TYPE	DATA	AGE
default-token-qf5xz	kubernetes.io/service-account-token	3	27d
frontend-tls	kubernetes.io/tls	3	19d
letsencrypt-staging	Opaque	1	19d
myapi-url-token	Opaque	2	2m37s
myapiurltoken-literal	Opaque	2	29s
myapiurltoken-yaml	Opaque	2	68s
tls-secret	kubernetes.io/tls	3	19d

Figure 10.32: Our secret was successfully created from the CLI

Thus, we have created secrets using literal values in addition to the preceding two methods.

Creating the Docker registry key

Connecting to a private Docker registry is a necessity in production environments. Since this use case is so common, Kubernetes has provided mechanisms to create a connection:

```
kubectl create secret docker-registry <secret-name> \
--docker-server=<your- registry-server> \
--docker-username=<your-name> \
--docker-password=<your-pword> --docker-email=<your-email>
```

The first parameter is the secret type, which is **docker-registry**. Then, you give the secret a name; for example, **regcred**. The other parameters are the Docker server (<https://index.docker.io/v1/> for Docker Hub), your username, your password, and your email.

You can retrieve the secret in the same way as other secrets by using **kubectl** to access secrets.

In Azure, **Azure Container Registry (ACR)** is most frequently used to store container images. There are two ways in which the cluster can connect to ACR. The first way is using a secret in the cluster like we just described. The second way – which is the recommended way – is to use a service principal. We will cover integrating AKS and ACR in *Chapter 11, Serverless Functions*.

The final secret type in Kubernetes is the TLS secret.

Creating the TLS secret

A TLS secret is used to store TLS certificates. To create a TLS secret that can be used in Ingress definitions, we use the following command:

```
kubectl create secret tls <secret-name> --key <ssl.key> --cert <ssl.crt>
```

The first parameter is **tls** to set the secret type, and then the **key** value and the actual certificate value. These files are usually obtained from your certificate registrar.

Note

We created TLS secrets in *Chapter 6, Managing your AKS*, cluster, where we used **cert-manager** to create the secrets on our behalf.

If you want to generate your own secret, you can run the following command to generate a self-signed SSL certificate:

```
openssl req -x509 -nodes -days 365 -newkey rsa:2048 - keyout /tmp/ssl.key -out /tmp/ssl.crt -subj "/CN=foo.bar.com"
```

In this section, we covered the different secret types in Kubernetes and saw how you can create secrets. In the next section, we will use the secrets in our application.

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 container 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.

Secrets as environment variables

Secrets are referenced in the Pod definition under the **containers** and **env** sections. We will use the secrets that we previously defined in a Pod and learn how to use them in an application:

1. We 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, we are setting the environment variables.

Lines 11-14: Here, we refer to the `secreturl.txt` file in the `myapi-url-token` secret.

Lines 16-19: Here, we refer to the `secrettoken.txt` file in the `myapi-url-token` secret.

2. Let's now create the Pod and see whether it really worked:

```
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 10.33:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter10$ kubectl exec -it secret-using-env sh
# echo $SECRET_URL
https://my-secret-url-location.topsecret.com
# echo $SECRET_TOKEN
superSecretToken
```

Figure 10.33: We can get the secrets inside the Pod

Any application can use the secret values by referencing the appropriate `env` variables. Please note that both the application and the Pod definition have no hard-coded secrets.

Secrets as files

Let's take a look at how to mount the same secrets as files. We will use the following Pod

definition to demonstrate how this can be done. It is provided in the `pod-with-env-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, we provide the mount details. We mount in the `/etc/secrets` directory as read-only.
- **Lines 13-16:** Here, we refer to the secret. Please note that both values in the secret will be mounted in the container.

Note that this is more succinct than the `env` definition, as you don't have to define a name for each and every secret. However, applications need to have a special code to read the contents of the file in order to load it properly.

Let's see whether the secrets made it through:

1. Create the Pod using the following command:

```
kubectl create -f pod-with-vol-secret.yaml
```

2. Echo the contents of the files in the mounted volume:

```
kubectl exec -it secret-using-volume bash
cd /etc/secrets/
cat secreturl.txt
cat /etc/secrets/secrettoken.txt
```

As you can see in *Figure 10.34*, our secrets are present in our Pod:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter10$ kubectl exec -it secret-using-volume bash
root@secret-using-volume:/# cd /etc/secrets
root@secret-using-volume:/etc/secrets# cat secreturl.txt
https://my-secret-url-location.topsecret.com
root@secret-using-volume:/etc/secrets# cat secrettoken.txt
superSecretToken
```

Figure 10.34: The secrets are available as files in our Pod

We have now discussed two ways in which secrets can be passed to a running container. In the next section, we will explain 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 Docker 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 instance that the secret Pod 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 10.35*:

```
Node:           aks-agentpool-39838025-vmss000000/10.240.0.4
Node-Selectors: <none>
```

Figure 10.35: Get the instance ID

2. Next, get the Docker ID of the running Pod:

```
kubectl describe pod secret-using-env | grep 'docker://'
```

This should give you the Docker ID:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter10$ kubectl describe pod secret-using-env | grep 'docker://'
Container ID: docker://3a9f8f8da17f3dc7a7d573a1e0f6dc57f5e5ab2f634878cee8923d5865da1ca1
```

Figure 10.36: Get the Docker ID

- Finally, we will execute a command in the node running our container to show the secret we have passed as an environment variable:

```
INSTANCE=<provide instance number>
DOCKERID=<provide Docker ID>

VMSS=$(az vmss list --query '[].name' -o tsv)
RGNAME=$(az vmss list --query '[].resourceGroup' -o tsv)
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 will show you both secrets in plain text:

```
user@Azure:~$ 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
message: "Enable succeeded: \n[stdout]\n[SECRET_URL=https://my-secret-url-lo
cation.topsecret.com]\n \
\ [SECRET_TOKEN=superSecretToken]\n KUBERNETES_PORT=tcp://10.0.0.1:443 APP_0
_PORT_7474_TCP_PORT=7474\
```

Figure 10.37: The secrets are decoded in the Docker runtime

As you can see, the secrets are decoded in the Docker runtime. This means any operator with access to the machine will have access to the secrets. This also means that most logging systems will log sensitive secrets.

Note

RBAC is also very important for controlling secrets. A person who has access to the cluster and has the right role has access to the secrets that are stored. As secrets are only base64-encoded, anybody with RBAC permissions to the secrets can decode them. It is recommended to treat access to secrets carefully and be very careful about giving people access to use the **kubectl exec** command to get a shell to containers.

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 we have explored secrets in Kubernetes using the default secrets mechanism, let's go ahead and use a more secure option, namely Key Vault.

Using secrets stored in Key Vault

In the previous section, we explored secrets that were stored natively in Kubernetes. This means they were stored base64-encoded on the Kubernetes API server (in the background, they will be stored in an etcd database, but that is part of the managed service provided by Microsoft). We 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 secrets storage solution called Azure Key Vault. It is a managed service that makes creating, storing, and retrieving secrets easy, and offers good monitoring of access to your secrets. Microsoft maintains an open-source project that allows you to mount secrets in Key Vault to your application. This solution is called Key Vault FlexVolume and is available here: <https://github.com/Azure/kubernetes-keyvault-flexvol>.

In this section, we will create a Key Vault and install Key Vault FlexVolume to mount a secret stored in Key Vault in a Pod.

Creating a Key Vault

We will use the Azure portal to create a Key Vault:

1. To start the creation process, look for *key vault* in the Azure search bar:

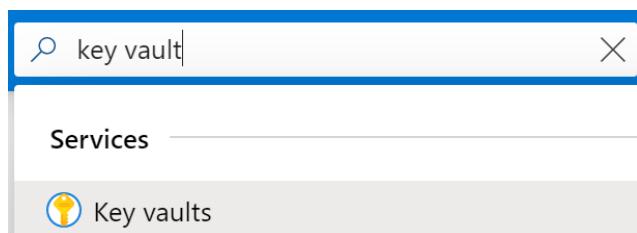


Figure 10.38: Looking for key vault in the Azure search bar

2. Click the **Add** button to start the creation process:

The screenshot shows the 'Key vaults' page in the Azure portal. At the top left is the 'Home' link and a breadcrumb trail 'Key vaults'. Below that is the title 'Key vaults' and the Microsoft logo. A navigation bar at the bottom includes a blue plus sign icon labeled 'Add', a gear icon labeled 'Manage view', and a circular arrow icon labeled 'Refresh'.

Figure 10.39: Click the Add button to start creating a Key Vault

3. Provide the details to create the Key Vault. The Key Vault's name has to be globally unique, so consider adding your initials to the name. It is recommended to create the Key Vault in the same region as your cluster:

The screenshot shows the 'Create key vault' wizard. The first step, 'Subscription', shows 'Azure subscription 1' selected. The second step, 'Resource group', shows 'rg-hansonaks' selected with a 'Create new' link. The third step, 'Instance details', includes fields for 'Key vault name' (set to 'hansonaks-kv'), 'Region' (set to 'West US 2'), 'Pricing tier' (set to 'Standard'), 'Soft-delete' (set to 'Enable'), 'Retention period (days)' (set to '90'), and 'Purge protection' (set to 'Disable'). At the bottom are 'Review + create' and 'Next : Access policy >' buttons.

Subscription *	Azure subscription 1
Resource group *	rg-hansonaks
Create new	
Instance details	
Key vault name *	hansonaks-kv
Region *	West US 2
Pricing tier *	Standard
Soft-delete	Enable Disable
Retention period (days) *	90
Purge protection	Enable Disable
Review + create	
< Previous	
Next : Access policy >	

Figure 10.40: Providing the details to create the Key Vault

4. Once you have provided the details, hit the **Review + create** button to review and create your Key Vault. Hit the **Create** button to finish the creation process.
5. It will take a couple of seconds to create your Key Vault. Once the vault is created, open it, go to secrets, and hit the **Generate/Import** button to create a new secret:

The screenshot shows the Azure Key Vault 'Secrets' page for a vault named 'handsonaks-kv'. The left sidebar has links for Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Events (preview), and Settings. Under Settings, there are 'Keys' and 'Secrets' (marked with a red circle containing '1'). The main area shows a table with columns Name, Type, and Status. A message says 'There are no secrets available.' Above the table are buttons for Generate/Import (marked with a red circle containing '2'), Refresh, and Restore Backup.

Name	Type	Status
There are no secrets available.		

Figure 10.41: Creating a new secret

6. In the secret creation wizard, provide the details about your secret. To make this demonstration easier to follow, it is recommended to use the name **k8s-secret-demo**. Hit the **Create** button at the bottom of the screen to create the secret:

Create a secret

Upload options
Manual

Name * ⓘ
k8s-secret-demo ✓

Value * ⓘ
***** ✓

Content type (optional)

Set activation date? ⓘ

Set expiration date? ⓘ

Enabled?

Yes
 No

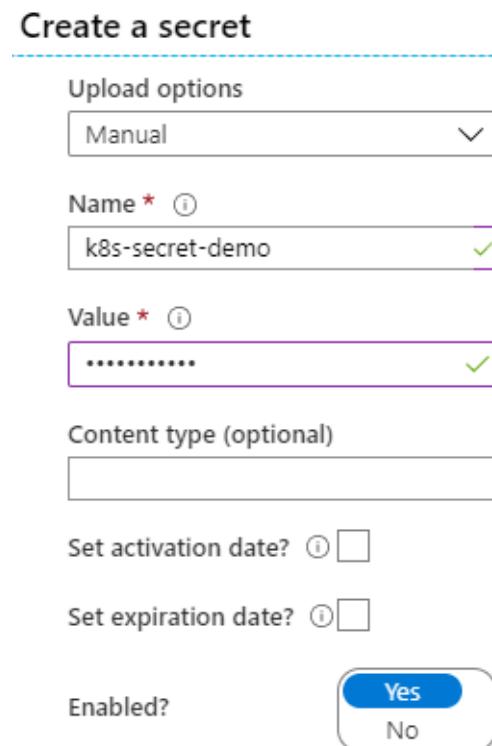


Figure 10.42: Providing the details for your new secret

Now that we have a secret in Key Vault, we can move ahead and configure Key Vault FlexVolume to access this secret in Kubernetes.

Setting up Key Vault FlexVolume

In this section, we will set up Key Vault FlexVolume in our cluster. This will allow us to retrieve secrets from Key Vault:

1. Create Key Vault FlexVolume using the following command. The **kv-flexvol-installer.yaml** file has been provided in the source code for this chapter:

```
kubectl create -f kv-flexvol-installer.yaml
```

Note

We have provided the **kv-flexvol-installer.yaml** file to ensure consistency with the examples in this book. For production use cases, we recommend installing the latest version, available at <https://github.com/Azure/kubernetes-keyvault-flexvol>.

2. FlexVolume requires credentials to connect to Key Vault. In this step, we will create a new service principal:

```
APPID=$(az ad app create \
    --display-name "flex" \
    --identifier-uris "https://flex" \
    --query appId -o tsv)
az ad sp create --id $APPID
```

```
APPPASSWD=$(az ad sp credential reset \
    --name $APPID \
    --credential-description "KeyVault" \
    --query password -o tsv)
```

3. We will now create two secrets in Kubernetes to store the service principal connection:

```
kubectl create secret generic kvcreds \
    --from-literal=clientid=$APPID \
    --from-literal=clientsecret=$APPPASSWD --type=azure/kv
```

4. We will now give this service principal access to the secrets in our Key Vault:

```
KVNAME=handsonaks-kv
az keyvault set-policy -n $KVNAME --key-permissions \
    get --spn $APPID
az keyvault set-policy -n $KVNAME --secret-permissions \
    get --spn $APPID
az keyvault set-policy -n $KVNAME --certificate-permissions \
    get --spn $APPID
```

You can verify that these permissions were successfully set in the portal. In your Key Vault, in the **Access policies** section, you should see that the flex application has **Get** permissions to keys, secrets, and certificates:

The screenshot shows the 'Access policies' page for a Key Vault named 'handsonaks-kv'. On the left, there's a sidebar with various navigation options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Events (preview), Keys, Secrets, Certificates, and Properties. The 'Access policies' option is selected. At the top, there are buttons for Save, Discard, and Refresh. Below these, there's a section titled 'Enable Access to:' with three checkboxes: 'Azure Virtual Machines for deployment', 'Azure Resource Manager for template deployment', and 'Azure Disk Encryption for volume encryption'. A '+ Add Access Policy' button is also present. The main area is titled 'Current Access Policies' and contains two sections: 'APPLICATION' and 'USER'. Under 'APPLICATION', there's a row for 'flex' with dropdown menus showing 'Get' for all three permission types (Key Permissions, Secret Permissions, Certificate Permissions). Under 'USER', there's a row for 'Nils Frans... handsonak...' with dropdown menus showing '9 selected', '7 selected', and '15 selected' respectively.

Figure 10.43: The flex application has Get permissions on keys, secrets, and certificates

Note

Key Vault FlexVolume supports a number of authentication options. We are now using a pre-created service principal. FlexVolume also supports the use of managed identities, either using a Pod identity or using the identity of the **Virtual Machine Scale Set (VMSS)** hosting the cluster. These other authentication options will not be explored in this book, but you are encouraged to read more at <https://github.com/Azure/kubernetes-keyvault-flexvol>.

That concludes the setup of Key Vault FlexVolume. In the next section, we will use FlexVolume to get access to a secret and mount it in a Pod.

Using Key Vault FlexVolume to mount a secret in a Pod

In this section, we will mount a secret from Key Vault into a new Pod.

1. We have provided a file called **pod_secret_flex.yaml** that will help create a Pod that mounts a Key Vault secret. You will need to make two changes to this file:

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: nginx-secret-flex
5  spec:
6    containers:
7      - name: nginx
8        image: nginx
9        volumeMounts:
10       - name: test
11         mountPath: /etc/secret/
12         readOnly: true
13    volumes:
14      - name: test
15        flexVolume:
16          driver: "azure/kv"
17          secretRef:
18            name: kvcreds
19          options:
20            keyvaultname: <keyvault name>
21            keyvaultobjectnames: k8s-secret-demo
22            keyvaultobjecttypes: secret
23            tenantid: "<tenant ID>"
```

Let's investigate this file:

Lines 9-12: Similar to the example of mounting a secret as a file, we are also providing a **volumeMount** to mount our secret as a file.

Lines 13-23: This is the volume configuration that points to FlexVolume.

Lines 17-18: Here, we refer to the service principal credentials we created in the previous example.

Lines 20-23: You need to make changes to these values to represent your environment.

2. We can create this Pod using the following command:

```
kubectl create -f pod_secret_flex.yaml
```

3. Once the Pod is created and running, we can execute in the Pod and verify that the secret is present:

```
kubectl exec -it nginx-secret-flex bash  
cd /etc/secret  
cat k8s-secret-demo
```

This should output the secret we created in Key Vault, as seen in Figure 10.43:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter10$ kubectl exec -it nginx-secret-flex bash  
root@nginx-secret-flex:/# cd /etc/secret  
root@nginx-secret-flex:/etc/secret# cat k8s-secret-demo  
superSecretroot@nginx-secret-flex:/etc/secret#
```

Figure 10.44: The secret we configured in Key Vault is mounted in the Pod as a file

We have successfully used Key Vault to store secrets. The secret is no longer stored poorly encoded using base64 on our cluster, but it is now stored securely outside of the cluster in Key Vault. We can still access the secret in the cluster using Key Vault FlexVolume.

Let's make sure to clean up our Deployment:

```
kubectl delete -f pod_secret_flex.yaml  
kubectl delete -f kv-flexvol-installer.yaml  
kubectl delete secret kvcreds
```

In this section, we have seen multiple ways to create and mount secrets. We have explored creating secrets using files, YAML files, and directly from the command line. We have also explored how secrets can be consumed, either as an environment variable or as a mounted file. We then looked into a more secure way to consume secrets, namely via Azure Key Vault.

In the next section, we'll explore using the Istio service mesh to configure additional network security in our cluster.

The Istio service mesh at your service

We have found a number of ways to secure our Pods, but our network connections are still open. Any Pod in the cluster can talk to any other Pod in the same cluster. As a site reliability engineer, you will want to enforce both ingress and egress rules. Additionally, you also want to introduce traffic monitoring and would like to have better traffic control. As a developer, you don't want to be bothered by all of those requirements as you won't know where your application will be deployed, or what is and isn't allowed. The best solution would be a tool for us to run the applications as is, while still specifying network policies, advanced monitoring, and traffic control.

Enter service mesh. This is defined as the layer that controls service-to-service communication. A service mesh is a network between microservices. A service mesh is implemented as a piece of software that controls and monitors traffic between those different microservices. Typically, a service mesh makes use of a sidecar to implement functionality transparently. If you remember, a Pod in Kubernetes can consist of one or more containers. A sidecar is a container that is added to an existing Pod to implement additional functionality; in a service mesh's case, this functionality is the service mesh's functionality.

Note

A service mesh controls much more than simply network security. If all you require is network security in your cluster, please consider adopting network policies.

Just as with microservices, a service mesh implementation is not a free lunch. If you don't have hundreds of microservices running, you probably don't need a service mesh. If you decide that you really do need one, you will need to choose one first. There are four popular options, each with its own advantages:

- Linkerd (<https://linkerd.io/>), including Conduit (<https://conduit.io/>)
- Istio (<https://istio.io/>)
- Consul (<https://www.consul.io/mesh.html>)

You should choose one service mesh based on your needs and feel comfortable in the knowledge that any one of these solutions will work for you. In this section, you will be introduced to the Istio service mesh. We chose the Istio service mesh because of its popularity. We used stars and commits on GitHub as a measure of popularity.

Describing the Istio service mesh

Istio is a service mesh created by IBM, Google, and Lyft. The project was announced in May 2017 and reached a stable v1 in July 2018. Istio is the control plane portion of the service mesh. Istio by default makes use of the **Envoy** sidecar. In this section, we will try to explain what a service mesh is and what the core functionalities of the Istio service mesh are.

Note

We are only touching on service meshes, and Istio specifically, very briefly in this book. Istio is a very powerful tool to not only secure but also manage the traffic for your cloud-native application. There are a lot of details and functionalities we do not cover in this book.

In terms of functionality, a service mesh (and Istio specifically) has a number of core capabilities. The first of these capabilities is traffic management. The term traffic management in this context means the control of traffic routing. By implementing a service mesh, you can control traffic in order to implement A/B testing or canary rollouts. Without a service mesh, you would need to implement that logic in your application's core code, whereas with a service mesh, that logic is implemented outside of the application.

With traffic management also comes the added security that Istio offers. Istio can manage authentication, authorization, and the encryption of service-to-service communication. This ensures that only authorized services communicate with each other. In terms of encryption, Istio can implement **mutual TLS (mTLS)** to encrypt service-to-service communication.

Istio also has the capability to implement policies. Policies can be used to either rate limit certain traffic (for example, only allow x transactions a minute), handle the whitelisting and blacklisting of access to services, or implement header rewrites and redirects.

Finally, Istio adds tremendous visibility of the traffic between the different services in your application. Out of the box, Istio can perform traffic tracing, monitoring, and logging of the traffic between your services. You can use this information to create dashboards to demonstrate application performance and use this information to more effectively debug application failures.

In the following section, we will install Istio and configure mTLS encryption of traffic between Pods. This is only one of the many capabilities of the platform. We encourage you to venture outside of the book and learn more about Istio.

Installing Istio

Installing Istio is easy; to do so, follow these steps:

1. Move to your home directory and download the **istio** package, as follows:

```
cd ~  
curl -L https://istio.io/downloadIstio | sh -
```

2. Add the **istio** binaries to your path. First, get the Istio release number you are running:

```
ls | grep istio
```

This should show you an output similar to *Figure 10.45*:

```
user@Azure:~$ ls | grep istio  
istio-1.4.4
```

Figure 10.45: Getting your Istio release number

Note down the Istio version and use it as follows to add the binaries to your path:

```
export PATH="$PATH:~/istio-<release-number>/bin"
```

3. Check whether your cluster can be used to run Istio using the following command:

```
istioctl verify-install
```
4. Install **istio** with the demo profile:

```
istioctl manifest apply --set profile=demo
```

Note

The demo profile is great to use for a demo of Istio, but it is not recommended for production installations.

5. Make sure everything is up and running, as follows:

```
kubectl get svc -n istio-system
```

You should see a number of services in the **istio-system** namespace:

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP
grafana	ClusterIP	10.0.212.170	<none>
istio-citadel	ClusterIP	10.0.223.149	<none>
istio-egressgateway	ClusterIP	10.0.53.193	<none>
istio-galley	ClusterIP	10.0.185.62	<none>
istio-ingressgateway	LoadBalancer	10.0.43.193	40.91.113.143
istio-pilot	ClusterIP	10.0.0.174	<none>
istio-policy	ClusterIP	10.0.117.236	<none>
istio-sidecar-injector	ClusterIP	10.0.219.27	<none>
istio-telemetry	ClusterIP	10.0.191.163	<none>
jaeger-agent	ClusterIP	None	<none>
jaeger-collector	ClusterIP	10.0.33.41	<none>
jaeger-query	ClusterIP	10.0.92.74	<none>
kiali	ClusterIP	10.0.55.38	<none>
prometheus	ClusterIP	10.0.165.7	<none>
tracing	ClusterIP	10.0.169.6	<none>
zipkin	ClusterIP	10.0.96.138	<none>

Figure 10.46: All Istio services are up and running

You now have Istio up and running.

Injecting Envoy as a sidecar automatically

As mentioned in the introduction of this section, a service mesh uses a sidecar to implement functionality. Istio has the ability to install the **Envoy** sidecar it uses automatically by using labels in the namespace. We can make it function in this way by using the following steps:

1. Let's label the default namespace with the appropriate label, namely **istio-injection=enabled**:

```
kubectl label namespace default istio-injection=enabled
```

2. Let's launch an application to see whether the sidecar is indeed deployed automatically (the **bookinfo.yaml** file is provided in the source code for this chapter):

```
kubectl create -f bookinfo.yaml
```

Get the Pods running on the default namespace. It might take a couple of seconds for the Pods to show up, and it will take a couple of minutes for all the Pods to become **Running**:

```
kubectl get pods
```

- Run the **describe** command on any one of the Pods:

```
kubectl describe pods/productpage-v1-<pod-ID>
```

You can see that the sidecar has indeed been applied:

```
Containers:
  productpage:
    Container ID:  docker://425288fe925ad8b3a3606e151fd72a4e73e057830423b2f97a69eebac17d89d2
    Image:         docker.io/istio/examples-bookinfo-productpage-v1:1.15.0
    Image ID:      docker-pullable://istio/examples-bookinfo-productpage-v1@sha256:0a5eb479595
    Port:          9080/TCP
    Host Port:    0/TCP
    State:         Running
    Started:      Sat, 29 Feb 2020 05:46:58 +0000
    Ready:         True
    Restart Count: 0
    Environment:   <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from bookinfo-productpage-token-bbbg6 (ro)
  istio-proxy:
    Container ID: docker://859a4439cf44031490b554ca749e74f7e88aa2e91f3651f97c2638b6e5d8acba
    Image:         docker.io/istio/proxyv2:1.4.5
    Image ID:      docker-pullable://istio/proxyv2@sha256:fc09ea0f969147a4843a564c5b677fbf3a6f9
    Port:          15090/TCP
    Host Port:    0/TCP
```

Figure 10.47: Istio automatically injected the sidecar proxy

Note that without making any modifications to the underlying application, we were able to get the Istio service mesh deployed and attached to the containers.

Enforcing mutual TLS

To encrypt all the service-to-service traffic, we will enable mTLS. By default, mutual TLS is not enforced. In this section, we will enforce mTLS authentication step by step.

Note

If you want more information on the end-to-end security framework within Istio, please read <https://istio.io/docs/concepts/security/#authentication-policies>.

For more details on mutual TLS, please read <https://istio.io/docs/concepts/security/#mutual-tls-authentication>.

Deploying sample services

In this example, you will be deploying two services, **httpbin** and **sleep**, under different namespaces. Two of these namespaces, **foo** and **bar**, will be part of the service mesh. This means they will have the **istio** sidecar proxy. You will learn a different way in which a sidecar can be injected here. The third namespace, **legacy**, will run the same services without the sidecar proxy:

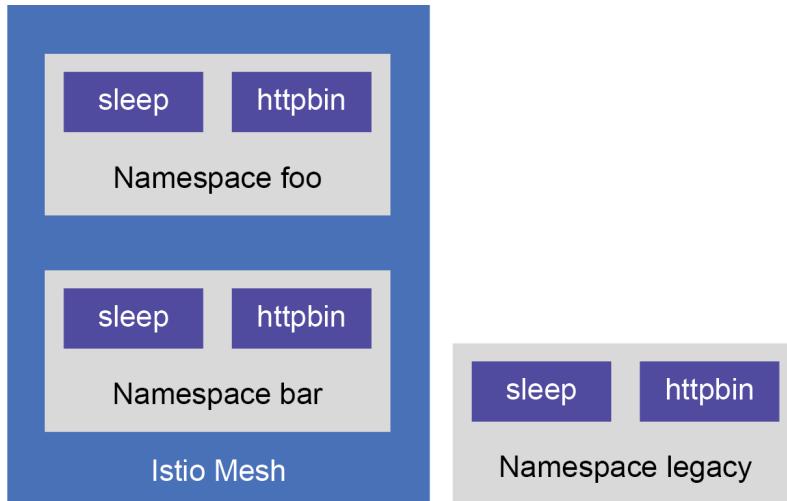


Figure 10.48: Two of the three namespaces are in the service mesh

We will look at the services of namespaces by using the following commands:

1. First, we create namespaces (**foo**, **bar**, and **legacy**) and create the **httpbin** and **sleep** services in those namespaces:

```
kubectl create ns foo
kubectl apply -f <(istioctl kube-inject \
-f httpbin.yaml) -n foo
kubectl apply -f <(istioctl kube-inject \
-f sleep.yaml) -n foo
kubectl create ns bar
kubectl apply -f <(istioctl kube-inject \
-f httpbin.yaml) -n bar
kubectl apply -f <(istioctl kube-inject \
-f sleep.yaml) -n bar
kubectl create ns legacy
kubectl apply -f httpbin.yaml -n legacy
kubectl apply -f sleep.yaml -n legacy
```

As you can see, we are now using the **istioctl** tool to inject the sidecar. It reads our YAML files and injects the sidecar in the Deployment. We now have a service in the **foo** and **bar** namespaces with the sidecar injected. However, they are not injected in the **legacy** namespace.

- Let's check whether everything deployed successfully. To check this, we have provided a script that makes a connection from each namespace to all the other namespaces. The script will output the HTTP status code for each connection.

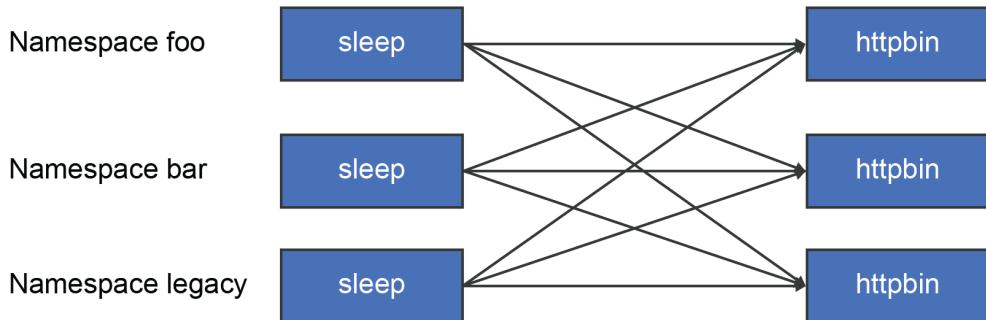


Figure 10.49: The script will test all connections

- Run the script using the following command:

```
bash test_mtls.sh
```

The preceding command iterates through all reachable combinations. You should see something similar to the following output. An HTTP status code of **200** means success:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter10$ bash test_mtls.sh
sleep.foo to      httpbin.foo: 200
sleep.foo to      httpbin.bar: 200
sleep.foo to      httpbin.legacy: 200
sleep.bar to      httpbin.foo: 200
sleep.bar to      httpbin.bar: 200
sleep.bar to      httpbin.legacy: 200
sleep.legacy to   httpbin.foo: 200
sleep.legacy to   httpbin.bar: 200
sleep.legacy to   httpbin.legacy: 200
```

Figure 10.50: Without any policy, we can successfully connect from each namespace to the other namespaces

This shows us that in the current configuration, all Pods can communicate with all other Pods.

4. Ensure that there are no existing policies except for the default, as follows:

```
kubectl get policies.authentication.istio.io \
--all-namespaces
kubectl get meshpolicies.authentication.istio.io
```

This should show you:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter10$ kubectl get policies.authentication.istio.io \
> --all-namespaces
NAMESPACE      NAME          AGE
istio-system   grafana-ports-mtls-disabled   27m
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter10$ kubectl get meshpolicies.authentication.istio.io
NAME      AGE
default  27m
```

Figure 10.51: There should only be two policies present

5. Additionally, ensure that there are no destination rules that apply:

```
kubectl get destinationrules.networking.istio.io \
--all-namespaces -o yaml | grep "host:"
```

In the results, there should be no hosts with **foo**, **bar**, **legacy**, or a wildcard (indicated as *****).

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter10$ kubectl get destinationrules.networking.istio.io \
> --all-namespaces -o yaml | grep "host:"
host: '*.global'
host: istio-policy.istio-system.svc.cluster.local
host: istio-telemetry.istio-system.svc.cluster.local
```

Figure 10.52: There should be no hosts with foo, bar, legacy, or a * wildcard

We have successfully deployed the sample services and were able to confirm that – in the default scenario – all services were able to communicate with each other.

Globally enabling mTLS

With an mTLS policy, you can state that all services must use mTLS when communicating with other services. If not using mTLS, a bad actor who has access to a cluster, even if they don't have access to the namespace, can communicate with any Pod. If given enough rights, they can also operate as the man in the middle between services. Implementing mTLS between services reduces the chances of man-in-the-middle attacks between services:

1. To enable mutual TLS globally, we will create the following **MeshPolicy** (provided in `mtls_policy.yaml`):

```

1  apiVersion: authentication.istio.io/v1alpha1
2  kind: MeshPolicy
3  metadata:
4    name: default
5  spec:
6    peers:
7      - mtls: {}

```

Since this **MeshPolicy** has no selectors, it will apply to all workloads. All workloads in the mesh will only accept encrypted requests using TLS. This means the **MeshPolicy** handles the incoming part of the connection:

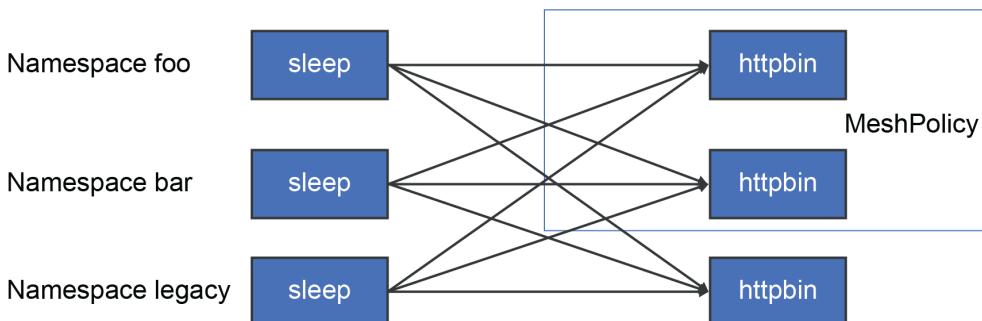


Figure 10.53: The **MeshPolicy** applies to the incoming connection

You can create the **MeshPolicy** using this command:

```
kubectl apply -f mtls_policy.yaml
```

Note

We are applying the mTLS very coarsely and aggressively. Typically, in a production system, you will introduce mTLS more slowly. Istio has a special mTLS enforcement mode called **permissive** to help achieve this. With mTLS in the **permissive** mode, Istio will try to implement mTLS where possible and log a warning where it is not possible. However, the traffic will continue to flow.

2. Now run the script again to test network connectivity:

```
bash test_mtls.sh
```

3. Those systems with sidecars will fail when running this command and will receive a **503** status code, as the client is still using plain text. It might take a few seconds for **MeshPolicy** to take effect. *Figure 10.54* shows the output:

```
sleep.foo to      httpbin.foo: 503
sleep.foo to      httpbin.bar: 503
sleep.foo to      httpbin.legacy: 200
sleep.bar to      httpbin.foo: 503
sleep.bar to      httpbin.bar: 503
sleep.bar to      httpbin.legacy: 200
sleep.legacy to   httpbin.foo: 000
command terminated with exit code 56
sleep.legacy to   httpbin.bar: 000
command terminated with exit code 56
sleep.legacy to   httpbin.legacy: 200
```

Figure 10.54: Traffic to Pods with a sidecar fails with a 503 status code

If you see a **200** status code more frequently in the preceding output, consider waiting a couple of seconds and then rerunning the test.

4. We will now allow certain traffic by setting the destination rule to use a * wildcard that is similar to the mesh-wide authentication policy. This is required to configure the client side:

```

1  apiVersion: networking.istio.io/v1alpha3
2  kind: DestinationRule
3  metadata:
4    name: default
5    namespace: istio-system
6  spec:
7    host: "*.local"
8    trafficPolicy:
9      tls:
10        mode: ISTIO_MUTUAL

```

Let's have a look at this file:

- **Line 2:** Here, we are creating a **DestinationRule** which defines policies that apply to traffic intended for a service after routing has occurred.
- **Line 7:** Traffic intended for any host in `.local` (which in our case is all the traffic in the cluster) should use this policy.
- **Lines 8-10:** Here, we define that mTLS is required for all traffic.

The **DestinationRule** applies to the outgoing part of the connection, as shown in Figure 10.55:

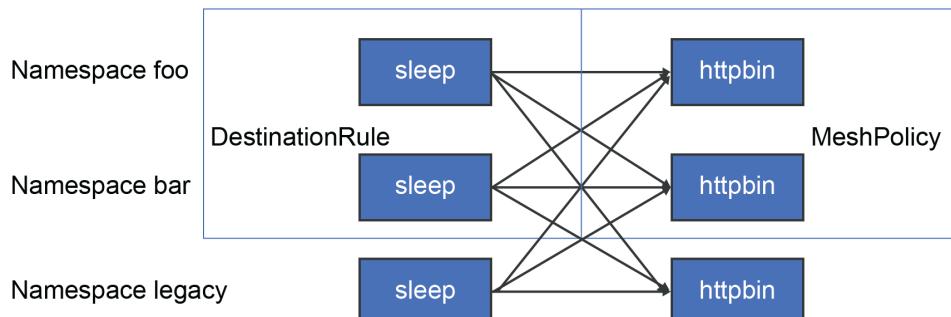


Figure 10.55: The DestinationRule applies to outgoing traffic

We can create this using the following:

```
kubectl create -f destinationRule.yaml
```

We can check the impact of this by running the same command again:

```
bash test_mtls.sh
```

This time, the returned codes will be as shown in *Figure 10.56*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter10$ bash test_mtls.sh
sleep.foo to      httpbin.foo: 200
sleep.foo to      httpbin.bar: 200
sleep.foo to      httpbin.legacy: 503
sleep.bar to      httpbin.foo: 200
sleep.bar to      httpbin.bar: 200
sleep.bar to      httpbin.legacy: 503
sleep.legacy to   httpbin.foo: 000
command terminated with exit code 56
sleep.legacy to   httpbin.bar: 000
command terminated with exit code 56
sleep.legacy to   httpbin.legacy: 200
```

Figure 10.56: Foo and bar can now connect to each other but they can't connect to legacy anymore

Let's consider for a minute what we implemented in this case. We implemented:

A cluster-wide policy that requires TLS traffic coming into Pods.

A cluster-wide DestinationRule that required mTLS on outgoing traffic.

The impact of this is that services within the mesh (also known as having the sidecar) can now communicate with each other using mTLS, and services that are completely out of the mesh (also known as not having the sidecar) can also communicate with each other, only without mTLS. Due to our current configuration, service-to-service communication is broken when only part of the flow is in the mesh.

Let's make sure to clean up any resources we deployed:

```
istioctl manifest generate --set profile=demo | kubectl delete -f -
for NS in "foo" "bar" "legacy"
do
  kubectl delete -f sleep.yaml -n $NS
  kubectl delete -f httpbin.yaml -n $NS
done
kubectl delete -f bookinfo.yaml
```

This concludes this demonstration of Istio.

Summary

In this chapter, we focused on security in Kubernetes. We started with a look into cluster RBAC using identities in Azure AD. After that, we continued with storing secrets in Kubernetes. We went into detail about creating, decoding, and using secrets. Finally, we installed and injected Istio, achieving the goal of being able to set system-wide policies without needing developer intervention or oversight. Since hackers like to pick on easy systems, the skills that you have learned in this chapter will help to make your setup less likely to be targeted.

In the next and final chapter, you will learn how to deploy serverless functions on [Azure Kubernetes Service \(AKS\)](#).

11

Serverless functions

Serverless and serverless functions have gained tremendous traction over the past few years. Cloud services such as Azure Functions, AWS Lambda, and GCP Cloud Run have made it very easy for developers 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 on 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. 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.

This automatic scaling is just one benefit of using serverless functions. 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. Another benefit of serverless functions is the ease of use for developers. 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 is run, and you are charged nothing for the idle time when your function is not run.

The popularity of public cloud serverless functions 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 Services (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's 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 multiple function platforms that are available for Kubernetes.

Multiple functions platforms

Functions platforms, such as Azure Functions, AWS Lambda, and Google Cloud Functions, have gained tremendously in popularity. The ability to run code without thinking about servers and having virtually limitless scale is very popular. The downside of using a cloud provider's functions implementation is that you are locked into their 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:

- **Serverless** (<https://serverless.com/>): 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.
- **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.
- **Fission.io** (<https://fission.io/>): Fission is written in the Go language and is Kubernetes-native. It is a serverless framework backed by the company Platform9.
- **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, and Docker Compose. It is primarily written in the Scala language.
- **Knative** (<https://cloud.google.com/knative/>): Knative is a serverless functions 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.

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 to scale to and from 0 Pods. Scaling from 0 to 1 Pod is important so that your application can start processing events. Scaling down to 0 instances is useful for preserving resources in your cluster. Scaling to and from 0 Pods is not possible using the default **Horizontal Pod Autoscaler (HPA)** in Kubernetes.

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, Scaling your application*.

In this chapter, we will deploy Azure Functions to Kubernetes in two examples:

- An HTTP-triggered function
- A queue-triggered function

Before we start, we need to set up an **Azure Container Registry (ACR)** and a development machine. The ACR will be used to store custom Docker images that contain the functions we will develop. We will use a development machine to build the functions and create Docker images.

Setting up prerequisites

In this section, we will set up the prerequisites we need in order to build and run functions. We need a container registry and a development machine.

We introduced container images and a container registry in *Chapter 1, Introduction to Docker and Kubernetes*, in the section on *Docker images*. A container image contains all the software required to start an actual running container. In this chapter, we will build custom Docker images that contain our functions. We need a place to store these images so that Kubernetes can pull these images and run the containers at scale. We will use the Azure Container Registry for this. Azure Container Registry is a private container registry that is fully managed by Azure.

Up to now in this book, we have run all the examples on the Azure Cloud Shell. For the example in this chapter, we need a separate development machine because the Azure Cloud Shell doesn't allow you to build Docker images. We will create a new development machine on 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, we will create an ACR and configure our Kubernetes cluster to have access to this cluster:

1. In the Azure search bar, look for **container registry** and click on **Container registries**:

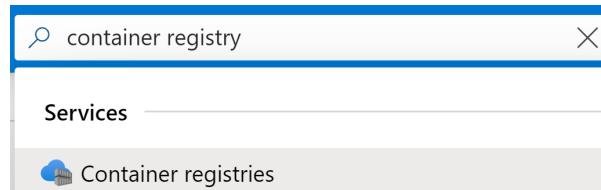


Figure 11.1: Looking for Container registries in the search bar

2. Click the **Add** button on the top to create a new registry. 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. Select the **Create** button to create the registry:

The screenshot shows the "Create container registry" dialog box. It includes the following fields:

- Registry name ***: handsonaksfunctions.azurecr.io
- Subscription ***: Azure subscription 1
- Resource group ***: rg-handsonaks
- Location ***: West US 2
- Admin user ***: A toggle switch between "Enable" (which is selected) and "Disable".
- SKU ***: Standard

Figure 11.2: Providing the details to create the registry

3. When your registry is created, open up Cloud Shell so that we can configure our AKS cluster to get access to our container registry. Use the following command to give AKS permissions on your registry:

```
az aks update -n handsonaks -g rg-handsonaks --attach-acr <acrName>
```

We now have an ACR that is integrated with AKS. In the next section, we will create a development machine that will be used to build the Azure functions.

Creating a development machine

In this section, we will create a development machine and install the tools necessary to run Azure functions on this machine:

- Docker runtime
- Azure CLI
- Azure Functions
- Kubectl

Note

To ensure a consistent experience, we will be creating a **virtual machine (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 machine:

1. To start, we will generate a set of ssh keys that will be used to connect to the VM:

```
ssh-keygen
```

You will be prompted for a location and a passphrase. Keep the default location and input an empty passphrase.

Note

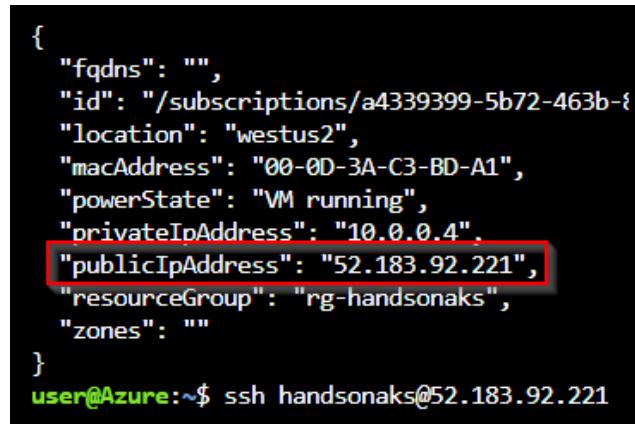
If you followed the example in *Chapter 10, Securing your AKS cluster*, where we created an Azure AD integrated cluster, you can skip step 1 as you will already have a set of SSH keys.

If you prefer to reuse the SSH keys that you already have, you can do that as well.

2. We will now create our development machine. We will create an Ubuntu VM using the following command:

```
az vm create -g rg-handsonaks -n devMachine \
--image UbuntuLTS --ssh-key-value ~/.ssh/id_rsa.pub \
--admin-username handsonaks --size Standard_D1_v2
```

3. 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 11.3:



```
{  
    "fqdns": "",  
    "id": "/subscriptions/a4339399-5b72-463b-8  
    "location": "westus2",  
    "macAddress": "00-0D-3A-C3-BD-A1",  
    "powerState": "VM running",  
    "privateIpAddress": "10.0.0.4",  
    "publicIpAddress": "52.183.92.221",  
    "resourceGroup": "rg-handsonaks",  
    "zones": ""  
}  
user@Azure:~$ ssh handsonaks@52.183.92.221
```

Figure 11.3: Connecting to the machine's public IP

Connect to the VM using the following command:

```
ssh handsonaks@<public IP>
```

You will be prompted whether you trust the machine's identity. Type **yes** to confirm.

4. You're now connected to a new machine 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
```

5. To verify that Docker is installed and running, you can run the following command:

```
sudo docker run hello-world
```

This should show you the **hello-world** message from Docker:

```
handsonaks@devMachine:~$ sudo docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
1b930d010525: Pull complete
Digest: sha256:9572f7cdcee8591948c2963463447a53466950b3fc15a247fcad1917ca215a2f
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
```

Figure 11.4: Running hello-world with Docker

6. To make the operation smoother, we will add our user to the Docker group, which will no longer require **sudo** in front of the Docker commands:

```
sudo usermod -aG docker handsonaks
newgrp docker
```

You should now be able to run the **hello-world** command without **sudo**:

```
docker run hello-world
```

7. Next, we will install the Azure CLI on this development machine. 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 logging in:

```
az login
```

This will display a login code that you need to enter at <https://microsoft.com/devicelogin>:

```
handsonaks@devMachine:~$ az login
To sign in, use a web browser to open the page https://microsoft.com/
/devicelogin and enter the code B2KC8AKJQ to authenticate.
```

Figure 11.5: Logging in to the az CLI

Browse to that website and paste in the login code that was provided to you to enable you to log in to Cloud Shell. Make sure to do this in a browser you are logged into with the user who has access to your Azure subscription.

We can now use the CLI to authenticate our machine to ACR. This can be done using the following command:

```
az acr login -n <registryname>
```

This will show you a warning that the password will be stored unencrypted. You can ignore that for the purpose of this demonstration.

The credentials to ACR expire after a certain time. If you run into the following error during this demonstration, you can log in to ACR again using the preceding command:

```
Running 'docker push handsonaksfunctions.azurecr.io/js-eventhub'..done
Error running docker push handsonaksfunctions.azurecr.io/js-eventhub.
output: The push refers to repository [handsonaksfunctions.azurecr.io/js-eventhub]
dbd793f76c1f: Preparing
587f5764a2fa: Preparing
77cd1c2449e0: Preparing
c642551d0d3a: Preparing
f65616f105d5: Preparing
b9f724514b30: Preparing
0cf75cb98eb2: Preparing
814c70fd9ae62: Preparing
b9f724514b30: Waiting
0cf75cb98eb2: Waiting
814c70fd9ae62: Waiting

unauthorized: authentication required
```

Figure 11.6: If this error is encountered, you can resolve this by logging into ACR again

9. Next, we'll install **kubectl** on our machine. The **az** CLI has a shortcut to install the CLI, which we will use:

```
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, we 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 -y
```

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 step to reflect your Ubuntu version.

We now have the prerequisites to start our work with functions on Kubernetes. We created an ACR to store our custom Docker images, and we have a development machine that we will use to create and build Azure functions. In the next section, we will build a first function that is HTTP triggered.

Creating an HTTP-triggered Azure function

In this first example, we will create an HTTP-triggered Azure function. This means that you can browse to the page hosting the actual function:

1. To begin, we will create a new directory and navigate to that directory:

```
mkdir http  
cd http
```

2. Now, we will initialize a function using the following command. The **--docker** parameter specifies that we will build our function as a Docker container. This will result in a Dockerfile being created for us. We will select the Python language, option 3 in the following screenshot:

```
func init --docker
```

This will create the required files for our function to work:

```
handsonaks@devMachine:~/http$ func init --docker
Select a number for worker runtime:
1. dotnet
2. node
3. python
4. powershell
Choose option: 3
python
Found Python version 3.6.9 (python3).
Writing .gitignore
Writing host.json
Writing local.settings.json
Writing /home/handsonaks/http/.vscode/extensions.json
Writing Dockerfile
Writing .dockerignore
```

Figure 11.7: Creating a Python function

3. Next, we will create the actual function. Enter the following code and select the fifth option, **HTTP trigger**, and name the function **python-http**:

```
func new
```

This should result in an output like *Figure 11.8*:

```
handsonaks@devMachine:~/http$ func new
Select a number for template:
1. Azure Blob Storage trigger
2. Azure Cosmos DB trigger
3. Azure Event Grid trigger
4. Azure Event Hub trigger
5. HTTP trigger
6. Azure Queue Storage trigger
7. Azure Service Bus Queue trigger
8. Azure Service Bus Topic trigger
9. Timer trigger
Choose option: 5
HTTP trigger
Function name: [HttpTrigger] python-http
Writing /home/handsonaks/http/python-http/_init__.py
Writing /home/handsonaks/http/python-http/function.json
The function "python-http" was created successfully from the "HTTP trigger" template.
```

Figure 11.8: Creating an HTTP-triggered function

4. The code of the function is stored in the directory called **python-http**. We are not going to make code changes to the 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. We will need to make one change to a function's configuration file. By default, functions require an authenticated request. We will change this to anonymous for our demo. We will make the change in **vi** by executing the following command:

```
vi python-http/function.json
```

We will replace the **authLevel** on line 5 with **anonymous**. To make that change, execute the following steps:

Press I to go into insert mode.

Remove **function** and replace it with **anonymous**:

```
{  
  "scriptFile": "__init__.py",  
  "bindings": [  
    {  
      "authLevel": "anonymous",  
      "type": "httpTrigger",  
    }  
  ]  
}
```

Figure 11.9: Changing the function to anonymous

- Hit Esc, type :wq!, and then Enter to save and quit vi.

Note

We changed the authentication requirement for our function to **anonymous**.

This will make our 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. We are now ready to deploy our function to AKS. We can deploy the function using the following command:

```
func kubernetes deploy --name python-http \
--registry <registry name>.azurecr.io
```

This will cause the function's runtime to do a couple of steps. First, it will build a container image, then it will push that image to our registry, and finally it will deploy the function to Kubernetes:

```
handsonaks@devMachine:~/http$ func kubernetes deploy --name python-http --registry handsonaksfunctions.azurecr.io
Running 'docker build -t handsonaksfunctions.azurecr.io/python-http /home/handsonaks/http'...done
Running 'docker push handsonaksfunctions.azurecr.io/python-http'.....done
secret/python-http created
service/python-http-http created
deployment.apps/python-http-http created
```

Figure 11.10: Deploying the function to AKS

7. This will create a regular Pod on top of Kubernetes. To check the Pods, you can run the following command:

```
kubectl get pods
```

8. Once that Pod is in a running state, you can get the public IP of the Service that was deployed and connect to it:

```
kubectl get service
```

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!, which is what our function is supposed to show:

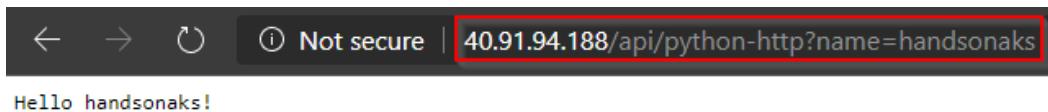


Figure 11.11: Our function is working correctly

We have now created a function with an HTTP trigger. Let's clean up this Deployment before moving to the next section:

```
Kubectl delete deploy python-http-http
kubectl delete service python-http-http
kubectl delete secret python-http
```

In this section, we 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 an autoscaler.

Creating a queue-triggered function

In the previous section, we created a sample HTTP function. In a real-world use case, 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, we'll create a function that is integrated with storage queues to consume events. We will also configure KEDA to allow scaling to/from 0 Pods in case of low traffic.

We still start by creating a queue in Azure.

Creating a queue

In this section, we will create a new storage account and a new queue in that storage account. We will connect functions to that queue in the next section.

1. To begin, we will create a storage account. Look for **storage** in the Azure search bar and select **Storage accounts**:

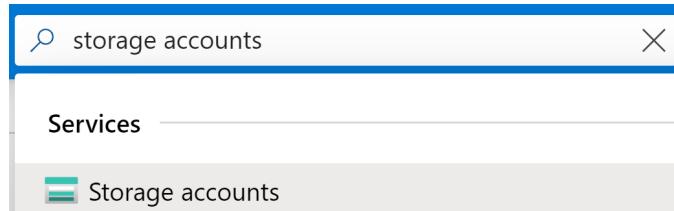


Figure 11.12: Looking for storage in the Azure search bar

2. Click the **Add** button on the top to create a new 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)**:

Basics Networking Advanced Tags Review + create

Azure Storage is a Microsoft-managed service providing cloud storage that is highly available, secure, durable, scalable, and redundant. Azure Storage includes Azure Blobs (objects), Azure Data Lake Storage Gen2, Azure Files, Azure Queues, and Azure Tables. The cost of your storage account depends on the usage and the options you choose below.

[Learn more about Azure storage accounts](#)

Project details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription *

Azure subscription 1

Resource group *

rg-hansonaks

[Create new](#)

Instance details

The default deployment model is Resource Manager, which supports the latest Azure features. You may choose to deploy using the classic deployment model instead. [Choose classic deployment model](#)

Storage account name * ⓘ

hoafunc

Location *

(US) West US 2

Performance ⓘ

Standard Premium

Account kind ⓘ

StorageV2 (general purpose v2)

Replication ⓘ

Locally-redundant storage (LRS)

Access tier (default) ⓘ

Cool Hot

Figure 11.13: Providing the details to create the storage account

If you're ready, click the **Review and Create** button at the bottom. In the review 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 blade, go to **Access keys**, and copy the primary connection string. Note down this string for now:

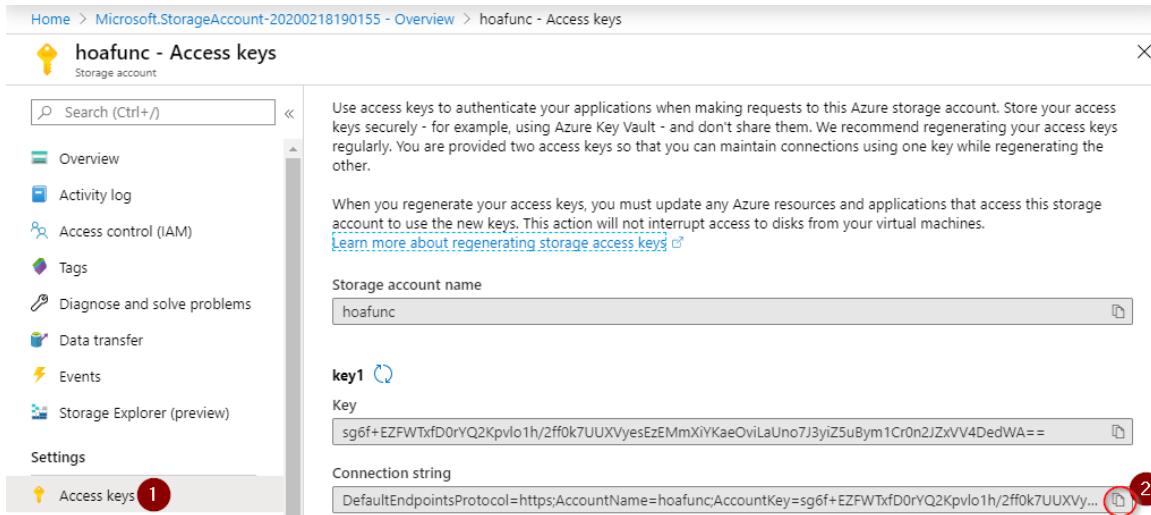


Figure 11.14: Copying the primary connection string

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>.

- 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, call the queue **function**:

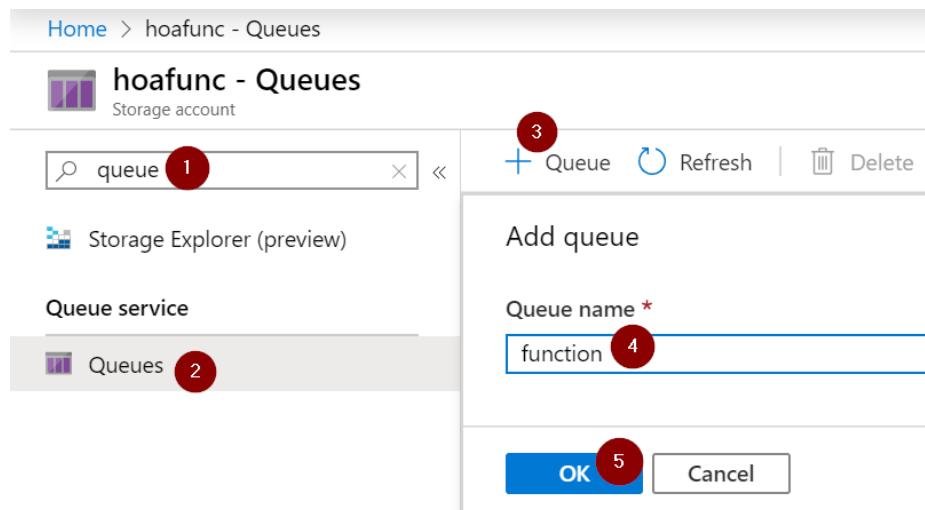


Figure 11.15: Creating a new queue

We have now created a storage account in Azure and have its connection string. We created a queue in this storage account. In the next section, we will create a function that will consume messages from the queue.

Creating a queue-triggered function

In the previous section, we created a queue on Azure. In this section, we will create a new function that will watch this queue. We will need to configure this function with the connection string to this queue:

- We will begin by creating a new directory and navigating to it:

```
mkdir ~/js-queue
cd ~/js-queue
```

- Now we can create the function. We will start with the initialization:

```
func init --docker
<select node, option 2>
<select javascript, option 1>
```

This should result in the output shown in Figure 11.16:

```
handsonaks@devMachine:~/js-queue$ func init --docker
Select a number for worker runtime:
1. dotnet
2. node
3. python
4. powershell
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 .dockerrcignore
```

Figure 11.16: Initializing a new function

Following the initialization, we can create the actual function:

```
func new
<select Azure queue storage trigger, option 10>
<provide a name, suggested name: js-queue>
```

This should result in the output shown in Figure 11.17:

```
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. SendGrid
12. Azure Service Bus Queue trigger
13. Azure Service Bus Topic trigger
14. SignalR negotiate HTTP trigger
15. 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 11.17: Creating a new function

3. We will now need to make a couple of configuration changes. We need to provide functions with the connection string 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.
- On the line of **AzureWebJobsStorage** (line 6), replace the value 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>"
```

```
{
  "IsEncrypted": false,
  "Values": {
    "FUNCTIONS_WORKER_RUNTIME": "node",
    "AzureWebJobsStorage": "DefaultEndpointsProtocol=https;AccountName=hoafunc;Acc
    ountKey=sg6f+EZFWTxFD0rYQ2Kpvlo1h/2ff0k7UUXVyesEzMmXiYKaeOviLaUno7J3yiZ5uBym1Cr0n
    2JZxVV4DedWA==;EndpointSuffix=core.windows.net",
    "QueueConnString": "DefaultEndpointsProtocol=https;AccountName=hoafunc;Account
    Key=sg6f+EZFWTxFD0rYQ2Kpvlo1h/2ff0k7UUXVyesEzMmXiYKaeOviLaUno7J3yiZ5uBym1Cr0n2JZx
    VV4DedWA==;EndpointSuffix=core.windows.net"
  }
}
```

Figure 11.18: 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 we need to edit is the function configuration itself. Here, we will refer to the connection string from earlier, and provide the queue name. 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.
- On line 7, change the queue name to the name of the queue we created (**function**).
- On line 8, add **QueueConnString** to the connection field:

```
{"bindings": [ { "name": "myQueueItem", "type": "queueTrigger", "direction": "in", "queueName": "function", "connection": "QueueConnString" } ]}
```

Figure 11.19: Editing the js-queue/function.json file

- Save and close the file by hitting the Esc key, type :wq!, and then press Enter.
5. We are now ready to publish our function to Kubernetes. We will start the publishing by setting up KEDA on our Kubernetes cluster:

```
kubectl create ns keda
func kubernetes install --keda --namespace keda
```

This will set up KEDA on our cluster. The installation doesn't take long. To verify that installation was successful, make sure that the KEDA Pod is running:

```
kubectl get pod -n keda
```

6. We can now deploy our function to Kubernetes. We 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, use the following command:

```
func kubernetes deploy --name js-queue \
--registry <registry name>.azurecr.io \
--polling-interval=5 --max-replicas=15 --cooldown-period=15
```

To verify the Deployment, 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 11.20*, this Deployment creates a Deployment, ReplicaSet, and an HPA. In the HPA, you should see that there are no replicas currently running:

user@Azure:~\$ kubectl get all						
NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE	
service/kubernetes	ClusterIP	10.0.0.1	<none>	443/TCP	12d	
NAME	READY	UP-TO-DATE	AVAILABLE	AGE		
deployment.apps/js-queue	0/0	0	0	74s		
NAME	DESIRED	CURRENT	READY	AGE		
replicaset.apps/js-queue-7b5b4c47cf	0	0	0	74s		
NAME			REFERENCE		TARGETS	
MINPODS	MAXPODS	REPLICAS	AGE			
1	15	0	73s	Deployment/js-queue	<unknown>/5 (avg)	

Figure 11.20: The Deployment created three objects, and we have zero replicas running now

7. We will create a message in the queue now to wake up the Deployment 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 open a new cloud shell session. To open a new session, select the *Open new session* button in the cloud shell:



Figure 11.21: Opening a new cloud shell instance

9. On this new shell, run the following command to create a message in the queue.

```
az storage message put --queue-name function --connection-string <your connection string> --content "test"
```

After creating this message, switch back to the previous shell. It might take a couple of seconds, but soon enough, your HPA should scale to 1 replica. Afterward, it should also scale back down to 0 replicas:

NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS	AGE
keda-hpa-js-queue	Deployment/js-queue	<unknown>/5 (avg)	1	15	0	7h48m
keda-hpa-js-queue	Deployment/js-queue	0/5 (avg)	1	15	1	7h49m
keda-hpa-js-queue	Deployment/js-queue	<unknown>/5 (avg)	1	15	0	7h49m

Figure 11.22: With one message in the queue, KEDA scales from 0 to 1 and back to 0 replicas

We have now created a function that is triggered on the number of messages in a queue. We were able to verify that KEDA scaled our Pods from 0 to 1 when we created a message in the queue, and back down to 0 when there were no messages left. In the next section, we will execute a scale test, and we will create multiple messages in the queue and see how the functions react.

Scale testing functions

In the previous section, we saw how functions reacted when there was a single message in the queue. In this example, we are going to send 1,000 messages into the queue and see how KEDA will first scale out our 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, we are going to open a new cloud shell session. To open a new session, select the *Open new session* button in the cloud shell:



Figure 11.23: Opening a new cloud shell instance

3. To send the 1,000 messages into the queue, we have provided a Python script called **sendMessages.py**, in the code bundle. Cloud Shell already has Python and pip (the Python package manager) installed. To be able to run this script, you will first need to install two dependencies:

```
pip3 install azure
pip3 install azure-storage-blob==12.0.0
```

When those are installed, open the **sendMessages.py** file:

```
code sendMessages.py
```

Edit the storage connection string on line 4 to your connection string:

```
1  from azure.storage.common import CloudStorageAccount
2  from azure.storage.queue import Queue, QueueService, QueueMessage
3
4  queue_service = QueueService(connection_string="DefaultEndpointsProtocol=https;AccountName=hoafunc;
5
6
7  for i in range(1, 1000):
8      messagename="test"
9      queue_service.put_message("function", messagename + str(i))
10     print ('Successfully added message: ', messagename + str(i))
11
```

Figure 11.24: Pasting in your connection string for your storage account on line 4

4. Once you have pasted in your connection string, you can execute the Python script and send 1,000 messages to your queue:

```
python3 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 maximum of 15 replicas. The HPA uses metrics provided by KEDA to take the scaling decisions. Kubernetes, by default, doesn't know about the number of messages in an Azure storage queue that KEDA provides to the HPA.

Once the queue is empty, KEDA will scale back down to 0 replicas:

NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS	AGE
keda-hpa-js-queue	Deployment/js-queue	<unknown>/5 (avg)	1	15	0	7h56m
keda-hpa-js-queue	Deployment/js-queue	219/5 (avg)	1	15	1	7h58m
keda-hpa-js-queue	Deployment/js-queue	38500m/5 (avg)	1	15	4	7h58m
keda-hpa-js-queue	Deployment/js-queue	33875m/5 (avg)	1	15	8	7h58m
keda-hpa-js-queue	Deployment/js-queue	21667m/5 (avg)	1	15	15	7h59m
keda-hpa-js-queue	Deployment/js-queue	4734m/5 (avg)	1	15	15	7h59m
keda-hpa-js-queue	Deployment/js-queue	<unknown>/5 (avg)	1	15	0	7h59m

Figure 11.25: KEDA will scale from 0 to 1, and the HPA will scale to 15 Pods

This concludes our examples of running serverless functions on top of Kubernetes. Let's make sure to clean up our Deployments. Run the following command from within the development machine we 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 scaled object js-queue  
kubectl delete deployment js-queue  
func kubernetes remove --namespace keda  
az vm delete -g rg-handsonaks -n devMachine
```

Note

The deletion of KEDA will show a couple of errors. That is because we only installed a subset of KEDA on top of our cluster, and the removal process tries to delete all components.

In this section, we ran a function that was triggered by messages on a storage queue on top of Kubernetes. We used a component called KEDA to achieve scaling in our cluster. We saw how KEDA can scale from 0 to 1 and back down to 0. We also saw how the HPA can use metrics provided by KEDA to scale out a Deployment.

Summary

In this chapter, we deployed serverless functions on top of our Kubernetes cluster. To achieve this, we first created a development machine and an Azure Container Registry.

We started our 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, we installed an additional component on our Kubernetes cluster called KEDA. KEDA allows serverless scaling in Kubernetes: it allows Deployments to scale from 0 Pods, and it also provides additional metrics to the **Horizontal Pod Autoscaler (HPA)**. We used a function that was triggered on messages in an Azure storage queue.

This chapter also concludes the book. Throughout this book, we've introduced AKS through multiple hands-on examples. The first part of the book focused on getting applications up and running. We created an AKS cluster, deployed multiple applications and learned how to scale those applications.

The second part of the book focused on the operational aspects of running AKS. We discussed common failures and how to solve them, integrated applications with Azure AD, and looked into the monitoring of the cluster.

In the final part of the book, we looked into advanced integration of AKS with other Azure services. We integrated our AKS cluster with Azure databases and Azure Event Hubs, we secured our cluster, and finally, we developed Azure Functions on top of our AKS cluster.

As a result of finishing this book, you should now be ready to build and run your applications at scale on top of AKS.



Microsoft.Source Newsletter - Inbox

Message

Microsoft

Microsoft.Source Newsletter | Issue 7

You're reading Microsoft.Source, the developer community newsletter featuring ideas and projects from your peers down the street—and around the world. If someone forwarded you this newsletter and you want to receive future editions, [sign up >](#)

[Give feedback](#) Get more of what you want in each edition.

Featured Story

Vanilla JS and HTML –No frameworks, no libraries, no problem >
Do you know what it takes to render HTML elements without the complexity of AngularJS, React, Svelte, or Vue.js? See how to create a simple web page with pure HTML, CSS, and JS.
Web, JavaScript, HTML

What's New

Build a web experience to send GIFs to MXChip >
IoT, project

The Making of Azure Mystery Mansion >
Game, Twine, PlayFab

Trying to make FETCH happen >
Serverless, IoT, Azure Functions

Events [See all events](#)

Cosmos DB Live Webcast / Online >
Expert-led, containers, .Net

OpenHack Serverless / Los Angeles >
In-person event, serverless, hack

Learning

Microsoft Ignite – Watch videos on demand >
Watch all keynotes, announcements, and sessions on demand

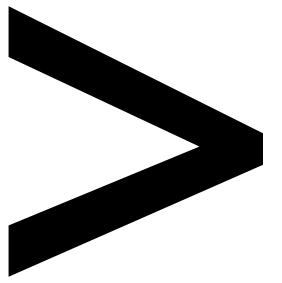
By developers, for developers

Microsoft.Source newsletter

Get technical articles, sample code, and information on upcoming events in Microsoft.Source, the curated monthly developer community newsletter.

- Keep up on the latest technologies
- Connect with your peers at community events
- Learn with hands-on resources

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

access: 10, 17, 34-35, 39, 41, 65, 67, 75, 78, 81, 142-143, 159, 173, 188, 200, 206-207, 211, 224, 236, 238, 253, 257, 263-267, 273-277, 282, 289, 294-295, 298-300, 302, 304, 311, 321-322, 324, 328, 332
accessible: 41, 73, 88, 146-147, 183
acme-v: 157
acrname: 322
add-on: 142, 147-148
address: 16, 31, 40, 46, 60, 62, 72, 89, 129, 148-151, 156, 165, 176, 179-180, 223, 247-248, 278
agentpool: 98, 100, 243, 259, 283
aks insights: 198, 206
alerts: 170, 173-174
algorithm: 7
anonymous: 328
apache: 241-243, 319
apigroup: 277-278
apiurl: 284
apiversion: 35-37, 49, 56-57, 61, 64, 66, 70, 77, 91, 145, 151, 153, 156-157, 165-168, 190, 192, 276-278, 287, 290-291, 301, 311, 313
appear: 13, 40, 186
append: 25
apppasswd: 299
approach: 6, 14, 187, 207
architects: 7
attack: 223, 238

auditing: 235
audit log: 212, 222, 233, 235, 236
authentication: 41, 158, 161, 163, 165, 166, 167, 168, 268, 298, 302, 305, 311, 330
authlevel: 328
authority: 141
authorization: 140, 157, 168
auth-url: 168
autoscaler: 86-87, 91-92, 95, 97-98, 100, 111, 123, 318, 320, 329, 340
autoscales: 91
azurecr: 329, 337
azure functions: 207, 315, 316, 317, 318, 319, 320, 324
azure monitor: 15, 171, 194, 198, 199, 201, 204, 206
azure-vote: 35, 38, 41, 276

B

backend: 50, 57-58, 61, 64, 145, 153, 157, 167-168, 215, 318
backup: 214, 224-225, 227, 233-235, 238
balancer: 17, 39-40, 70-72, 105, 111, 148
bitnami: 77-78, 109, 232, 243
bookinfo: 306, 315
breakers: 7
browser: 21, 40, 73, 89, 93, 141, 144, 156, 158, 169, 183, 186-187, 192-193, 224, 246-249, 279, 324, 329

built-in: 64-65, 67, 142, 214, 283
bypass: 62

C

calculate: 120
callback: 163
cert-manager: 140, 146
cgroups: 9
charts: 74-75, 148, 215, 243-244, 254
clientid: 218, 266, 299
cloned: 49, 86
cloudapp: 153, 157, 163, 165, 167-169
cluster: 3, 17-18, 21-32, 34-35, 40-41, 43, 46-47, 54, 60, 62, 64-65, 67-68, 71-74, 76, 78-79, 82, 86-87, 90-91, 94-101, 109, 111, 113-115, 117-118, 120-121, 123-125, 133-138, 141, 143, 146-148, 151, 156, 158-160, 164, 168-170, 173-174, 196, 198-205, 208, 211, 213-217, 219-220, 224-225, 230, 235, 238, 242-243, 249-251, 254-255, 259-260, 263-266, 270-271, 273-276, 278, 281-283, 289, 294, 296, 299-300, 302-303, 305, 311, 313, 315, 318-322, 325, 330, 336, 340-341
clusterip: 62, 68, 88, 101, 106, 115, 183
compliance: 214
conduit: 303
cfgmap: 57

console: 23
consul: 303
container: 3-4, 9-10,
12-15, 46, 48, 51, 54,
59-60, 109, 123-124, 131,
137, 166, 179-181, 184,
188, 191, 193, 195-196,
200, 206-208, 232-233,
289-290, 292-294, 303,
320-322, 326, 329, 340
controller: 143-144,
153, 158, 205, 232
cowsay: 10, 12
crashes: 179
credential: 266-267, 299
critical: 7, 54, 85, 173, 264

D

daemonset: 205
dangerous: 264
dashboards: 304
database: 15, 17, 48, 75,
109-110, 186-187, 208,
211, 213-215, 220-230,
232-233, 235-236, 238,
242-243, 247, 249, 295
databricks: 242
datacenter: 319
datadoghq: 48
deadlocks: 188
decode: 283, 286, 294
deleterole: 277-278
demand: 6, 85, 111, 317-318
deploy: 4, 7, 18, 22, 25, 35,
41, 43, 45-49, 55, 60, 62,
65-66, 73-74, 114-115,
135, 173, 191, 216-220,
230, 242-243, 245,
254, 260, 275-276, 315,
318-320, 329, 337, 340

describe: 52, 55-56, 94,
101, 107-110, 122-123,
125-126, 134, 136-137,
155, 174, 177, 180,
182, 195-197, 199,
230, 285, 293, 307

design: 7, 15, 82

devops: 5, 7-9, 100

diagnose: 120, 123

directory: 17, 23, 35,
86, 138, 142, 152, 157,
160, 164-165, 218,
267-269, 271, 292,
305, 326, 328, 333

disaster recovery:

212, 222, 232

diskuri: 127

docker: 3-6, 9-14, 18,
39, 43, 50-51, 59-60,
109, 166, 179, 182, 284,
288-289, 293-294,
319-320, 322-324, 326

dockerfile: 12-13, 326

dockerid: 294

dockerized: 18

downgrade: 102, 330

E

encrypt: 141-144, 147-148,
151-152, 154, 156,
167, 170, 304, 307

enforcing: 307

errors: 123, 135, 170, 174,
181-184, 188, 192, 340

eventhub: 255

event hubs: 209, 236,
239, 240, 247, 248, 249,
252, 254, 255, 257, 258

events: 56, 103, 132, 136,
173, 176-177, 179-180,
182, 236, 241-243, 249,
254, 257, 260, 320, 330
extensions: 54, 91, 145,
153, 157, 165, 167-168

F

facebook: 142, 159-160

facilitate: 21

failbacks: 100

failover: 4, 91, 111

failure: 113-114, 117, 119,
123, 130, 132-133,
138, 183, 192, 196

filter: 26, 72, 103

filtering: 103, 208

firewall: 224

flexvolume: 295, 298-302

flowchart: 146

forbidden: 282

framework: 7, 188, 307, 319

friend: 242, 256-257

front-end: 68, 70-71, 85,
89-90, 102-103, 107,
145-147, 156, 168, 187

functions: 41, 211, 242, 289,
315, 317-322, 326, 328,
330, 335, 338, 340-341

fqdn: 60, 145, 146, 154

G

gateway: 17, 143, 168

gibibyte: 121

gigabyte: 121

github: 9, 35, 49, 75, 86, 93, 114, 142, 148, 153, 159-160, 168-169, 215, 243, 295, 299-300, 303, 319
gitlab: 9, 142, 160
google: 4, 9, 14, 21, 64, 100, 142, 147, 159-160, 169, 304, 319
googleapis: 75, 143, 216, 243
gopath: 93
graphs: 257-258
grepping: 110
guestbook: 46-48, 60, 66-68, 70-73, 75, 82, 86-87, 89-90, 97, 101-102, 105, 109, 114-115, 118-120, 123-124, 142, 144-147, 158-160, 170, 174, 179, 181, 183-185, 187-188, 208

H

hackers: 315
handsonaks: 25, 34, 98, 100, 243, 259, 266, 283, 322-325, 329
hashicorp: 22
helmsman: 263
high availability: 45, 61, 211, 212, 222, 223
hpa: 84, 89, 90, 91, 92, 93, 109, 318, 336, 337, 338
http-based: 17
httpbin: 308, 315
httpget: 190

I

images: 9-10, 12, 48, 54, 103, 182, 283, 289, 320-321, 326
import: 297
incubator: 243
ingress: 17, 142-149, 152-154, 156-158, 160, 164, 166-170, 289, 303
inject: 309
injected: 152, 307-309, 315
invoke: 294
issuer: 147, 151-154, 156-157, 160, 167, 170
istio: 258, 262, 300, 301, 302, 303, 304, 305, 306, 310, 311, 313
istioctl: 305, 308-309, 315

J

javascript: 7, 333
jenkins: 9
jeopardize: 263
jetstack: 148

K

kanban: 8
keda: 318, 328, 334, 335, 336, 337, 338
kafka: 239, 240, 247, 248, 249, 252, 254, 257, 258
keyvault: 299, 301
keyword: 23

kubectl: 34, 38-41, 49, 52-53, 55-57, 59-63, 65-66, 71, 73, 75-76, 78-80, 82, 86-92, 94-95, 97-111, 114-116, 118, 120-128, 130-138, 144, 146-148, 151, 154-155, 158, 160, 166-167, 169, 173-177, 180-184, 187-188, 190-200, 208, 216, 218, 220-221, 223, 226, 230-233, 238, 244-247, 249, 256-257, 259, 276-278, 281-282, 284-285, 287-289, 291-295, 299, 302, 306-308, 310, 312, 314-315, 322, 325, 329, 336-338, 340
kubeless: 319

L

linkedin: 142, 160, 241
localhost: 15, 185, 187

M

manually: 14, 51, 74, 86-87, 91, 95-97, 100, 111, 136, 138, 153, 207, 286-287
mapping: 148, 156
mariadb: 75-78, 109-110, 133, 213
master: 17, 31, 48-51, 53-54, 57-63, 65, 67, 75, 77, 124, 185, 234

maxmemory: 54-56, 59
meshpolicy: 311-312
middleware: 318
monolithic: 241-242
mountpath: 58, 77,
 190-191, 292, 301
multi-tier: 47, 73

N

namespace: 15, 56, 62,
 82, 148, 166, 175, 177,
 179, 181, 198, 216, 218,
 244-245, 251, 265,
 275, 277-278, 282-284,
 306-309, 311, 313
navigate: 21, 35, 41,
 86, 88, 106, 115, 187,
 231-232, 243, 326
network: 6, 15, 17, 60, 73,
 75-76, 186, 222, 230,
 242-243, 246-247,
 249, 254, 257, 260,
 302-303, 312
neo4j: 240, 241,
 245, 246, 247
nginx-: 190
nodepool: 98, 100,
 243, 259, 266, 283
nodeport: 69-70
nslookup: 62

O

oauth-: 169
openssl: 266, 289
orchestration: 1, 2, 12

P

package: 46, 74, 82,
 305, 326, 339
padlock: 158
permission: 265,
 267, 270, 282
persistent volume: 123
php-redis: 67, 107-108, 180
pod-id: 59, 62, 122,
 134, 136-137, 307
policy: 309, 311-314
polyglot: 7
powershell: 22, 32
probes: 173, 176, 188-189,
 192, 194, 196
proxies: 60, 65
public: 3-4, 10, 12, 39-40,
 70-73, 88-89, 102,
 105-106, 115-116, 127,
 143, 147-149, 151, 159,
 183, 185, 238, 247,
 318-319, 323, 329
pusher: 160, 165, 169
pvc: 74, 76, 77, 128, 131
python: 7, 326-327, 339

Q

queries: 207, 235

R

rbac: 15, 157, 216, 258,
 261, 262, 263, 264,
 268, 276, 277, 278,
 279, 281, 292, 313
rakyll: 93
readonly: 276-277,
 292, 301

real-time: 242, 250
real-world: 330
redeploy: 105, 254
redhat: 166
redundancy: 225
registry: 10, 12, 180,
 283-284, 288-289,
 320-322, 329, 337, 340
replica: 50, 52, 68, 94,
 184, 234-235, 338
repository: 10, 12, 35, 49,
 148, 166, 168, 180, 243
reschedule: 120
research: 141
resource: 15, 22, 24-25,
 30, 87, 91, 107,
 136-137, 174, 200, 216,
 238, 243, 251, 274,
 283-284, 287, 332
resources: 15, 17, 22,
 36-37, 39, 41, 50-51,
 58, 63-64, 67, 72-73,
 77, 85, 94-95, 97-101,
 107, 114, 120-123, 135,
 138, 152, 154, 159,
 169, 174-176, 179-180,
 196-198, 213, 215, 235,
 254, 264-265, 276-277,
 283, 295, 315, 320, 337
restart: 33, 138, 188,
 195-196, 219, 232
restore: 194, 225-229,
 233-234, 238
return: 32, 60, 62, 80,
 92, 155, 194, 284
runtime: 9-10, 14,
 293-294, 322, 329

S

sas: 330
scalable: 4, 6, 43, 85, 250
scaled-out: 99
scale-down: 94, 97
scaling: 4, 16, 31, 43, 47, 82, 85-87, 89-98, 115, 121, 123, 184, 214, 216, 238, 266, 318, 320, 330, 337, 339-340
script: 22, 107, 193, 266, 269, 309, 312, 339
secret: 54, 109, 128, 154, 162, 164-165, 218, 226, 230-232, 255, 260, 267, 283-295, 297-302, 329, 340
server: 3, 5, 10, 31, 61-62, 109, 141, 151-152, 156-157, 180, 185-187, 189-196, 222, 229, 231-232, 234-238, 267, 269, 289, 295
serverless: 211, 289, 315, 317-319, 340
serverless functions: 315, 316, 338
server log: 233, 234, 235, 236
service mesh: 261, 262, 300, 301, 302, 304, 306
set-policy: 299
shutdown: 118, 122
sidecar: 15, 142, 303-304, 306-309, 312, 314
slaves: 62-63
source: 3, 5, 12, 57, 115, 145, 151, 153, 158-159, 164, 167, 170, 299, 306, 328

sprints: 8

ssh-keygen: 322
static: 46, 69, 74, 95
switch: 156, 254, 285, 338-339

T

template: 22, 36-37, 50, 57, 64, 67, 79, 107-108, 165
tenant: 165, 217-218, 301
tenantid: 218, 266, 270, 301
terraform: 22
throughput: 251, 258
thumbnail: 318
tiller: 74
timeout: 136

U

ubuntu: 323, 326
upstream: 165
utility: 219

V

validate: 28
valuable: 236
vmware: 4
vnet rules: 220, 221

W

warning: 144, 244, 312, 325
whalesay: 10, 12
wildcard: 310, 313
worked: 90, 187, 291
wp-config: 231-232
wp-mariadb: 109
wrapper: 50

Y

your-name: 288
your-pword: 288

Z

zookeeper: 243

