# Hands-on Kubernetes on Azure

## Third Edition

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

Nills Franssens, Shivakumar Gopalakrishnan, and Gunther Lenz

# Hands-on Kubernetes on Azure, Third Edition

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

Nills Franssens

Shivakumar Gopalakrishnan

Gunther Lenz

Packt

BIRMINGHAM–MUMBAI

## Hands-on Kubernetes on Azure, Third Edition



**Authors:** Nills Franssens, Shivakumar Gopalakrishnan, and Gunther Lenz

**Technical Reviewers:** Richard Hooper and Swaminathan Vetri

**Managing Editor:** Aditya Datar and Siddhant Jain

**Acquisitions Editor:** Ben Renow-Clarke

**Production Editor:** Deepak Chavan

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

**First Published:** March 2019

**Second Published:** May 2020

**Third Published:** April 2021

**Production Reference:** 3230421

**ISBN:** 978-1-80107-994-5

Published by Packt Publishing Ltd.

Livery Place, 35 Livery Street

Birmingham, B3 2PB, UK.

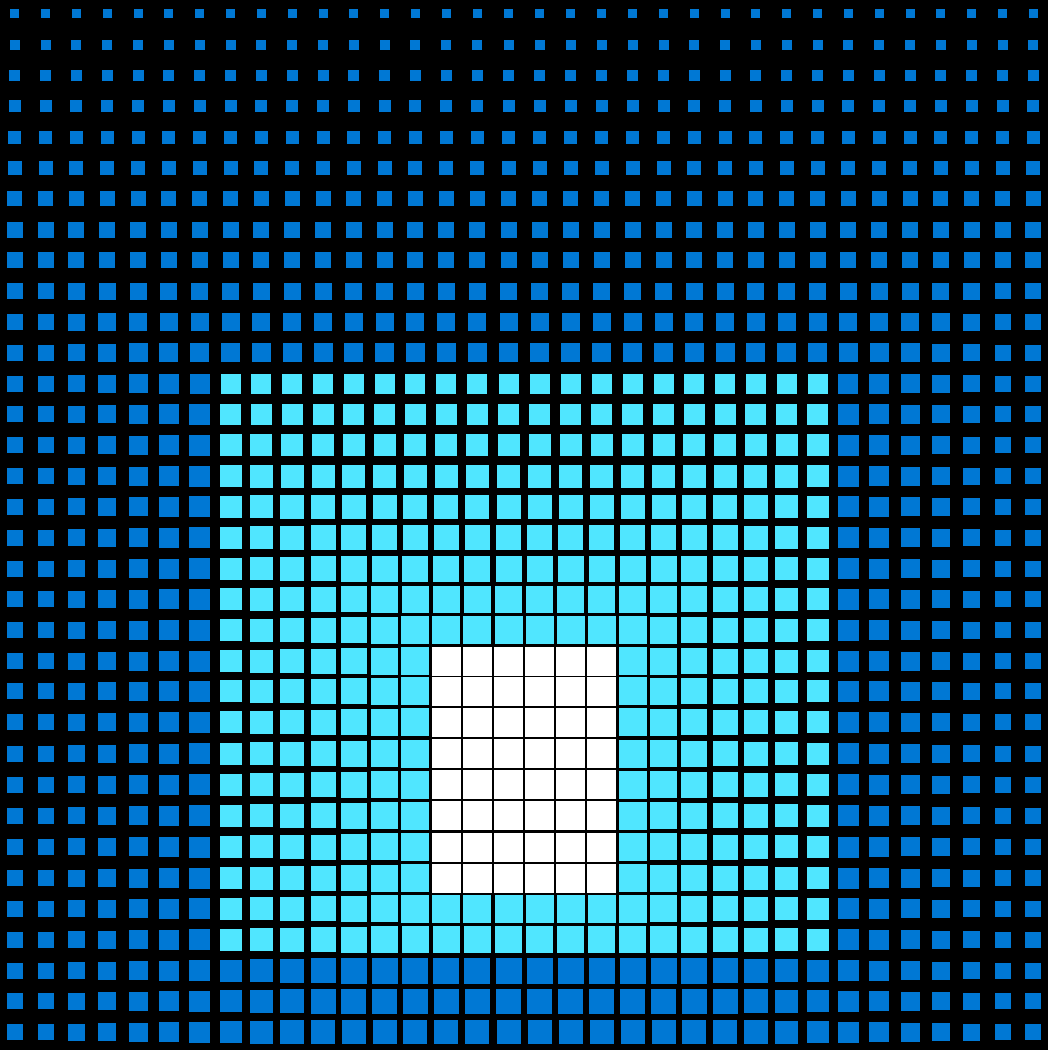
*To mama and papa. This book would not have been possible without everything you did for me. I love you both.*

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

**- Nills Franssens**

Microsoft Azure

kubernetes



Get started

# Kubernetes on Azure

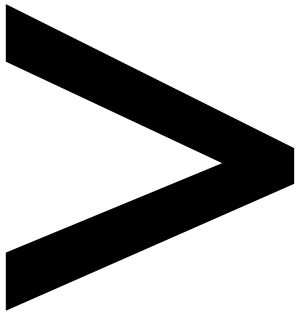
Find out what you can do with a fully managed service for simplifying Kubernetes deployment, management and operations, including:

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

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

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

# Table of Contents



# Preface

**About**

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

## Hands-on Kubernetes on Azure – Third Edition

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

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

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

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

### About the authors

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

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

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

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

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

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

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

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

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

## About the reviewers

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

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

## Learning objectives

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

## Audience

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

## Approach

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

## Hardware and software requirements

### Hardware requirements

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

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

### Software requirements

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

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

## Conventions

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

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

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

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

## Downloading resources

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

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

# Foreword

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

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

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

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

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

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

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

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

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

# Section 1: The Basics

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

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

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

This section contains the following chapters:

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

# 1

# Introduction to containers and Kubernetes

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

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

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

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

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

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

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

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

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

The following topics will be covered in this chapter:

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

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

## The software evolution that brought us here

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

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

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

### Microservices

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

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

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

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

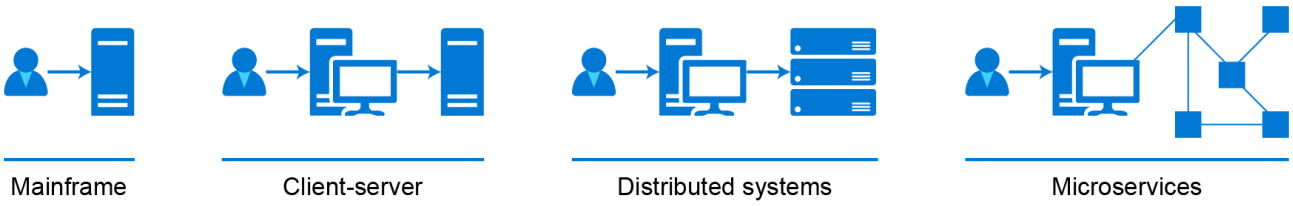



**Figure 1.1: A standard microservices architecture**

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

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

## Advantages of running microservices

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

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

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

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

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

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

## Disadvantages of running microservices

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

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

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

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

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

## DevOps

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

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

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

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

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

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

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

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

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

## Fundamentals of containers

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

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

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

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

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

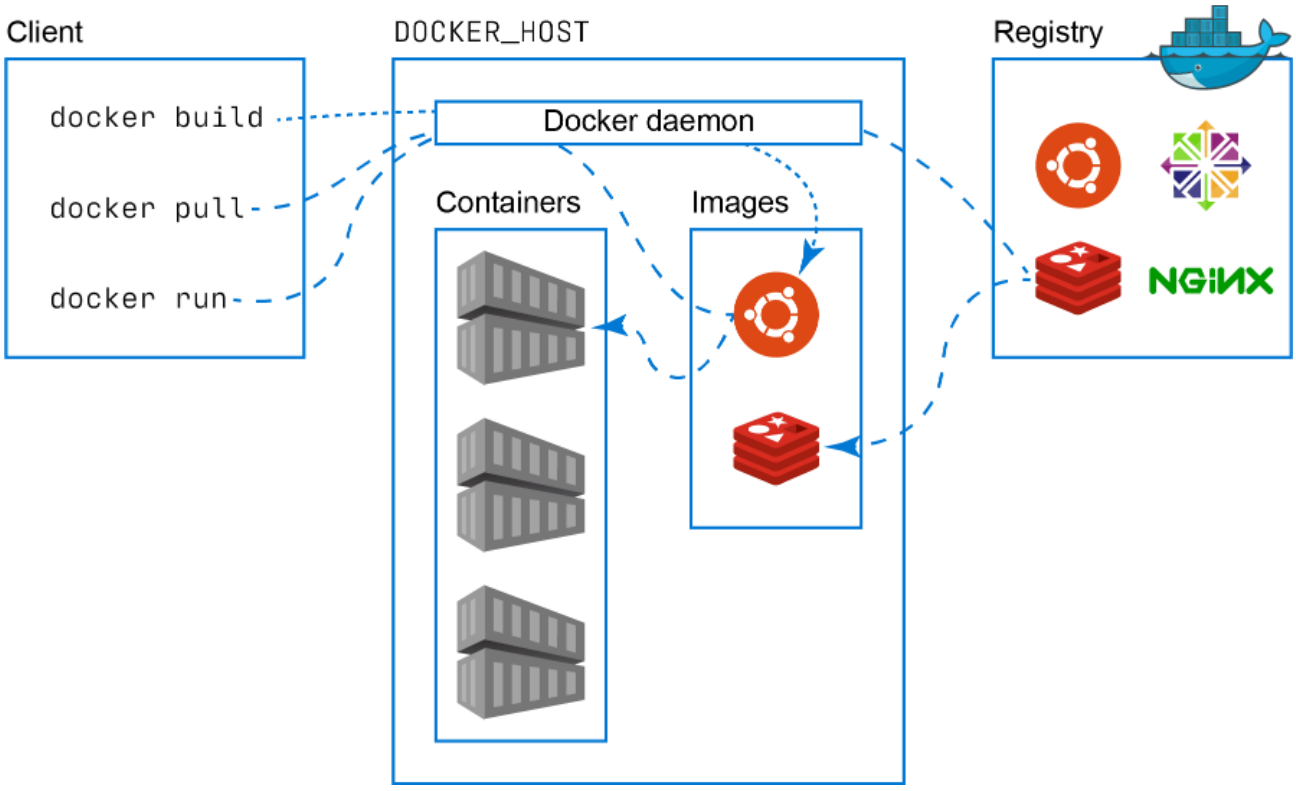



Figure 1.2: Fundamentals of Docker architecture

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

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

## Container images

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

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

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

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

```
#                         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.                                                #
################################################################
[node1] (local) root@192.168.0.8 ~
$ 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 u
pgrade 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
[node1] (local) root@192.168.0.8 ~
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
docker/whalesay     latest              6b362a9f73eb        5 years ago         247MB
[node1] (local) root@192.168.0.8 ~
$ docker run docker/whalesay cowsay boo
 _____
< boo >
 -----
    \
     \
      \
                    ##        .
              ## ## ##       ==
           ## ## ## ##      ===
       /""""""""""""""""___/ ===
  ~~~ {~~ ~~~~ ~~~ ~~~~ ~~ ~ /  ===- ~~~
       \______ o          __/
        \    \        __/
          \____\______/
[node1] (local) root@192.168.0.8 ~
$
```

**Figure 1.3: An example of running containers in Docker Labs**

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

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

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

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

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

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

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

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

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

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


Figure 1.4: Running a custom container

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

## Kubernetes as a container orchestration platform

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

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

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

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

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

## Pods in Kubernetes

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

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

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

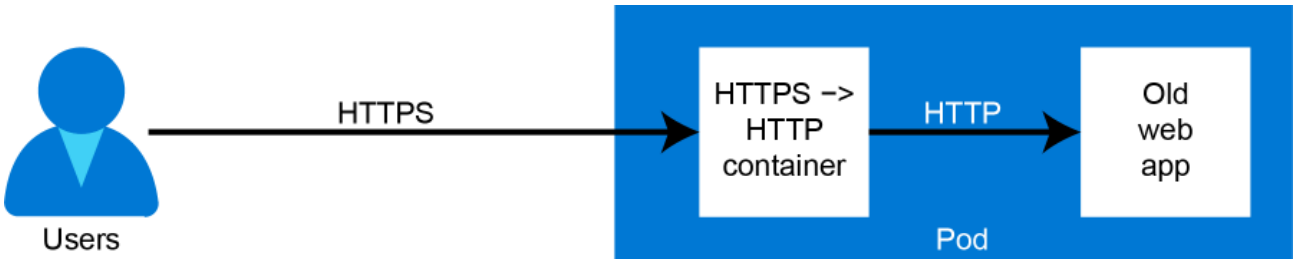



**Figure 1.5: An example of a multi-container pod that does HTTPS offloading**

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

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

## Deployments in Kubernetes

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

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


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

## Services in Kubernetes

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

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

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

## Azure Kubernetes Service

AKS makes creating and managing Kubernetes clusters easier.

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

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

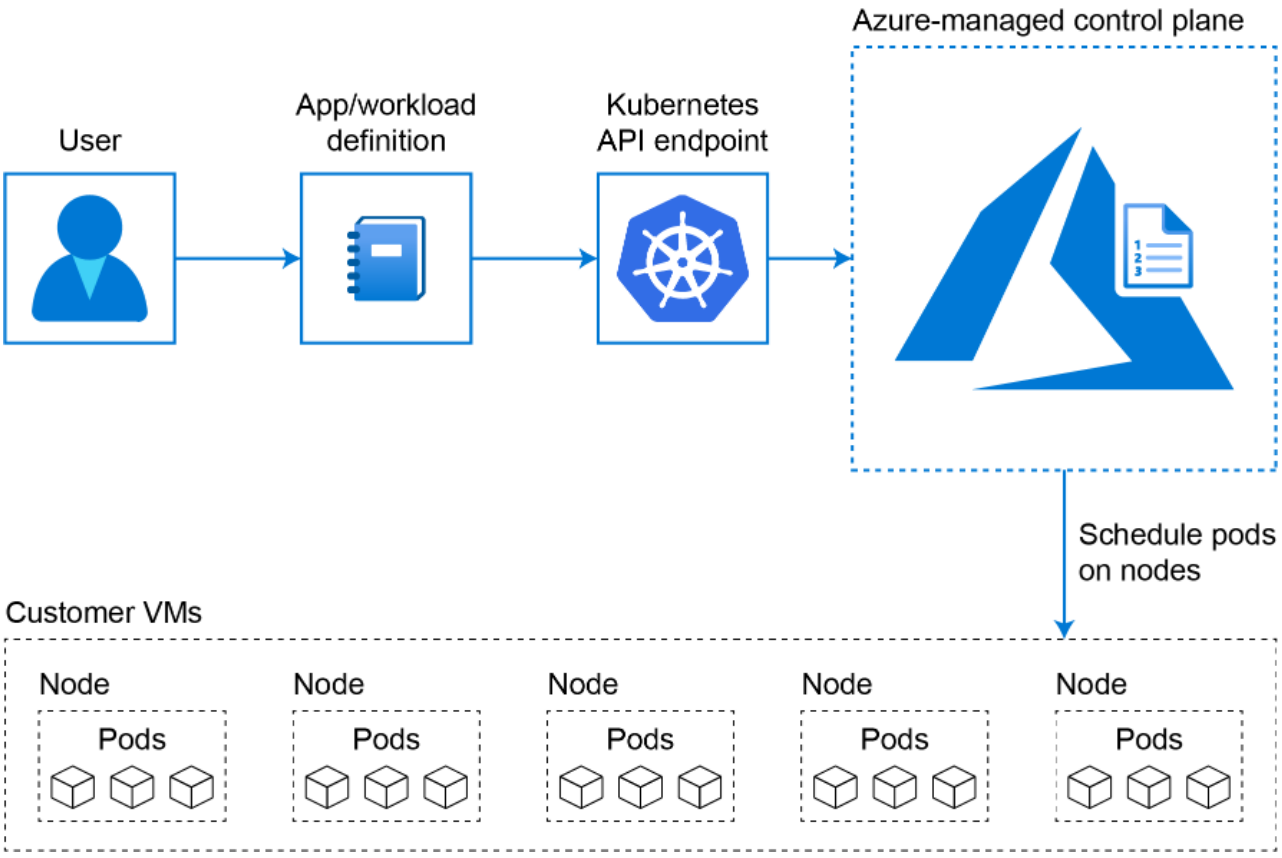



Figure 1.7: Scheduling of pods in AKS

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

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

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

## Summary

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

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

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

# 2

# Getting started with Azure Kubernetes Service

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

The following topics will be covered in this chapter:

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

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

## Different ways to create an AKS cluster

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

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

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

# Getting started with the Azure portal

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

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

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

## Creating your first AKS cluster

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

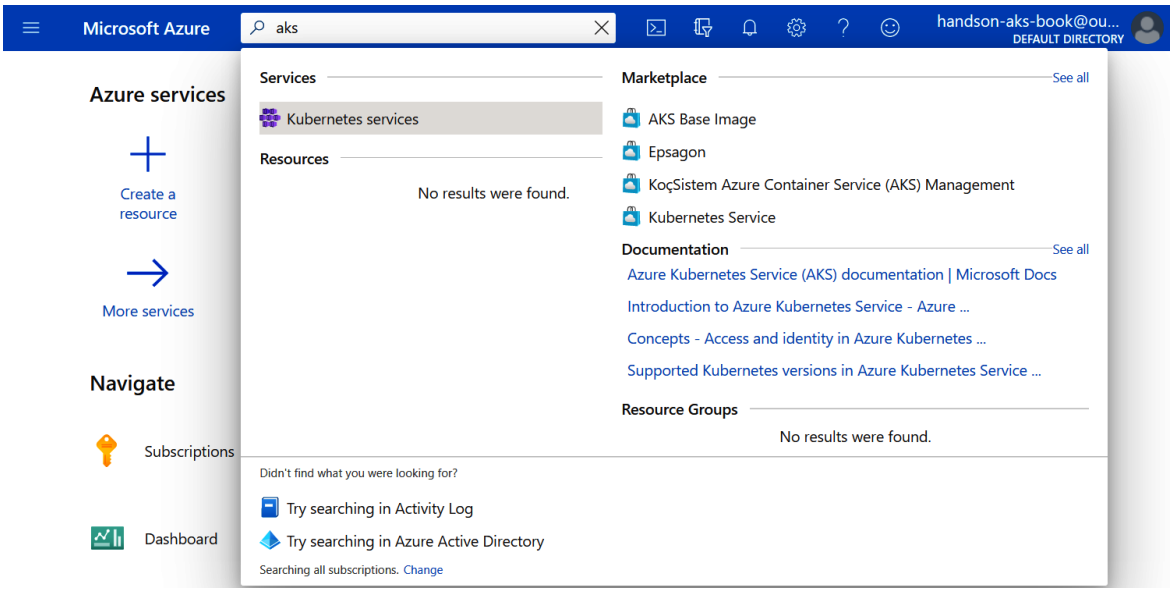



Figure 2.1: Searching for AKS with the search bar

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

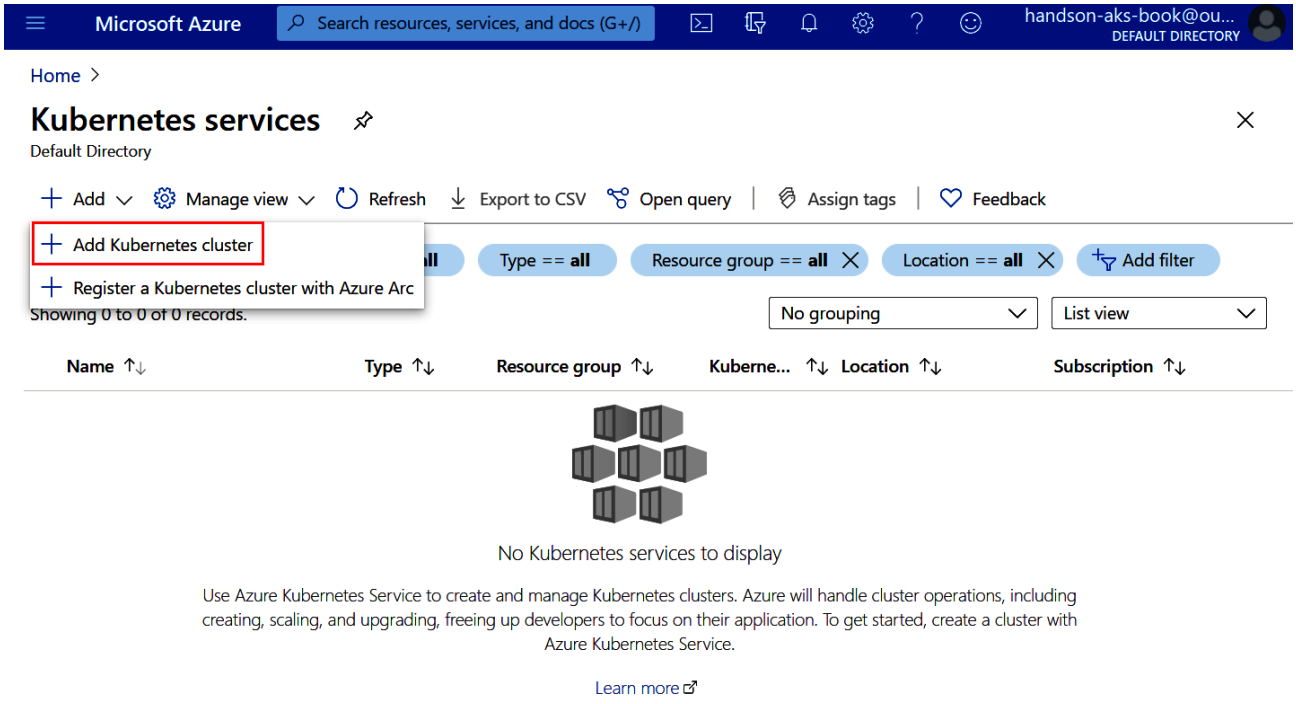



**Figure 2.2: Clicking the + Add button and the + Add Kubernetes cluster button to start the cluster creation process**

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

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

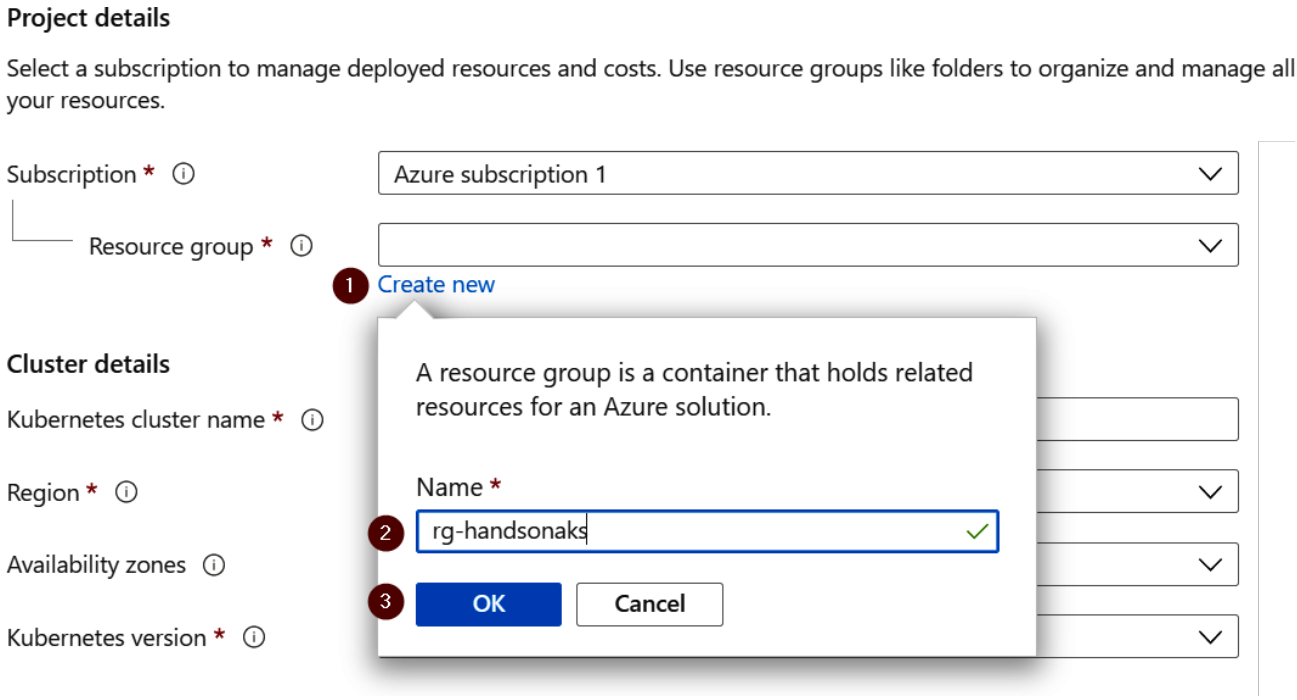



**Figure 2.3: Creating a new resource group**

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

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

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

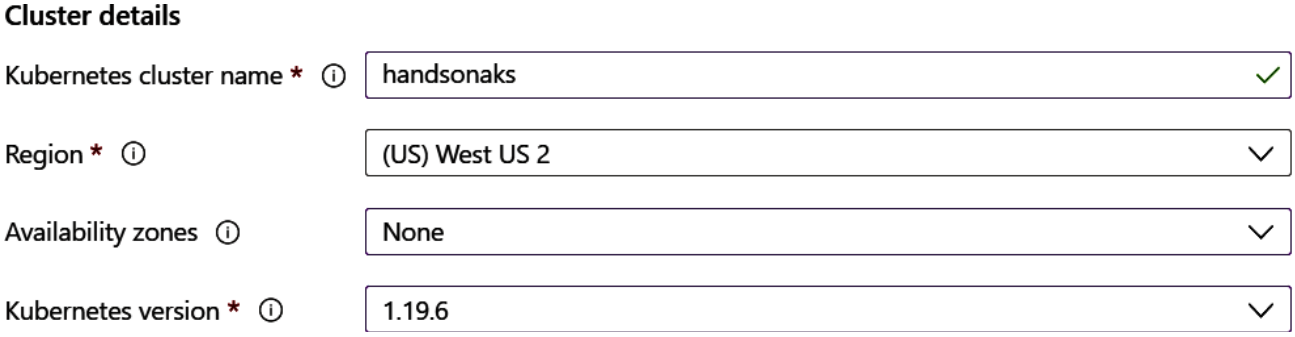



Figure 2.4: Providing the cluster details

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

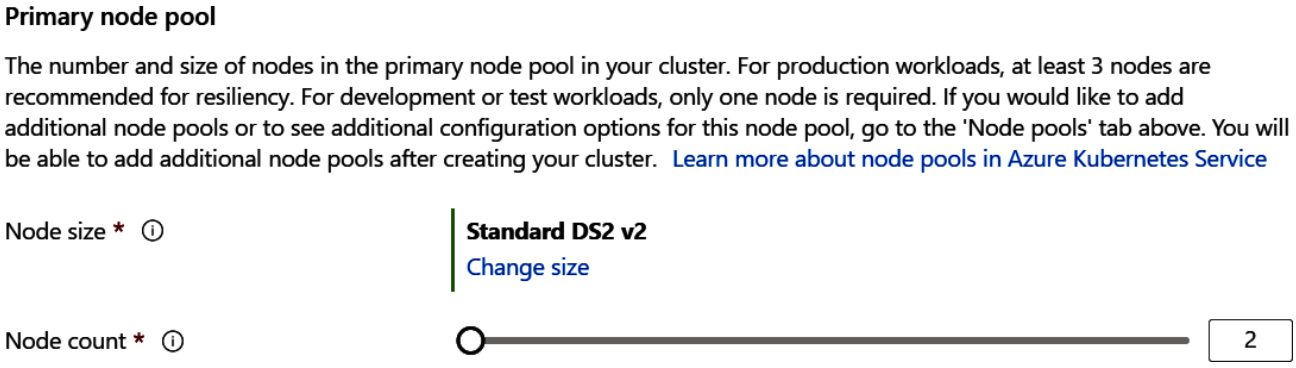



Figure 2.5: Updated Node size and Node count

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

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

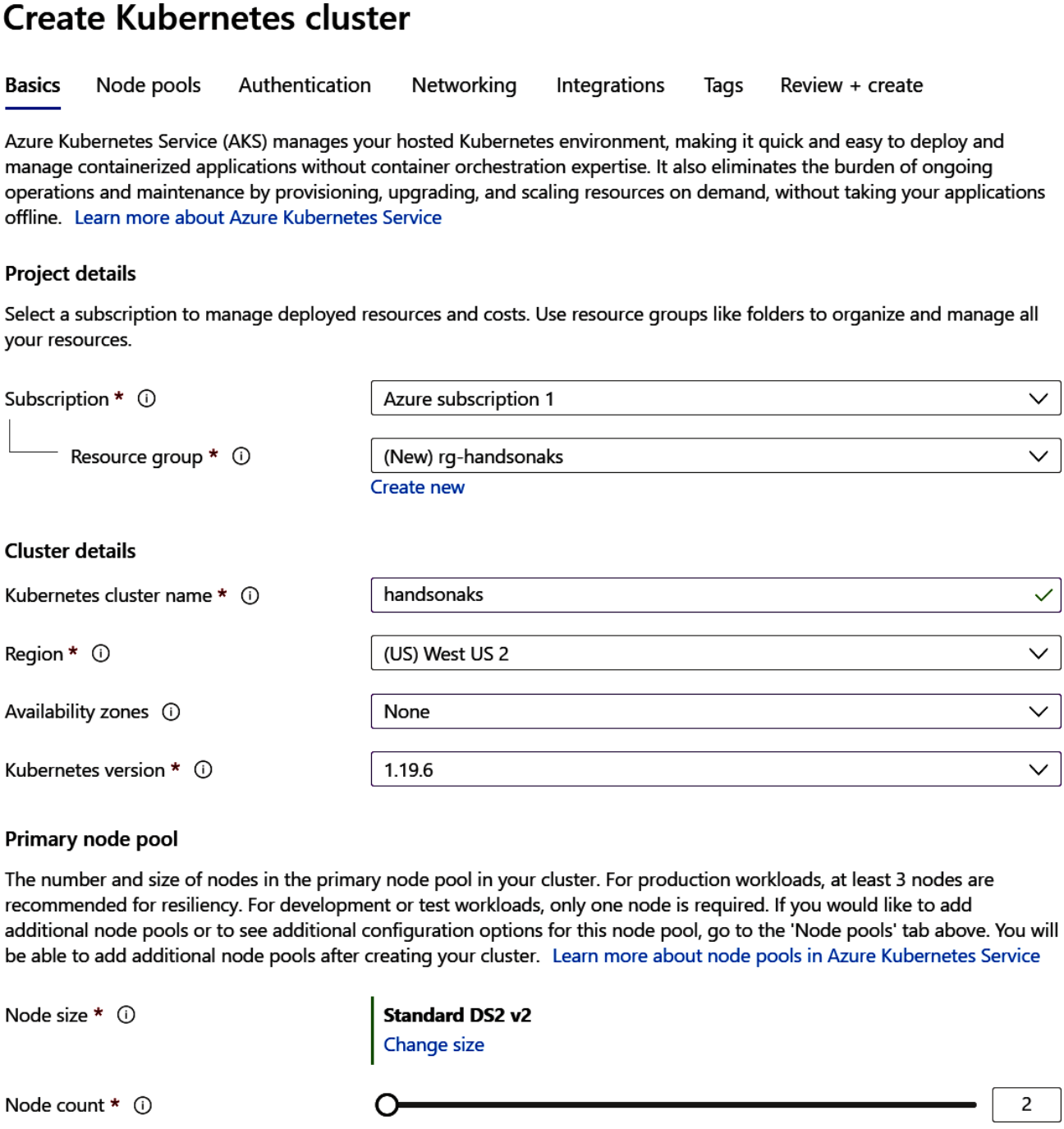



Figure 2.6: Setting the cluster configuration

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

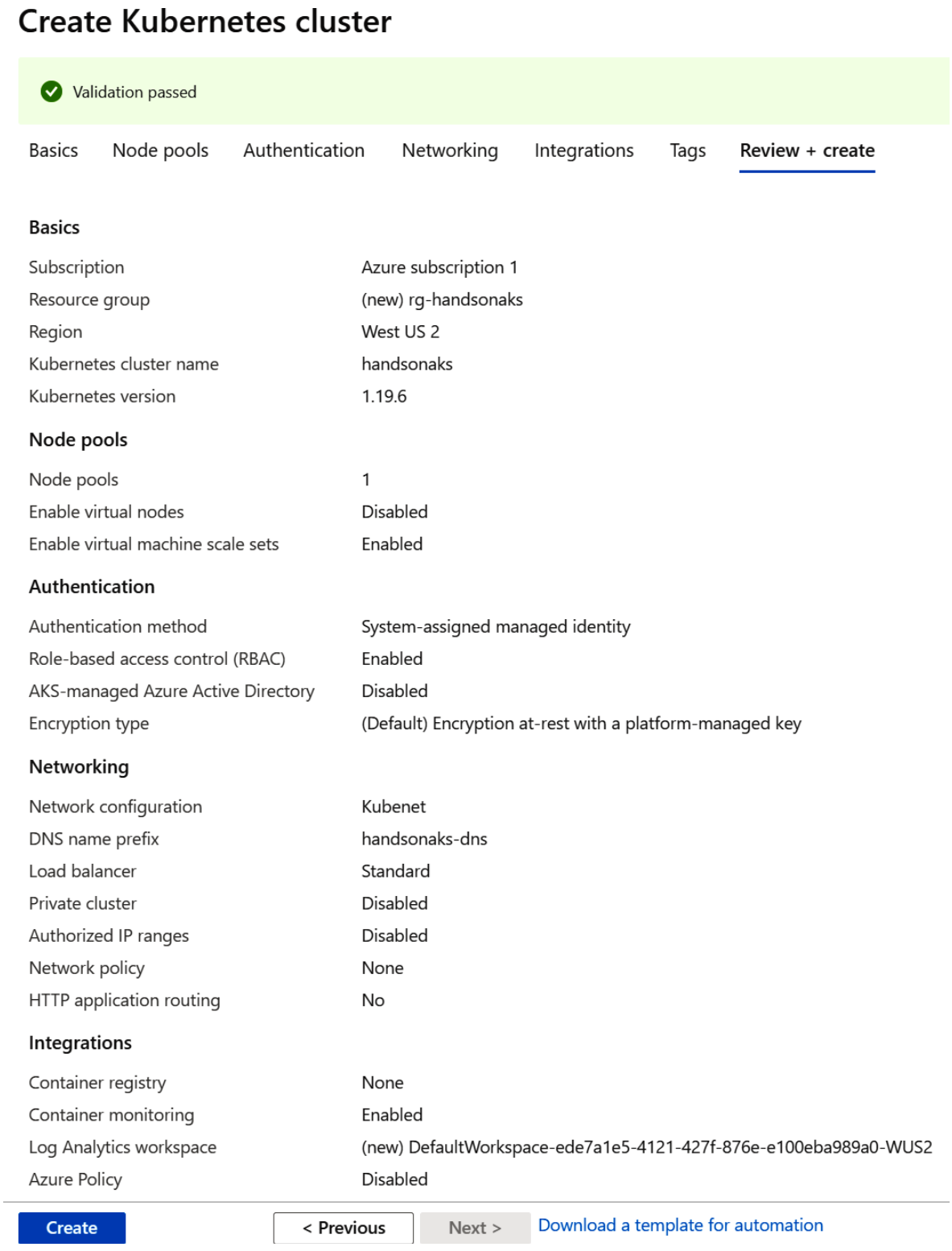



**Figure 2.7: The final validation of your cluster configuration**

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

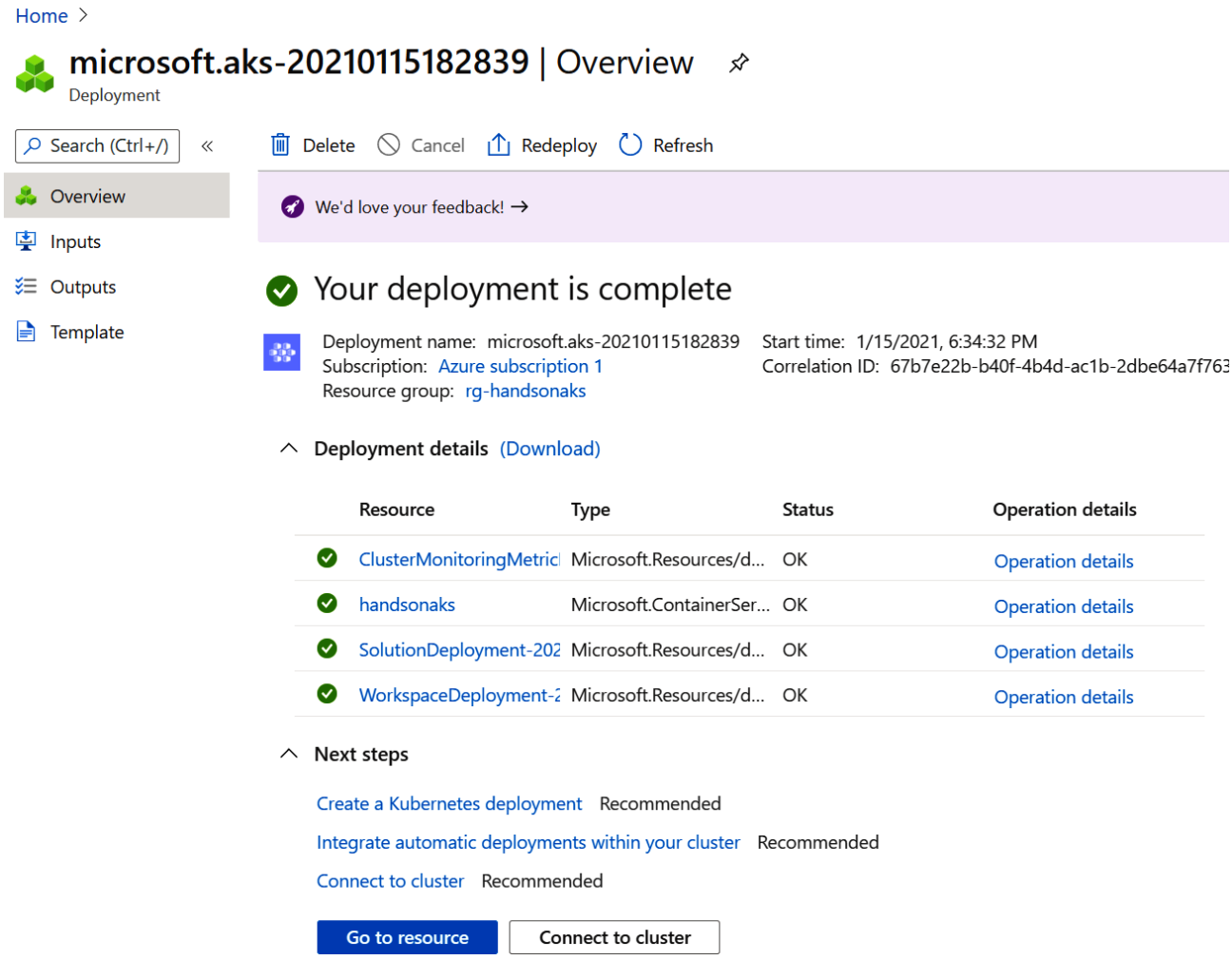



**Figure 2.8: Deployment details once the cluster is successfully deployed**

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

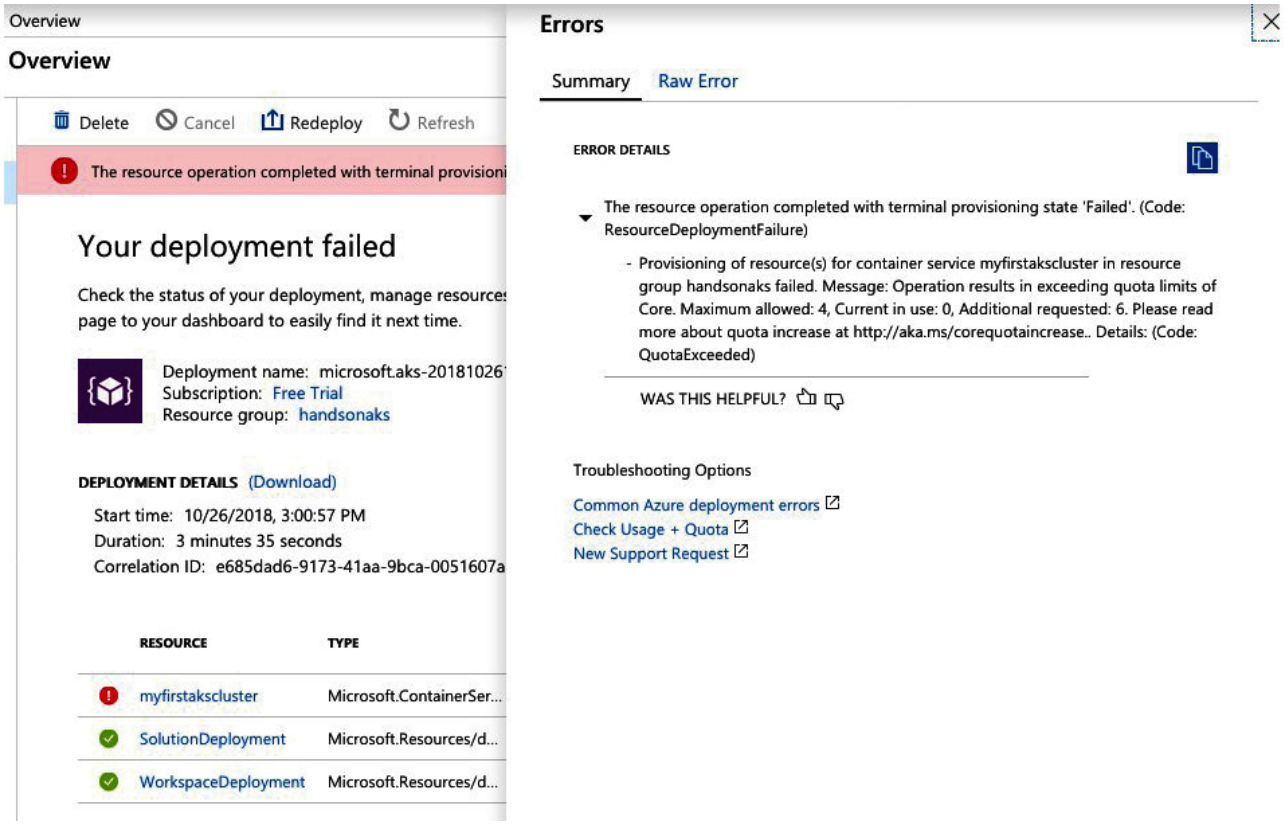



**Figure 2.9: Retrying with a smaller cluster size due to a quota limit error**

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

## A quick overview of your cluster in the Azure portal

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

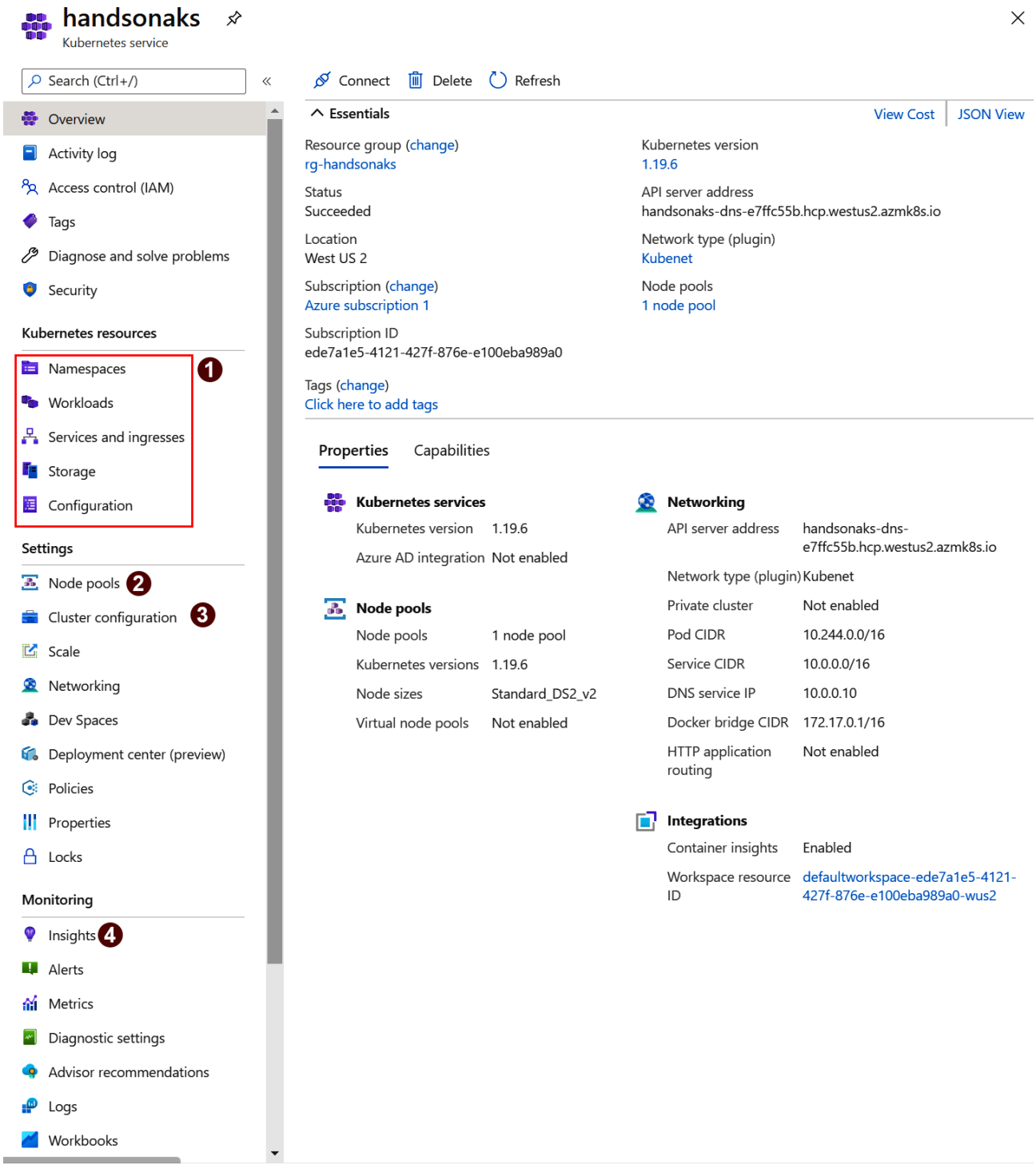



**Figure 2.10: The AKS pane in the Azure portal**

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

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

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

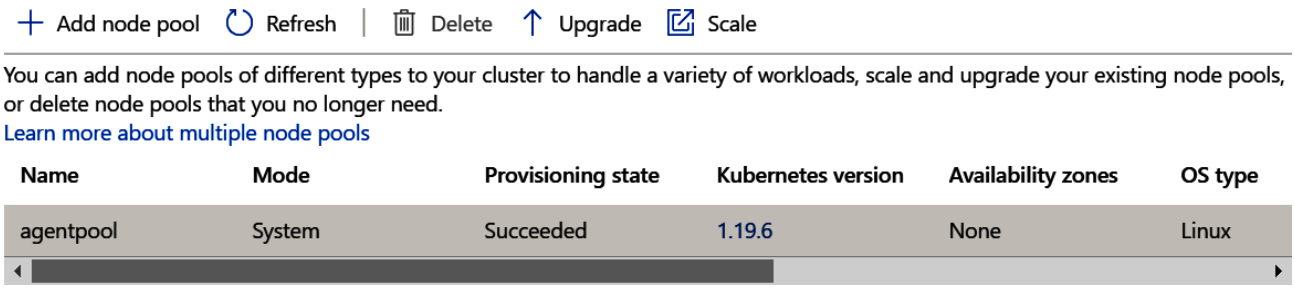



**Figure 2.11: Adding, scaling, and upgrading node pools**

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

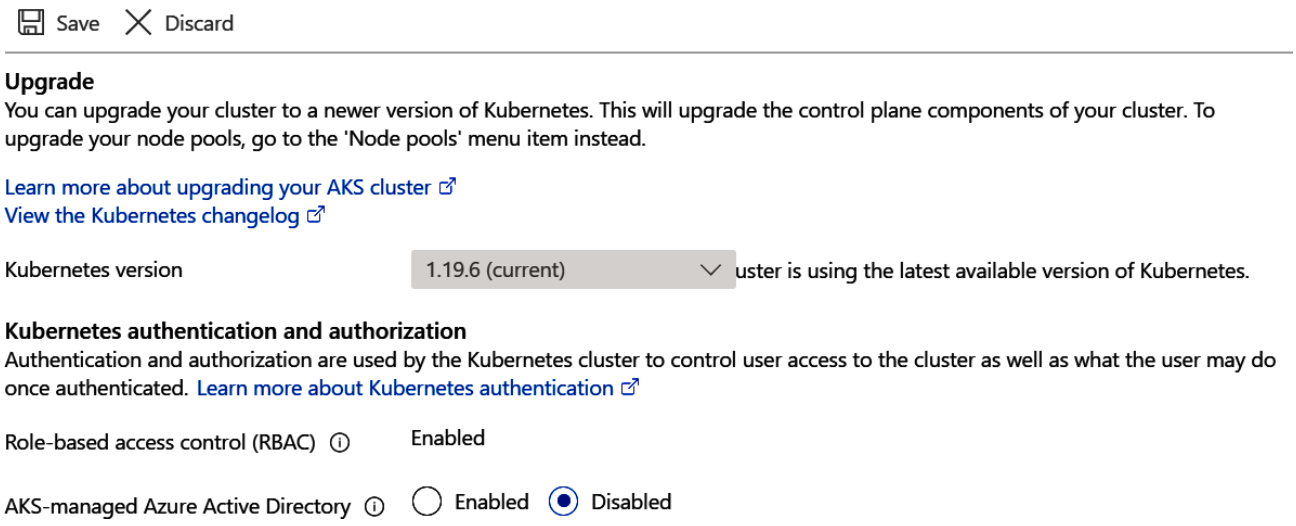



Figure 2.12: Upgrading the Kubernetes version of the API server using the Upgrade pane

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

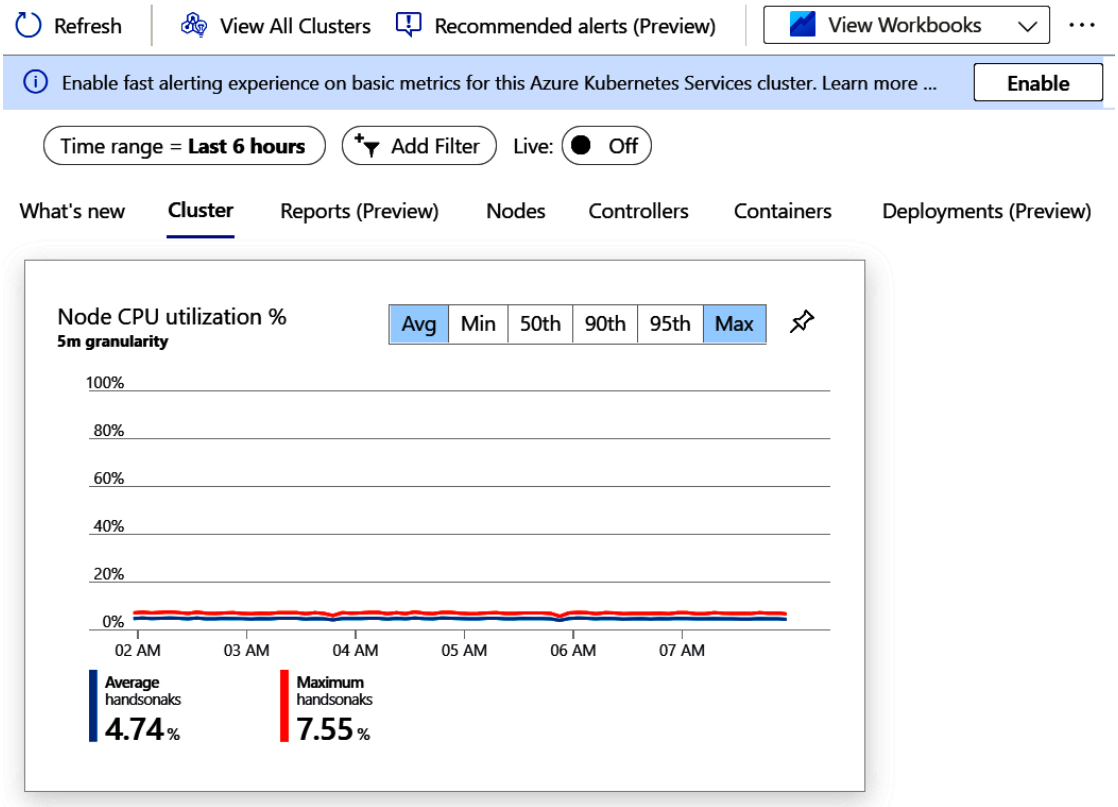



Figure 2.13: Displaying cluster utilization using the Insights pane

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

## Accessing your cluster using Azure Cloud Shell

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

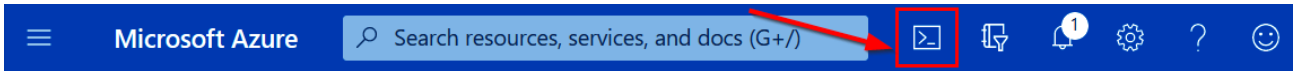



**Figure 2.14: Clicking the Cloud Shell icon to open Azure Cloud Shell**

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

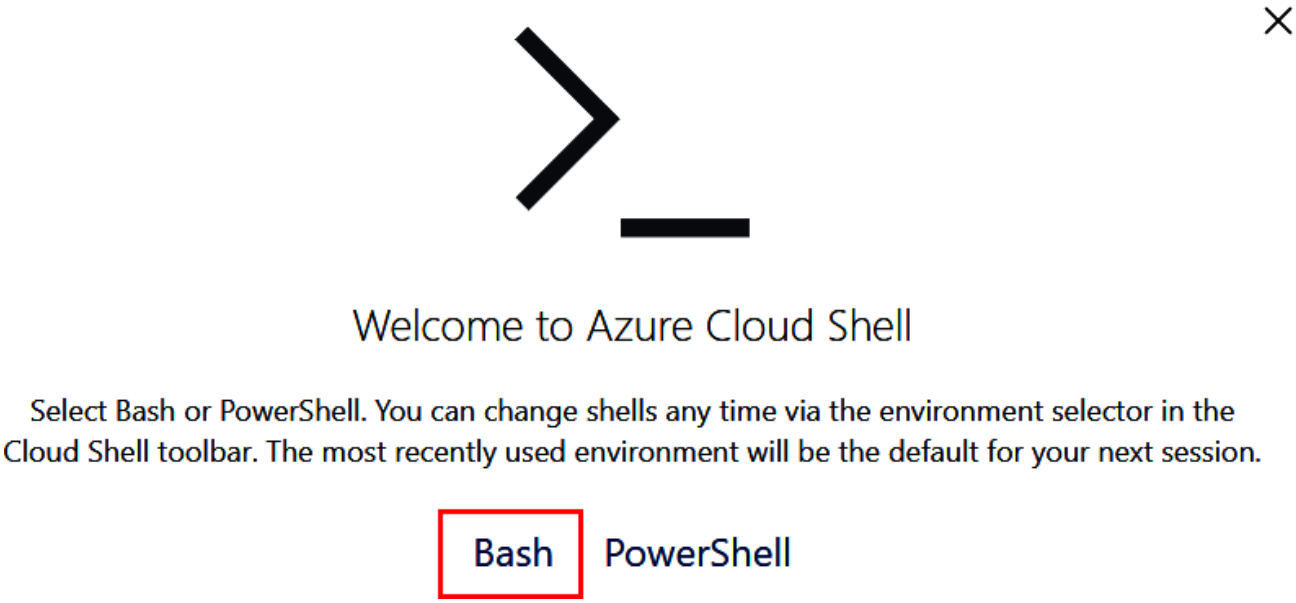



**Figure 2.15: Selecting the Bash option**

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

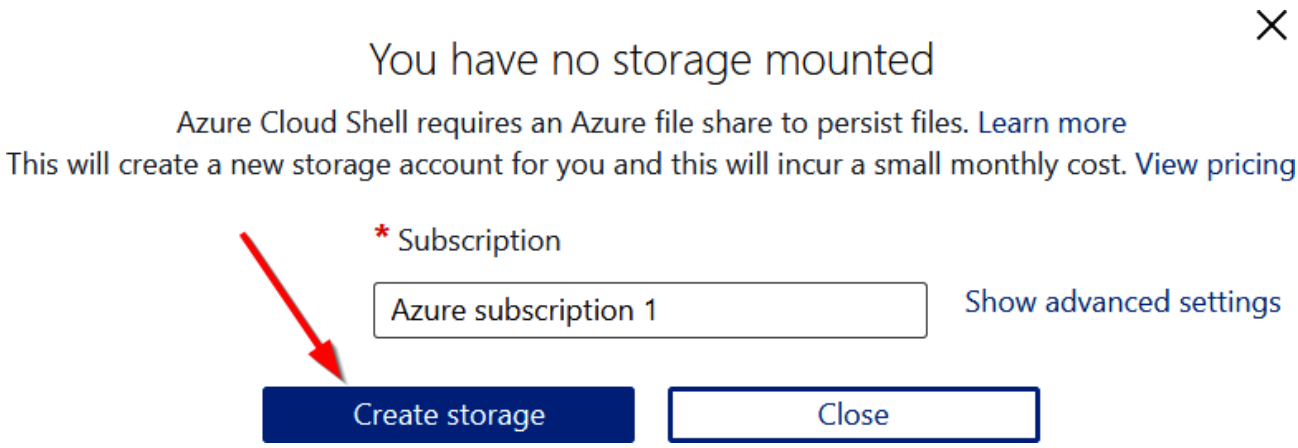



**Figure 2.16: Creating a new storage account for Cloud Shell**

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

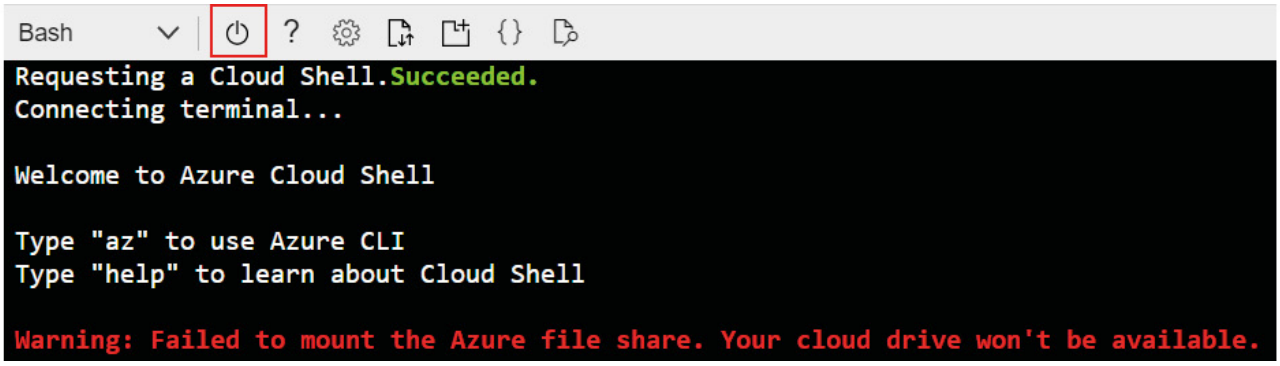



**Figure 2.17: Hitting the restart button upon receiving a mount storage error**

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

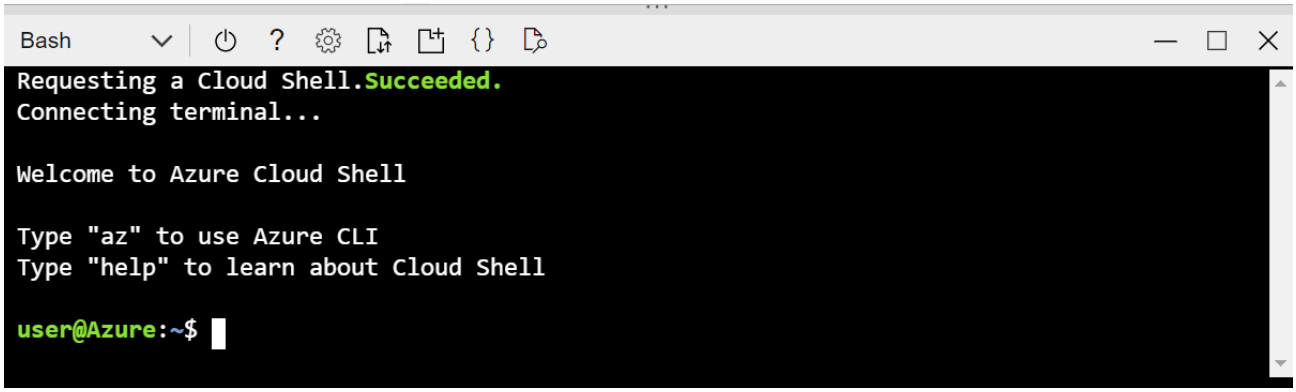



**Figure 2.18: Launching Cloud Shell successfully**

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

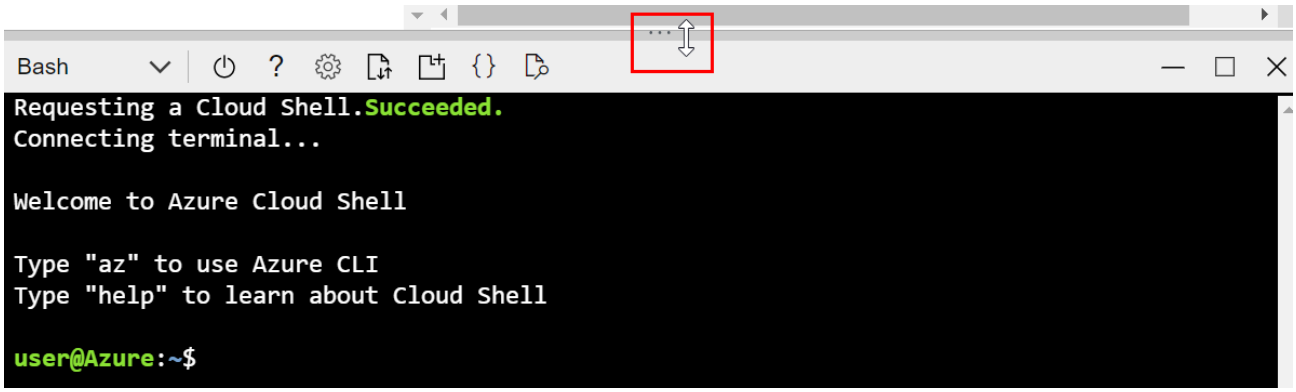



**Figure 2.19: Using the divider to make Cloud Shell larger or smaller**

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

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

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

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

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

To verify that you have access, type the following:

```
kubectl get nodes
```

You should see something like *Figure 2.20*:

```
user@Azure:~$ kubectl get nodes
NAME                                STATUS   ROLES   AGE     VERSION
aks-agentpool-39838025-vmss000000   Ready    agent   7m8s    v1.19.6
aks-agentpool-39838025-vmss000001   Ready    agent   6m46s   v1.19.6
```

**Figure 2.20: Output of the kubectl get nodes command**

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

## Deploying and inspecting your first demo application

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

## Deploying the demo application

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

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

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

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

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

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

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

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

Here is the code example for your convenience:

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

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

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

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

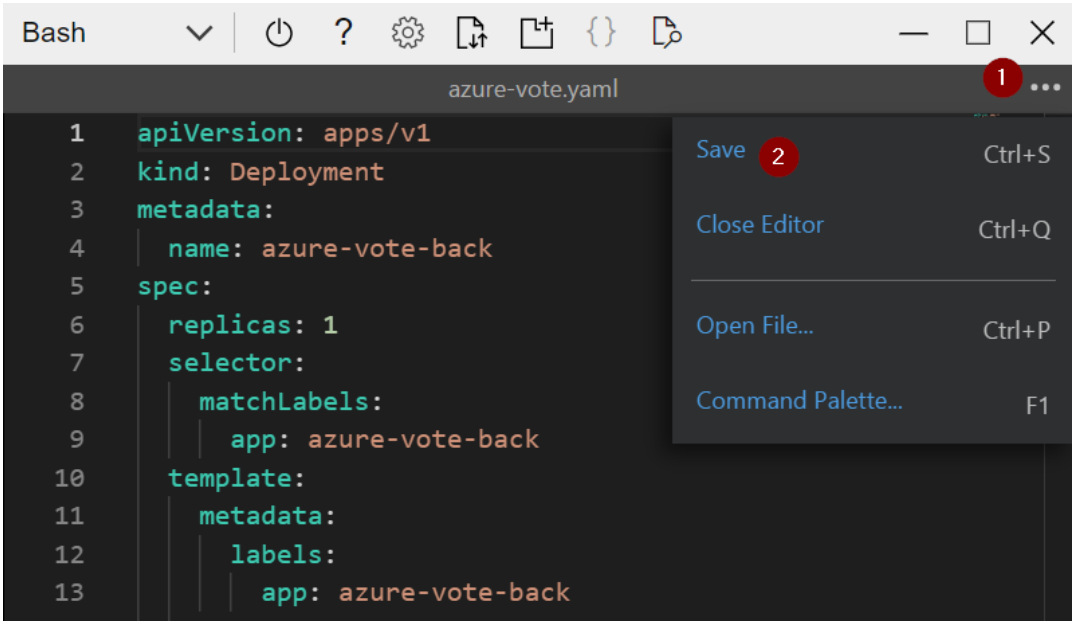



**Figure 2.21: Save the azure-vote.yaml file**

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

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

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

Now, let's launch the application:

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

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

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

**Figure 2.22: Output of the kubectl create command**

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

## Exploring the demo application

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

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

```
kubectl get pods
```

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

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter02$ kubectl get pods
NAME                                READY   STATUS              RESTARTS   AGE
azure-vote-back-5f8bd8b-8npsf       0/1     ContainerCreating   0          3s
azure-vote-front-7797fb8f5d-n4n9d   0/1     ContainerCreating   0          3s
```

**Figure 2.23: Output of the kubectl get pods command**

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

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

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

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

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

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

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

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

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

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

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

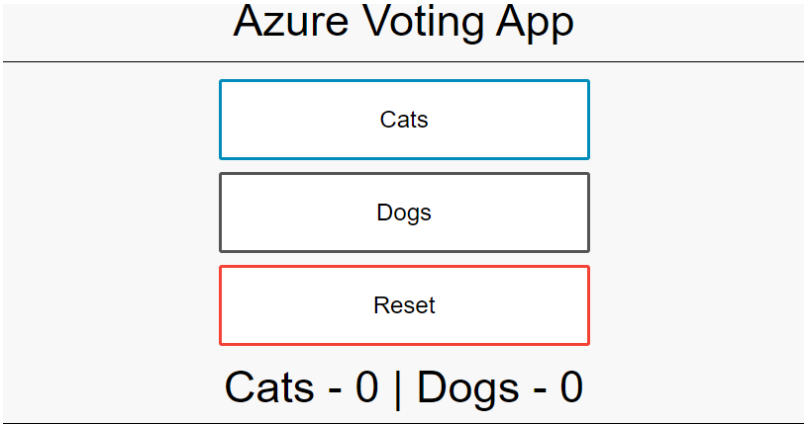



**Figure 2.25: The actual application you just launched**

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

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

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

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

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

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

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

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

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

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

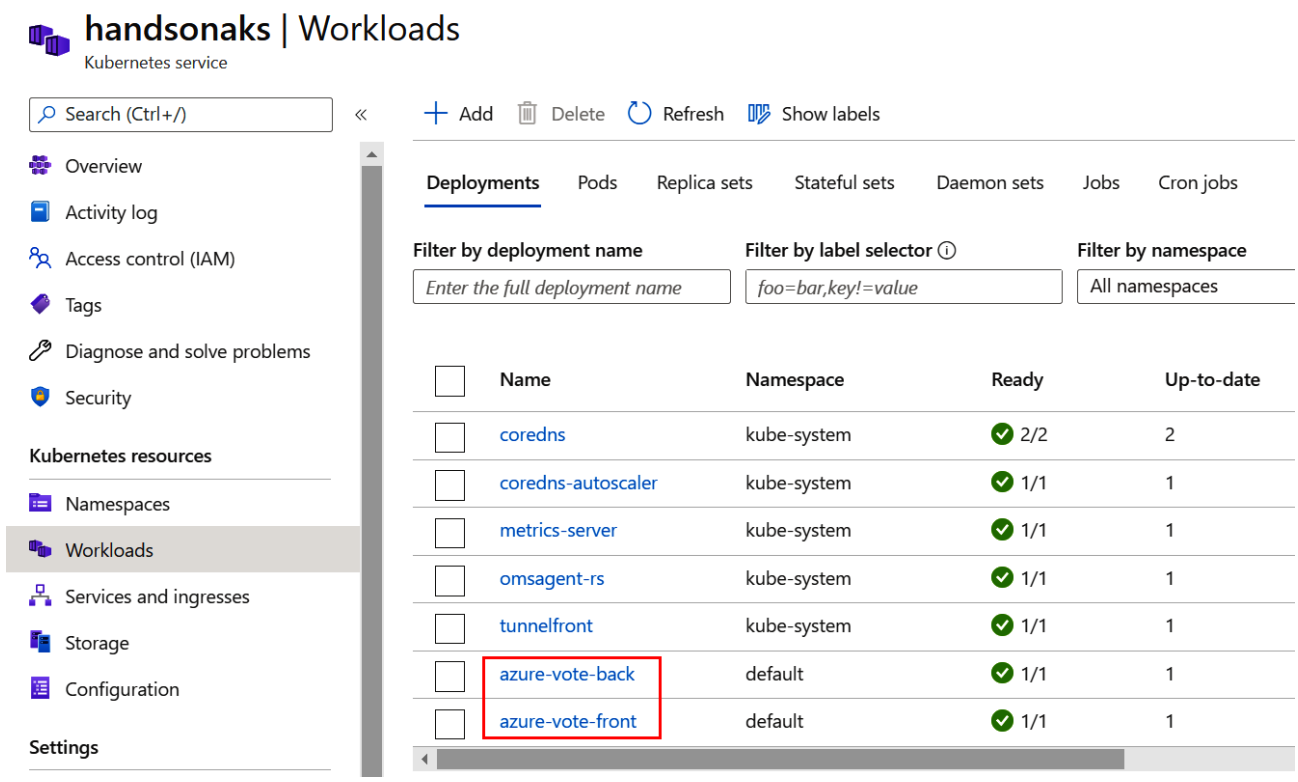



Figure 2.27: Exploring the two deployments part of your application in the Azure portal

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

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

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

Figure 2.28: Cleaning up the application

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

## Summary

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

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

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

# Section 2: Deploying on AKS

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

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

This section contains the following chapters:

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

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

# 3

# Application deployment on AKS

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

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

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


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

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

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

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

The following topics will be covered in this chapter:

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

We'll begin with the sample guestbook application.

## Deploying the sample guestbook application step by step

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

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

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

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

### Introducing the application

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

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

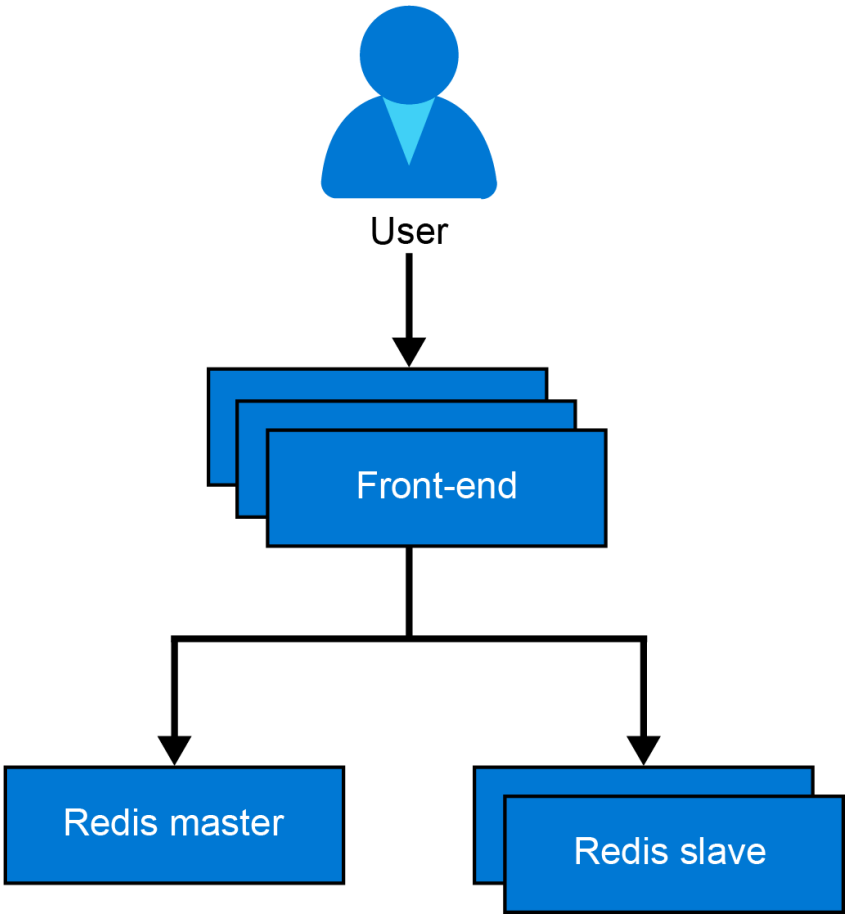



Figure 3.2: High-level overview of the guestbook application

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

## Deploying the Redis master

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

Perform the following steps to complete the task:

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

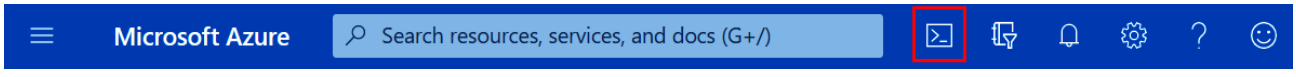



Figure 3.3: Opening the Cloud Shell

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

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

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

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

4. Enter the following command to deploy the master:

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

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

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

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

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

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

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

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

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

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

## Examining the deployment

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

```
kubectl get all
```

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

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter03$ kubectl get all
NAME                                 READY   STATUS    RESTARTS   AGE
pod/redis-master-f46ff57fd-b8cjp     1/1     Running   0          16m

NAME                 TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
service/kubernetes   ClusterIP   10.0.0.1     <none>        443/TCP   38h

NAME                           READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/redis-master   1/1     1            1           16m

NAME                                      DESIRED   CURRENT   READY   AGE
replicaset.apps/redis-master-f46ff57fd    1         1         1       16m
```

**Figure 3.4: Objects that were created by your deployment**

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

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

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

This will generate an output as follows:

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

**Figure 3.5: Description of the deployment**

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

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

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

Executing this command will produce the following output:

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

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

## Redis master with a ConfigMap

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

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

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

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

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

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

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

## Creating a ConfigMap from a file

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

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

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

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

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

   You should get an output as follows:

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

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

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

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

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

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

**Figure 3.6: Description of the ConfigMap**

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

## Creating a ConfigMap from a YAML file

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

1. To start, delete the previously created ConfigMap:

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

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

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

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

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

   You should get an output as follows:

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

4. Next, run the following command:

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

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

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

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

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

### Using a ConfigMap to read in configuration data

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

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

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

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

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

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

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

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

1. Let's create this updated deployment:

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

   This should output the following:

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

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

   ```
   kubectl get pods
   ```

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

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

**Figure 3.7: Details of the pod**

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

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

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

   ```
   CONFIG GET maxmemory
   ```

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

   ```
   CONFIG GET maxmemory-policy
   ```

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

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter03$ kubectl exec -it redis-master-766c5cf5c8-pw7qg -- redis-cli
127.0.0.1:6379> CONFIG GET maxmemory
1) "maxmemory"
2) "2097152"
127.0.0.1:6379> CONFIG GET maxmemory-policy
1) "maxmemory-policy"
2) "allkeys-lru"
127.0.0.1:6379> exit
```

**Figure 3.8: Verifying the Redis configuration in the pod**

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

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

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

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

## Complete deployment of the sample guestbook application

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

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

## Exposing the Redis master service

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

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

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

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

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

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

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

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

```
kubectl get service
```

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

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

**Figure 3.9: Properties of the created service**

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

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

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

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

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

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter03$ kubectl get pods
NAME                           READY   STATUS    RESTARTS   AGE
redis-master-766c5cf5c8-pw7qg   1/1     Running   0          13m
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter03$ kubectl exec -it redis-master-766c5cf5c8-pw7qg -- bash
[ root@redis-master-766c5cf5c8-pw7qg:/data ]$ ping redis-master
PING redis-master.default.svc.cluster.local (10.0.106.207) 56(84) bytes of data.
^C
--- redis-master.default.svc.cluster.local ping statistics ---
2 packets transmitted, 0 received, 100% packet loss, time 1002ms

[ root@redis-master-766c5cf5c8-pw7qg:/data ]$ exit
exit
command terminated with exit code 1
```

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

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

## Deploying the Redis replicas

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

Let's set this up:

1. Create the deployment by running the following command:

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

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

```
kubectl get all
```

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

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

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

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

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

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

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

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

Everything is the same except for the following:

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

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

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

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

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

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

```
kubectl get service
```

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

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

**Figure 3.12: Redis-master and redis-replica service**

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

## Deploying and exposing the front end

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

You can create the front end using the following command:

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

To verify the deployment, run this command:

```
kubectl get pods
```

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

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

**Figure 3.13: Verifying the front end deployment**

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

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

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

Let's see these changes:

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

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

## Exposing the front-end service

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

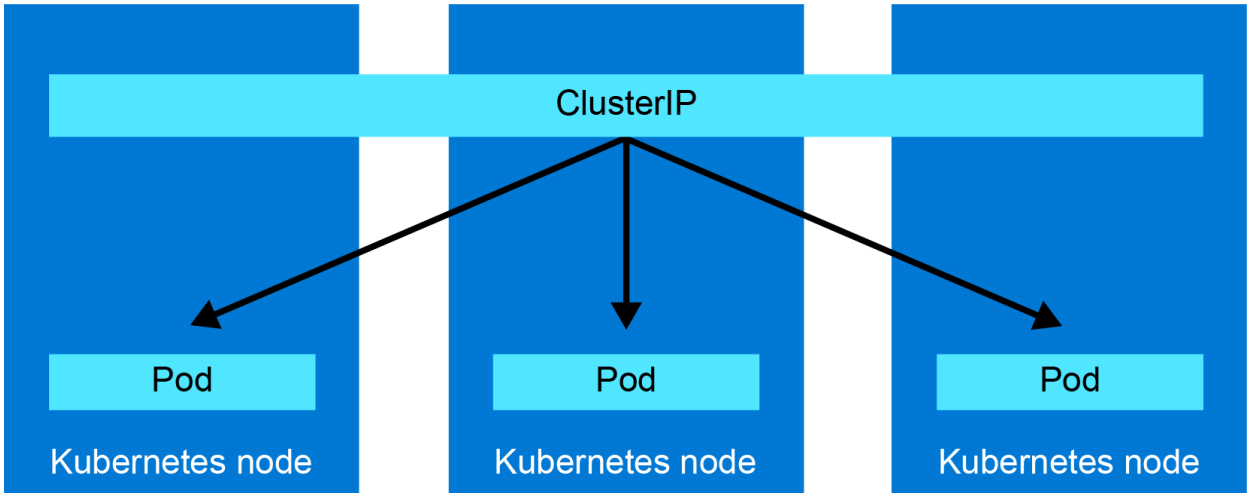



**Figure 3.14: Kubernetes service of type ClusterIP**

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

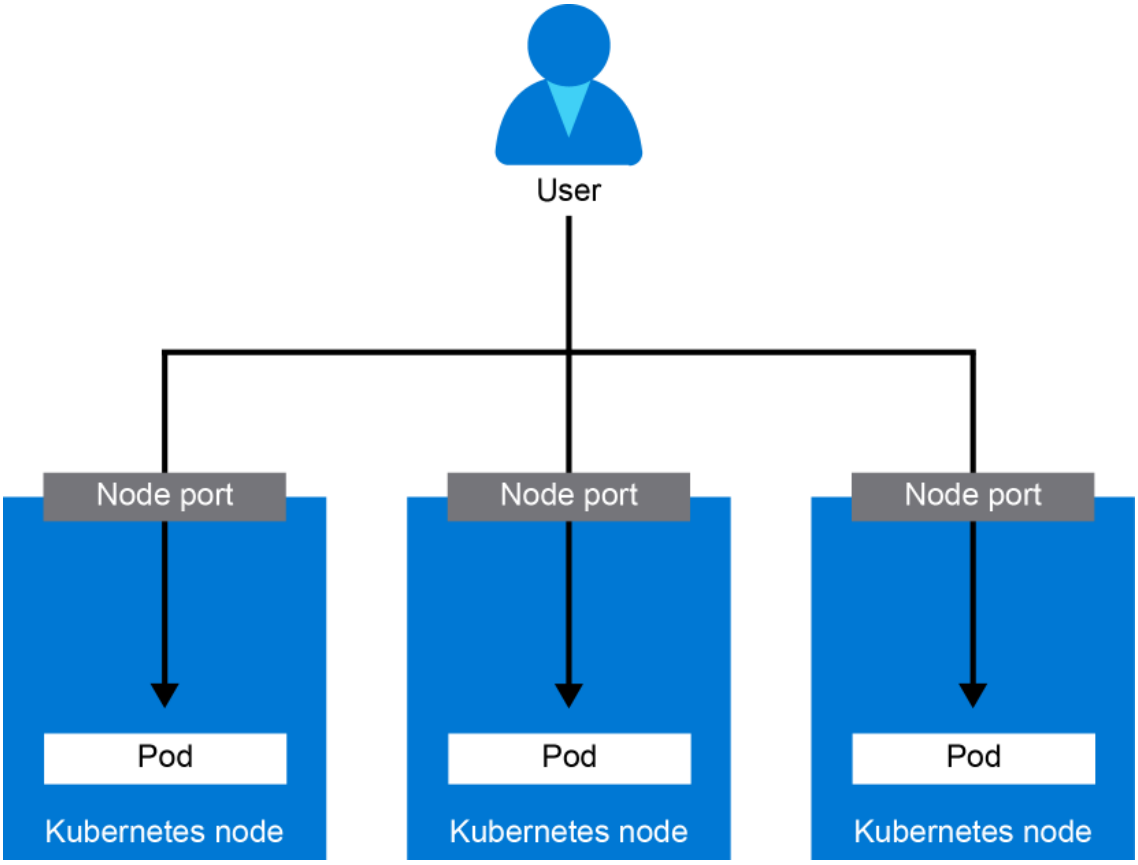



**Figure 3.15: Kubernetes service of type NodePort**

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

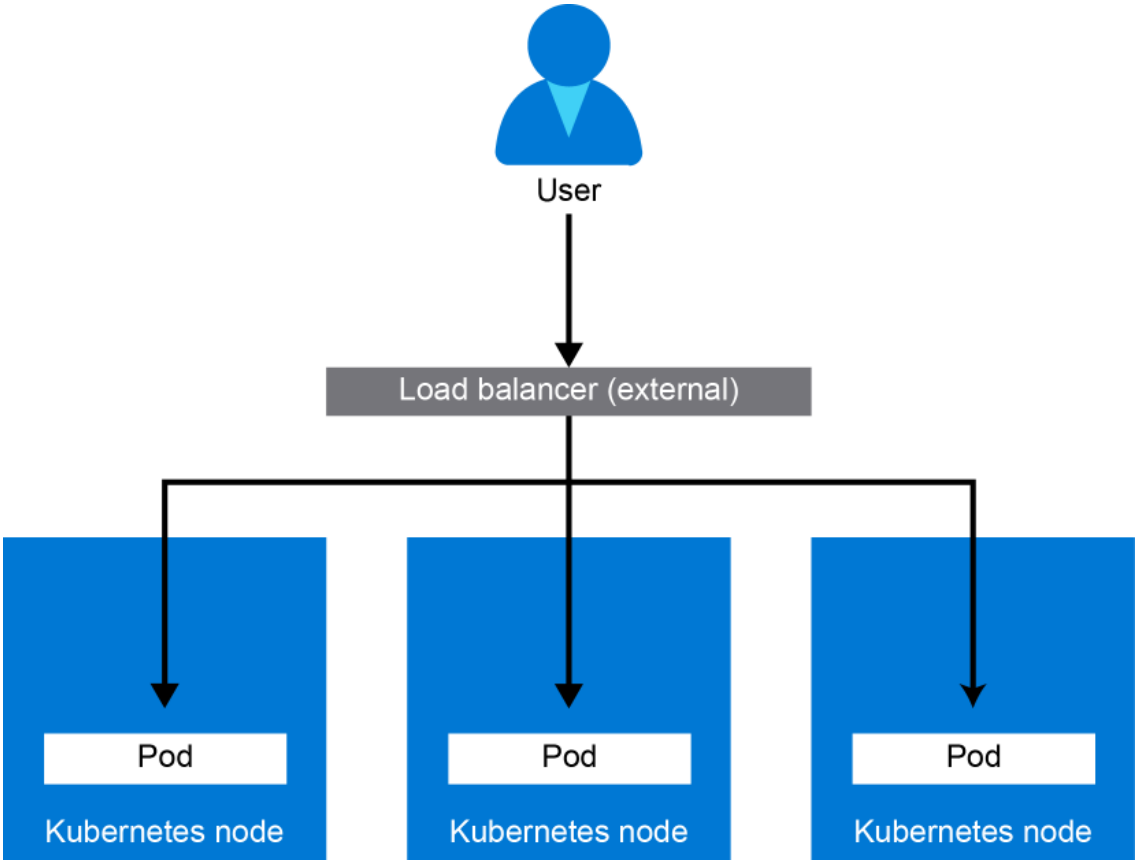



**Figure 3.16: Kubernetes service of type LoadBalancer**

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

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

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

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

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

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

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

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

```
kubectl get service -w
```

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

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

**Figure 3.17: External IP value**

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

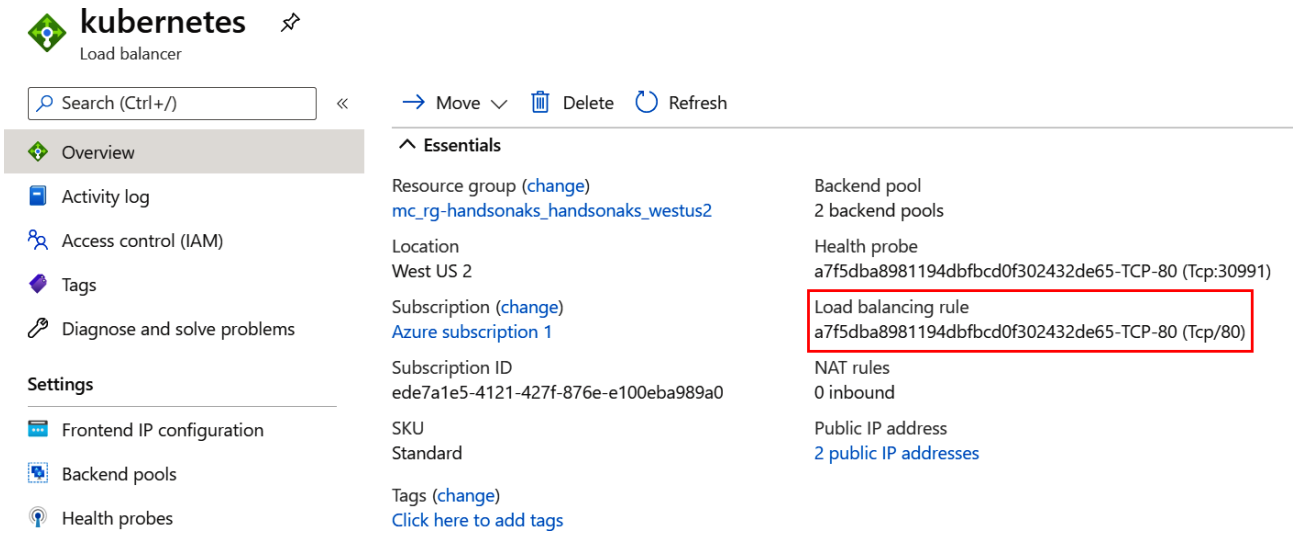



**Figure 3.18: kubernetes Load balancer in the Azure portal**

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

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

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

## The guestbook application in action

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

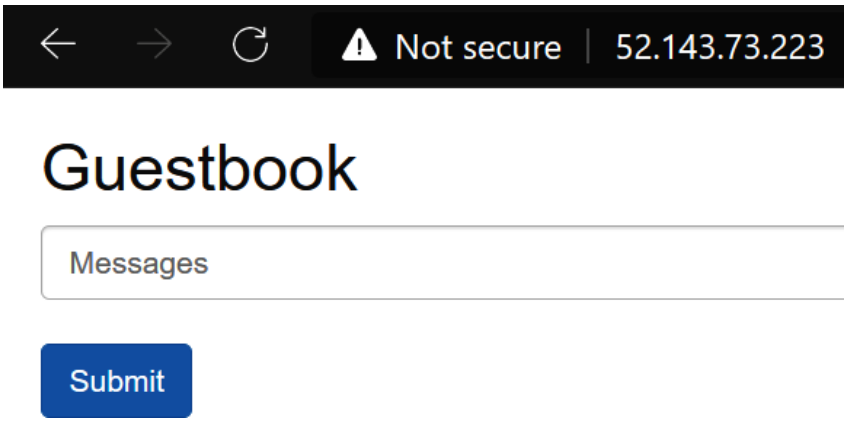



**Figure 3.19: The guestbook application in action**

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

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

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

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

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

## Installing complex Kubernetes applications using Helm

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

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

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

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

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

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

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

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

## Installing WordPress using Helm

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

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

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

2. To install WordPress, run the following command:

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

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

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

### PersistentVolumeClaims

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

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

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

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

```
kubectl get pods
```

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

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

**Figure 3.20: Numbers attached to MariaDB and WordPress pods**

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
kubectl get storageclass
```

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

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

**Figure 3.21: Different storage classes in your cluster**

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

```
kubectl get pvc
```

The output generated is displayed in *Figure 3.22*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter03$ kubectl get pvc
NAME                          STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS   AGE
data-handsonakswp-mariadb-0   Bound    pvc-c68da151-777c-4efa-ac72-c05dd0b33801   8Gi        RWO            default        13m
handsonakswp-wordpress        Bound    pvc-102a8509-5f0b-411d-8ef0-518f2759ca36   10Gi       RWO            default        13m
```

**Figure 3.22: Different PVCs in the cluster**

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

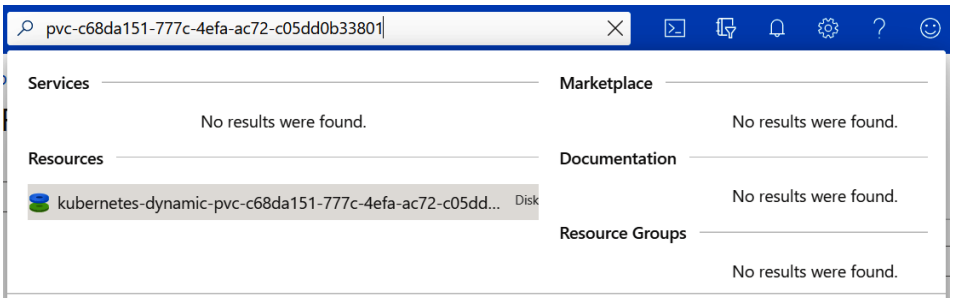



**Figure 3.23: Getting the disk linked to a PVC**

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

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

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

## Checking the WordPress deployment

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

```
helm ls
```

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

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter03$ helm ls
NAME            NAMESPACE    REVISION     UPDATED                                 STATUS       CHART                APP VERSION
handsonakswp    default      1            2021-01-17 22:49:58.470139624 +0000 UTC deployed     wordpress-10.4.2     5.6.0
```

**Figure 3.24: WordPress application deployment status**

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

```
helm status handsonakswp
```

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

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter03$ helm status handsonakswp
NAME: handsonakswp
LAST DEPLOYED: Sun Jan 17 22:49:58 2021
NAMESPACE: default
STATUS: deployed
REVISION: 1
NOTES:
** Please be patient while the chart is being deployed **

Your WordPress site can be accessed through the following DNS name from within your cluster:

    handsonakswp-wordpress.default.svc.cluster.local (port 80)

To access your WordPress site from outside the cluster follow the steps below:

1. Get the WordPress URL by running these commands:

  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. Open a browser and access WordPress using the obtained URL.

3. Login with the following credentials below to see your blog:

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

**Figure 3.25: Getting more details about the deployment**

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

```
kubectl get all
```

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

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

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

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

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

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

**Figure 3.26: List of objects created by Helm**

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

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

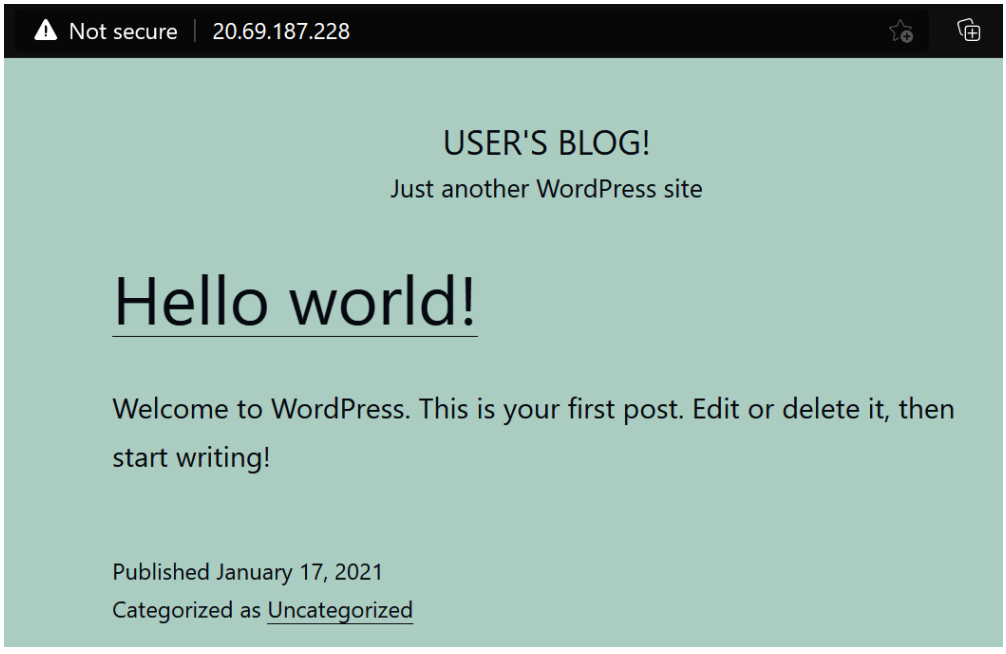



**Figure 3.27: WordPress site being displayed on connection with the external IP**

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

```
helm delete handsonakswp
```

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

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

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

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

## Summary

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

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

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

# 4

# Building scalable applications

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

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

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

In this chapter, we will cover the following topics:

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

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

## Scaling your application

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

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

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

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

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

## Manually scaling your application

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

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

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

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

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

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

**Figure 4.1: Launching the guestbook application**

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

```
kubectl get service
```

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

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

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

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

```
kubectl edit service frontend
```

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

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

**Figure 4.3: Changing this line to type: LoadBalancer**

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

```
kubectl get service -w
```

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

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

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

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

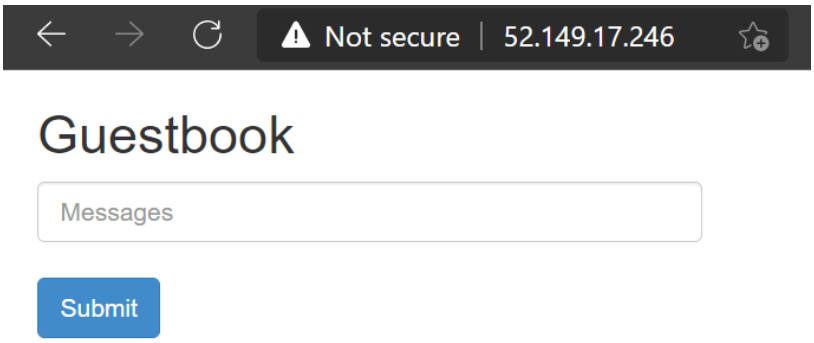



**Figure 4.5: Browse to the guestbook application**

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

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

## Scaling the guestbook front-end component

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

```
kubectl get pods
```

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

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04$ kubectl get pods
NAME                              READY   STATUS    RESTARTS   AGE
frontend-766d4f77cb-5ltc2         1/1     Running   0          3m15s
frontend-766d4f77cb-r4m7k         1/1     Running   0          3m15s
frontend-766d4f77cb-sw6b4         1/1     Running   0          3m15s
redis-master-f46ff57fd-wmsn5      1/1     Running   0          3m16s
redis-replica-5bc7bcc9c4-n9gjt    1/1     Running   0          3m15s
redis-replica-5bc7bcc9c4-w2ld4    1/1     Running   0          3m15s
```

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

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

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

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

```
kubectl get pods
```

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

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

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

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

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

This will generate the following output:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04$ kubectl get pods -o wide
NAME                                READY   STATUS    RESTARTS   AGE     IP            NODE
frontend-766d4f77cb-5ltc2           1/1     Running   0          4m22s   10.244.1.6    aks-agentpool-39838025-vmss000001
frontend-766d4f77cb-6xwvz           1/1     Running   0          30s     10.244.1.8    aks-agentpool-39838025-vmss000001
frontend-766d4f77cb-gmd5p           1/1     Running   0          30s     10.244.0.11   aks-agentpool-39838025-vmss000000
frontend-766d4f77cb-r4m7k           1/1     Running   0          4m22s   10.244.0.8    aks-agentpool-39838025-vmss000000
frontend-766d4f77cb-sw6b4           1/1     Running   0          4m22s   10.244.1.7    aks-agentpool-39838025-vmss000001
frontend-766d4f77cb-vz726           1/1     Running   0          30s     10.244.0.10   aks-agentpool-39838025-vmss000000
redis-master-f46ff57fd-wmsn5        1/1     Running   0          4m23s   10.244.1.4    aks-agentpool-39838025-vmss000001
redis-replica-5bc7bcc9c4-n9gjt      1/1     Running   0          4m22s   10.244.1.5    aks-agentpool-39838025-vmss000001
redis-replica-5bc7bcc9c4-w2ld4      1/1     Running   0          4m22s   10.244.0.9    aks-agentpool-39838025-vmss000000
```

**Figure 4.8: Showing which nodes the pods are running on**

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

## Using the HPA

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

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

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

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

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

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

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

Let's investigate what is configured in this file:

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

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

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

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

```
kubectl get hpa
```

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

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

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

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

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

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

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

```
kubectl get pods -w
```

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

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


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

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

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

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

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

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

**Figure 4.12: New pods get started by the HPA**

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

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

   ```
   kubectl describe hpa
   ```

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

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04$ kubectl describe hpa
Name:                                                  frontend-scaler
Namespace:                                             default
Labels:                                                <none>
Annotations:                                           <none>
CreationTimestamp:                                     Wed, 20 Jan 2021 01:10:58 +0000
Reference:                                             Deployment/frontend
Metrics:                                               ( current / target )
  resource cpu on pods  (as a percentage of request):  384% (38m) / 50%  (1)
Min replicas:                                          1
Max replicas:                                          10
Deployment pods:                                       10 current / 10 desired
Conditions:
  Type            Status  Reason            Message
  ----            ------  ------            -------
  AbleToScale     True    ReadyForNewScale  recommended size matches current size
  ScalingActive   True    ValidMetricFound  the HPA was able to successfully calculate a replica count from cpu resource utilization (percentage of request)
  ScalingLimited  True    TooManyReplicas   (2) he desired replica count is more than the maximum replica count
Events:
  Type     Reason                        Age                  From                       Message
  ----     ------                        ----                 ----                       -------
  Warning  FailedGetResourceMetric       18m (x3 over 18m)    horizontal-pod-autoscaler  unable to get metrics for resource cpu: no metrics returned from resource metrics API
  Warning  FailedComputeMetricsReplicas  18m (x3 over 18m)    horizontal-pod-autoscaler  invalid metrics (1 invalid out of 1), first error is: failed to get cpu utilization:
unable to get metrics for resource cpu: no metrics returned from resource metrics API
  Normal   SuccessfulRescale             13m                  horizontal-pod-autoscaler  New size: 1; (3) reason: All metrics below target
  Normal   SuccessfulRescale             11m                  horizontal-pod-autoscaler  New size: 4; reason: cpu resource utilization (percentage of request) above target
  Normal   SuccessfulRescale             11m                  horizontal-pod-autoscaler  New size: 8; reason: cpu resource utilization (percentage of request) above target
  Normal   SuccessfulRescale             11m                  horizontal-pod-autoscaler  New size: 10; reason: cpu resource utilization (percentage of request) above target
```

**Figure 4.13: Detailed view of the HPA**

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

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

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

```
kubectl get hpa -w
```

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

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

**Figure 4.14: Watching the HPA scale down**

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

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

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

# Scaling your cluster

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

## Manually scaling your cluster

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

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

To scale your cluster, follow these steps:

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

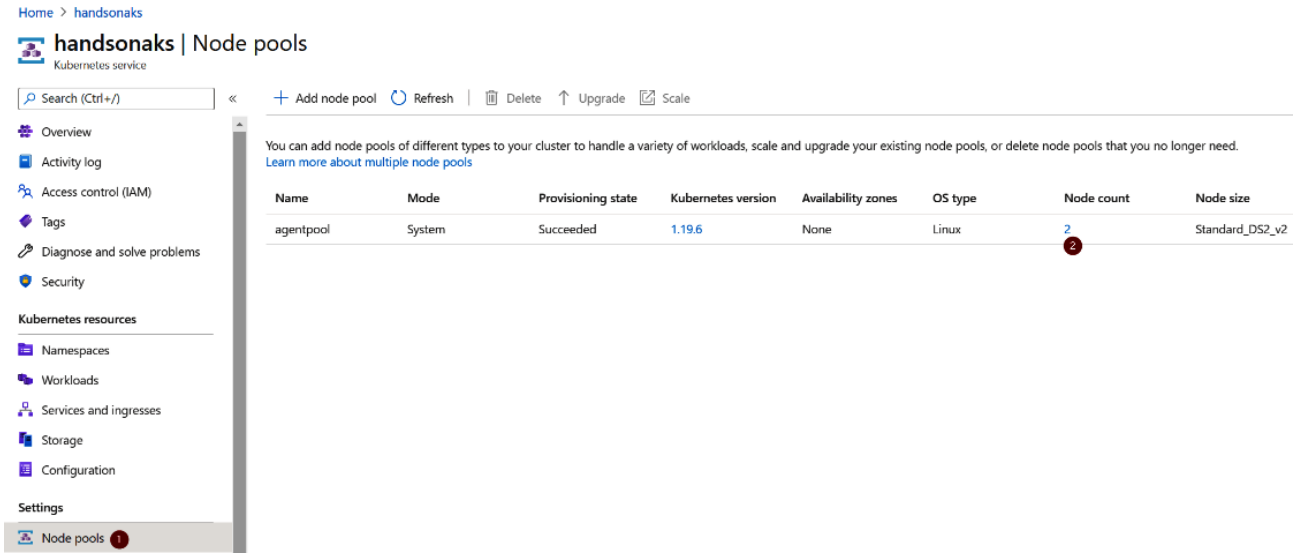



Figure 4.15: Manually scaling the cluster

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

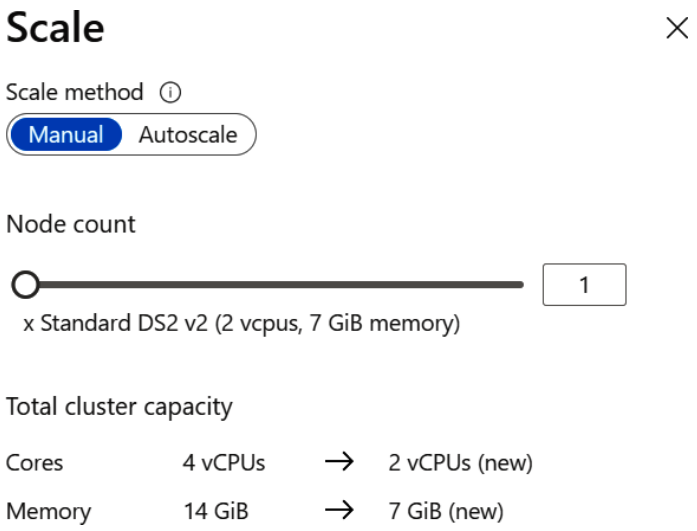



**Figure 4.16: Pop-up window confirming the new cluster size**

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

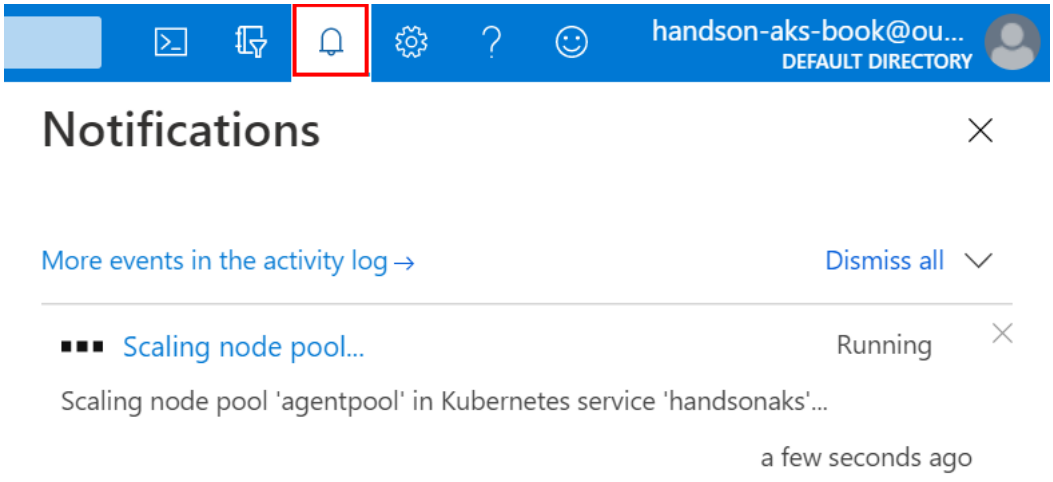



**Figure 4.17: Cluster scaling can be followed using the notifications in the Azure portal**

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

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

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

## Scaling your cluster using the cluster autoscaler

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

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

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

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

```
kubectl get pods
```

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

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04$ kubectl get pods
NAME                             READY   STATUS    RESTARTS   AGE
frontend-766d4f77cb-578vs        1/1     Running   0          30m
frontend-766d4f77cb-p64vw        1/1     Running   0          30m
frontend-766d4f77cb-wjwj4        1/1     Running   0          30m
redis-master-f46ff57fd-b95l8     1/1     Running   0          30m
redis-replica-5bc7bcc9c4-btkzp   1/1     Running   0          30m
redis-replica-5bc7bcc9c4-ckvz2   1/1     Running   0          30m
redis-replica-5bc7bcc9c4-mwmcm   1/1     Running   0          30m
redis-replica-5bc7bcc9c4-vdxcl   0/1     Pending   0          30m
redis-replica-5bc7bcc9c4-vjrg5   0/1     Pending   0          30m
```

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

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

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

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

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

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

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

```
kubectl get nodes -w
```

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

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

Figure 4.19: The new node joins the cluster

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

```
kubectl get pods
```

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

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04$ kubectl get pods
NAME                              READY   STATUS    RESTARTS   AGE
frontend-766d4f77cb-578vs         1/1     Running   0          37m
frontend-766d4f77cb-p64vw         1/1     Running   0          37m
frontend-766d4f77cb-wjwj4         1/1     Running   0          37m
redis-master-f46ff57fd-b95l8      1/1     Running   0          37m
redis-replica-5bc7bcc9c4-btkzp    1/1     Running   0          37m
redis-replica-5bc7bcc9c4-ckvz2    1/1     Running   0          37m
redis-replica-5bc7bcc9c4-mwmcm    1/1     Running   0          37m
redis-replica-5bc7bcc9c4-vdxcl    1/1     Running   0          37m
redis-replica-5bc7bcc9c4-vjrg5    1/1     Running   0          37m
```

Figure 4.20: All pods are now in a Running state

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

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

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

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

## Upgrading your application

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

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

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

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

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

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

## Upgrading by changing YAML files

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
kubectl get service
```

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

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

**Figure 4.23: Output displaying a public IP**

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

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

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

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

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

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

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

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

**Figure 4.24: Pods from a new ReplicaSet are created**

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

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

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

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

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

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

```
kubectl get replicaset
```

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

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

**Figure 4.26: Two different ReplicaSets**

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

```
kubectl rollout history deployment frontend
```

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

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

**Figure 4.27: Deployment history of the application**

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

```
kubectl rollout undo deployment frontend
```

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

```
kubectl get replicaset
```

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

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

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

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

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

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

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

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

## Upgrading an application using kubectl edit

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

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

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

```
git reset --hard
```

2. You will then deploy the guestbook application:

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

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

```
kubectl edit service frontend
```

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

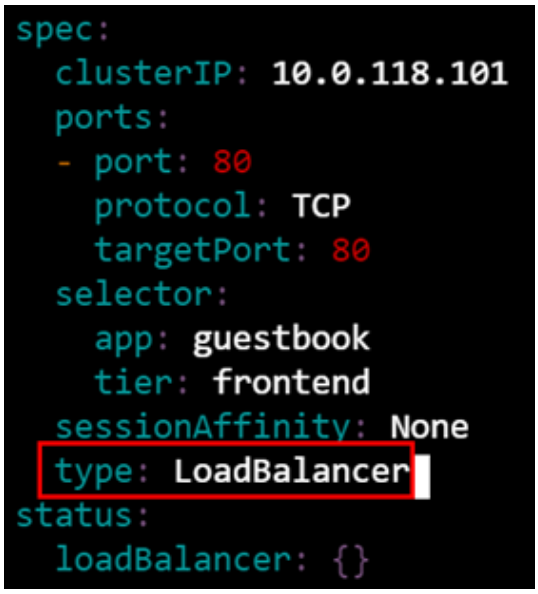



**Figure 4.29: Changing this line to type: LoadBalancer**

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

```
kubectl get svc -w
```

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

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

## Upgrading an application using kubectl patch

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

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

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

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

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

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

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

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

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

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

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

```
kubectl describe deployment frontend
```

This will display an output as follows:

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

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

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

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

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

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

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

   ```
   kubectl describe deployment frontend
   ```

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

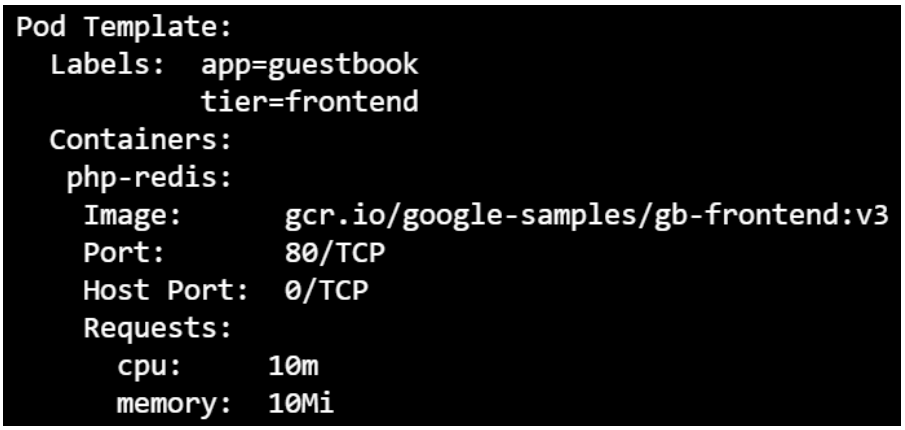



**Figure 4.31: After another patch, we are running the new version again**

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

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

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

## Upgrading applications using Helm

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

1. Run the following command:

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

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

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

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

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

**Figure 4.32: Getting the current image of the StatefulSet**

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

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

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

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

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

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

**Figure 4.33: The encrypted secrets that MariaDB uses**

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

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

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

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04$ echo "OHpveVdWUmRUWA==" | base64 -d
8zoyWVRdTXuser@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04$ echo "NTg4TUJrVUk2dA==" | base64 -d
588MBkUI6tuser@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04$
```

**Figure 4.34: The decoded root and database passwords**

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

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

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

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

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

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

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

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

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

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

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

**Figure 4.36: Showing the new image**

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

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

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

## Summary

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

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

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

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

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

# 5

# Handling common failures in AKS

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

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

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

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

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

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

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

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

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

## Handling node failures

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

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

```
kubectl get nodes
```

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

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

**Figure 5.1: List of nodes in the cluster**

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

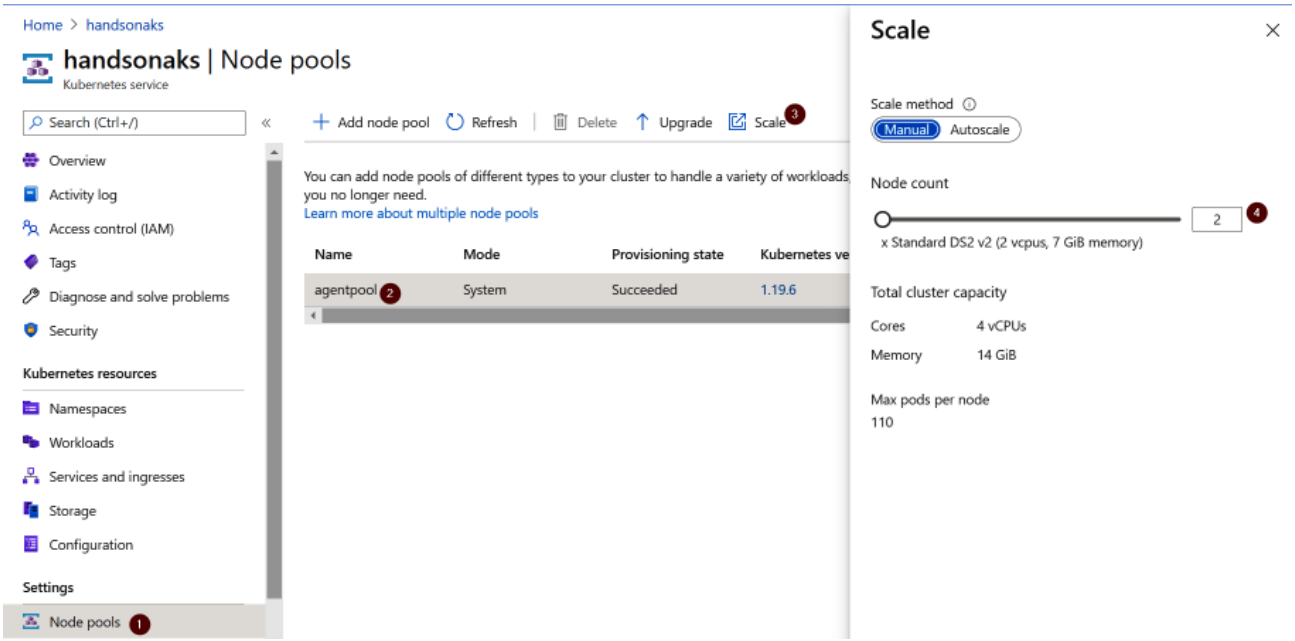



**Figure 5.2: Scaling the cluster**

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

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