Beginning Kubernetes on the Google Cloud Platform

A Guide to Automating Application Deployment, Scaling, and Management

—

Ernesto Garbarino

Apress®

Beginning Kubernetes on the Google Cloud Platform

A Guide to Automating Application Deployment, Scaling, and Management

Ernesto Garbarino

Apress®

Beginning Kubernetes on the Google Cloud Platform: A Guide to Automating Application Deployment, Scaling, and Management

Ernesto Garbarino
EGHAM, UK

ISBN-13 (pbk): 978-1-4842-5490-5 ISBN-13 (electronic): 978-1-4842-5491-2
https://doi.org/10.1007/978-1-4842-5491-2

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Susan McDermott
Development Editor: Laura Berendson
Coordinating Editor: Jessica Vakili

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit http://www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-5490-5. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

Table of Contents

About the Author

Ernesto Garbarino is a consultant in the Digital, Cloud, and DevOps domains. His 20-year experience ranges from working with early startups and entrepreneurial organizations during the dot-com era, to senior consultancy work in blue chip industries including telecoms, logistics, and banking. He holds an MSc in Software Engineering with Distinction from the University of Oxford. He lives in Egham, United Kingdom with his two loves; his lovely wife, Adriana, and his sweet black and tan dachshund dog, Daisy.

About the Technical Reviewer

Jing Dong is an entrepreneur and technologist with a wealth of experience in software engineering, container orchestration, and cloud native magics. He started programming in 1994 when he was nine years old. His passion for distributed systems led him into reaching data and scalability challenges in both technical and cultural aspects. He architected and engineered many large-scale applications and data platforms for F1 motorsport, disaster prediction, online media, and financial services. He holds an MSc in Computing in Distributed Systems from Imperial College London. Electronic components and flying robots occupy the rest of his free time. He lives in the United Kingdom with his beloved wife and son.

CHAPTER 1

Introduction

In 2016, my end customer was a global logistics company in the process of upgrading a 15-year-old monolithic clearance system based on an application server (WebLogic), a relational SQL database (Oracle), and messaging queues (TIBCO). Every component in each tier was a bottleneck in its own right. The four-node WebLogic cluster could not cope with the load, the slowest SQL queries would take several seconds to process, and publishing a message to a TIBCO EMS queue was akin to throwing a feather into a 100 meter well. This was a mission critical system that made possible the transit of goods through various clearance centers across the European Union. A crash (or significant slowdown) of this system would have implied empty shelves in several countries and offices, waiting for overnight documents, grinding to a halt.

I was appointed with the task of leading a team in charge of rearchitecting the system. The team consisted of highly qualified subject matter experts in each technology tier. We had expertise in most of the tools that would help create a modern, highly scalable, and highly available application: Cassandra for NoSQL databases, Kafka for messaging, Hazelcast for caching, and so on. But what about the Java monolith? We had broken down the application logic into suitable bounded contexts and had a solid microservices architecture blueprint but could not get a container—let alone container *orchestration*—expert to give us a hand with the design.

E. Garbarino, *Beginning Kubernetes on the Google Cloud Platform*,
https://doi.org/10.1007/978-1-4842-5491-2_1

I was reluctant to adding more activities to my already busy agenda, but I was so desperate to see the project through that I said to myself: "what the heck, I'll take care of it myself." The first week, after struggling to wrap my head around the literature on the topic, I thought I had made a big mistake and that I should talk myself out of this area of expertise. There were an overwhelming number of self-proclaimed "container orchestrators" in the marketplace: Docker Swarm, Apache Mesos, Pivotal Cloud Foundry—the customer's favorite—and of course, Kubernetes. Each had radically different concepts, tooling, and workflows for the management of containers. For example, in Cloud Foundry, pure Docker containers were a second-class citizen; developers were supposed to "push" code directly into the platform and let it be matched with its runtime in the background via a mechanism known as "build packs." I was dubious about Pivotal Cloud Foundry's approach which felt just like WebLogic all over again where a single missing dependency on the server side would result in a failed deployment.

I tried to deal with the problem with an architect's glasses, by understanding concepts, looking at diagrams, and working out how the components worked together, but I simply "did not get it." When it came to hard questions such as "How does a zero-downtime migration *precisely* work?" I could not articulate it other than in vague terms. So, I decided that if I wanted to be able to answer these complex questions and get to the guts of a container orchestrator, I had to roll out my sleeves.

Before I could get my hands dirty, I had to pick one orchestrator. I simply did not have the time to tinker with each of them. Docker Swarm felt too simplistic—in 2016—for an enterprise system in which the likes of load balancers and storage systems needed to be orchestrated as well. Mesos felt like a general solution in which Docker was shoehorned at a later stage in its development. This left me with Kubernetes, which was poorly documented and only had its Google heritage as its seemingly unique selling proposition. In spite of its infancy at that time, Kubernetes appeared to be concerned with *general* strategies to achieve high scalability and high availability—plus the complex orchestration of network and storage resources to support said architectural properties. Kubernetes seemed to be trying to solve the *right problems*.

The Kubernetes literature, at the time, though, was overwhelming in its complexity. Most texts started explaining all of the internal components that underpin a Kubernetes cluster (Kubelet, Kube Proxy, etcd, etc.) and how they worked together. All good to explain how Kubernetes works, but not ideal to demonstrate *how it helps* solve mundane problems. Some other texts, instead, started with a 30,000 feet overview quickly throwing complicated jargon, all at once: Pods, replication controllers, DeamonSets, etc.

As I have done may times over in my life, I decided to organize my findings into a simpler structure that my primitive simian brain could understand; so I started my research adventure with the intention of documenting my insights in a blog post. But something unexpected happened in the midst of this process.

As I was a handful of weeks into my lab experiments, I realized that I was no longer researching a so-called "container orchestrator." The realization was that containers—or the use of Docker images to be more precise—were rather inconsequential from an architectural standpoint. Kubernetes was much more than this. It was the answer to a much more fundamental problem; how to achieve *general-purpose, distributed computing, at the operating system level, rather than at the application level.*

When the penny dropped, I felt a mix of excitement and fear. "This is the future" kept playing in my head like a broken record. Kubernetes was no longer a "choice" anymore; I was not dealing with the problem of what container orchestrator flavor was the most appropriate to implement a clearance microservices-based application. I was in front of a technology that would possibly shape how everyone would do general-purpose distributed computing in a few years time. This was like the moment I booted Slackware Linux for the first time on a PC that could only speak MS-DOS and Windows until then.

See, in the 1980s, an Intel PC running MS-DOS allowed a single user to run one single program at a time. When GNU/Linux came into the picture—acknowledging the existence of other Unix-like operating systems at the time such as Xenix, SCO, and MINIX—ordinary users, like me, had, for the first time, the ability to run multiple programs concurrently, on a single computer. Kubernetes takes the evolutionary path one step further by adding the ability to run multiple programs concurrently *in different computers*.

In other words, the "magic" of a modern operating system—other than hardware abstraction—is that it helps run programs concurrently and manage them in a horizontal fashion abstracting away from how many CPU cycles are being devoted to each program, which CPU and/or core is supporting them, and so on. When you launch a job to compress an audio file to MP3, say, by using the LAME encoder and typing `lame equinoxe.wav -o equinoxe.mp3`, you *don't need to care* which CPU or core the process will be allocated to. Imagine if you had 1000 computers and you did not have to care about *which computers* pick up the job? How many concurrent MP3 (or video encoding) jobs could you run? This is exactly what Kubernetes allows you to do in a seamless fashion in a way that is not fundamentally harder than how regular Linux isolates the user from the vagaries of task switching and CPU/core allocation.

To address the original dilemma, "Is Kubernetes a container orchestrator?". Well, yes, in so far as Linux is an orchestrator of executable and linkable format (ELF) files. Kubernetes is much more than a Docker host running on multiple boxes—in fact, it can orchestrate Windows containers and other container types such as *rkt* (Rocket) too. What is more, Kubernetes is already today the de facto platform for general-purpose distributed computing (including network and storage resources) as proved by its ubiquitous support by every single large cloud and on-prem vendor.

Why Kubernetes on the Google Cloud Platform

Kubernetes was originally designed by Google and announced in 2014. Google first released Kubernetes 1.0 in July 21, 2015, when it donated it to the Cloud Native Computing Foundation (CNCF). The CNCF, formed initially with the primary aim of making Kubernetes vendor neutral, is run by the Linux Foundation. The Kubernetes design was influenced by Borg which is the proprietary system that Google uses to run its data centers.

The CNCF includes, among its platinum members, every single major cloud vendor (Google itself, Amazon, Microsoft, IBM, Oracle, and Alibaba) as well as companies more familiar in the *on-prem* space, such as RedHat, Intel, VMWare, Samsung, and Huawei. Kubernetes wears different skins—it is branded differently—depending on the vendor. In AWS, for example, it is called “Amazon Elastic Kubernetes Service” (EKS); in Azure, instead, it is called “Azure Kubernetes Service” (AKS). Nearly every major public and private cloud provider offers a Kubernetes incarnation, but we have chosen the Google Container Engine (GKE). Why?

The reader should not jump to the conclusion that “If it is made by Google, it runs better on Google.” This reasoning is wrong in two counts. First, the Kubernetes project, since it was open sourced and handed over to the CNCF in 2015, has received significant contributions from its various sponsors in addition to those from individual enthusiasts. Second, whether Kubernetes is perceived to “run well” depends on a myriad of contextual factors: the proximity of the user to the closest Google data center, specific storage and networking needs, the choice of peculiar cloud native services that may or may not be available in a given cloud vendor’s product catalogue, and so on.

Instead, the author has chosen the Google Cloud Platform (GCP) for its *pedagogic* convenience, in other words, the benefits that it brings to the reader who is making their first steps in the Kubernetes world:

- A generous offer of a $300 (US dollars) credit, or the equivalent in a local currency, for readers to experiment with a real-world, production-grade cloud platform.
- An integrated shell (the Google Cloud Shell) which relieves the reader from the burden of setting up a complex workstation configuration. The end-to-end learning journey requires only a web browser: a Chromebook or a tablet is sufficient.
- The availability of highly integrated network (load balancers) and storage resources which may be absent or difficult to set up in local/on-prem environments such as Minikube or RedHat OpenShift.

Finally, GKE is a production-ready, world-class product used by blue-chip companies such as Philips. The reader may choose to migrate their company's workloads to GCP directly rather than—necessarily—having to reinvest skills in the specifics of AWS or Azure, if the GCP's service offering is suitable for the needs at hand.

Who This Book Is For

This book was written for the *absolute beginner* in mind, who is looking to internalize the *foundational* capabilities that Kubernetes brings to the table such as dynamic scaling, zero-downtime deployments, and externalized configuration. Junior Linux administrators, operations engineers, developers of monolithic Linux-based apps in *any* language, and solution architects—who are happy to roll up their sleeves—are ideal candidates.

The book is self-contained and does not demand advanced knowledge in areas such as cloud, virtualization, or programming. Not even proficiency in Docker is required. Since Kubernetes is effectively a Docker host, the reader can experiment with Docker and public Docker images found in Docker Hub for the first time on the Google Cloud Platform itself without installing the Docker runtime on their local machine.

The reader is only required to possess minimal familiarity with the Linux's command-line interface and the ability to use, *but not necessarily write*, simple shell scripts. Chapter 9 is an exception, in which Python is extensively used for the examples—as a necessity given the challenge in demonstrating the dynamics of StatefulSets—however, the reader only needs to understand *how to run* the provided Python scripts rather than their exact inner workings.

How This Book Is Different

This book differentiates itself from other seemingly similar publications in that it prioritizes comprehension over covering an extensive syllabus, and that it is written in a strict bottom-up fashion, using, in general, small code examples.

As a consequence of being a didactically focused book, a fewer number of topics are covered. The selected topics are explored slowly, on a step-by-step basis and using examples that help the reader observe, prove, and internalize *how Kubernetes helps solve problems*, as opposed to what its internals are.

The bottom-up approach allows the reader to be productive from the first chapter, without having to "read ahead" or wait until the end of the book to set up Kubernetes-based workloads. This method also decreases the number of concepts that the reader must joggle in their mind at any given time.

How to Use This Book

This book can be read in two ways: passively and actively.

The passive approach consists on reading the book away from a computer. Many books in the 1970s and 1980s were written in a passive style since it was often the case that the readers did not have a computer at home.

Given that the author grew up reading books from this era, this text treats the passive reader as a first-class citizen. What this means is that all relevant results are presented on the book; the reader is seldom required to run a command or script themselves to understand their effect. Likewise, all code is reproduced in the book—barring cases of small variations in which the changes are described in writing. The only issue, in using this approach, is that the reader may be tempted to read too fast and skimp on important details. An effective way to avoid falling into this trap is by using a highlighter and writing notes on the margins, which induces a natural and beneficial slowdown.

The active reading style is the one in which the reader runs the suggested commands and scripts as they read the book. This approach helps internalize the examples in a faster manner, and it also allows the reader to try out their own variations to the supplied code. If following this approach, the author recommends using the Google Cloud Shell rather than battling with a—potentially—problematic local environment.

Finally, regardless of whether the reader has opted for a passive or active reading approach, the author recommends that one entire chapter is covered in "one go" per reading session. Each section within a chapter introduces a vital new concept that the author expects the reader to have retained by the time he moves onto the next one. Table 1-1 provides the estimated reading time in hours for each chapter.

Table 1-1. *Estimated reading time for each chapter in hours*

Chapter	Passive	Active
Chapter 1	01:00 HRs	02:30 HRs
Chapter 2	02:00 HRs	04:30 HRs
Chapter 3	01:30 HRs	03:30 HRs
Chapter 4	01:30 HRs	03:30 HRs
Chapter 5	01:00 HRs	02:30 HRs
Chapter 6	01:00 HRs	02:30 HRs
Chapter 8	01:00 HRs	02:30 HRs
Chapter 9	02:00 HRs	04:30 HRs

Conventions

The following typographical conventions are used in this book:

- *Italics* introduces a new term or concept that is worthy of the reader's attention.
- Capitalized nouns such as "Service" or "Deployment" refer to Kubernetes object types as opposed to their regular meaning in English.
- `Fixed-width` text is used to refer to syntactical elements such as YAML or JSON, commands, or arguments. The latter also include user-defined names such as `my-cluster`.
- `<IN-BRACKETS-FIXED-WIDTH>` text refers to command arguments.

- dot.separated.fixed-width-text is used to refer to properties within various Kubernetes object types. A detailed description for such properties can be obtained by running the kubectl explain <PROPERTY> command, for example, kubectl explain pod.spec.containers. Running kubectl requires that we set up a Kubernetes cluster first which we cover in the following sections. In certain cases, the parent properties may be skipped if implied by the context. For example, spec.containers or simply containers may be used at times rather than pod.spec.containers.

In addition, and in the interest of brevity, and making this text more accessible, the output of commands (kubectl in most cases) may be altered as follows:

- If a command produces a tabular display with multiple columns, columns that do not contribute to the discussion at hand may be omitted.
- If a command produces warning that are not contextually relevant (e.g., they alert the user of features in new APIs), such warnings may not be shown.
- White space may be added or removed whenever it improves the formatting.
- Long identifiers may be shortened by replacing a boilerplate word fragments within said identifiers with an asterisk. For example, the identifier gke-my-cluster-default-pool-4ff6f64a-6f4v represents a Kubernetes Node. We may show such identifier as gke-*-4ff6f64a-6f4v where * stands for my-cluster-default-pool .

- Date components may be omitted whenever they make the examples overflow the book's column length and/or when including the whole datetime string does not contribute to the discussion at hand. For example, a log line such as `Wed Aug 28 09:17:27 DST 2019 - Started` may be shown simply as `17:27 - Started`.

Finally, source code listings as well as commands may use a backward slash \ to present multiple arguments that otherwise would have been written using one single long line:

```
$ gcloud container clusters create my-cluster \
    --issue-client-certificate \
    --enable-basic-auth
```

The reader is free to ignore the backward slashes (and thus the carriage return) and write examples consisting of multiple lines as one single long line instead, if necessary.

Setting Up a GCP Environment

As the time this book went to press, Google is offering a $300 (US dollars) credit toward new accounts on their platform. Three hundred dollars is sufficient to try out the examples in this book many times over and even run some extra pet projects for a while. The steps to create an account are as follows:

1. Go to https://cloud.google.com/.
2. Click the “Try Free” button.
3. Follow through prompts and questions.
4. Enter address and credit card when asked.

The credit card will not be charged until the credit is exhausted, but it is necessary to verify the user's identity. It can also be used to pay for Google services when the credit runs out.

Once an account is set up, the next step is to enable the Google Kubernetes Engine (GKE) API so that we can interact with it from the command line. The steps are as follows:

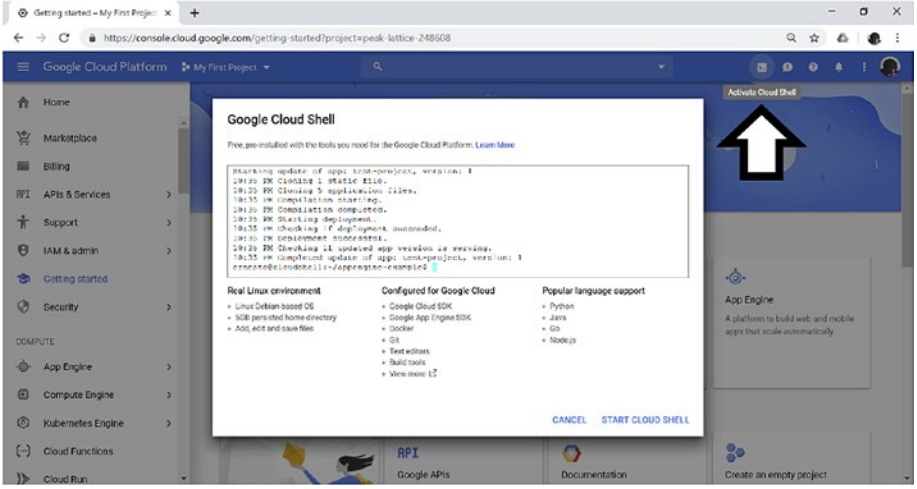
1. Go to `https://console.google.com`.
2. Click the hamburger button menu on the top left corner to display the main menu.
3. Select "Kubernetes Engine" which is located under the "COMPUTE" header and which should redirect the browser to `https://console.cloud.google.com/kubernetes`.
4. Select "Clusters."
5. Look for a message that says "Kubernetes Engine API is being enabled."

Please do not click "Create cluster," "Deploy container," or other similar options since we will be working strictly from the command line as explained in the next section.

Using the Google Cloud Shell

Google follows a *code first* philosophy. This is because whatever actions we perform on GCP—such as launching a Kubernetes cluster—we prefer to script to avoid repeating ourselves in the future. It follows that learning a GUI-based workflow, to then relearn the same equivalent workflow using the command line, is an unnecessary duplication of efforts. With the aim of embracing the code first approach, we will be using the Google Cloud Shell as the *de facto* environment for all examples in this book.

The Google Shell Environment, being a first-class citizen feature, is always available on the top menu, to the left, using a command prompt–looking icon as show in Figure 1-1. The reader may wonder whether using the Google Cloud Shell is not a false economy in terms of learning a technology that is tied up only to Google. Not in the slightest.



Google Cloud Shell

Free, pre-installed with the tools you need for the Google Cloud Platform. Learn More

```
Starting update of app: test-project, version: 1
10:35 PM Cloning 1 static file.
10:35 PM Cloning 5 application files.
10:35 PM Compilation starting.
10:35 PM Compilation completed.
10:35 PM Starting deployment.
10:35 PM Checking if deployment succeeded.
10:35 PM Deployment successful.
10:35 PM Checking if updated app version is serving.
10:35 PM Completed update of app: test-project, version: 1
ernesto@cloudshell:~/appengine-example$
```

Real Linux environment
- Linux Debian-based OS
- 5GB persisted home directory
- Add, edit and save files

Configured for Google Cloud
- Google Cloud SDK
- Google App Engine SDK
- Docker
- Git
- Text editors
- Build tools
- View more

Popular language support
- Python
- Java
- Go
- Node.js

CANCEL START CLOUD SHELL

***Figure 1-1.** The Google Cloud Shell icon on the top left menu bar*

The Google Cloud Shell, contrary to the likes of Microsoft PowerShell, is not an alternative to Bash. It is a web-based terminal—think Putty or iTerm running on a web browser—that is automatically connected to a tiny fully featured Linux virtual environment. Said virtual machine gives us a home directory to store our files as well as all the utilities and commands that we need for this book preinstalled and pre-authenticated:

- The `gcloud` command, which is part of the Google SDK and that is already authenticated and pointing to our default project as soon as we log in
- The `kubectl` command, which is part the Kubernetes client-side suite and will be authenticated and pointing to our Google-based cluster as soon as it is created

- The `python3` command, which is used in Chapter 9 to demonstrate StatefulSets
- The `curl` command, which is used to interact with web servers
- The `git` command, to download code examples from GitHub

The steps for installing said packages in a local machine depend on whether the reader's machine runs Windows, macOS, or Linux—as well as which specific distribution in the latter case. Google provides instructions for each of operating system at `https://cloud.google.com/sdk/install`. We recommend that the reader only sets up a local environment after finishing the book. Likewise, if the reader is running Microsoft Windows 10, we highly advice the use of *Windows Subsystem for Linux* (WSL) rather than Cygwin or other pseudo-Unix environment. For more information about what the benefits of WSL are, and how to install it, please refer to `https://docs.microsoft.com/en-us/windows/wsl/`.

Note Many examples in this book prompt the reader to open multiple windows or tabs to observe the real-time behavior of various Kubernetes' objects. The Google Cloud Shell allows opening multiple tabs, but they are not useful for "side by side" comparisons. It is often more convenient to use TMUX, which is preinstalled and running by default. TMUX is a terminal multiplexer which allows to divide a screen into panels, among many other features. For the scope of this book, the following TMUX commands should be enough:

Split the screen horizontally

Ctrl+B (once) and then " (double quote character)

Split the screen vertically

Ctrl + B (once) and then % (percent character)

Move across open panels (so they are selected)

Ctrl + B (once) and then arrow keys

Increase/decrease the size of the selected panel

Ctrl + B (once) and then Ctrl+arrow keys

Close panel

Type `exit` on selected panel

For more information on TMUX, please type `man tmux`

Downloading Source Code and Running Examples

The large program listings included in this book include a comment with a file name normally as the first or second row, depending on whether the first row is used to invoke a command—using the *shebang* notation—as in the following case:

```
#!/bin/sh
# create.sh
gcloud container clusters create my-cluster \
  --issue-client-certificate \
  --enable-basic-auth \
  --zone=europe-west2-a
```

The source code for all files included in this book is located on GitHub at https://github.com/egarbarino/kubernetes-gcp. Google Cloud Shell has Git installed by default. The following sequence of commands clone the repo to the local home directory and provide a listing of folders, one for each chapter:

```
$ cd ~ # Be sure we are in the home directory
$ git clone \
    https://github.com/egarbarino/kubernetes-gcp.git
$ cd kubernetes-gcp
$ ls
chp1  chp2  chp3  chp4  chp5  chp6  chp7  chp8 ...
```

For brevity, the code listings throughout this text do not include the chapter number as part of the file name. For example, the code seen before shows `create.sh` rather than `chp1/create.sh`; the reader is expected to change to the relevant directory before executing code for a given chapter. For example:

```
$ cd chp1
$ ls
create.sh  destroy.sh
```

Creating and Destroying a Kubernetes Cluster

Before we create a Kubernetes cluster, we have to decide the geographical location in which we want to run it. Google uses the term *region* to refer to the geographical location (e.g., Dublin as opposed to London) and *zone* for the segregated (in terms of power supplies, networks, compute, etc.) environment within such regions. However, the zone *identifier* includes both the region and the zone. For example, `europe-west2-a` selects the zone a within `europe-west2` which, in turn, identifies a Google data center in London, UK. Here, we will use the `gcloud config set compute/zone <ZONE>` command.

```
$ gcloud config set compute/zone europe-west2-a
Updated property [compute/zone]:
```

The same setting can be provided as a flag, as we will learn in a few moments. Now we are ready to launch our Kubernetes cluster using the `gcloud container clusters create <NAME>` command:

```
$ gcloud container clusters create my-cluster \
    --issue-client-certificate \
    --enable-basic-auth
Creating cluster my-cluster in europe-west2-a...
Cluster is being health-checked (master is healthy)
done
kubeconfig entry generated for my-cluster.
NAME       LOCATION       MASTER_VERSION NUM_NODES
my-cluster europe-west2-a 1.12.8-gke.10  3
```

The default cluster consists of three virtual machines (called *Nodes*). Before Kubernetes 1.12, the `gcloud container clusters create` command, on its own, was sufficient to create a simple Kubernetes cluster, but now, it is necessary to use explicit flags so that the user is made aware that the default settings are not necessarily the most secure. In particular

- The `--issue-client-certificate` flag is necessary to avoid the complication of setting up custom certificates.
- The `--enable-basic-auth` flag is necessary to avoid setting up a more robust authentication mechanism.

Other flags may or not may be necessary depending on whether we have global defaults in place:

- The `--project=<NAME>` flag indicates whether we want to create the cluster in some other project rather than the default one. The Google Cloud Shell will normally point to the default project automatically. If in doubt, we can list the number of projects by issuing the `gcloud project lists` command. Likewise, if we want to change the default project permanently, we can use the `gcloud config set project <NAME>` command.
- The `--zone=<ZONE>` flag indicates the compute zone in which the cluster will be created. The complete list of available zones can be obtained by issuing the `gcloud compute zones list` command. However, for a more human-friendly list, that includes the actual cities and countries in which the facilities are located, the URL `https://cloud.google.com/compute/docs/regions-zones/` is more useful. The default zone may be specified using the `gcloud config set compute/zone <ZONE>` command beforehand, which is what we have exactly done in the previous example. The provided scripts under this chapter's folder, `create.sh` and `destroy.sh`, set the zone using this flag. Such scripts may be altered by the reader to suit their preference.
- The `--num-nodes=<NUMBER>` flag sets the number of virtual machines (Nodes) that the Kubernetes cluster will consist of. This is useful to experiment with smaller and larger clusters especially in high-availability scenarios.

- The --cluster-version=<VERSION> flag specifies the Kubernetes release for the master and slave Nodes. The command gcloud container get-server-config lists the currently available versions. This book has been tested against the 1.13 version. If this flag is ignored, the server will pick the most recent stable version. If some of the examples fail because of backward-breaking changes introduced in newer versions, the reader can specify whichever is the latest build under the 1.13 release, when running the gcloud container get-server-config command. Alternatively, the reader can use the provided misc/create_on_v13.sh script under this chapter's folder.

Once we are done with our Kubernetes cluster, we can dispose of it by issuing the gcloud container clusters delete <NAME> command. It is also useful to add the --async and --quiet flags so that the deletion process takes place in the background and without polluting the screen with text, respectively:

```
$ gcloud container clusters delete my-cluster \
    --async --quiet
```

The author recommends running a Kubernetes cluster only as long as it is necessary to run the examples, to make the most out of the allocated credit. Leaving a three Node Kubernetes cluster running for a few days, by mistake, can easily rank up a bill worth over $100, vaporizing a third of the free credit in the process.

All of the examples in the text assume a Kubernetes cluster named my-cluster. For convenience, the scripts create.sh and destroy.sh are included under this chapter's folder. There is also a script called misc/create_on_v13.sh should GKE select a default version that is way too disconnected from 1.13 and makes reproducing examples impossible.

Thinking in Kubernetes

In Kubernetes, nearly every component type is implemented as a generic *resource* that, in this text, we also often call *Kubernetes object*, or *object* for short—because resources have attributes that we can examine and change. Resources are managed in a relatively horizontal fashion; most of our day-to-day actions using the kubectl command involve

- Listing existing objects
- Examining an object's properties
- Creating new objects and altering an existing object's properties
- Deleting objects

Let us start with the listing of objects. We can list the supported resource types using the kubectl api-resources command. These aren't the instances themselves—yet—but the type of objects that exist in the Kubernetes world:

```
$ kubectl api-resources
NAME          SHORTNAMES  NAMESPACED  KIND
...
configmaps    cm          true        ConfigMap
endpoints     ep          true        Endpoints
events        ev          true        Event
limitranges   limits      true        LimitRange
namespaces    ns          false       Namespace
nodes         no          false       Node
...
```

Note that one of the resource types is called nodes. Nodes are the computing resources that support our Kubernetes cluster, so objects of this class are guaranteed to exist beforehand—unless we had decided to set up a cluster with no worker Nodes. To list object instances, we use kubectl get <RESOURCE-TYPE> command. In this case, <RESOURCE-TYPE> is nodes:

```
$ kubectl get nodes
NAME                  STATUS  AGE  VERSION
gke-*-4ff6f64a-6f4v  Ready   92m  v1.12.8-gke.10
gke-*-4ff6f64a-d8nx  Ready   92m  v1.12.8-gke.10
gke-*-4ff6f64a-nw0m  Ready   92m  v1.12.8-gke.10
```

The <RESOURCE-TYPE> has also a *shortname* which is no for nodes. Kubectl normally also accepts singular (in addition to plural) versions of the same resource type. For instance, in the case of Nodes, kubectl get nodes, kubectl get node, and kubectl get no are all equivalent.

Once that we have obtained a listing of object instances, we can examine the specific properties of a given, selected object. Here we use the command kubectl describe <RESOURCE-TYPE>/<OBJECT-IDENTIFIER> to obtain a quick onscreen summary. We select the first Node, so <RESOURCE-TYPE> will be nodes, and <OBJECT-IDENTIFIER> will be gke-*-4ff6f64a-6f4v:

```
$ kubectl describe nodes/gke-*4ff6f64a-6f4v
...
Addresses:
  InternalIP:   10.154.0.23
  ExternalIP:   35.197.220.185
  ...
System Info:
  Machine ID:       af2b11a0b29eb76e2592b4dd64f16308
  System UUID:      AF2B11A0-B29E-B76E-*-B4DD64F16308
  Boot ID:          22ba7558-7748-45f4-*-789d16d343d2
  Kernel Version:   4.14.127+
  Operating System: linux
  Architecture:     amd64
...
```

The actual object's underlying logical attribute structure may be obtained either in JSON or YAML formats using the `kubectl get <RESOURCE-TYPE>/<OBJECT-IDENTIFIER>` command by adding the `-o json` or `-o yaml` flags, respectively. For example:

```
$ kubectl get nodes/gke-*-4ff6f64a-6f4v -o yaml
kapiVersion: v1
kind: Node
metadata:
  name: gke-my-cluster-default-pool-4ff6f64a-6f4v
  resourceVersion: "19152"
  uid: 89b70bf4-c330-11e9-9bab-42010a9a0178
  ...
spec:
  podCIDR: 10.0.0.0/24
  ...
status:
  addresses:
  - address: 10.154.0.23
    type: InternalIP
  - address: 35.197.220.185
    type: ExternalIP
...
```

The syntax and description for each property can be learned using the `kubectl explain <PROPERTY>` command where `<PROPERTY>` consists of a dot-separated string in which the first element is the resource type and the subsequent elements are the various nested attributes. For example:

```
$ kubectl explain nodes.status.addresses.address
KIND:     Node
VERSION:  v1
```

```
FIELD:    address <string>

DESCRIPTION:
     The node address.
```

In terms of altering an object's properties, most commands such as kubectl scale (covered in Chapter 3) ultimately change one or more object's properties under the hood. However, we can modify the properties directly using the kubectl patch <RESOURCE-TYPE>/<OBJECT-IDENTIFIER> -p '<NEW-JSON>' command. In the following example, we add a new label called greeting with the value hello to gke-*-4ff6f64a-6f4v:

```
$ kubectl patch nodes/gke-*-4ff6f64a-6f4v \
    -p '{"metadata":{"labels":{"greeting":"hello"}}}'
node/gke-*-4ff6f64a-6f4v patched
```

The results can be checked using the kubectl get command shown a few moments ago. In most cases, though, the use of kubectl patch is seldom required given that there is either a dedicated command that will indirectly produce the intended change—such as kubectl scale—or we will refresh the object's entire set of attributes using the kubectl apply command. This command, as well as kubectl create, will be covered extensively throughout this book.

Finally, disposing of objects involves running the kubectl delete <RESOURCE-TYPE>/<OBJECT-IDENTIFIER> command. Given that the Node named gke-*-4ff6f64a-6f4v is a treated as any other regular object, we can go ahead and delete it:

```
$ kubectl delete nodes/gke-*-6f4v
node "gke-*-4ff6f64a-6f4v" deleted
```

Adding the `--force` flag will speed up the process, and the `--all` flag can be used when `<OBJECT-IDENTIFIER>` is omitted so that all object instances under the declared resource type are deleted. Alternatively, `kubectl delete all --all` can be used to wipe out all user-created resources in the *default namespace*, but it will not destroy Nodes which are namespace agnostic. Most object types are declared within specific namespaces; since Pods are the archetypical type of object that is homed in a namespace, we will cover this topic toward the end of Chapter 2.

Note This book will often prompt the reader to "clean up the environment," or to start with a "fresh environment." Such a request may also be included in the form of comments. For example:

```
# Clean up the environment first
$ kubectl apply -f server.yaml
```

There is also the implicit assumption that the reader will start each chapter with a clean environment as well. Cleaning up the environment means deleting all objects that have been previously creating in the Kubernetes cluster so that they do not take up resources needed by the example at hand, and to avoid name conflicts as well.

The reader may list all user-created objects in the default namespace by issuing the `kubectl get all –all` command and delete each object one by one or alternatively use the `kubectl delete all –all` command. If this does not work, or the reader feels that the environment may still be polluted with previous artifacts, the solution is to shut down the Kubernetes cluster and start a new one.

How This Book Is Organized

This book consists of nine chapters. Figure 1-2 provides a high-level architectural diagram which works as a pictorial chapter guide:

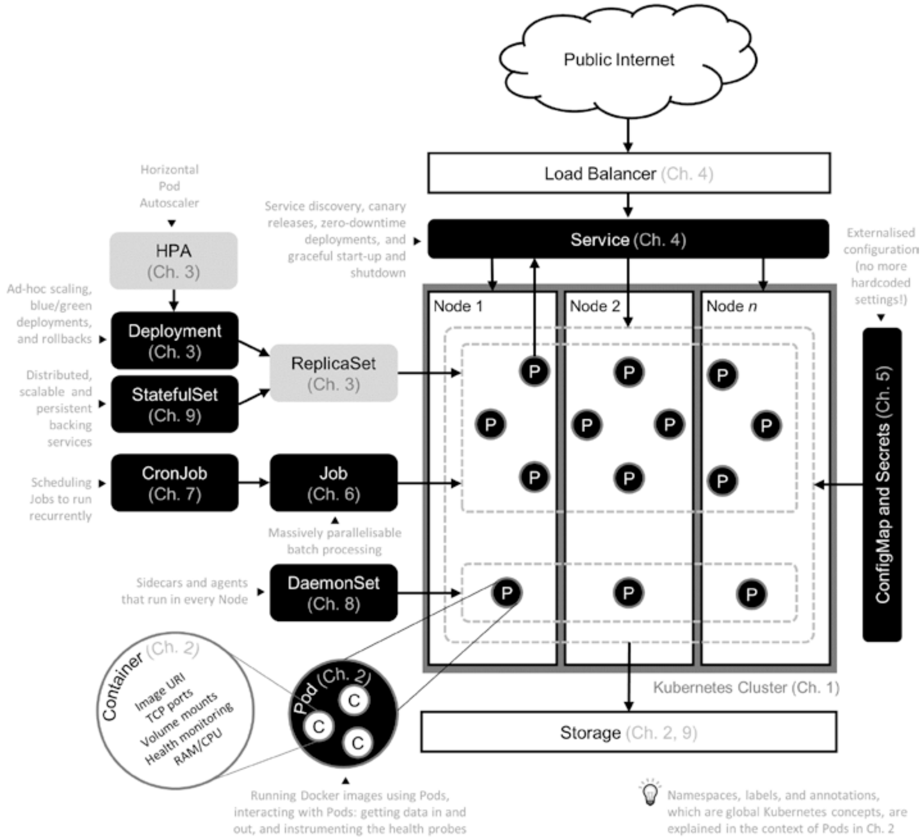
- Chapter 1, “Introduction,” includes the instructions to use the Google Cloud Shell, download the source code examples, and set up a Kubernetes cluster.
- Chapter 2, “Pods,” introduces the Pod resource type, which is the most fundamental building block in Kubernetes. The reader will learn how to instrument Pods to run, from simple one-off commands to web servers. Operational aspects such as setting up CPU and RAM constraints and the organization of Pods using labels and annotations will also be also be covered. By the end of the chapter, the reader will be capable of running and interacting with applications running in a Kubernetes cluster as though they were running on a local machine.
- Chapter 3, “Deployments and Scaling,” helps the reader take Pods to the next level by introducing the Deployment controller which allows the on-demand scaling of Pods and seamless migrations—including blue/green deployments and rollbacks—when upgrading Pod versions. Also, the dynamics of autoscaling are demonstrated using the Horizontal Pod Autoscaler (HPA) controller.
- Chapter 4, “Service Discovery,” teaches the reader how to make Pods available on the public Internet as well as within the Kubernetes cluster. In addition, this chapter explains how the Service controller interacts with the

Deployment controller to facilitate zero-downtime deployments as well as graceful startup and shutdown of applications.

- Chapter 5, "ConfigMap and Secrets," shows how configuration can be externalized away from applications by storing it using the ConfigMap or Secret controllers. Likewise, the Secrets controller capability of storing Docker Registry credentials and TLS certificates will also be examined.
- Chapter 6, "Jobs," looks at the case of running batch processes—different from steady web servers—using the Job controller. Kubernetes' parallelization capabilities, which help reduce the total processing time of large, computational-expensive, multiple-item batch jobs are also given special attention.
- Chapter 7, "CronJobs," describes the CronJob controller which can be instrumented to run Jobs in a recurrent manner, and that relies on the same syntax used by the cron daemon's crontab file found in most Unix-like systems.
- Chapter 8, "DaemonSets," explains how to deploy Pods that are locally available in every Node of a Kubernetes cluster so that consuming Pods can benefit from faster access time either through a local TCP connection or the local file system.
- Chapter 9, "StatefulSets," the final chapter, demonstrates the nature of highly scalable stateful backing services by taking the reader through the process of implementing a primitive key/value store using the StatefulSet controller. By the end of the chapter, the reader will

have an appreciation of why cloud native (managed) data stores offer a nearly unbeatable advantage, and the instrumentation mechanisms that the StatefulSet controller offers should the reader choose to roll out their own data store instead.

Please note that the arrows in Figure 1-2 indicate logical flow rather than a network connection. Likewise, the depicted Nodes are *Worker Nodes*. Master Nodes, and the objects that run in them, are managed by GCP and fall outside the scope of this beginner's book.


Public Internet
Load Balancer (Ch. 4)
Service (Ch. 4)
Service discovery, canary releases, zero-downtime deployments, and graceful start-up and shutdown
Externalised configuration (no more hardcoded settings!)
Horizontal Pod Autoscaler
HPA (Ch. 3)
Ad-hoc scaling, blue/green deployments, and rollbacks
Deployment (Ch. 3)
Distributed, scalable and persistent backing services
StatefulSet (Ch. 9)
ReplicaSet (Ch. 3)
Scheduling Jobs to run recurrently
CronJob (Ch. 7)
Job (Ch. 6)
Massively parallelisable batch processing
Sidecars and agents that run in every Node
DaemonSet (Ch. 8)
Node 1
Node 2
Node n
P
ConfigMap and Secrets (Ch. 5)
Kubernetes Cluster (Ch. 1)
Storage (Ch. 2, 9)
Container (Ch. 2)
Image URI
TCP ports
Volume mounts
Health monitoring
RAM/CPU
Pod (Ch. 2)
C

Running Docker images using Pods, interacting with Pods: getting data in and out, and instrumenting the health probes

Namespaces, labels, and annotations, which are global Kubernetes concepts, are explained in the context of Pods in Ch. 2

Figure 1-2. *Kubernetes high-level architecture and chapter guide*

CHAPTER 2

Pods

Pods are the most atomic unit of work in a Kubernetes cluster. A Pod encompasses one or more containers, which will be deployed together on the same machine and can, thus, use local data exchange mechanisms (Unix sockets, the TCP loopback device, and even memory-backed shared folders) for faster communication.

Not only containers grouped in Pods achieve faster communication by avoiding a network roundtrip, they can also use shared resources such as file systems.

A key characteristic of a Pod is that, once deployed, they share the same IP address and port space. Unlike vanilla Docker, containers within a Kubernetes Pod do not run in an isolated virtual network.

From a conceptual perspective, it is worth understanding that *Pods are the runtime objects that live in specific Nodes*. A Pod deployed into Kubernetes is not an "image" or a "disk" but the actual, tangible, CPU-cycle consuming, network-accessible resource.

In this chapter, we will first look at how to launch and interact with Pods as though they were local Linux processes; we will learn how to specify arguments, pipe data in and out, and connect to an exposed network port. We will then look at more advanced aspects of Pod management, such as interacting with multi-container Pods, setting CPU and RAM constraints, mounting external storage, and instrumenting health checks. Finally, we will show the usefulness of *labels* and *annotations*, which help tag, organize, and select not only Pods but most other Kubernetes object types as well.

E. Garbarino, *Beginning Kubernetes on the Google Cloud Platform*,
https://doi.org/10.1007/978-1-4842-5491-2_2

The Fastest Way to Launch a Pod

The quickest way to launch a Pod, using the least amount of keystrokes, is by issuing the `kubectl run <NAME> --image=<URI>` command where `<NAME>` is the *Pod's prefix* (more on this in a few moments) and `<URI>` is either a Docker Hub image such as `nginx:1.7.9` or a fully qualified Docker URI in some other Docker Registry such as Google's own; for example, the Google's *Hello World* Docker image is located at `gcr.io/google-samples/hello-app:1.0`.

For simplicity, let us launch a Pod running the latest version of the Nginx web server from Docker Hub:

```
$ kubectl run nginx --image=nginx
deployment.apps/nginx created
```

Now, even though this is the easiest way of running a Pod, it actually produces a more complex setup than what we might need. Specifically, it creates a Deployment controller and a ReplicaSet controller (to be covered in Chapter 3). The ReplicaSet controller, in turn, controls exactly one Pod that is assigned a random name: `nginx-8586cf59-8t9z9`. We can check the resulting Deployment, ReplicaSet, and Pod objects using the `kubectl get <RESOURCE-TYPE>` command where `<RESOURCE-TYPE>` is `deployment`, `replicaset` (for ReplicaSet), and `pod`, respectively:

```
$ kubectl get deployment
NAME   DESIRED CURRENT UP-TO-DATE AVAILABLE AGE
nginx  1       1       1          1         0s
```

```
$ kubectl get replicaset
NAME            DESIRED  CURRENT  READY  AGE
nginx-8586cf59  1        1        1      5m
```

```
$ kubectl get pod
NAME                 READY STATUS  RESTARTS AGE
nginx-8586cf59-8t9z9 1/1   Running 0        12s
```

Although this is the easiest way to run a Pod, in terms of brevity, Pods are assigned random names that we need to figure out using `kubectl get pod` or other mechanism such as label selectors—covered at the end of this chapter. Furthermore, we can't delete the Pod directly when we are done with it because the Deployment and ReplicaSet controllers will recreate it again. In fact, to dispose of a Pod we've just created, we need to delete the Pod's Deployment object rather than the Pod itself:

```
$ kubectl delete deployment/nginx
deployment.extensions "nginx" deleted
```

The reason as to why `kubectl run` command creates a Deployment by default is because of its restart policy, which is controlled by the `--restart=<VALUE>` flag, and it is set to `Always` if left unspecified. If we set `<VALUE>` to `OnFailure`, instead, the Pod will only be restarted only if it fails. The `OnFailure` policy, though, does not create a clean Pod either though. A Job object (a controller, like a Deployment) is created which we will cover when we reach Chapter 6. The third and last possible value is `Never` which creates exactly one single Pod and nothing else; this is exactly what we need for the scope of this chapter.

Note Kubernetes will deprecate in the future the behavior wherein the `kubectl run` command creates a Deployment by "accident" whenever the `--restart` flag is omitted. We will revisit this topic again when we reach Chapter 3.

Launching a Single Pod

The kubectl run <NAME> --image<IMAGE> --restart=Never command (please note the --restart=Never flag) creates a single Pod and no other additional objects. The resulting Pod will be named exactly as the provided <NAME> argument:

```
$ kubectl run nginx --image=nginx --restart=Never
pod/nginx created
```

```
$ kubectl get pod
NAME      READY     STATUS    RESTARTS   AGE
nginx     1/1       Running   0          0s
```

How to access the Nginx web server itself is something we will cover in a few sections ahead. For now, it suffices to know that Nginx is running as per its Dockerfile initialization settings. We can override the entry command by specifying a single command as the last argument of kubectl run. For example:

```
# Clean up the environment first
$ kubectl run nginx --image=nginx --restart=Never \
    /usr/sbin/nginx
pod/nginx created
```

The problem here, though, is that by default, the nginx starts the process inside the container and exits with a success status code which ends the execution of the Pod—and therefore the Nginx web server itself:

```
$ kubectl get pod
NAME      READY     STATUS      RESTARTS   AGE
nginx     0/1       Completed   0          0s
```

When running Pods imperatively, we have to keep always in mind that the entry process, be it a web server or other application, must remain suspended in some sort of loop rather than completing and exiting right away. In the case of Nginx, we need to pass the `-g 'daemon off;'` flag. However, passing a command as the last argument doesn't work here since the flags will be interpreted as extra arguments to `kubectl run`. The solution here is to use the `--command` flag and the double hyphen syntax: `kubectl run ... --command -- <CMD> [<ARG1> ... <ARG2>]`. After the double hyphen, `--`, we can write a long command with arguments:

```
# Clean up the environment first
$ kubectl run nginx --image=nginx --restart=Never \
    --command -- /usr/sbin/nginx -g 'daemon off;'
pod/nginx created

$ kubectl get pod
NAME     READY    STATUS    RESTARTS   AGE
nginx    1/1      Running   0          0s
```

If the `--command` flag is not present, then the arguments after `--` will be considered to be the arguments applicable to the image' Dockerfile command. Note, however, that most images do not have an entry command, so the `--command` flag is seldom necessary. In other words, the first argument after `--` is typically interpreted as a command even in the absence of the `--command` flag since public Docker images, such as those found on Docker Hub, seldom declare an `ENTRYPOINT` attribute. More on this in a few sections ahead.

Note The presented example shows HTTP logs which require a HTTP client such as `curl` to interact with the Pod first. The reader will not experience any output when attaching to a Pod using the Nginx image otherwise. The exposure of a Pod's network ports and its interaction with an external TCP client will be explained a few sections ahead in this chapter and in in-depth in Chapter 4.

Another consequence of running Pods using the simple syntax shown earlier is that Pods are sent to the background and we don't get to see their output. We can attach to the running container's first process by issuing the `kubectl attach <POD-NAME>` command. For example:

```
$ kubectl attach nginx
Defaulting container name to nginx.
127.0.0.1 - [12:16:00] "GET / HTTP/1.1" 200 612
127.0.0.1 - [12:16:01] "GET / HTTP/1.1" 200 612
127.0.0.1 - [12:16:01] "GET / HTTP/1.1" 200 612
...
```

Ctrl+C will return control to us. We can also use `kubectl logs <POD-NAME>` which we will treat separately further ahead. To dispose from a Pod running in the background, the `kubectl delete pod/<NAME>` command is used.

Launching a Single Pod to Run a Command

Pods run in the background by default. Debugging them requires that we attach to them, query their logs, or run a shell inside of them—before they terminate. However, we can launch a Pod and stay attached to it so that we can immediately see its output. This is achieved by adding the `--attach` flag:

```
$ kubectl run nginx --image=nginx --restart=Never \
    --attach
127.0.0.1 - [13:11:03] "GET / HTTP/1.1" 200 612
127.0.0.1 - [13:11:04] "GET / HTTP/1.1" 200 612
127.0.0.1 - [13:11:05] "GET / HTTP/1.1" 200 612
```

Please note, as commented before, that we will not see traffic logs unless we access the nginx web server which we will explain in a few moments.

Logs such as those generated by an HTTP server are typically being continuously generated; what if we wanted to simply run a command and then forget about the Pod? Well, in principle we just need to use the `--attach` flag and pass the desired command to the Pod's container. Let us, for example, use the `alpine` Docker image from Docker Hub to run the `date` command:

```
$ kubectl run alpine --image=alpine \
    --restart=Never --attach date
Fri Sep 21 13:25:02 UTC 2018
```

Note The `alpine` Docker image is based on Alpine Linux. It is only 5 MB in size, which is an order of magnitude smaller than a vanilla `ubuntu` image—the go-to choice of beginner Docker users. The `alpine` image has the benefit of offering access to a package repository that is fairly complete whenever extra utilities are required.

Considering that we are not passing arguments to `date`, we can just use it as the last argument to `kubectl` rather than using the double hyphen `--` syntax. However, what happens if we want to know what the date and time are again?

```
$ kubectl run alpine --image=alpine \
    --restart=Never --attach date
Error from server (AlreadyExists): pods "alpine"
already exists
```

Oh, this is certainly annoying; we can't run a Pod for the second time using the same name. This issue may be solved by deleting the Pod using the `kubectl delete pod/alpine` command, but this gets tedious after a while. Luckily, the Kubernetes team thought about this use case and added an optional flag, `--rm` (remove), which results in the Pod being deleted after the command ends:

```
$ kubectl run alpine --image=alpine \
    --restart=Never --attach --rm date
Fri Sep 21 13:30:35 UTC 2018
pod "alpine" deleted
```

```
$ kubectl run alpine --image=alpine --restart=Never --attach
--rm date
Fri Sep 21 13:30:40 UTC 2018
pod "alpine" deleted
```

Please note that `--rm` only works when launching a Pod in attached mode and that Ctrl+C will not terminate the Pod if it is running in a loop, like the Nginx process does by default.

So far so good. Now we know how to run one-off commands in Kubernetes as though it was a local Linux box. Our local Linux box, though, does not only run fire-and-forget commands but allows to pipe input into them. For instance, let us say that we wanted to run the `wc` (Word Count) command and provide as an input, the *local* `/etc/resolv.conf` file:

```
$ wc /etc/resolv.conf
  4  24 182 /etc/resolv.conf
```

```
$ cat /etc/resolv.conf | \
    kubectl run alpine --image=alpine \
     --restart=Never --attach --rm wc
        0         0         0
pod "alpine" deleted
```

The previous example didn't work. Why? This is because the `--attach` flag only attaches the Pod's container *STDOUT* (Standard Output) to the console but not *STDIN* (Standard Input). To pipe STDIN into our Alpine-based Pod, a different flag, `--stdin` or `-i` for short, is required. The `-i` flag also sets `--attach` to true automatically, so there is no need to use both:

```
$ cat /etc/resolv.conf | \
    kubectl run alpine --image=alpine \
    --restart=Never -i --rm wc
        4        24       182
pod "alpine" deleted
```

Running a Pod Interactively

So far we have seen how to run background applications like web servers and one-off commands. What about if we wanted to execute commands interactively either through the shell or by initiating a command such as the `mysql` client? Then, all we have to do is specify that we want a terminal using the `--tty` flag or `-t`. We can combine both `-t` and `-i` in one flag, resulting in `-ti`:

```
$ kubectl run alpine --image=alpine \
    --restart=Never -ti --rm sh
If you don't see a command prompt, try pressing enter.
/ # ls
bin    etc    lib    mnt    root   sbin   sys    usr
dev    home   media  proc   run    srv    tmp    var
/ # date
Fri Sep 21 16:16:03 UTC 2018
/ # exit
pod "alpine" deleted
```

Interacting with an Existing Pod

As we had mentioned before, we can use the kubectl attach command to get access to the Pod's container main process, but this does not allow for running other commands: we can only contemplate the output of the process that is already running in a passive manner. Running a shell and then attaching to it does not work because the shells exits immediately unless we create an artificial loop:

```
$ kubectl run alpine --image=alpine \
    --restart=Never sh
pod/alpine created

$ kubectl attach alpine
error: cannot attach a container in a completed pod; current
phase is Succeeded
```

Let us now go ahead, create an artificial loop, and try to attach again:

```
# Clean up the environment first

$ kubectl run alpine --image=alpine \
    --restart=Never -- \
    sh -c "while true; do echo 'doing nothing' ; \
    sleep 1; done"
pod/alpine created

$ kubectl attach alpine
Defaulting container name to alpine.
Use 'kubectl describe pod/alpine -n default' to see
all of the containers in this pod.
If you don't see a command prompt, try pressing enter.
doing nothing
doing nothing
doing nothing
...
```

Now that the container stays on the *Running* status (we can use `kubectl get pod` just to be sure) and therefore commands can be run against it using the `kubectl exec <POD-NAME> <COMMAND>` command:

```
$ kubectl get pod
NAME      READY     STATUS    RESTARTS   AGE
alpine    1/1       Running   0          30s
```

```
$ kubectl exec alpine date
Fri Sep 21 16:50:58 UTC 2018
```

The `kubectl exec` command, similarly to `kubectl run`, takes the `-i` flag so that it can pipe STDIN into a Pod's container:

```
$ cat /etc/resolv.conf | kubectl exec alpine -i wc
        4        24       182
```

The `-t` flag can be used to open a console and run a new shell so that we can perform troubleshooting drills and/or run new commands directly inside of the Pod's container:

```
$ kubectl exec alpine -ti sh
/ # ps
PID USER     TIME  COMMAND
  1 root     0:00 sh -c while true; do ...
408 root     0:00 sh
417 root     0:00 sleep 1
418 root     0:00 ps
/ # exit
```

Retrieving and Following a Pod's Logs

In Kubernetes, a Pod's log is the output of a container's first process (which runs with PID 1) rather than a physical log file (e.g., a file with the `.log` extension somewhere in `/var/log`). In principle, we could just use `kubectl attach`, but this command does not remember the output that was produced before issuing it. We can only see the output from the moment of attachment onward.

The `kubectl logs <POD-NAME>`, instead, shows everything that the default container's first process has dumped on STDOUT since it has initiated—barring buffering limits that are outside the scope of this book:

```
$ kubectl logs alpine
doing nothing
doing nothing
doing nothing
...
```

If we count the lines emitted by `kubectl logs alpine`, we will see that they will keep increasing:

```
$ kubectl logs alpine | wc
   1431    2862   20034

# Wait one second
$ kubectl logs alpine | wc
   1432    2864   20048
```

In most cases, though, we want a behavior similar to `kubectl attach`. Yes, we want to know what happened *before*, but once we catch up, we want to keep watching for new changes, similarly to the `tail -f` Unix command. Well, just like in the case of `tail`, the `-f` flag allows us to "follow" the log as more output is produced:

```
$ kubectl logs -f alpine
doing nothing
doing nothing
...
```

In this example, the command prompt will not appear until we abort the session by pressing Ctrl+C.

Interacting with a Pod's TCP Port

In previous sections, we had seen examples of launching a Pod containing the Nginx web server:

```
# Clean up the environment first

$ kubectl run nginx --image=nginx \
    --restart=Never --rm --attach
127.0.0.1 - [06:22:08] "GET / HTTP/1.1" 200 612
127.0.0.1 - [06:22:44] "GET / HTTP/1.1" 200 612
127.0.0.1 - [06:22:45] "GET / HTTP/1.1" 200 612
```

Now, how do we access the web server in the first place, say, by using the `curl` command, so that we can generate the requests we see in the shown output? Well, it depends on whether we want to access the Pod from our local computer or from another Pod within the Kubernetes cluster.

Let us start with the first case. When accessing a Pod from our local computer, we need to create a bridge (called a port forward) from some local available port, say `1080`, to the `nginx` Pod, which is `80` by default.

The command used for this purpose is kubectl port-forward <POD-NAME> <LOCAL-PORT>:<POD-PORT>

```
# Assume the nginx Pod is still running
```

```
$ kubectl port-forward nginx 1080:80
Forwarding from 127.0.0.1:1080 -> 80
Forwarding from [::1]:1080 -> 80
```

Now, on a different window, we can interact with the nginx Pod by accessing our current local port 1080:

```
# run on a different window, tab or shell
$ curl http://localhost:1080
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
```

The second case is that of accessing the Pod's TCP port from inside another Pod rather than our local computer. The challenge here is that unless we set up a Service (covered in Chapter 4), the Pod names do not automatically become reachable hostnames:

```
$ kubectl run alpine --image=alpine \
    --restart=Never --rm -ti sh
/ # ping nginx
ping: bad address 'nginx'
```

What we need, instead, is to find out the Pod's IP address. Every Pod is assigned a unique IP address so that there are not port collisions among different Pods. The quickest way to find out a Pod's IP address is just by issuing the kubectl get pod command with the -o wide flag which includes the IP column:

```
$ kubectl get pod -o wide
NAME   READY STATUS  RESTARTS AGE IP
alpine 1/1   Running 0        2m  10.36.2.8
nginx  1/1   Running 0        7m  10.36.1.5
```

Now we can return to our `alpine` window and use `10.36.1.5` rather than `nginx`:

```
/ # ping 10.36.1.5
PING 10.36.1.5 (10.36.1.5): 56 data bytes
64 bytes from 10.36.1.5: seq=0 ttl=62 time=1.370 ms
64 bytes from 10.36.1.5: seq=1 ttl=62 time=0.354 ms
64 bytes from 10.36.1.5: seq=2 ttl=62 time=0.364 ms
...
```

On Alpine, `wget` is preinstalled rather than `curl`, but it serves the same purpose:

```
# wget -q http://10.36.1.5 -O -
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
```

Applying the `-o wide` flag to `kubectl get pod` is fine for brief manual checks but in a scripted, automated scenario, we might want to obtain a Pod's IP address in a programmatic fashion. In this case, we can query the Pod's `pod.status.podIP` field from its JSON representation using the following command:

```
$ kubectl get pod/nginx -o jsonpath \
    --template="{.status.podIP}"
10.36.1.5
```

We will cover the Pod's JSON representation later in this chapter. More information about JSONPath queries can be obtained from http://goessner.net/articles/JsonPath/.

Transferring Files from and to a Pod

In addition to connecting to Pods via TCP, piping data into and out of them, and opening shells inside of them, we can also download and upload files. File transfer (Copy, or cp, in the Kubernetes jargon) is achieved using the kubectl cp <FROM-FILE> <TO-FILE> command where <*-FILE> becomes a Pod source or sink whenever the <POD-NAME>:path format is used.

For example, the nginx's index.html file is downloaded to our current directory as follows:

```
$ kubectl cp \
    nginx:/usr/share/nginx/html/index.html \
    index.html
$ head index.html
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
```

Let us now overwrite this file and upload it back to the nginx Pod

```
$ echo "<html><body>Hello World</body></html>" > \
    index.html
$ kubectl cp \
    index.html \
    nginx:/usr/share/nginx/html/index.html
```

and, finally, set up a bridge to prove the results of our file transfer:

```
$ kubectl port-forward nginx 1080:80
Forwarding from 127.0.0.1:1080 -> 80
# In a different window or tab
```

```
$ curl http://localhost:1080
<html><body>Hello World</body></html>
```

Selecting a Pod's Container

As mentioned in the introduction, a Pod may house multiple containers. In all of the examples we've seen so far, especially when running commands or fetching the logs from an existing Pod, it seemed as though there was a `1:1` relationship between a Pod and a Docker image. For example, when we issued the command `kubectl logs nginx`, it seemed as though the `nginx` Pod and container were the same thing:

```
$ kubectl logs nginx
127.0.0.1 - [06:22:08] "GET / HTTP/1.1" 200 612
127.0.0.1 - [06:22:44] "GET / HTTP/1.1" 200 612
127.0.0.1 - [06:22:45] "GET / HTTP/1.1" 200 612
...
```

Well, this is just Kubernetes being nice and selecting the first and sole container automatically for us. In fact, `kubectl logs nginx` can be considered a simplified version of `kubectl logs nginx -c nginx`. The flag `-c` is a shortcut for `--container`:

```
$ kubectl logs nginx -c nginx
127.0.0.1 - [06:22:08] "GET / HTTP/1.1" 200 612
127.0.0.1 - [06:22:44] "GET / HTTP/1.1" 200 612
127.0.0.1 - [06:22:45] "GET / HTTP/1.1" 200 612
...
```

Now, how do we tell if a Pod has more than one container? The simplest way is just to issue a `kubectl get pod` command and see the RUNNING/DECLARED value under the `READY` column. For example, each Kubernetes' DNS pod consists of four containers, and each one of them is up and running (as suggested by the `4/4` value) in the following example:

```
$ kubectl get pod --all-namespaces
NAMESPACE    NAME              READY  STATUS
default      nginx             1/1    Running
kube-system  fluentd-*-tr69s   2/2    Running
kube-system  heapster-*-5rks2  3/3    Running
kube-system  kube-dns-*-48wxf  4/4    Running
...
```

There is no need to be concerned with *namespaces* now since we will cover this toward the end of this chapter, but it suffices to say that we need to specify the `kube-system` namespace (via the `-n kube-system` flag) whenever we are referring to a nonuser created Pod. User-created Pods, such as `nginx`, live in the `default` namespace unless otherwise specified.

Note As explained in Chapter 1, long identifiers may be shortened by replacing a boilerplate word fragments within said identifiers with an asterisk. In this specific section, the Pod named `kube-dns-5dcfcbf5fb-48wxf` is also referred as `kube-dns-*-48wxf` whenever space constraints apply. Please note that this is not a wildcard syntax; the Pods must be always referenced by their full name.

Getting back to the original discussion, if we try to fetch the logs for (or execute a command) against a four-container Pod such as `kube-dns-5dcfcbf5fb-48wxf`, the illusion of the 1:1 mapping between Pods and containers disappears:

```
$ kubectl logs -n kube-system pod/kube-dns-*-48wxf
Error from server (BadRequest):
  a container name must be specified
  for pod kube-dns-*-48wxf, choose one of:
[kubedns dnsmasq sidecar prometheus-to-sd]
```

As it can be appreciated in the shown result, we are requested to specify a specific Pod which is accomplished using the `-c` flag. Next, we run `kubectl logs` again but specify the `sidecar` container using the `-c sidecar` flag:

```
$ kubectl logs -n kube-system -c sidecar \
    pod/kube-dns-*-48wxf
I0922 06:14:50 1 main.go:51] Version v1.14.8.3
I0922 06:14:50 1 server.go:45] Starting server ...
...
```

In this case, the command has been kind enough to inform us of what containers are available, but not all commands necessarily do this. We can find out the names of the containers by running `kubectl describe pod/<NAME>` and looking the first indented names below `Containers`:

```
$ kubectl describe -n kube-system \
    pod/kube-dns-*-48wxf
...
Containers:
  kubedns:
    ...
  dnsmasq:
    ...
  sidecar:
    ...
  prometheus-to-sd:
    ...
...
```

A more programmatic approach involves querying the Pod's JSON `pod.spec.containers.name` field using JSONPath:

```
$ kubectl get -n kube-system pod/kube-dns-*-48wxf \
  -o jsonpath --template="{.spec.containers[*].name}"
kubedns dnsmasq sidecar prometheus-to-sd
```

Troubleshooting Pods

In all Pod interaction use cases we have examined so far, the assumption has always been that the Pod under consideration had managed to run at least once. What if the Pod does not start at all and there are no logs and neither TCP services like web servers available to us? A Pod may fail to run for a variety of reasons: it may have an erratic startup configuration, it may require excessive CPU and RAM that are not currently available in the Kubernetes cluster, and so on. However, a common reason as to why Pods often fail to initialize is that the referenced Docker image is incorrect. For instance, in the following example, we intentionally misspell our favorite web server as `nginex` (adding an `e` before `x`) rather than `nginx`:

```
# Clean up the environment first

$ kubectl run nginx --image=nginex \
    --restart=Never
pod/nginx created

$ kubectl get pod
NAME     READY    STATUS            RESTARTS   AGE
nginx    0/1      ErrImagePull      0          2s
nginx    0/1      ImagePullBackOff  0          15s
```

Even though `ImagePullBackOff` tell us something about the image, we can find out more details using the `kubectl describe pod/<NAME>`

command. This command provides a comprehensive report and an account on relevant Pod life cycle events at the end:

```
$ kubectl describe pod/nginx
...
Type    Reason  Age     Message
----    ------  ----    -------
...
Normal  Pulling 1m (x3) pulling image "nginex"
Warning Failed  1m (x3) Failed to pull image "nginex"

rpc error: code = Unknown desc =
  Error response from daemon:
   repository nginex not found:
    does not exist or no pull access
 ...
```

In this example we see that `nginex` was not found, and the Pod controller tried three times (`x3`) to fetch the image without success.

The `kubectl describe` command's main advantage is that it summarizes the Pod's most important details in a human-readable report. This is, naturally, not useful whenever we want to capture a specific piece of detail, such as the Node in which the Pod has been allocated or its IP address.

All Kubernetes objects, including Pods, are represented as an object whose attributes can be rendered in both JSON and YAML formats. To obtain said object structure, all we have to use is the regular `kubectl get pod/<NAME>` command and add the `-o json` or `-o yaml` flag, respectively:

```
$ kubectl get pod/nginx -o json | head
{
  "apiVersion": "v1",
  "kind": "Pod",
  "metadata": {
        "annotations": {
            "kubernetes.io/limit-ranger":
               "LimitRanger plugin set:
                cpu request for container nginx"
        },
        "creationTimestamp": "2018-09-22T10:19:10Z",
        "labels": {
            "run": "nginx"
```

```
$ kubectl get pod/nginx -o yaml | head
apiVersion: v1
kind: Pod
metadata:
  annotations:
    kubernetes.io/limit-ranger:
        'LimitRanger plugin set:
           cpu request for container nginx'
  creationTimestamp: 2018-09-22T10:19:10Z
  labels:
    run: nginx
  name: nginx
```

All the attributes follow the hierarchical structure of the JSON format and can be interrogated using the `kubectl explain <RESOURCE-TYPE>[.x][.y][.z]` command where `x.y.z` are nested attributes. For example:

```
$ kubectl explain pod
$ kubectl explain pod.spec
$ kubectl explain pod.spec.containers
$ kubectl explain pod.spec.containers.ports
```

In general, most Kubernetes objects follow a fairly consistent structure:

```
apiVersion: v1 # The object's API version
kind: Pod      # The object/resource type.
metadata:      # Name, label, annotations, etc.
   ...
spec:          # Static properties (e.g. containers)
   ...
status:        # Runtime properties (e.g. podIP)
   ...
```

Specific fields may be retrieved using a JSONPath query specified using the `--template={}` flag and changing the output type to `jsonpath` using the `-o jsonpath` flag. For example:

```
$ kubectl get pod/nginx -o jsonpath \
    --template="{.spec.containers[*].image}"
nginex
```

Pod Manifests

A Pod manifest is a file that describes a Pod's properties declaratively. All Pods are formulated as an object structure. Whenever we use the imperative command such as `kubectl run`, we are, in reality, creating a Pod manifest on the fly. In fact, we can see the resulting manifest by adding the `--dry-run` and `-o yaml` flags to most imperative commands. For example:

```
# Clean up the environment first

$ kubectl run nginx --image=nginx --restart=Never \
    --dry-run=true -o yaml
apiVersion: v1
kind: Pod
```

```
metadata:
  creationTimestamp: null
  labels:
    run: nginx
  name: nginx
spec:
  containers:
  - image: nginx
    imagePullPolicy: IfNotPresent
    name: nginx
    resources: {}
  dnsPolicy: ClusterFirst
  restartPolicy: Never
status: {}
```

We can save this output to a file such as `nginx.yaml` and create the Pod from this file itself by issuing the `kubectl apply -f <MANIFEST>` command:

```
$ kubectl run nginx --image=nginx --restart=Never \
    --dry-run=true -o yaml > nginx.yaml

$ kubectl apply -f nginx.yaml
pod/nginx created

$ kubectl get pods
NAME     READY    STATUS    RESTARTS   AGE
nginx    1/1      Running   0          0s
```

We can also clean up `nginx.yaml` a little bit, by removing empty attributes, those that get sensible defaults and those that are populated only at runtime—all attributes and values under `.status`. The following version, called `nginx-clean.yaml`, is a Pod manifest consisting of the minimum, mandatory number of attributes:

```
# nginx-clean.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - image: nginx
    name: nginx
```

A Pod created using `kubectl apply -f <MANIFEST>` can be deleted by referencing the object name (e.g., `kubectl delete pod/nginx`), but Kubernetes can pick up the object's name from the manifest file directly when using the `kubectl delete -f <MANIFEST>` syntax:

```
$ kubectl delete -f nginx-clean.yaml
pod "nginx" deleted
```

Note In principle, a brand new Pod should be created by issuing the `kubectl create -f <MANIFEST>` command rather than the `apply`-based form used in this textbook. The reason as to why we prefer the `apply`-based form is that it will also update an existing Pod (or other resource type) if it is already running.

Declaring Containers' Network Ports

All containers within a Pod share the same port space. Likewise, even though it is not mandatory to specify port numbers (and naming them), it is a requisite to name ports whenever two or more of them are declared. Moreover, service exposure (Chapter 4) takes less steps when ports are formally declared on the Pods manifests themselves. The bottom line is that declaring network ports is good Pod manifest hygiene, so let us see how to do it in the following example of a manifest named nginx-port.yaml:

```
# nginx-port.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - image: nginx
    name: nginx
    ports:
    - containerPort: 80
      name: http
      protocol: TCP
```

The .containerPort attribute is mandatory. The value of the .protocol attribute is TCP by default. The .name attribute is optional only if there is one port declared as in the presented example. If there are multiple ports, then each must have a distinct name. There are other optional attributes that may be listed using kubectl explain pod.spec.containers.ports.

Setting Up the Container's Environment Variables

Many Docker application images expect the definition of settings in the form of environment variables. Mysql is a good example which requires at least the MYSQL_ROOT_PASSWORD env variable to be present. Environment variables are defined as an array at `pod.spec.containers.env` wherein each element consists of the `name` and `value` attributes:

```
# mysql.yaml
apiVersion: v1
kind: Pod
metadata:
  name: mysql
spec:
  containers:
  - image: mysql
    name: mysql
    ports:
    - containerPort: 3306
      name: mysql
      protocol: TCP
    env:
    - name: MYSQL_ROOT_PASSWORD
      value: mypassword
```

The password is indeed required in order to connect to the MySQL server even when using the internal `mysql` client:

```
$ kubectl apply -f mysql.yaml
pod/mysql created

# Wait until mysql transitions to Running

$ kubectl get pod/mysql
NAME    READY   STATUS    RESTARTS   AGE
mysql   1/1     Running   0          105s
$ echo "show databases" | kubectl exec -i mysql \
    -- mysql --password=mypassword
```

```
Database
information_schema
mysql
performance_schema
sys
```

Overwriting the Container's Command

Early in this chapter, we have seen that the `kubectl run` command allows us to override the Docker image's default command. This is also the mechanism that we use to run arbitrary commands supported by a disposable Docker image. For example, if we just wanted to check the date in UTC, we can proceed as follows:

```
$ kubectl run alpine --image=alpine \
    --restart=Never --rm --attach -- date --utc
Sun Sep 23 11:32:03 UTC 2018
pod "alpine" deleted
```

The same mechanism can also be used to execute simple shell scripts, and it is not limited to running one-off commands. For example, the following shell script prints the current date every second in an infinite loop:

```
$ kubectl run alpine --image=alpine \
    --restart=Never --attach -- \
    sh -c "while true; do date; sleep 1; done"
Sun Sep 23 11:40:59 UTC 2018
Sun Sep 23 11:41:00 UTC 2018
Sun Sep 23 11:41:01 UTC 2018
...
```

In both of these cases, what `kubectl run` is doing is populating the `spec.containers.args` attribute with an array whose first element is the command and the second and subsequent arguments are the command's arguments. We can check this by running `kubectl get pod/<POD-NAME> -o yaml` and looking at the contents below `pod.spec`:

```
$ kubectl get pod/alpine -o yaml | \
    grep "spec:" -A 5
spec:
  containers:
  - args:
    - sh
    - -c
    - while true; do echo date; sleep 1; done
```

When creating our Pod manifests from scratch, we may use the angle bracket array notation, for example, `args: ["sh","-c","while true; do date; sleep 1; done"]`. However, for long scripts, we can use the YAML pipe syntax in addition to the YAML hyphen syntax for array elements shown before. We often use this approach in this text to improve readability. This is one of the many multiline YAML options approaches available. For more information, please refer to https://yaml-multiline.info/, which helps find out which is the best strategy for each given multiline use case.

This capability is useful because it allows us to embed scripts in a more human-readable fashion, as shown in the manifest named `alpine-script.yaml`:

```
# alpine-script.yaml
apiVersion: v1
kind: Pod
metadata:
 name: alpine
```

```
spec:
  containers:
  - name: alpine
    image: alpine
    args:
    - sh
    - -c
    - |
      while true;
        do date;
        sleep 1;
      done
```

We still have to keep in mind that the script will be passed as a single argument, so statement ending tokens such as semicolons are still necessary. We can check the JSON representation of the `alpine-script.yaml` script, to see how it is translated, as follows:

```
$ kubectl apply -f alpine-script.yaml
pod/alpine created
```

```
$ kubectl get pod/alpine -o json | \
    grep "\"args\"" -A 4
    "args": [
        "sh",
        "-c",
        "while true;\n  do date;\n  sleep 1;\ndone\n"
    ],
```

Before we finish this section, it is worth mentioning that Docker images have the concept of an *entry point* command defined in a Dockerfile using the `ENTRYPOINT` declaration. For historical reasons and a difference in terminology between Kubernetes and Docker, the `pod.spec.containers.args` attribute as well as the arguments supplied after `kubectl run ... --`

or `kubectl exec ... --` overrides a Dockerfile's `CMD` declaration. On its own, `CMD` declares both a command and, potentially, its arguments, for example, `CMD ["sh", "-c", "echo Hello"]`. However, if an `ENTRYPOINT` declaration is also present, as a curse to developers, the rules change in a rather twisted way: `CMD` becomes just the default arguments to whatever the entry point command is and, consequently, the Kubernetes' `pod.spec.container.args` attribute.

Most off-the-shelf Docker Hub images, such as `nginx`, `busybox`, `alpine` and so on, *do not* include an `ENTRYPOINT` declaration. But if one is present, then we would need to use `pod.spec.containers.command` to override it and then treat `pod.spec.containers.args` as the arguments to said command. Hopefully, the examples in Table 2-1 will help clarify the difference.

Table 2-1. *The result of Docker and Kubernetes-specified commands*

CMD	ENTRYPOINT	K8S .args	K8S .command	Result
bash	n/a	n/a	n/a	bash
bash	n/a	["sh"]	n/a	sh
n/a	["bash"]	n/a	n/a	bash
["-c","ls"]	["bash"]	n/a	n/a	bash -c ls
n/a	["bash"]	["-c","date"]	n/a	bash -c date
["-c","ls"]	["bash"]	["-c","date"]	n/a	bash -c date
["-c","ls"]	["bash"]	["-c","date"]	["sh"]	sh -c ls

Whenever overriding the Dockerfile's `ENTRYPOINT` is required, together with the imperative forms, `kubectl run` and `kubectl exec`, the `--command` flag must be added so that the first argument after the double hyphen `--` is treated as a command rather than as the first argument to the entry point. For example, the following imperative form

```
# Clean up the environment first

$ kubectl run alpine --image=alpine \
    --restart=Never --command -- sh -c date
pod/alpine created

$ kubectl logs alpine
Tue Sep 25 20:28:22 UTC 2018
```

is equivalent to the following declarative one:

```
# alpine-mixed.yaml
apiVersion: v1
kind: Pod
metadata:
 name: alpine
spec:
  containers:
  - name: alpine
    image: alpine
    command:
    - sh
    args:
    - -c
    - date
$ kubectl apply -f alpine-mixed.yaml
pod/alpine created

$ kubectl logs alpine
Tue Sep 25 20:30:10 UTC 2018
```

In practice, as explained earlier, it is rarely necessary to deal with the permutations presented in Table 2-1 since most Docker images do not declare the troublesome `ENTRYPOINT` argument, and therefore, it is customary to simply use `pod.spec.containers.args` as an array where the first element is the command and the second and following ones are its arguments.

Managing Containers' CPU and RAM Requirements

Whenever we launch a Pod, Kubernetes finds a Node with sufficient CPU and RAM resources to run the containers declared within said Pod. Likewise, whenever a Pod container is running, Kubernetes does not normally allow it to take over the entire Node's CPU and memory resources in detriment of other containers running in the same Node.

If we don't specify any CPU or memory bounds, Pod's containers will normally be assigned default values typically defined using a namespace-wide *LimitRanger* object—which is outside the scope of this book.

Why is it necessary to exert fine-grained control of compute resources rather than letting Kubernetes use default values? Because we want to be frugal when it comes to our Kubernetes' compute resources. Every Node is typically supported by an entire virtual machine (or even a physical one in extreme cases) that must be funded even if no containers are running on it.

This means that for a production system, letting Kubernetes assign arbitrary CPU and memory bounds for our Pods' containers is a not a good cost and utilization strategy. For instance, a small C or Golang application may require a handful of megabytes, whereas a single monolithic, containerized Java application may require over 1GB all by itself. In the first case, we want to tell Kubernetes to allocate only the bare minimum required resources: in other words, it is preferable to allocate a much smaller compute footprint to a C or Golang application than to a Java one—all things being equal.

Let us now cut to the chase and show what a Pod manifest looks like, which includes explicit bounds for CPU and memory:

```
# nginx-limited.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
    - image: nginx
      name: nginx
      # Compute Resource Bounds
      resources:
        requests:
          memory: "64Mi"
          cpu: "500m"
        limits:
          memory: "128Mi"
          cpu: "500m"
```

We can see that on `nginx-limited.yaml`, we specify the `.memory` and `.cpu` attributes twice, once under `pod.spec.resources.requests` and then again under `pod.spec.resources.limits`. What is the difference? Well, the difference is that the first is a *prerequisite* bound, whereas the second one is a *runtime* bound. *Requests* define the minimum level of compute resources that must be available in a Node *before* Kubernetes deploys the Pod and associated containers. *Limits*, instead, establish the maximum level of compute resources that the containers will be allowed to take *after* they have been deployed into a Node by Kubernetes.

Let us now discuss, in a bit more detail, the way in which CPU and memory bounds are expressed.

CPU resources are measured in *cpu units* which is the same magnitude used by AWS and Azure as well (vCPU and vCore, respectively). However, it is also equivalent to one hyperthread on an Intel CPU—in this case, it may not be a single physical core.

The default metric used by Kubernetes is *millicores*, and values are suffixed with an *m*. One millicore (1000m) allocates exactly one CPU unit. Some examples are shown in Table 2-2.

Table 2-2. *CPU allocation for sample millicore values*

Example	Result	Meaning
64m	64/1000 = 0.064	6.4% of a CPU core
128m	128/1000 = 0.128	12.8% of a CPU core
500m	500/1000 = 0.5	50% of a CPU core
1000m	1000/1000 = 1	Exactly one CPU core
2000m	2000/1000 = 2	Exactly two CPU cores
2500m	2500/1000 = 2.5	Two CPU cores + 50% of another CPU core

Fractions are also allowed, for example, a 0.5 value would be interpreted as 500m. However, millicores appears to be the preferred choice by the Kubernetes community judging by most online examples.

Let us now turn our attention to memory. Memory, unlike CPU, always defines an absolute value rather than a relative one. Memory is ultimately specified in bytes, but larger units of measurement are typically used. Values can be specified both in decimal and binary forms: see Table 2-3.

Table 2-3. *The result, in bytes, for sample memory values*

Suffix	Value	Example	Example in Bytes
n/a	512	512	512
K (kilo)	1000	128K	128,000
Ki (kibi)	1024	128Ki	131,072
M (mega)	1000^2	128M	128,000,000
Mi (mebi)	1024^2	128Mi	134,217,728
G (giga)	1000^3	1G	1,000,000,000
Gi (gibi)	1024^3	1Gi	1,073,741,824

One last comment about *requests* and *limits* is that they are not mutually exclusive. *Requests* specify the bounds under which the container is meant to run under *normal circumstances,* whereas limits represent a *maximum ceiling*. In the case of CPU, most Kubernetes implementations, including GKE, will typically throttle the container, but memory consumption beyond the limit value may result in an abrupt termination.

Note Specifying an overly pessimistic `pod.spec.resources.limits` value may lead to catastrophic effects; the entire fleets of Pods may be continuously killed and recreated as they repeatedly exceed the specified ceiling. It is best to sample the applicable application’s runtime behavior under real-world conditions before deciding on which values to use.

Pod Volumes and Volume Mounts

Volumes in Kubernetes are an abstraction for making Unix-like file systems accessible from within a Pod's container. The key difference between a container's own file system and a volume is that most volume types transcend the container's life cycle. In other words, files written to a container's file system are lost whenever the container crashes, exists, or is restarted.

Similarly to the `mount` command in Unix, volumes provide an abstraction that encapsulates the actual storage mechanism and its location. As far as the container is concerned, a volume is just a local directory. However, the implementation and properties of a volume may vary drastically:

- It may be simply a *temporary file system* so that containers within a single Pod can exchange data. Such a volume type is called `emptyDir`.
- It may be *a directory within the Node's file system* such as `hostPath`—which will not be reachable if the Pod is scheduled to a different Node.
- It may be a *network storage device* such as a *Google Cloud Storage* volume (referred as `gcePersistentVolume`) or a NFS server.

Let us start with the most common and simple kind of volume: a temporary file system within a Pod which is called `emptyDir`. The `emptyDir` volume type is tied up to the Pod life cycle and can use tmpfs (a RAM-backed file system) for faster read/write speed. This is the default volume type for two or more containers in the same Pod to exchange data. Declaring a Pod volume involves two aspects:

1. Declaring and *naming* (we will use `data`) the volume at the Pod level under `spec.volumes` and specifying the volume type: `emptyDir` in our case

2. *Mounting* the relevant volume in each appropriate container under `spec.containers.volumeMounts` and specifying the path (we have chosen `/data/`) within the container that will be used to access the referred volume

We will now assemble the `spec.volumes` and `spec.containers.volumeMounts` declaration for `data` into a complete Pod manifest file, called `alpine-emptyDir.yaml`:

```
# alpine-emptyDir.yaml
apiVersion: v1
kind: Pod
metadata:
 name: alpine
spec:
  volumes:
    - name: data
      emptyDir:
  containers:
  - name: alpine
    image: alpine
    args:
    - sh
    - -c
    - |
      date >> /tmp/log.txt;
      date >> /data/log.txt;
      sleep 20;
      exit 1; # exit with error
    volumeMounts:
      - mountPath: "/data"
        name: "data"
```

The alpine-emptyDir.yaml manifest will run a shell script that logs the output of the date command to /tmp/log.txt, and, /data/log.txt. It will then wait for 20 seconds and exit with an error, which will force a container restart unless the pod.spec.restartPolicy attribute is set to Never.

The objective is to run the Pod and let it “crash” for at least two times:

```
$ kubectl apply -f alpine-emptyDir.yaml
pod/alpine created
```

```
$ kubectl get pod -w
NAME    READY STATUS           RESTARTS  AGE
alpine  1/1   Running          0          0s
alpine  0/1   Error            0         18s
alpine  1/1   Running          1         19s
alpine  0/1   Error            1         39s
alpine  0/1   CrashLoopBackOff 1         54s
alpine  1/1   Running          2         54s
...
```

At the time of the third restart, we query the contents of /tmp/log.txt, and /data/log.txt:

```
$ kubectl exec alpine -- \
    sh -c "cat /tmp/log.txt ; \
    echo "---" ; cat /data/log.txt"
Wed Sep 26 07:20:38 UTC 2018
---
Wed Sep 26 07:19:43 UTC 2018
Wed Sep 26 07:20:04 UTC 2018
Wed Sep 26 07:20:38 UTC 2018
```

As expected, /tmp/log.txt only shows one instance of the date timestamp, whereas /data/log.txt shows three even though the container has crashed three times. This is because, as mentioned before, emptyDir is tied up to the Pod life cycle. In fact, deleting and restarting the Pod will erase emptyDir so querying /data/log.txt right after starting the Pod again will show only one entry:

```
$ kubectl delete -f alpine-emptyDir.yaml
pod "alpine" deleted

$ kubectl apply -f alpine-emptyDir.yaml
pod/alpine created

$ kubectl exec alpine -- cat /data/log.txt
Wed Sep 26 11:25:09 UTC 2018
```

A *seemingly* more stable alternative to the emptyDir volume type is the hostPath one. This type of volume mounts an actual directory within the Node's file system:

```
# alpine-hostPath.yaml
...
spec:
  volumes:
    - name: data
      hostPath:
        path: /var/data
...
```

The hostPath volume type is useful for accessing Kubernetes files such as /var/log/kube-proxy.log, in a read-only basis, but it is not a good volume type to store our own files. There are two main reasons. The first is that unless we specify a Node selector (more on labels and selectors in a few sections ahead), Pods may be scheduled to run on any random Node.

This means that a Pod may initially run on Node 6m9k but then on 7tck following a deletion and recreation event:

```
$ kubectl get pods -o wide
NAME   READY STATUS  RESTARTS AGE IP         NODE
alpine 1/1   Running 8        23m 10.36.0.10 *-6m9k

$ kubectl delete pod/alpine
pod/alpine deleted
$ kubectl apply -f alpine-hostPath.yaml
pod/alpine created

$ kubectl get pods -o wide
NAME   READY STATUS  RESTARTS AGE IP         NODE
alpine 1/1   Running 1        23m 10.36.0.11 *-7tck
```

The second reason as to why storing user files using hostPath is discouraged is that Kubernetes Nodes themselves may be destroyed and recreated, for upgrades, patching, etc., without any guarantees as to the preservation of user-created data and/or the use of a stable name. Likewise, a general fault may also prevent a Node from being recovered. In this case, its system volumes are not expected to be preserved, or recycled, whenever a new Node is restored, and/or recreated again.

External Volumes and Google Cloud Storage

The emptyDir and hostPath volume types that we have seen in the last sections are only applicable within the scope of a Kubernetes cluster. The former is bound to the Pod's life cycle, whereas the latter is bound to the Node's allocation.

For serious, long-term data persistence, we often need to access enterprise-grade storage that is completely detached not only from the Pod but from the entire Kubernetes cluster itself. One such kind of storage is *Google Cloud Storage* in GCP, and the volumes we defined in it are called simply *disks*.

Let us go ahead and create a 1GB disk using the gcloud command:

```
$ gcloud compute disks create my-disk --size=1GB \
    --zone=europe-west2-a
Created
NAME     ZONE            SIZE_GB  TYPE         STATUS
my-disk  europe-west2-a  1        pd-standard  READY
```

Now, all we have to do in our Pod manifest is declare a gcePersistentDisk volume type and reference my-disk using the pdName attribute. Since disks are general-purpose block devices, we also need to specify the specific file system type using the fsType attribute:

```
# alpine-disk.yaml
...
spec:
  volumes:
    - name: data
      gcePersistentDisk:
        pdName: my-disk
        fsType: ext4
...
```

Other than changing the volume type, we will also modify the shell script so that it cats /data/log.txt rather than ending with an error:

```
# alpine-disk.yaml
...
spec:
  containers:
  - name: alpine
    image: alpine
    args:
    - sh
    - -c
    - date >> /data/log.txt; cat /data/log.txt
...
```

The Pod will run once and complete, generating a date entry that can now be checked using `kubectl logs`:

```
$ kubectl apply -f alpine-disk.yaml
pod/alpine created

# Wait until the pod's status is Running first

$ kubectl logs alpine
Fri Sep 28 15:26:08 UTC 2018
```

We will now delete *the Kubernetes cluster* itself and start a new, fresh one, to prove that storage has a decoupled life cycle:

```
$ ~/kubernetes-gcp/chp1/destroy.sh
# Wait a couple of minutes
$ ~/kubernetes-gcp/chp1/create.sh
Creating cluster...
```

If we apply the `alpine-disk.yaml` Pod manifest, we will see that the date entry from the last run is still present in addition to the one produced just now:

```
$ kubectl apply -f alpine-disk.yaml
pod/alpine created

$ kubectl logs alpine
Fri Sep 28 15:26:07 UTC 2018
Fri Sep 28 15:46:11 UTC 2018
```

Other cloud vendors have similar volume types. For example, in AWS, there is `awsElasticBlockStorage` and in Azure, `azureDisk`. The Kubernetes team keeps adding support for other storage mechanisms almost with every new release. For an up-to-date list, please check https://kubernetes.io/docs/concepts/storage/volumes/.

For more information about the settings and fields for each volume type, the `kubectl explain` command can also be used, for instance, `kubectl explain pod.spec.volumes.azureDisk`.

Pod Health and Life Cycle

Kubernetes has the capability of continuously monitoring a Pod's health status via a mechanism called *probes*. Probes may be declared under two different categories: *readiness* and *liveness*:

- **Readiness:** The container is *ready* to serve user requests so that Kubernetes can decide whether to *add* or *remove* the Pod from Service *load balancers*.
- **Liveness:** The container is *running* as intended by their designers so that Kubernetes can decide whether the container is "stuck" and must be *restarted.*

```
# Pod manifest file snippet
spec:
  containers:
     - image: ...
       readinessProbe:
         # configuration here
       livenessProbe:
         # configuration here
```

The same types of Probes are declared under `readinessProbe` and `livenessProbe` of which the *command-based*, *HTTP*, and *TCP* ones are the typical ones:

- **Command-based:** Kubernetes runs a command within the container and checks whether the result is successful (return code = 0).
- **HTTP:** Kubernetes queries a HTTP URL and checks whether the return code is greater or equal to 200 but lower than 400.
- **TCP:** Kubernetes simply checks whether it manages to open a specified TCP port.

Let us start with the command-based probe. This is the easiest to implement since it involves running an arbitrary command; as long as the exit status code is 0, the container will be deemed healthy. For example:

```
# Pod manifest file snippet
spec:
  containers:
    - image: ...
      livenessProbe:
        exec:
          command:
          - cat
          - /tmp/healthy
        initialDelaySeconds: 5
        periodSeconds: 5
```

In this snippet, `cat /tmp/healthy` will return exit code 0 only if `/tmp/healthy` is present. This approach allows adding a health probe to applications that do not expose a network interface, or that, if they do, such interface cannot be instrumented to provide health status information.

Let us now look into the HTTP Probe. The HTTP Probe, in its most basic form, simply looks into the HTTP response status of a web server to check whether it is between 200 and 399; it does not demand any specific return body. Any other code results in an error. For example, the following snippet could be thought of as of running the `curl -I http://localhost:8080/healthy` command and checking the status of the `HTTP` header:

```
# Pod manifest file snippet
spec:
  containers:
     - image: ...
       livenessProbe:
         httpGet:
           path: /healthy
           port: 8080
         initialDelaySeconds: 5
         timeoutSeconds: 1
```

Other attributes include

- `host`: The hostname that hosts the URL; it is the Pod IP's by default.
- `scheme`: HTTP (default) or HTTPS (certificate verification will be skipped).
- `httpHeaders`: Custom HTTP headers.

The last type of Probe is TCP. A TCP Probe, in its most basic configuration, simply tests whether a TCP port can be opened. In this sense, it is even more primitive than the HTTP Probe since it does not require a particular response. To implement a TCP Probe, we just need to specify the `tcpSocket` probe type and its `port` attribute:

```
# Pod manifest file snippet
spec:
  containers:
     - image: ...
       livenessProbe:
         tcpSocket:
           port: 9090
         initialDelaySeconds: 15
         periodSeconds: 20
```

Now that we have covered the three different probe types, let us look at the difference between using them under a readiness and liveness contexts and the meaning of other additional attributes, such as `initialDelaySeconds` and `periodSeconds`, which we have not yet discussed.

The difference between readiness and liveness probes is that readiness probes tell Kubernetes whether the container should be taken off a Service object, which is typically, in turn, exposed to external consumers through a load balancer (see Chapter 4), whereas liveness probes tell Kubernetes whether the container must be restarted. From a configuration perspective, the low-level checks are the same in both cases. It is just a matter of placing the specific probe command (e.g., `httpGet`) under `livenessProbe` or `readinessProbe`:

```
# Pod manifest file snippet
spec:
  containers:
     - image: ...
       # A health check failure will result in a
       # container restart
       livenessProbe:
         httpGet:
         ...
```

```
# Pod manifest file snippet
spec:
  containers:
     - image: ...
       # A health check failure will result in the
       # container being taken off the load balancer
       readinessProbe:
         httpGet:
         ...
```

A single container definition typically has both readiness and liveness probes. They are not mutually exclusive.

What is left to cover before we finish this section is how *frequently* and under which *conditions* a probe will result in declaring a Pod as unresponsive, or unable to service requests, in the case of liveness and readiness contexts, respectively. The fined-grained behavior control of probes is achieved using the `initialDelaySeconds`, `timeoutSeconds`, `periodSeconds`, and `failureThreshold` attributes. For example:

```
# Pod manifest file snippet
spec:
  containers:
     - image: ...
       livenessProbe:
         httpGet:
           path: /healthy
           port: 8080
         initialDelaySeconds: 5
         timeoutSeconds: 1
         periodSeconds: 10
         failureThreshold: 3
         successThreshold: 1
```

Let us look at each attribute, one at a time:

```
initialDelaySeconds: 5
```

Here we say that we want to wait at least five seconds before the probe starts. This is useful to account, for instance, for a web server that may take a while to start:

```
timeoutSeconds: 1
```

If our service is a bit slow to respond, we may want to give it extra time. In this case, our http server on port 8080 must respond almost right away (within one second):

```
periodSeconds: 10
```

We can't be spamming `http://localhost/healthy:8080` every nanosecond. We should run a check, be happy with the result, and come back later for another test. This attribute controls how frequently the probe is run:

```
failureThreshold: 3
```

Should we consider the probe check a fail just because of one error? Surely not. This attribute controls how many failures are required to deem the container failed to the external world:

```
successThreshold: 1
```

This is odd. What is the use of a success threshold? Well, with `failureThreshold` we control how many *sequential* failures we need to actually account for a container failure (be it in the readiness or liveness context). This works like a counter: one failure, two failures, three failures, and …. bang!. But how do we reset this counter? By counting successful probe checks. By default, it takes just one successful result to reset the counter to zero, but we may be more pessimistic and wait for two or more.

Table 2-4 summarizes the discussed attributes and shows their default and minimum values.

Table 2-4. Default and minimum values for Probe attributes

Probe Attribute Name	Default	Minimum
initialDelaySeconds	n/a	n/a
periodSeconds	10	1
timeoutSeconds	1	1
successThreshold	1*	1
failureThreshold	3	1

We can also look at how these attributes fit into the Pod life cycle in a number of key stages. This breakdown may help clarify their applicability from another angle:

1. **Container Created:** At this point, the probe isn't running yet. There is a wait state before transitioning to (2) set by the `initialDelaySeconds` attribute.

2. **Probe Started:** This is when the failure and success counters are set to zero and a transition to (3) occurs.

3. **Run Probe Check:** When a specific check is conducted (e.g., HTTP), a time-out counter starts which is set by the `timeoutSeconds` attribute. If a failure or a time-out is detected, there is a transition to (4). If there is no time-out and a success state is detected, there is a transition to (5).

4. **Failure:** In this case, the failure counter is incremented, and the success counter is set to zero. Then, there is a transition to (6).

5. **Success:** In this case, the success counter is incremented, and there is a transition to (6). If the success counter is greater or equal than the `successThreshold` attribute, the failures counter is set to zero.

6. **Determine Failure:** If the failures counter is greater or equal than the value specified by the `failureThreshold` attribute, a probe reports a failure—the action will depend on whether it is a readiness or a liveness probe. Otherwise, there will be a wait state determined by the `periodSeconds` attribute, and then a transition to (3) will occur.

This view on the Pod life cycle is centered around the behavior of probes. A more comprehensive account on the Pod life cycle that is useful to understand its implication in terms of the service controller and stateful services is included in Chapter 9.

Namespaces

Namespaces are a universal concept in Kubernetes and not the exclusivity of Pods, but it is convenient to learn about them in the context of Pods because, ultimately, all workloads in Kubernetes are housed in Pods and all Pods live in a namespace.

Namespaces is the mechanism that Kubernetes uses to segregate resources by a user-defined criteria. For example, namespaces can isolate development life cycle environments such as development, testing, staging, and production. They can also help group related resources, without necessarily intending to establish a Chinese wall; for example, a namespace may be used to group together “product catalogue” components, whereas other for “order fulfilment” ones.

Let us look at namespaces in a rather empirical manner. When running `kubectl get pod`, there appears to be nothing unless we launch a Pod of our own such as `alpine`:

```
$ kubectl get pod
NAME      READY     STATUS    RESTARTS   AGE
alpine    1/1       Running   0          31m
```

Well, this is an illusion created by the fact that most Kubernetes commands target the `default` namespace which is where the user's first objects run. In fact, most Kubernetes commands can be thought as of implicitly carrying the flag `-n default` which is a shortcut for `--namespace=default`:

```
$ kubectl get pod -n default
NAME      READY     STATUS    RESTARTS   AGE
alpine    1/1       Running   0          33m
```

As we have mentioned before, almost all workloads in Kubernetes are housed in Pods. This does not only apply to user-created artifacts but to Kubernetes infrastructure components as well. Most of Kubernetes' own utilities and processes are implemented as regular Pods, but these appear to be invisible by default because they happen to live in a separate namespace called `kube-system`:

```
$ kubectl get pod -n kube-system
NAME                       READY STATUS    RESTARTS   AGE
event-exporter-*-vsmlb     2/2   Running   0          2d
fluentd-gcp-*-gz4nc        2/2   Running   0          2d
fluentd-gcp-*-lq2lx        2/2   Running   0          2d
fluentd-gcp-*-srg92        2/2   Running   0          2d
heapster-*-xwmvv           3/3   Running   0          2d
```

```
kube-dns-*-p95tp        4/4   Running  0          2d
kube-dns-*-wjzqz        4/4   Running  0          2d
...
```

These are all regular Pods. We can run all of the commands we have learned so far against them. We just need to remember to add the `-n kube-system` flag to every command we use. Otherwise, Kubernetes will assume `-n default`. For example, let us look into what processes are running inside the first container found in the Pod named `kube-dns-*-p95tp`:

```
$ kubectl exec -n kube-system kube-dns-*-p95tp ps
Defaulting container name to kubedns.
PID USER TIME COMMAND
  1 root 4:35 /kube-dns --domain=cluster.local. ...
 12 root 0:00 ps
```

If we would like to identify the namespace to which a given Pod is assigned, we can use the flag `--all-namespaces` together with the `kubectl get` command. For example:

```
$ kubectl get pod --all-namespaces
NAMESPACE    NAME                   READY STATUS
default      alpine                 1/1   Running
kube-system  event-exporter-*-vsmlb 2/2   Running
kube-system  fluentd-gcp-*-gz4nc    2/2   Running
...
```

Note in this output how the first column identifies the namespace in which every Pod is defined. We can also list the existing namespaces themselves using `kubectl get namespaces`:

```
$ kubectl get namespace
NAME         STATUS    AGE
default      Active    2d
kube-public  Active    2d
kube-system  Active    2d
```

Namespaces are the hardest form of logical separation between Kubernetes objects. Let us suppose that we define three distinct namespaces called ns1, ns2, and ns3:

```
$ kubectl create namespace ns1
namespace/ns1 created
$ kubectl create namespace ns2
namespace/ns2 created
$ kubectl create namespace ns3
namespace/ns3 created
```

We can now run a Pod called nginx in each namespace without any Pod name collisions:

```
$ kubectl run nginx --image=nginx --restart=Never \
    --namespace=ns1
pod/nginx created
$ kubectl run nginx --image=nginx --restart=Never \
    --namespace=ns2
pod/nginx created
$ kubectl run nginx --image=nginx --restart=Never \
    --namespace=ns3
pod/nginx created
```

```
$ kubectl get pod --all-namespaces | grep nginx
ns1       nginx   1/1   Running     0    1m
ns2       nginx   1/1   Running     0    1m
ns3       nginx   1/1   Running     0    1m
```

Labels

Labels are simply user-defined (or Kubernetes-generated) key/value pairs that are associated with a Pod (as well as any other Kubernetes object). They are useful for describing small elements (both keys and values are restricted to 63 characters) of meta-information such as

- A family of related objects (e.g., replicas of the same Pod)
- Version numbers
- Environments (e.g., dev, staging, production)
- Deployment type (e.g., canary release or A/B testing)

Labels are a fundamental concept in Kubernetes because it is the mechanism that facilitates orchestration. When multiple Pods (or other object types) are "orchestrated," the way in which a controller object (such as Deployment) manages a swarm of Pods is by selecting their label as we will see in Chapter 3.

The innocent `kubectl run` command, for example, adds a label called "run" to every Pod whose value is the Pod's *given* name:

```
$ kubectl run nginx --image=nginx --restart=Never
pod/nginx created
```

```
$ kubectl get pods --show-labels
NAME   READY  STATUS    RESTARTS   AGE    LABELS
nginx  1/1    Running   0          44s    run=nginx
```

As seen in this output, the `--show-labels` flag displays the labels that have been declared for the listed objects. Labels can be set both imperatively and declaratively. For instance, the following Pod manifest sets the labels `env` and `author` to `prod` and `Ernie`, respectively:

```
# nginx-labels.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: prod
    author: Ernie
spec:
  containers:
  - image: nginx
    name: nginx
  restartPolicy: Never
```

This is equivalent to the following imperative syntax:

```
$ kubectl run nginx --image=nginx --restart=Never \
    -l "env=prod,author=Ernie"
```

The -l flag is a shortcut for --labels="<LABELS>" where <LABELS> is a comma-separated list of key/value pairs. Both by running kubectl apply -f nginx-labels.yaml and using the imperative command seen before, we can observe that two user-defined labels, author and env, have been set instead of the default run=nginx label:

```
$ kubectl get pods --show-labels
NAME  READY STATUS  AGE LABELS
nginx 1/1   Running 3m  author=Ernie,env=prod
```

Before we continue, let us add a bit of complexity by creating two more nginx Pods with slightly different labels:

```
$ kubectl run nginx1 --image=nginx --restart=Never \
    -l "env=dev,author=Ernie"
pod/nginx1 created
```

```
$ kubectl run nginx2 --image=nginx --restart=Never \
    -l "env=dev,author=Mohit"
pod/nginx2 created
```

The usefulness of labels is not in just adding arbitrary metadata to discrete objects (just Pods for now since we haven't covered other resource types yet) but in dealing with collections of them. Even though labels are schema free, it helps to think that we are defining column types in a database. For example, now that we know that Pods `nginx`, `nginx1`, and `nginx2` have in common that they declare their environment and author, via the `env` and `author` labels, respectively, we can specify that we want these values to be listed as specific columns using the `-L <LABEL1,LABEL2,...>` flag:

```
$ kubectl get pods -L env,author
NAME    READY   STATUS  RESTARTS  AGE   ENV   AUTHOR
nginx   1/1     Running 0         11m   prod  Ernie
nginx1  1/1     Running 0         6m    dev   Ernie
nginx2  1/1     Running 0         5m    dev   Mohit
```

Labels would not be useful if we could not formulate queries such as "give me the objects whose author is Ernie" or "those that have a `caution` label." This sort of expressions are called *selector expressions*. The first question would be an *equality-based* expression, whereas the second is a *set-based* one. Selector expressions or *selectors*, for short, are declared in the manifest of many controller objects to connect them with their dependencies, but they can also be expressed imperatively via the `-l <SELECTOR-EXPRESSION>` flag which is a shortcut for `--selector=<SELECTOR-EXPRESSION>`.

The first question, for instance, is expressed as follows:

```
$ kubectl get pods -l author=Ernie
NAME      READY     STATUS    RESTARTS   AGE
nginx     1/1       Running   0          30m
nginx1    1/1       Running   0          24m
```

We can also negate the equality expression and ask for those Pods whose author *is not* Ernie:

```
$ kubectl get pods -l author!=Ernie
NAME      READY     STATUS    RESTARTS   AGE
nginx2    1/1       Running   0          24m
```

The set-based questions are about membership. We have asked a few moments ago about Pods that have a label called `caution`. In this case we simply specify the label name:

```
$ kubectl get pods -l caution
No resources found.
```

Indeed, we haven't defined a `caution` label yet. To ask for those objects without a label, we just prefix the label with an exclamation mark as follows:

```
$ kubectl get pods -l \!caution
NAME      READY     STATUS    RESTARTS   AGE
nginx     1/1       Running   0          37m
nginx1    1/1       Running   0          32m
nginx2    1/1       Running   0          31m
```

A more advanced type of set-based selector is one in which we test for multiple values using the `<LABEL> in (<VALUE1>,<VALUE2>,...)` syntax (`notin` is used for negation). For example, let us list those Pods whose authors are Ernie or Mohit:

```
$ kubectl get pods -l "author in (Ernie,Mohit)"
NAME      READY     STATUS    RESTARTS   AGE
nginx     1/1       Running   0          50m
nginx1    1/1       Running   0          44m
nginx2    1/1       Running   0          43m
```

A label can be changed also at runtime using the `kubectl label <RESOURCE-TYPE>/<OBJECT-IDENTIFIER> <KEY>=<VALUE>` command. The `--overwrite` flag must also be added if we are altering an existing label. For example:

```
$ kubectl label pod/nginx author=Bert --overwrite
pod/nginx labeled
```

```
$ kubectl get pods -L author
NAME   READY  STATUS    RESTARTS   AGE    AUTHOR
nginx  1/1    Running   0          51m    Bert
nginx1 1/1    Running   0          46m    Ernie
nginx2 1/1    Running   0          45m    Mohit
```

Now we can run the set-based query again by asking for those Pods whose author is neither Ernie nor Mohit:

```
$ kubectl get pods -l "author notin (Ernie,Mohit)"
NAME      READY     STATUS    RESTARTS   AGE
nginx     1/1       Running   0          54m
```

Finally, labels can be removed using `kubectl label <RESOURCE-TYPE>/<OBJECT-IDENTIFIER> <KEY>-` (note the minus sign at the end). The following two statements add and remove the `caution` label to `nginx`:

```
$ kubectl label pod/nginx caution=true
pod/nginx labeled
```

```
$ kubectl label pod/nginx caution-
pod/nginx labeled
```

Annotations

Annotations are similar to labels in the sense that they are a kind of key/value-based metadata. However, they are intended for the purpose of storing non-identifying, non-selectable data—selection expressions don't work against annotations.

In most cases, annotations are static rather than volatile metadata. They are declared within `pod.metadata.annotations`:

```
# Pod manifest file snippet
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  annotations:
    author: Michael Faraday
    e-mail: michael@faraday.com
```

Furthermore, annotations do not have a 63 character limit as labels' values do. They may contain unstructured, long strings.

The retrieval of annotations, since is not meant for selector expressions, is typically accomplished via JSONPath. For example, if author was defined as an annotation field on nginx, we could use the following command:

```
$ kubectl get pod/nginx -o jsonpath \
    --template="{.metadata.annotations.author}"
Michael Faraday
```

Summary

In this chapter, we learned how to launch, interact, and manage containerized applications by using Kubernetes Pods. This includes the passing of arguments, piping data in and out, as well as the exposure of network ports. We then explored more advanced features such as the setting of CPU and RAM constraints, the mounting of external storage, and the instrumentation of health checks using Probes. Finally, we showed how Labels and Annotations help tag, organize, and select Pods—and almost any other Kubernetes object type as well.

The understanding gained in this chapter is enough to treat a Kubernetes cluster like a giant computer with lots of CPU and RAM in which monolithic workloads can be deployed. In this sense, this chapter is self-contained. For example, installing a traditional three-tier application such as WordPress, in a monolithic fashion—barring exposure on the public Internet, covered in Chapter 4—would not require features behind what we had covered here.

The next chapters are all about going *beyond* the features of traditional monoliths, by employing Kubernetes controllers that give us advanced capabilities such as high availability, fault tolerance, service discovery, job scheduling, and the instrumentation of distributed data stores.

CHAPTER 3

Deployments and Scaling

A *Deployment* is a uniformly managed set of Pod instances, all based on the same Docker image. A Pod instance is called a *Replica*. The Deployment controller uses multiple Replicas to achieve *high scalability*, by providing more compute capacity than is otherwise possible with a single monolithic Pod, and in-cluster *high availability*, by diverting traffic away from unhealthy Pods (with the aid of the Service controller, as we will see in Chapter 4) and restarting—or recreating—them when they fail or get stuck.

As per the definition given here, a Deployment may appear simply as a fancy name for "Pod cluster," but "Deployment" is not actually a misnomer; the Deployment controller's true power lies in its actual *release capabilities*—the deployment of new Pod versions with near-zero downtime to its consumers (e.g., using blue/green or rolling updates) as well as the seamless transition between different scaling configurations, while preserving compute resources at the same time.

We will start this chapter with an overview on the relationship between the Deployment controller, the ReplicaSet controller, and Pods. We will then learn how to launch, monitor, and control Deployments. We will also see the various ways in which we can locate and reference the objects that Kubernetes creates as a result of instructing a Deployment.

E. Garbarino, *Beginning Kubernetes on the Google Cloud Platform*,
https://doi.org/10.1007/978-1-4842-5491-2_3

Once the essentials about Deployments are covered, we will focus on the available deployment strategies including rolling and blue/green deployments and the parameters that allow achieving the best compromise between resource utilization and service consumer impact.

Finally, we will cover the subject of autoscaling both at the Pod level, using Kubernetes' out-of-the-box Horizontal Pod Autoscaler (HPA), and at the Node level, using the GKE's autoscaling flag at the time of cluster creation.

ReplicaSets

For historical reasons, the process of running replicas is handled through a separate component called *ReplicaSet*. This component, in turn, replaces an even older component called *ReplicationController*.

To avoid any confusion, let us spend some time understanding the relationship between the Deployment controller and the ReplicaSet controller. A Deployment is a higher-level controller which manages both deployment transitions (e.g., rolling updates) as well as replication conditions via ReplicaSets. It is not that Deployments *replace* or *embed* the ReplicaSet object (the camel case spelling is used to refer the object name), they simply control it; ReplicaSets are still visible as discrete Kubernetes objects even though the user is encouraged to interact with them through Deployments.

In conclusion, the ReplicaSet is a discrete, fully qualified object in Kubernetes, but running a ReplicaSet outside of Deployments is discouraged, and, whenever a ReplicaSet is under a Deployment's control, all interaction should be mediated by the Deployment object.

Our First Deployment

Deployments are such a fundamental feature in Kubernetes that it is easier to create a Deployment than it is to create a monolithic, singleton Pod. If we look at the examples in the last chapter, we will notice that every instance of `kubectl run` had to be accompanied by a `--restart=Never` flag. Well, one "cheap" way of creating a Deployment is by simply *dropping* this flag:

```
$ kubectl run nginx --image=nginx
deployment.apps/nginx created
```

Omitting the `--restart=Never` flag has a dramatic effect. Rather than creating a single Pod, we have now created two additional objects: a Deployment controller which controls a ReplicaSet, which then, in turn, controls a Pod!

```
$ kubectl get deployment,replicaset,pod
NAME               DESIRED CURRENT UP-TO-DATE AVAIL.
deployment.*/nginx 1       1       1          1

NAME                         DESIRED   CURRENT   READY
replicaset.*/nginx-8586cf59 1         1         1

NAME                        READY   STATUS    RESTARTS
pod/nginx-8586cf59-b72sn   1/1     Running   0
```

Although creating a Deployment in this way is convenient, Kubernetes will deprecate their creation using the `kubectl run` command in the future. The new *preferred method* is via the `kubectl create deployment <NAME>` command as follows:

```
$ kubectl create deployment nginx --image=nginx
deployment.apps/nginx created
```

The two versions are equivalent at the moment, but the "old method," via `kubectl run`, is still the one used in most textbooks and examples on the official `http://kubernetes.io` web site itself; therefore, the reader is advised to remember the two presented approaches for the time being.

Note As of Kubernetes v1.15, the `kubectl create <RESOURCE-TYPE>` command is still not a sound replacement for the traditional `kubectl run` approach. For example, when the Kubernetes team deprecated the creation of CronJobs (covered in Chapter 7) commands via the traditional run-based form, they did not include the `--schedule` flag. This issue was then fixed in a subsequent release following a feature request on GitHub.

In the case of `kubectl create deployment`, the `--replicas` flag is missing. This does not mean that the command is "broken," but it forces the user to take more steps to achieve a goal that used to take one single command. A deployment's number of replicas may still be set imperatively via the `kubectl scale` command (to be covered in the next sections) or by declaring a JSON fragment.

Turning our attention back to the objects that are created as a result of creating a Deployment, the given name `nginx` now applies to the Deployment controller instance rather than the Pod. The Pod has a random name: `nginx-8586cf59-b72sn`. The reason as to why Pods have now random names is that they are ephemeral. Their number may vary; some may be killed, some new ones may be created, and so on. In fact, a Deployment that controls a single Pod is not very useful. Let us specify a number of replicas other than 1 (the default) by using the `--replicas=<N>` flag:

```
# Kill the running Deployment first
$ kubectl delete deployment/nginx

# Specify three replicas
$ kubectl run nginx --image=nginx --replicas=3
deployment.apps/nginx created
```

We will now see that three Pods running rather than one:

```
$ kubectl get pods
NAME                    READY STATUS   RESTARTS  AGE
nginx-64f497f8fd-8grlr  1/1   Running  0         39s
nginx-64f497f8fd-8svqz  1/1   Running  0         39s
nginx-64f497f8fd-b5hxn  1/1   Running  0         39s
```

The number of replicas is dynamic and can be specified at runtime using the kubectl scale deploy/<NAME> --replicas=<NUMBER> command. For example:

```
$ kubectl scale deploy/nginx --replicas=5
deployment.extensions/nginx scaled
```

```
$ kubectl get pods
NAME                    READY STATUS   RESTARTS  AGE
nginx-64f497f8fd-8grlr  1/1   Running  0         5m
nginx-64f497f8fd-8svqz  1/1   Running  0         5m
nginx-64f497f8fd-b5hx   1/1   Running  0         5m
nginx-64f497f8fd-w8p6k  1/1   Running  0         1m
nginx-64f497f8fd-x7vdv  1/1   Running  0         1m7
```

Likewise, the specified Pod image is also dynamic and may be altered using the kubectl set image deploy/<NAME> <CONTAINER-NAME>=<URI> command. For example:

```
$ kubectl set image deploy/nginx nginx=nginx:1.9.1
deployment.extensions/nginx image updated
```

More on Listing Deployments

The kubectl get deployments command shows a number of columns:

```
$ kubectl get deployments
NAME    DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE
nginx   5        5        5           5          10m
```

The displayed columns refer to the number of *Pod replicas* within the Deployment:

- DESIRED: The *target state*: specified in deployment.spec.replicas
- CURRENT: The number of replicas running but not necessarily available: specified in deployment.status.replicas
- UP-TO-DATE: The number of Pod replicas that have been updated to achieve the current state: specified in deployment.status.updatedReplicas
- AVAILABLE: The number of replicas actually available to users: specified in deployment.status.availableReplicas
- AGE: The amount of time that the deployment controller has been running since first created

Deployments Manifests

An example of a minimal, but complete, Deployment manifest may be intimidating. It is, therefore, easier to think of a Deployment manifest as a two-step process.

The first step consists in defining a Pod template. A Pod template is almost the same as the definition of a stand-alone Pod, except that we only populate the metadata and spec sections:

```
# Pod Template
...
spec:
  template:
    metadata:
      labels:
        app: nginx-app # Pod label
    spec:
      containers:
      - name: nginx-container
        image: nginx:1.7.1
```

It is also necessary to declare an explicit Pod label key/pair since we need to reference the Pod. Earlier we have used app: nginx-app, which is then used in the Deployment's spec to bind the controller object to the Pod template:

```
# Deployment Spec
...
spec:
  replicas: 3        # Specify number of replicas
  selector:
    matchLabels:     # Select Pod using label
      app: nginx-app
```

Under spec:, we also specify the number of replicas which is equivalent to the --replicas=<N> flag used in the imperative form.

Finally, we assemble these two definitions into a complete Deployment manifest:

```
# simpleDeployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-declarative
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx-app
  template:
    metadata:
      labels:
        app: nginx-app
    spec:
      containers:
      - name: nginx-container
        image: nginx:1.7.1
```

This Deployment is first created using the `kubectl apply -f <FILE>` command:

```
$ kubectl apply -f simpleDeployment.yaml
deployment.apps/nginx-declarative created
```

The effect will be similar to that of its imperative counterpart; three Pods will be created:

```
$ kubectl get pods
NAME                       READY STATUS  RESTARTS AGE
nginx-declarative-*-bj4wn 1/1   Running 0        3m
nginx-declarative-*-brhvw 1/1   Running 0        3m
nginx-declarative-*-tc6hv 1/1   Running 0        3m
```

Monitoring and Controlling a Deployment

The `kubectl rollout status deployment/<NAME>` command is used to monitor a Deployment that is in progress. For example, let us suppose that we create a new imperative Deployment for Nginx and we want to keep track of its progress:

```
$ kubectl run nginx --image=nginx --replicas=3 \
    ; kubectl rollout status deployment/nginx
deployment.apps/nginx created
Waiting for deployment "nginx" rollout to finish:
0 of 3 updated replicas are available...
Waiting for deployment "nginx" rollout to finish:
1 of 3 updated replicas are available...
Waiting for deployment "nginx" rollout to finish:
2 of 3 updated replicas are available...
deployment "nginx" successfully rolled out
```

Simple deployments—especially those that do not involve an update over an existing one—typically execute in a handful of seconds; more complex deployments, however, may take several minutes. In this case, we may want to pause a Deployment that is in progress to clean up resources or perform additional monitoring.

A Deployment is paused and resumed using the `kubectl rollout pause deploy/<NAME>` and `kubectl rollout resume deploy/<NAME>` commands, respectively. For example:

```
$ kubectl rollout pause deploy/nginx
deployment "nginx" paused
```

```
$ kubectl rollout resume deploy/nginx
deployment "nginx" resumed
```

Finding Out a Deployment's ReplicaSets

Replicas (Pod instances) are not controlled directly by the Deployment controller but, by an intermediary, the ReplicaSet controller. As such, it is often useful to work out which are the ReplicaSet controller(s) that are subordinate to a given Deployment controller.

This can be achieved via the *label selector*, using the `kubectl describe` command or simply by relying on *visual matching*.

Let us start with the label selector approach. In this case, we simply list the ReplicaSets using the `kubectl get rs` command but add the `--selector=<SELECTOR-EXPRESSION>` flag to match the Pod's label and selector expression in the Deployment manifest. For example:

```
$ kubectl get rs --selector="run=nginx"
NAME                DESIRED   CURRENT   READY     AGE
nginx-64f497f8fd    3         3         3         2m
```

Please note that the label `run=nginx` is added automatically by the `kubectl run` command; in `simpleDeployement.yaml`, instead, we have used a custom label: `app=nginx-app`.

Now, let us consider the kubectl describe approach. Here we simply type kubectl describe deploy/<NAME> command and locate the OldReplicaSets and NewReplicaSet fields. For example:

```
$ kubectl describe deploy/nginx
...
OldReplicaSets: <none>
NewReplicaSet:  nginx-declarative-381369836
                (3/3 replicas created)
...
```

The last and more straightforward approach consists in simply typing kubectl get rs and identifying the ReplicaSets whose prefix is the Deployment's name.

Sometimes, we may want to find out the parent Deployment controller given a ReplicaSet. To do this, we can use the kubectl describe rs/<NAME> command and search for the value of the Controlled By field. For example:

```
$ kubectl describe rs/nginx-381369836
...
Controlled By:  Deployment/nginx
...
```

Alternatively, we may use JSONPath if a more programmatic approach is required:

```
$ kubectl get pod/nginx-381369836-g4z5r \
    -o jsonpath \
    --template="{.metadata.ownerReferences[*].name}"
nginx-381369836
```

Finding Out a ReplicaSet’s Pods

In the last section, we have seen how to identify a Deployment’s ReplicaSet. ReplicaSets, in turn, control Pods; thus, the next natural question is how to find out which are the Pod’s under a given ReplicaSet’s control. Fortunately, we use the same three techniques seen before: label selectors, the `kubectl describe` command, and visual matching.

Let us start with the label selector. This is exactly the same as before, the `--selector=<SELECTOR-EXPRESSION>` flag is used except that we ask `kubectl get` for objects of type `pod` (Pod) rather than `rs` (ReplicaSet). For example:

```
$ kubectl get pod --selector="run=nginx"
NAME                    READY  STATUS   RESTARTS  AGE
nginx-64f497f8fd-72vfm 1/1    Running  0         18m
nginx-64f497f8fd-8zdhf 1/1    Running  0         18m
nginx-64f497f8fd-skrdw 1/1    Running  0         18m
```

Similarly, the use of the `kubectl describe` command involves simply specifying the ReplicaSet’s object type (`rs`) and name:

```
$ kubectl describe rs/nginx-381369836
...
Events:
  FirstSeen LastSeen Count Message
  --------- -------- ----- -------
  55m       55m      1     Created pod: nginx-*-cv2xj
  55m       55m      1     Created pod: nginx-*-8b5z9
  55m       55m      1     Created pod: nginx-*-npkn8
...
```

The visual matching technique is the easiest. We can work out the ReplicaSet's name by considering the two prefix strings of a Pod. For example, for Pod `nginx-64f497f8fd-72vfm`, its controlling ReplicaSet would be `nginx-64f497f8fd`:

```
nginx-64f497f8fd-8zdhf  1/1  Running   0     18m
```

Last but not least, if we are starting with a Pod's object and want to find out which are its controlling objects, we can use the `kubectl describe pod/<NAME>` command and locate the controller object followed by the `Controlled By:` attribute:

```
$ kubectl describe pod/nginx-381369836-g4z5r
...
Controlled By:  ReplicaSet/nginx-381369836
...
```

If a more programmatic approach is desired, the same can be obtained using JSONPath as follows:

```
$ kubectl get pod/nginx-381369836-g4z5r \
    -o jsonpath \
    --template="{.metadata.ownerReferences[*].name}"
nginx-declarative-381369836
```

Deleting Deployments

A Deployment is deleted using the `kubectl delete deploy/<NAME>` command which triggers a *cascade* deletion; all descendant ReplicaSet and associated Pod objects will be deleted as a result:

```
$ kubectl delete deploy/nginx
deployment.extensions "nginx" deleted
```

The `kubectl delete` command can also pick up the Deployment's name directly from a manifest file as follows:

```
$ kubectl delete -f simpleDeployment.yaml
deployment.apps "nginx-declarative" deleted
```

The cascade default deletion behavior (which results in all of the ReplicaSet and Pods being deleted) may be prevented by adding the `--cascade=false` flag. For example:

```
$ kubectl delete -f simpleDeployment.yaml \
    --cascade=false
deployment.apps "nginx-declarative" deleted
```

Revision-Tracking vs. Scaling-Only Deployments

Deployments may be categorized in two types: *revision-tracking* and *scaling-only*.

A revision-tracking Deployment is one that changes some aspect of the Pod's specification, most likely the number and/or version of container images declared within. A Deployment that only alters the number of replicas—both imperatively and declaratively—does not trigger a revision.

For example, issuing the `kubectl scale` command will not create a revision point that can be used to undo the scaling change. Returning to the precedent number of replicas involves setting the previous number again.

A scaling Deployment is achieved imperatively by using the `kubectl scale` command or by setting the `deployment.spec.replicas` attribute and applying the corresponding file using the `kubectl apply -f <DEPLOYMENT-MANIFEST>` command.

Since scaling-only Deployments are managed entirely by the ReplicaSet controller, there is no revision tracking (e.g., rollback capabilities) provided by the master Deployment controller. Objectively, although this behavior is rather inconsistent, one can argue that altering the number of replicas is not as consequential as changing an image.

Deployment Strategy

Until now, we have looked at Deployments just as a mechanism to deploy multiple Pod replicas, but we have not described how to control this process: should Kubernetes delete all existing Pods (causing downtime) or should it proceed in a more graceful manner? This is what a Deployment strategy is all about. Kubernetes offers two broad types of Deployment strategies:

- **Recreate:** Oppenheimer's approach to upgrading a Deployment: destroy everything first and only then create the replicas declared by the new Deployment's manifest.
- **RollingUpdate:** Fine-tune the upgrade process to achieve from something as "careful" as updating one Pod at a time, all the way up to a fully-fledged blue green deployment in which the entire new set of Pod replicas are stood up before disposing of the old ones.

The Strategy is configured using the `deployment.spec.strategy.type` attribute which may be set either to `Recreate` or `RollingUpdate`. The latter is the default. We will look at each option in detail in the next two sections.

Recreate Deployments

A Recreate Deployment is one that literally terminates all existing Pods and creates the new set as per the specified target state. In this sense, a Recreate Deployment is downtime-causing since once the number of Pods reaches zero, there will be some delay before the new Pods are created.

Recreate Deployments are useful for non-production scenarios in which we want to see the expected changes as soon as possible without the waiting for the rolling update ritual to complete.

Typically, a Deployment type would be specified declaratively, in the Deployment's manifest:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-declarative
spec:
  replicas: 5
  strategy:
    type: Recreate # Recreate setting
...
```

The actual Recreate Deployment will take place when applied using the `kubectl apply -f <MANIFEST>` command.

Rolling Update Deployments

A rolling update is *typically* one in which one Pod is updated at a time (and the load balancer is managed accordingly behind the scenes) so that the consumer does not experience downtime and resources (e.g., number of Nodes) are rationalized.

In practice, both *conventional "one at a time" rolling update deployments* and *blue/green* deployments (more on blue/green deployments in a few sections ahead) can be achieved using the same Rolling Update mechanism by setting the `deployment.spec.rolling.Update.maxSurge` and `deployment.spec.rolling.Update.maxUnavailable` variables. We will see how to achieve this further on.

For now, let us contemplate the following specification:

```
...
spec:
  replicas: 5
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
...
```

The `maxSurge` and `maxUnavailable` values shown here allow us to tune the nature of the Rolling Update:

- `maxSurge`: This property specifies the number of Pods for the target (New ReplicaSet) that must be created before the termination process on the baseline (Old ReplicaSet) starts. If this value is 1, this means that the number of replicas will remain constant during the deployment at the cost of Kubernetes allocating resources for one extra Pod at any given time.

- `maxUnavailable`: This property specifies the maximum number of Nodes that may be unavailable at any given time. If this number is zero, as in the preceding example, a `maxSurge` value of 1 is at least required since spare resources are required to keep the number of desired replicas constant.

Please note that the maxUnavailable and maxSurge variables accept percent values. For example, in this case, 25% of Kubernetes' extra resources will be used to guarantee that that there is no decrease in the number of running replicas:

```
...
spec:
  replicas: 4
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 0%
...
```

This example is equivalent to the one initially discussed since 25% of four replicas is exactly one replica.

The Pros and Cons of a Higher MaxSurge Value

A higher maxSurge setting creates more Pod instances of the target, *to be* (new) ReplicaSet, before the transition from the *baseline* (old) takes place. Therefore, the higher the maxSurge number, the smaller the transition period during which users may hit both the *baseline* and *to be* versions—whenever Deployments are used in combination with the Service controlled to be covered in Chapter 4.

The disadvantage is that the cluster needs the extra resources to be able to create the new Pods while keeping the old ones running. For example, a maxSurge setting of 50% will demand exactly 150% compute resources at the peak of the migration, assuming that the baseline ReplicaSet compute utilization is 100%.

The Pros and Cons of a High MaxUnavailable Value

In an ideal world, the `maxUnavailable` property should simply be set to 0 to ensure that the number of replicas remains constant. However, this not always *possible*, or *required*.

It may not be possible if compute resources are scarce in the Kubernetes cluster and thus `maxSurge` must be 0. In this case, the transition from the *baseline* to the *to be* state will involve taking one or more Pods down.

For example, the following settings ensure that no extra resources are allocated but that, at least, 75% of the Nodes are available at any given time during the deployment:

```
...
maxSurge: 0%
maxUnavailable: 25%
...
```

The disadvantage of a higher `maxUnavailable` value is that it involves reducing the cluster's capacity—that consisting of the replicas associated with the Deployments, not the Kubernetes cluster as a whole. However, this is not necessarily a *bad thing*. The deployment may be performed at a time during which demand is low and reducing the number of replicas may not have visible effects. Furthermore, even though the higher the value, the less number of replicas are available, the entire process is faster since multiple Pods may be updated at the same time.

Blue/Green Deployments

A blue/green deployment is one in which we deploy the entire new *to be* Pod cluster in advance, and when ready, we switch the traffic away from the *baseline* Pod cluster in one go. The entire process is orchestrated transparently with the help of the Service controller (see Chapter 4).

In Kubernetes, this is not a brand new strategy type; we still use the `RollingUpdate` Deployment type, but we set the `maxSurge` property to 100% and the `maxUnavailability` property to 0 as follows:

```
...
maxSurge: 100%
maxUnavailable: 0%
...
```

Provided that the Deployment constitutes a revision change—the underlying image type or image's version is changed—Kubernetes will execute a blue/green deployment in three broad steps:

1. Create new Pods for the *to be* New ReplicaSet until the `maxSurge` limit is reached; 100% in this case.
2. Redirect traffic to the New ReplicaSet—this requires assistance of the Service controller which we cover in Chapter 4.
3. Terminate Nodes in the Old ReplicaSet.

Summary of MaxSurge and MaxUnavailability Settings

As we have seen in the previous sections, most deployment strategies consist of setting the `maxSurge` and `maxUnavailability` properties to different values. Table 3-1 provides a summary of the most typical use cases along with the trade-off involved and sample values.

Table 3-1. *Appropriate values for different Deployment strategies*

Scenario	Trade-Off	`maxSurge`	`maxUnavailbility`
Destroy and Deploy*	Capacity and Avail.	0	100%
One-at-a-time Rolling Update	Resources	1	0
One-at-a-time Rolling Update	Capacity	0	1
Faster Rolling Update	Resources	25%	0
Faster Rolling Update	Capacity	0	25%
Blue/Green	Resources	100%	0

**Same as a Recreate Deployment*

Controlled Deployments

A controlled deployment is one that allows the operator (or an equivalent automated system) to react to a potential failed deployment. A Deployment may fail in a number of ways, but there are two scenarios in which Kubernetes offers the operator more control.

The first scenario is the most general one: Kubernetes is unable to update the requested number of replicas because there are insufficient cluster resources, or because the Pods themselves fail to start—for example, when an inexistent container image has been specified. It is unlikely that new compute resources or container images will pop up all of a sudden and correct the situation. What we want, instead, is that Kubernetes considers the Deployment failed after a set time rather than keeping the Deployment under progress forever. This is achieved by setting the `deployment.spec.progressDeadlineSeconds` attribute to an appropriate value.

For instance, let us consider the example of specifying a significant number of replicas (30) assuming a tiny, three Node Kubernetes cluster with insufficient compute resources:

```
# nginxDeployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  progressDeadlineSeconds: 60
  replicas: 30 # excessive number for a tiny cluster
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.1
        ports:
        - containerPort: 80
```

The progressDeadlineSeconds property will force Kubernetes to fail the Deployment after 60 seconds if all the replicas are not available by that time:

```
$ kubectl apply -f nginxDeployment.yaml ; \
    kubectl rollout status deploy/nginx
deployment.apps/nginx created
Waiting for deployment "nginx" rollout to finish:
0 of 30 updated replicas are available...
Waiting for deployment "nginx" rollout to finish:
7 of 30 updated replicas are available...
Waiting for deployment "nginx" rollout to finish:
11 of 30 updated replicas are available...
error: deployment "nginx" exceeded its
progress deadline
```

The second scenario is that in which the Deployment itself is successful but the containers fail after a while due to some internal problem such as a memory leak, a null pointer exception in the bootstrap code, and so on. In a controlled Deployment, we may want to deploy a single replica, wait a few seconds to be sure that it is stable, and only then proceed with the next replica. Kubernetes' default behavior is to update all replicas in parallel which may not be what we want.

The `deployment.spec.minReadySeconds` property allows us to define how long Kubernetes must wait *after* a Pod becomes ready before processing the next replica. The value for this property is a trade-off between the amount of time that it takes for most obvious problems to emerge and how long the deployment will take.

In combination with the `progressDeadlineSeconds` property, the `minReadySeconds` property helps prevent a catastrophic Deployment, especially when it is updating an existing healthy one.

For example, let us suppose that we have a healthy Deployment called myapp (declared in a manifest called myApp1.yaml) that consists of three replicas:

```
# myApp1.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
  labels:
    app: myapp
spec:
  progressDeadlineSeconds: 60
  minReadySeconds: 10
  replicas: 3
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
      - name: myapp
        image: busybox:1.27 # 1.27
        command: ["bin/sh"]
        args: ["-c", "sleep 999999;"]
```

Let us now deploy myApp1.yaml and check that it is up and running by ensuring that the status of each of the three pods is Running:

```
$ kubectl apply -f myApp1.yaml
deployment.apps/myapp created
```

```
$ kubectl get pods -w
NAME                    READY  STATUS  RESTARTS   AGE
myapp-54785c6ddc-5zmvp 1/1    Running 0          17s
myapp-54785c6ddc-rbcv8 1/1    Running 0          17s
myapp-54785c6ddc-wlf8r 1/1    Running 0          17s
```

Suppose that we want to update the Deployment with a new, "problematic" version, stored in a manifest called `myApp2.yaml`: more on why *problematic* after the source code snippet:

```
# myApp2.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
  labels:
    app: myapp
spec:
  progressDeadlineSeconds: 60
  minReadySeconds: 10
  replicas: 3
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
      - name: myapp
        image: busybox:1.27 # 1.27
        command: ["bin/sh"]
        args: ["-c", "sleep 5; exit 1"] # bug!
```

We have purposely introduced a bug in the preceding manifest; the application exits after five seconds with an error. We now apply the new, *problematic* Deployment manifest:

```
$ kubectl apply -f myApp2.yaml
deployment.apps/myapp configured
```

If we watch the Pods' status, we will see that a new replica controller 5d79979bc9 will be spawned and that it will create only one pod named nm7pv. Please note that the healthy replica controller 54785c6ddc remains unaltered:

```
$ kubectl get pods -w
NAME                       STATUS            RESTARTS
myapp-54785c6ddc-5zmvp Running           0
myapp-54785c6ddc-rbcv8 Running           0
myapp-54785c6ddc-wlf8r Running           0
myapp-5d79979bc9-sgqsn Pending           0
myapp-5d79979bc9-sgqsn Pending           0
myapp-5d79979bc9-sgqsn ContainerCreating 0s
myapp-5d79979bc9-sgqsn Running           0
myapp-5d79979bc9-sgqsn Error             0
myapp-5d79979bc9-sgqsn Running           1
myapp-5d79979bc9-sgqsn Error             1
myapp-5d79979bc9-sgqsn CrashLoopBackOff  1
...
```

After one second, the pod sgqsn is seen as Running, but because we had set minReadySeconds to ten seconds, Kubernetes does not move to spin up a new Pod yet. As expected, after five seconds, the Pod exists with error, and Kubernetes proceeds to restart the Pod.

Since we had also set the deadlineProgressSeconds to 60, the new faulty Deployment will expire after a while:

```
$ kubectl rollout status deploy/myapp
Waiting for deployment "myapp" rollout to finish:
1 out of 3 new replicas have been updated...
Waiting for deployment "myapp" rollout to finish:
1 out of 3 new replicas have been updated...
error: deployment "myapp" exceeded its
progress deadline
```

The end result will be that the existing healthy ReplicaSet myapp-54785c6ddc has remained unaltered; this behavior allows us fix the failed Deployment and try again without disruptions to the users of the healthy Pods:

```
$ kubectl get pods
NAME                       STATUS           RESTARTS
myapp-54785c6ddc-5zmvp Running          0
myapp-54785c6ddc-rbcv8 Running          0
myapp-54785c6ddc-wlf8r Running          0
myapp-5d79979bc9-sgqsn CrashLoopBackOff 5
```

Please note that when using the minReadySeconds property, the time specified in deadlineProgressSeconds may be slightly extended.

Rollout History

The rollout history for a given Deployment consists of the *revision-tracking* updates that have been performed against it. As we have explained before, scaling-only updates do not create a revision and are not part of a Deployment's history. Revisions are the mechanism that allows performing rollbacks. A rollout's history is a list of revisions in ascending order, and it is retrieved using the `kubectl rollout history deploy/<NAME>` command. For example:

```
$ kubectl rollout history deploy/nginx
deployments "nginx"
REVISION        CHANGE-CAUSE
1               <none>
2               <none>
```

Note that the `CHANGE-CAUSE` field's value is `<none>` because change commands are not recorded by default. Kubernetes assigns an incremental revision number for each revision-tracking Deployment update. In addition, it can also record the command (e.g., `kubectl apply`, `kubectl set image`, etc.) that was used to create each revision. To enable this behavior, simply append the `--record` flag to each command. This will populate the `CHANGE-CAUSE` column—in turn obtained from `deployment.metadata.annotations.kubernetes.io/change-cause`. For example:

```
$ kubectl run nginx --image=nginx:1.7.0 --record
deployment.apps/nginx created

$ kubectl set image deploy/nginx nginx=nginx:1.9.0 \
    --record
deployment.extensions/nginx image updated

$ kubectl rollout history deploy/nginx
deployments "nginx"
```

```
REVISION CHANGE-CAUSE
1        kubectl run nginx --image=nginx:1.7.0 ...
2        kubectl set image deploy/nginx ...
```

It is also useful to be aware that the command used to create a revision may not be sufficient to tell the difference between a revision and another especially in the case when the Deployment's details are captured in a file—declarative approach. To get details about a revision's images and other metadata, we should use the `kubectl rollout history --revision=<N>` command. For example:

```
$ kubectl rollout history deploy/nginx --revision=1
deployments "nginx" with revision #1
Pod Template:
  Labels:       pod-template-hash=4217019353
        run=nginx
  Annotations:  kubernetes.io/change-cause=kubectl
                run nginx --image=nginx --record=true
  Containers:
   nginx:
    Image:      nginx
    Port:       <none>
    Environment:        <none>
    Mounts:     <none>
  Volumes:      <none>
...
```

To keep the number of items in the revision history manageable, the `deployment.spec.revisionHistoryLimit` helps establish a maximum limit.

Rolling Back Deployments

In Kubernetes, rolling back a Deployment is known as an *undo* procedure, and it is performed using the `kubectl rollout undo deploy/<NAME>` command. The undo procedure actually creates a new revision whose parameters are the same as the previous one. In this way, a further undo will result in getting back to the version prior to it. In the following example, we create two Deployments, check the rollout history, perform an undo operation, and then check the rollout history again:

```
$ kubectl run nginx --image=nginx:1.7.0 --record
deployment.apps/nginx created

$ kubectl set image deploy/nginx nginx=nginx:1.9.0 \
    --record
deployment.extensions/nginx image updated

$ kubectl rollout history deploy/nginx
deployments "nginx"
REVISION CHANGE-CAUSE
1        kubectl run nginx --image=nginx:1.7.0 ...
2        kubectl set image deploy/nginx ...

$ kubectl rollout undo deploy/nginx
deployment.extensions/nginx

$ kubectl rollout history deploy/nginx
deployments "nginx"
REVISION CHANGE-CAUSE
2       kubectl set image deploy/nginx ...
3       kubectl run nginx --image=nginx:1.7.0 ...
```

It is also possible to roll back to a specific revision rather than the previous one. This is achieved by using the regular undo command and adding the `--to-revision=<N>` flag. For example:

```
$ kubectl rollout undo deploy/nginx --to-revision=2
```

Using `0` as the revision number causes Kubernetes to revert to the previous version—it is equivalent to omitting the flag.

The Horizontal Pod Autoscaler

Autoscaling refers to the runtime system's capability of allocating extra compute and storage resources, in an unattended manner, based on an observable metric such as CPU load. In Kubernetes, in particular, the current de facto autoscaler capability is provided by a service called the Horizontal Pod Autoscaler (HPA). The Horizontal Pod Autoscaler (HPA) is a regular Kubernetes API resource and controller which manages a Pod's number of replicas in an unattended manner based on observed resource utilization. It can be thought of as a bot which issues `kubectl scale` commands on behalf of the human administrator based on a Pod's scaling criteria, typically average CPU load.

To avoid confusion, it is worth understanding that "horizontal scaling" refers to the creation or deletion of Pod replicas across multiple Nodes—this is the only type of autoscaling that is officially implemented in Kubernetes at the time of writing through the Horizontal Pod Autoscaler (HPA) service. Horizontal scaling should not be mistaken for vertical scaling. Vertical scaling involves increasing the compute resources of a specific Node (e.g., RAM and CPU). The actual "other" type of scaling, besides horizontal scaling and vertical scaling, is *cluster scaling* which adjusts the number of Nodes (virtual or physical machines) rather than the number of Pods within a fixed number of Nodes. We provide an overview on cluster scaling toward the end of the chapter.

Setting Up Autoscaling

Setting up autoscaling in an imperative manner can be achieved using the `kubectl autoscale` command. We should have a running Deployment—or ReplicaSet—first, which will be controlled by the Horizontal Pod Autoscaler (HPA). We must also specify the minimum and maximum number of instances and, finally, the CPU percent to be used as a basis for scaling. The full command syntax is as follows: `kubectl autoscale deploy/<NAME> --min=<N> --max=<N> --cpu-percent=<N>`.

For example, for running Nginx in autoscaling mode, we would follow these two steps:

```
$ kubectl run nginx --image=nginx
deployment.apps/nginx created
```

```
$ kubectl autoscale deployment nginx \
    --min=1 --max=3 --cpu-percent=5
horizontalpodautoscaler.autoscaling/nginx autoscaled
```

The CPU percent of five is on purpose so that it is easy to observe the HPA behavior when the Deployment is subjected to a minimum amount of load.

Setting up an autoscaling declaratively requires the creation of a new manifest file which specifies the target Deployment—or ReplicaSet—the minimum and maximum number of replicas, as well as the CPU threshold:

```
# hpa.yaml
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: nginx
spec:
  maxReplicas: 3
  minReplicas: 1
  scaleTargetRef:
    kind: Deployment
    name: nginx
  targetCPUUtilizationPercentage: 5
```

To run the manifest, the `kubectl apply -f <FILE>` command is used, but we must make sure that there is a Deployment up and running before the autoscaling manifest is applied. For example:

```
$ kubectl run nginx --image=nginx --replicas=1
deployment.apps/nginx created
```

```
$ kubectl apply -f hpa.yaml
horizontalpodautoscaler.autoscaling/nginx created
```

Observing Autoscaling in Action

Observing autoscaling in action is simply a matter of setting a low CPU threshold (such as 5%) and then creating some CPU load on the running replicas(s). Let us see this process in action.

We first create a single replica and attach an HPA to it, as shown earlier:

```
$ kubectl run nginx --image=nginx
deployment.apps/nginx created
```

```
$ kubectl autoscale deployment nginx \
    --min=1 --max=3 --cpu-percent=5
horizontalpodautoscaler.autoscaling/nginx autoscaled
```

We then watch the behavior of the autoscaler:

```
$ kubectl get hpa -w
NAME  REFERENCE    TARGETS MINPODS MAXPODS REPLICAS
nginx Deployment/* 0% / 5% 1       3       1
```

We then open a separate shell, access the Deployment's Pod, and generate an infinite loop to cause some CPU load:

```
$ kubectl get pods
NAME                   READY STATUS    RESTARTS   AGE
nginx-4217019353-2fb1j 1/1   Running   0          27m
```

```
$ kubectl exec -it nginx-4217019353-sn1px \
    -- sh -c 'while true; do true; done'
```

If we go back to the shell in which we are watching the HPA controller, we will see how it ramps up the number of replicas:

```
$ kubectl get hpa -w
NAME  REFERENCE    TARGETS   MINPODS MAXPODS REPLICAS
nginx Deployment/*  0% / 5%  1       3       1
nginx Deployment/* 20% / 5%  1       3       1
nginx Deployment/* 65% / 5%  1       3       2
nginx Deployment/* 80% / 5%  1       3       3
nginx Deployment/* 90% / 5%  1       3       3
```

If we interrupt the infinite loop, we will see that the number of replicas will be scaled down to one after a while.

Another alternative is generating load on the Nginx HTTP server itself. This would be a more realistic scenario, but it takes some extra steps to set up. First, we need to get a tool to generate load such as ApacheBench:

```
$ sudo apt-get update
$ sudo apt-get install apache2-utils
```

Then we need to expose the Deployment on the external load balancer. This uses the Service controller command—to be explained further in-depth in Chapter 4:

```
$ kubectl expose deployment nginx \
    --type="LoadBalancer" --port=80 --target-port=80
service/nginx exposed
```

We wait until we obtain a public IP address:

```
$ kubectl get service -w
NAME       TYPE         CLUSTER-IP    EXTERNAL-IP
kubernetes ClusterIP    10.59.240.1   <none>
nginx      LoadBalancer 10.59.245.138 <pending>
nginx      LoadBalancer 10.59.245.138 35.197.222.105
```

Then, we “throw” excessive traffic at it, in this case, 1,000,000 requests using 100 separate threads to the external IP/port:

```
$ ab -n 1000000 -c 100 http://35.197.222.105:80/
This is ApacheBench, Version 2.3
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.
zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.
org/

Benchmarking 35.197.222.105 (be patient)
```

We can observe the results by running `kubectl get hpa -w`. Please note that nginx is highly efficient, and significant load, together with a fast connection, is equally required to spike the Pods’ CPUs above 5%.

Another aspect to consider, when watching the HPA in action, is that it does not react immediately. The reason is that the HPA normally queries resource utilization every 30 seconds—unless the defaults have been changed—and then reacts based on the last minute average across all available Pods. This means that, for example, a brief 99% CPU spike on a single Pod, during a 30-second window, may not be sufficient to take the aggregate average above the defined threshold.

Moreover, the HPA algorithm is implemented in such a way that bumpy scaling behavior is avoided. The HPA is relatively faster at scaling up than at scaling down since making services available takes precedence over saving compute resources.

For more information, please refer to https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.

Last but not least, HPA objects, when no longer required, can be disposed of by using the kubectl delete hpa/<NAME> command. For example:

```
$ kubectl delete hpa/nginx
horizontalpodautoscaler.autoscaling "nginx" deleted
```

Scaling the Kubernetes Cluster Itself

The HPA controller is confined to the resources available within a fixed Kubernetes cluster. It spawns new Pods, not brand new virtual machines to host Kubernetes Nodes.

GKE supports cluster-wise scaling; however, it is based on a different principle than the HPA. The scaling trigger is not CPU load or a similar metric; the allocation of new Nodes takes place when Pods request compute resources that are unavailable within the current cluster's Nodes. Likewise, scaling "down" involves consolidating Pods into fewer Nodes; this may have an impact on the workloads running on Pods that do not manage unexpected shutdowns in a graceful manner.

The most practical way to enable cluster-wise autoscaling in GCP is at the time of cluster creation using the gcloud container clusters create <CLUSTER-NAME> --num-nodes <NUM> (see Chapter 1 for other additional flags that may be required in the case of errors) command but adding --enable-autoscaling --min-nodes <MIN> --max-nodes <MAX> as additional arguments. In the following example, we set <MIN> to 3 but <MAX> to 6 so that double the capacity of the standard cluster can be made available if required:

```
$ gcloud container clusters create my-cluster \
    --num-nodes=3 --enable-autoscaling \
    --min-nodes 3 --max-nodes 6
```

```
Creating cluster my-cluster...
Cluster is being health-checked (master is healthy)
done.
Creating node pool my-pool...done.
NAME        LOCATION        MASTER_VERSION  NUM_NODES
my-cluster  europe-west2-a  1.12.8-gke.10   3
```

The easiest way to see cluster-wise autoscaling in action is by launching a Deployment with a significant number of replicas and see the number of available Nodes before and after:

```
$ kubectl get nodes
NAME                                        STATUS
gke-my-cluster-default-pool-3d996410-7307   Ready     gke-my-
cluster-default-pool-3d996410-d2wz   Ready
gke-my-cluster-default-pool-3d996410-gw59   Ready

$ kubectl run nginx --image=nginx:1.9.1 --replicas=25
deployment.apps/nginx created

# After 2 minutes
$ kubectl get nodes
NAME                                        STATUS
gke-my-cluster-default-pool-3d996410-7307   Ready
gke-my-cluster-default-pool-3d996410-d2wz   Ready
gke-my-cluster-default-pool-3d996410-gw59   Ready
gke-my-cluster-default-pool-3d996410-rhnp   Ready
gke-my-cluster-default-pool-3d996410-rjnc   Ready
```

Conversely, deleting the Deployment will prompt the autoscaler to reduce the number of Nodes:

```
$ kubectl delete deploy/nginx
deployment.extensions "nginx" deleted

# After some minutes
$ kubectl get nodes
NAME                                        STATUS
gke-my-cluster-default-pool-3d996410-7307   Ready
gke-my-cluster-default-pool-3d996410-d2wz   Ready
gke-my-cluster-default-pool-3d996410-gw59   Ready
```

Summary

In this chapter, we learned how to scale Pods and release new versions using different strategies (such as rolling and blue/green deployments) using the Deployment controller. We also saw how to monitor and control Deployments that are underway, for example, by pausing, resuming them, and even rolling back to a previous version. Finally, we learned how to set up autoscaling mechanisms, both at the Pod and Node levels, to achieve better cluster resource utilization.

CHAPTER 4

Service Discovery

Service discovery is the ability to locate the address of one or more Pods from within other Pods as well as the external world—the Internet. Kubernetes provides a Service controller to satisfy service discovery and connectivity use cases such as Pod-to-Pod, LAN-to-Pod, and Internet-to-Pod.

Service discovery is a necessity because Pods are volatile; they may be created and destroyed many times over throughout their life cycle, acquiring different IP address each time. The self-healing and scaling nature of Kubernetes also means that we often want a virtual IP address—and a round-robin load balancing mechanism—as opposed to the discrete address of specific, pet-like Pods.

In this chapter, we will first explore the three aforementioned connectivity use cases (Pod-to-Pod, LAN-to-Pod, and Internet-to-Pod). Then, we will look at the peculiarities of publishing services across different spaces and the exposure of multiple ports. Finally, we will contemplate how the Service controller helps instrument graceful startup and shutdown, as well as zero-downtime deployments.

Connectivity Use Cases Overview

The Service controller performs a variety of functions, but its main purpose is keeping track of Pods addresses and ports and publishing this information to interested service consumers. The Service controller also provides a single point of entry in a cluster scenario—multiple Pod replicas.

E. Garbarino, *Beginning Kubernetes on the Google Cloud Platform*,
https://doi.org/10.1007/978-1-4842-5491-2_4

To achieve its purpose, it uses other Kubernetes services such as `kube-dns` and `kube-proxy` which in turn leverage the underlying kernel and networking resources from the OS such as *iptables*.

The Service controller caters for a variety of use cases, but these are the most typical ones:

- **Pod-to-Pod:** This scenario involves a Pod connecting to other Pods within the same Kubernetes cluster. The *ClusterIP* service type is used for this purpose; it consists of a virtual IP address and DNS entry which is addressable by all Pods within the same cluster and that will add and remove replicas from the Cluster as they become ready and "unready," respectively.
- **LAN-to-Pod:** In this case, service consumers typically sit outside of the Kubernetes cluster—but are located within the same local area network (LAN). The *NodePort* service type is appropriate in typically used to satisfy this use case; the Service controller publishes a discrete port in every worker Node which is mapped to the exposed deployment.
- **Internet-to-Pod:** In most cases, we will want to expose at least one Deployment to the Internet. Kubernetes will interact with the Google Cloud Platform's load balancer so that an external—public IP—address is created and routed to the NodePorts. This is a *LoadBalancer* service type.

There is also the special case of a *headless service* used primarily in conjunction with StatefulSets to provide DNS-wise access to every individual Pod. This special case is covered separately in Chapter 9.

It is worth understanding that the Service controller provides a layer of indirection—be in the form of a DNS entry, an extra IP address, or port number—to Pods that have their own discrete IP address and that can be accessed directly. If what we need is simply to find out a Pods' IP addresses, we can use the `kubectl get pods -o wide -l <LABEL>` command where `<LABEL>` differentiates the Pods we are looking for from others. The `-l` flag is only required when a large number of Pods are running. Otherwise, we can probably tell the Pods apart by their naming convention.

In the next example, we first create three Nginx replicas and then find out their IP address:

```
$ kubectl run nginx --image=nginx --replicas=3
deployment.apps/nginx created
```

```
$ kubectl get pods -o wide -l run=nginx
NAME            READY STATUS    RESTARTS   IP
nginx-*-5s7fb   1/1   Running   0          10.0.1.13
nginx-*-fkjx4   1/1   Running   0          10.0.0.7
nginx-*-sg9bv   1/1   Running   0          10.0.1.14
```

If instead, we want the IP addresses by themselves—say, for piping them into some program—we can use a programmatic approach by specifying a JSON Path query:

```
$ kubectl get pods -o jsonpath \
   --template="{.items[*].status.podIP}" -l run=nginx
10.36.1.7 10.36.0.6 10.36.2.8
```

Pod-to-Pod Connectivity Use Case

Addressing a Pod from an external Pod involves creating a Service object which will observe the Pod(s) so that they are added or removed from a virtual IP address—called a *ClusterIP*—as they become ready and *unready*, respectively. The instrumentation of a container's "readiness" may be implemented via custom probes as explained in Chapter 2. The Service controller can target a variety of objects such as bare Pods and ReplicaSets, but we will concentrate on Deployments only.

The first step is creating a Service controller. The imperative command for creating a service that observes an existing deployment is `kubectl expose deploy/<NAME>`. The optional flag `--name=<NAME>` will give the service a name different than its target deployment, and `--port=<NUMBER>` is only necessary if we had not specified a port on the target object. For example:

```
$ kubectl run nginx --image=nginx --replicas=3
deployment.apps/nginx created

$ kubectl expose deploy/nginx --port=80
service/nginx exposed
```

The declarative approach is similar, but it requires the use of a label selector (Chapter 2) to identify the target deployment:

```
# service.yaml
kind: Service
apiVersion: v1
metadata:
  name: nginx
spec:
  selector:
    run: nginx
  ports:
  - protocol: TCP
    port: 80
```

This manifest can be applied right after creating a deployment as follows:

```
$ kubectl run nginx --image=nginx --replicas=3
deployment "nginx" created

$ kubectl apply -f service.yaml
service "nginx" created
```

So far, we have solved the problem of "exposing" a deployment, but what is the outcome? What is it that we can do now that was not possible before? First, let us type `kubectl get services` to see the details about the object we have just created:

```
$ kubectl get services
NAME       TYPE      CLUSTER-IP    EXT-IP PORT(S)
kubernetes ClusterIP 10.39.240.1   <none> 443/TCP
nginx      ClusterIP 10.39.243.143 <none> 80/TCP
```

The IP address `10.39.243.143` is now ready to receive ingress traffic from any Pod on port 80. We can check this by running a dummy, disposable Pod that connects to this endpoint:

```
$ kubectl run test --rm -i --image=alpine \
    --restart=Never \
    -- wget -O - http://10.39.243.143 | grep title
<title>Welcome to nginx!</title>
```

We have a virtual IP to reach out to our Deployment's Pod replicas, but how does a Pod get to find out about the IP address in the first place? Well, the good news is that Kubernetes creates a DNS entry for each Service controller. In a simple, single cluster, single namespace scenario—like all the examples in this text—we can simply use the Service name itself. For example, assuming that we are inside another Pod (e.g., an Alpine instance), we can access the Nginx web server as follows:

```
$ kubectl run test --rm -i --image=alpine \
    --restart=Never \
    -- wget -O - http://nginx | grep title
<title>Welcome to nginx!</title>
```

Note that rather than using `http://10.39.243.143`, we now use `http://nginx` instead. Each HTTP request to Nginx will hit one of the three Pod replicas in a round-robin fashion. If we want to convince us that this is indeed the case, we can change the `index.html` content of each Nginx Pod so that it displays the Pod's name rather than the same default welcome page. Assuming our three-replica Nginx deployment is still running, we can apply the suggested change by first extracting the Deployment's Pod names, and then by overwriting the contents of `index.html` in each Pod with the value of `$HOSTNAME`:

```
# Extract Pod names
$ pods=$(kubectl get pods -l run=nginx -o jsonpath \
       --template="{.items[*].metadata.name})"

# Change the contents of index.html for every Pod
$ for pod in $pods; \
    do kubectl exec -ti $pod \
           -- bash -c "echo \$HOSTNAME > \
           /usr/share/nginx/html/index.html"; \
    done
```

Now we can launch an adhoc Pod again; this time, though, we will be running requests against `http://nginx` in a loop until we press Ctrl+C:

```
$ kubectl run test --rm -i --image=alpine \
    --restart=Never -- \
    sh -c "while true; do wget -q -O \
    - http://nginx ; sleep 1 ; done"
nginx-dbddb74b8-t728t
nginx-dbddb74b8-h87s4
nginx-dbddb74b8-mwcg4
nginx-dbddb74b8-h87s4
```

```
nginx-dbddb74b8-h87s4
nginx-dbddb74b8-t728t
nginx-dbddb74b8-h87s4
nginx-dbddb74b8-h87s4
...
```

As we can see in the resulting output, a round-robin mechanism is in action since each request lands on a random Pod.

LAN-to-Pod Connectivity Use Case

Accessing Pods from an external host to the Kubernetes cluster involves exposing the services using the `NodePort` service type (the default service type is `ClusterIP`). This is just a matter of adding the `--type=NodePort` flag to the `kubectl expose` command. For example:

```
$ kubectl run nginx --image=nginx --replicas=3
deployment.apps/nginx created
```

```
$ kubectl expose deploy/nginx --type="NodePort" \
    --port=80
service/nginx exposed
```

The result is that Nginx HTTP server can be accessed now through a discrete port on any of the Node's IP addresses. First, let us find what the assigned port is:

```
$ kubectl describe service/nginx | grep NodePort
NodePort:                 <unset>  30091/TCP
```

We can see that the automatically assigned port is 30091. We can now make requests to the Nginx's web server through any of the Kubernetes' worker Nodes on this port from a machine *outside the Kubernetes cluster* in the same local area using the external IP address:

```
$ kubectl get nodes -o wide
NAME                   STATUS  AGE EXTERNAL-IP
gke-*-9777d23b-9103    Ready   7h  35.189.64.73
gke-*-9777d23b-m6hk    Ready   7h  35.197.208.108
gke-*-9777d23b-r4s9    Ready   7h  35.197.192.9
```

```
$ curl -s http://35.189.64.73:30091 | grep title
<title>Welcome to nginx!</title>
$ curl -s http://35.197.208.108:30091 | grep title
<title>Welcome to nginx!</title>
$ curl -s http://35.197.192.9:30091 | grep title
<title>Welcome to nginx!</title>
```

Note The Lan-to-Pod examples may not work directly in the Google Cloud Shell unless further security/network settings are applied. Such settings are outside the scope of this book.

To conclude this section, the declarative version of the `kubectl expose deploy/nginx --type="NodePort" --port=80` command is provided here:

```
# serviceNodePort.yaml
kind: Service
apiVersion: v1
metadata:
  name: nginx
```

```
spec:
  selector:
    run: nginx
  ports:
  - protocol: TCP
    port: 80
  type: NodePort
```

The only difference between the ClusterIP manifest used for Pod-to-Pod access is that the attribute `type` is added and set to `NodePort`. This attribute's value is `ClusterIP` by default if left undeclared.

Internet-to-Pod Connectivity Use Case

Accessing Pods from the Internet involves creating `LoadBalancer` service type. The `LoadBalancer` service type is similar to the `NodePort` service type in that it will publish the exposed object on a discrete port in every Node of the Kubernetes cluster. The difference is that, in addition to this, it will interact with the Google Cloud Platform's load balancer and allocate a public IP address that can direct traffic to these ports.

The following example creates a cluster of three Nginx replicas and exposes the Deployment to the Internet. Please note that the last command uses the `-w` flag in order to wait until the external—public—IP address is allocated:

```
$ kubectl run nginx --image=nginx --replicas=3
deployment.apps/nginx created

$ kubectl expose deploy/nginx --type=LoadBalancer \
    --port=80
service/nginx exposed
```

```
$ kubectl get services -w
NAME  TYPE         CLUSTER-IP    EXT-IP
nginx LoadBalancer 10.39.249.178 <pending>
nginx LoadBalancer 10.39.249.178 35.189.65.215
```

Whenever the load balancer is assigned a public IP address, the `service.status.loadBalancer.ingress.ip` property—or `.hostname` on other cloud vendors such as AWS—will be populated. If we want to capture the public IP address programmatically, all we must do is wait until this attribute is set. We can instrument this solution through a *while loop* in Bash, for example:

```
while [ -z $PUBLIC_IP ]; \
 do PUBLIC_IP=$(kubectl get service/nginx \
 -o jsonpath \
 --template="{.status.loadBalancer.ingress[*].ip}");\
 sleep 1; \
 done; \
 echo $PUBLIC_IP
35.189.65.215
```

The declarative version of `kubectl expose deploy/nginx --type=LoadBalancer --port=80` is presented in the next code listing. The only difference between the manifest used for the LAN-to-Pod use case is that the `type` attribute is set to `LoadBalancer`. The manifest is applied using the `kubectl apply -f serviceLoadBalancer.yaml` command. We might want to dispose of any running clashing Service before applying this command by issuing the `kubectl delete service/nginx` command first.

```
# serviceLoadBalancer.yaml
kind: Service
apiVersion: v1
metadata:
  name: nginx
```

```
spec:
  selector:
    run: nginx
  ports:
  - protocol: TCP
    port: 80
  type: LoadBalancer
```

Accessing Services in Different Namespaces

All the examples we have seen so far live in the *default* namespace. This is as though every `kubectl` command had the `-n default` flag added by default. As such, we never had to be concerned with full DNS names. Whenever we expose the Nginx deployment by typing `kubectl expose deploy/nginx`, we can access the resulting service from Pods without any additional domain components, for example, by typing `wget http://nginx`.

If more namespaces are in use, though, things may get tricky, and it might be useful to understand the shape of the full DNS record associated with each service. Let us suppose that there is a `nginx` Service in the `default` namespace—in which case there is no need to indicate a specific namespace since this is the default—and another equally named service in the `production` namespace as follows:

```
# nginx in the default namespace

$ kubectl run nginx --image=nginx --port=80
deployment.apps/nginx created

$ kubectl expose deploy/nginx
service/nginx exposed
```

```
# nginx in the production namespace

$ kubectl create namespace production
namespace/production created

$ kubectl run nginx --image=nginx --port=80 \
    -n production
deployment.apps/nginx created

$ kubectl expose deploy/nginx -n production
service/nginx exposed
```

The result is two Services named `nginx` but that live in different namespaces:

```
$ kubectl get services --all-namespaces | grep nginx
NAMESPACE  NAME  TYPE      CLUSTER-IP      PORT(S)
default    nginx ClusterIP 10.39.243.143   80/TCP
production nginx ClusterIP 10.39.244.112   80/TCP
```

Whether we get the Nginx service published on `10.39.243.143` or `10.39.244.112` will depend on the namespace the requesting Pod is running on:

```
$ kubectl run test --rm -ti --image=alpine \
    --restart=Never \
    -- getent hosts nginx | awk '{ print $1 }'
10.39.243.143

$ kubectl run test --rm -ti --image=alpine \
    --restart=Never \
    -n production \
    -- getent hosts nginx | awk '{ print $1 }'
10.39.244.112
```

The Pods within the default space will connect to 10.39.243.143 when using nginx as a host, whereas those in the production namespace will connect to 10.39.244.112. The way to reach a default ClusterIP from production and vice versa is to use the full domain name.

The default configuration uses the service-name.namespace.svc.cluster.local convention where service-name is nginx and namespace is either default or production in our example:

```
$ kubectl run test --rm -ti --image=alpine \
    --restart=Never \
    -- sh -c \
    "getent hosts nginx.default.svc.cluster.local; \
    getent hosts nginx.production.svc.cluster.local"
10.39.243.143     nginx.default.svc.cluster.local
10.39.244.112     nginx.production.svc.cluster.local
```

Exposing Services on a Different Port

The kubectl expose command and its equivalent declarative form will introspect the target object and expose it on its declared port. If no port information is available, then we specify the port using the --port flag or the service.spec.ports.port attribute. In our previous examples, the exposed port has always coincided with the actual Pod's port; whenever the exposed port differs from published one, it must be specified using either the --target-port flag or the service.spec.ports.targetPort attribute in the Service manifest.

In the next example, we create a Nginx deployment as usual—on port 80—but expose it on port 8000 on the public load balancer. Please note that given that the exposed port and published port are different, we must specify the exposed port using the --target-port flag:

```
$ kubectl run nginx --image=nginx
deployment.apps/nginx created
```

```
$ kubectl expose deploy/nginx --port=8000 \
    --target-port=80 \
    --type=LoadBalancer
service/nginx exposed
```

```
$ kubectl get services -w
NAME  TYPE         EXTERNAL-IP   PORT(S)
nginx LoadBalancer <pending>     8000:31937/TCP
nginx LoadBalancer 35.189.65.99  8000:31937/TCP
```

The result is that Nginx is now accessible on the public Internet on port 8000 even though it is exposed on port 80 at the Pod level:

```
$ curl -s -i http://35.189.65.99:8000 | grep title
<title>Welcome to nginx!</title>
```

For completeness, here we have the declarative equivalent of the presented `kubectl expose` command; it is applied using the `kubectl apply -f serviceLoadBalancerMapped.yaml` command. We might need to delete the service created using the imperative approach first, by running `kubectl delete service/nginx`:

```
# serviceLoadBalancerMapped.yaml
kind: Service
apiVersion: v1
metadata:
  name: nginx
spec:
  selector:
    run: nginx
  ports:
  - protocol: TCP
    port: 8000
    targetPort: 80
  type: LoadBalancer
```

Exposing Multiple Ports

A Pod may expose multiple ports either because it contains multiple containers or because a single container listens on multiple ports.
For example, a web server typically listens both on port 80 for regular, unencrypted traffic, and on port 443 for TLS traffic. The `spec.ports` attribute in the service manifest expects an array of port declarations, so all we have to do is append more elements to this array, keeping in mind that whenever two or more ports are defined, each must be given a unique name so that they can be disambiguated:

```
# serviceMultiplePorts.yaml
kind: Service
apiVersion: v1
metadata:
  name: nginx
spec:
  selector:
    run: nginx
  ports:
  - name: http  # user-defined name
    protocol: TCP
    port: 80
    targetPort: 80
  - name: https # user-defined name
    protocol: TCP
    port: 443
    targetPort: 443
  type: LoadBalancer
```

Canary Releases

The idea behind a canary release is that we expose a new version of a service only to a subset of users—before rolling it out to the entire userbase—so that we can observe the new service's behavior for a while until we are convinced that it does not present runtime defects.

An easy way to implement this strategy is by creating a Service object that—during the canary release—includes a new Pod, "the canary" in its load balanced cluster. For example, say that the production cluster includes three replicas of Pod version 1.0 in its current version, we can include an instance of the Pod v2.0 so that 1/4 of the traffic (in average) reaches the new Pod.

The key ingredients of this strategy are labels and selectors, which we have covered in Chapter 2. All we must do is add a label to the Pods that are meant to be in production and a matching selector in the service object. In this way, we can create the Service object in advance and let the Pods declare a label that will make them be selected by the Service object automatically. This is easier to see in action than to digest in words; let us follow this process step by step.

We first create a Service manifest whose selector will be looking for Pods whose label `prod` is equals to `true`:

```
# myservice.yaml
kind: Service
apiVersion: v1
metadata:
  name: myservice
spec:
  selector:
    prod: "true"
  ports:
  - protocol: TCP
```

```
    port: 80
    targetPort: 80
  type: LoadBalancer
```

After applying the manifest, we can keep a second window open in which we will see, interactively, which Endpoints join and leave the cluster:

```
$ kubectl apply -f myservice.yaml
service/myservice created
```

```
$ kubectl get endpoints/myservice -w
NAME        ENDPOINTS   AGE
myservice   <none>      29m
```

Given that we haven't created any Pods yet, there are no Endpoints in the cluster as evidenced by the value `<none>` under the `ENDPOINTS` column. Let us create the "incumbent" v1 production Deployment consisting of three replicas:

```
$ kubectl run v1 --image=nginx --port=80 \
    --replicas=3 --labels="prod=true"
deployment.apps/v1 created
```

If we check the terminal window in which we had left `kubectl get endpoints/myservice -w` running, we will notice that three new endpoints will be added. For example:

```
$ kubectl get endpoints/myservice -w
NAME      ENDPOINTS
myservice 10.36.2.10:80
myservice <none>
myservice 10.36.2.11:80
myservice 10.36.0.6:80,10.36.2.11:80
myservice 10.36.0.6:80,10.36.1.8:80,10.36.2.11:80
```

Since we have requested an external IP, we can check out that our v1 service is operational using `curl`:

```
$ kubectl get service/myservice
NAME      TYPE         EXTERNAL-IP   PORT(S)
myservice LoadBalancer 35.197.192.45 80:30385/TCP
```

```
$ curl -I -s http://35.197.192.45 | grep Server
Server: nginx/1.13.8
```

Now it is time to introduce a canary Pod. Let us create a v2 deployment. The differences are that the label `prod` is set to `false` and that we will be using the Apache server rather than Nginx as the container image for the new version:

```
$ kubectl run v2 --image=httpd --port=80 \
    --replicas=3 --labels="prod=false"
deployment.apps/v2 created
```

As of now, we can see that there are six Pod replicas in total. The `-L <LABEL>` displays the label's value:

```
$ kubectl get pods -L prod
NAME                READY  STATUS   RESTARTS  PROD
v1-3781799777-219m3  1/1    Running  0         true
v1-3781799777-qc29z  1/1    Running  0         true
v1-3781799777-tbj4f  1/1    Running  0         true
v2-3597628489-2kl05  1/1    Running  0         false
v2-3597628489-p8jcv  1/1    Running  0         false
v2-3597628489-zc95w  1/1    Running  0         false
```

In order to get one of the v2 Pods into the `myservice` cluster, all we have to do is set the label accordingly. Here we pick the Pod named `v2-3597628489-2kl05` and set its `prod` label to `true`:

```
$ kubectl label pod/v2-3597628489-2kl05 \
    prod=true --overwrite
pod "v2-3597628489-2kl05" labeled
```

Right after the label operation, if we check the window in which the command `kubectl get endpoints/myservice -w` is running, we will see that an extra endpoint will have been added. At this moment, if we hit the public IP address repeatedly, we will notice that some of the requests land on the Apache web server:

```
$ while true ; do curl -I -s http://35.197.192.45 \
    | grep Server ; done
Server: nginx/1.13.8
Server: nginx/1.13.8
Server: nginx/1.13.8
Server: nginx/1.13.8
Server: Apache/2.4.29 (Unix)
Server: nginx/1.13.8
Server: nginx/1.13.8
...
```

Once we are happy with the behavior of v2, we can promote the rest of the v2 fleet to production. This can be accomplished on a step-by-step basis by applying labels as shown before; however, at this point, it is best to create a formal deployment manifest so that, once applied, Kubernetes takes care of introducing the v2 Pods and retiring the v1 ones in an seamless fashion—please refer to Chapter 3 for further details on rolling and blue/green Deployments.

To conclude this section, the combination of labels and selectors provide us with flexibility in terms of what Pods are exposed to service consumers. One practical application is the instrumentation of canary releases.

Canary Releases and Inconsistent Versions

A canary release may consist of an internal code enhancement—or bug fix—but it may also introduce new features to users. Such new features may relate both to the visual user interface itself (e.g., HTML and CSS) as well as to the data APIs that power such interfaces. Whenever the latter is the case, the fact that each request may land on any random Pod may be problematic. For example, the first request may retrieve a new v2 AngularJS code that relies on a v2 REST API, but when the second request hits the load balancer, the selected Pod may be v1 and serve an incorrection version of such said API. In essence, when a canary release introduces external changes—both UI and data wise—we usually want users to stay on the same version, either the current release or the canary one.

The technical term for users that land on the same instance of a service is *sticky sessions* or *session affinity*—the latter is the one used by Kubernetes. There are myriad of approaches to implementing session affinity depending on how much data is available to identify a single user. For example, cookies or session identifiers appended to the URL may be used in the scenario of web applications, but what if the interface is, say, Protocol Buffers or Thrift rather than HTTP? The only detail that can *somewhat* identify a given user from another is their client IP address, and this is exactly what the Service object can use to implement this behavior.

By default, session affinity is disabled. Implementing session affinity in an imperative context is simply a matter of adding the `--session-affinity=ClientIP` flag to the `kubectl expose` command. For example:

```
# Assume there is a Nginx Deployment running
$ kubectl expose deploy/nginx --type=LoadBalancer \
    --session-affinity=ClientIP
service "nginx" exposed
```

The declarative version involves setting the `service.spec.sessionAffinity` property and applying the manifest running the `kubectl apply -f serviceSessionAffinity.yaml` command:

```
# serviceSessionAffinity.yaml
kind: Service
apiVersion: v1
metadata:
  name: nginx
spec:
  sessionAffinity: ClientIP
  selector:
    run: nginx
  ports:
  - protocol: TCP
    port: 80
  type: LoadBalancer
```

The limitation of IP-based session affinity is that multiple users may share the same IP address as it is typically the case in companies and schools. Likewise, the same logical user may appear as originating from multiple IP addresses as in the case of a user who watches Netflix using their Wi-Fi broadband router at home and through their smartphone using LTE or similar. For such scenarios, the service's capabilities are insufficient; therefore, it is best to use a service that has layer 7 introspection capabilities such as the *Ingress* controller. For more information, refer to `https://kubernetes.io/docs/concepts/services-networking/ingress/`.

Graceful Startup and Shutdown

An application that benefits from the graceful startup property is only ready to accept user requests once its bootstrapping process is completed, whereas graceful shutdown means that an application should not stop abruptly—breaking its users or the integrity of its dependencies as a result. In other words, applications should be able to tell "I had my coffee and took my shower, I am ready to work" as well as "I call it a day. I am going to brush my teeth and go to bed now."

The Service controller uses two main mechanisms to decide whether a given Pod should form part of its cluster: the Pod's label and the Pod's readiness status which is implemented using a Readiness probe (see Chapter 2). In a Spring-based Java application, for example, there are a few seconds of delay between the time that application is launched and the time the Spring Boot framework has fully initialized and is ready to accept http requests.

Zero-Downtime Deployments

Zero-downtime deployments are achieved by the Kubernetes' Deployment controller by registering new Pods with the Service controller and removing old ones, in a coordinated fashion, so that the end user is always being served by a minimum number of Pod replicas. As explained in Chapter 3, zero-downtime deployments may be implemented either as rolling deployments or as blue/green ones.

Seeing a zero-downtime deployment in action takes only a few steps. First, we create a regular Nginx Deployment object and expose it through the public load balancer. We set `--session-affinity=ClientIP` so that the consuming client experiences a seamless transition to the new upgraded Pod(s) once ready:

```
$ kubectl run site --image=nginx --replicas=3
deployment.apps/site created

$ kubectl expose deploy/site \
    --port=80 --session-affinity=ClientIP \
    --type=LoadBalancer
service/site exposed

# Confirm public IP address
$ kubectl get services -w
NAME    TYPE          EXTERNAL-IP     PORT(S)
site    LoadBalancer  35.197.210.194  80:30534/TCP
```

We then open a separate terminal window that we will use to leave a simple http client running in an infinite loop:

```
$ while true ; do curl -s -I http://35.197.210.194/ \
    | grep Server; sleep 1 ; done
Server: nginx/1.17.1
Server: nginx/1.17.1
Server: nginx/1.17.1
...
```

Now all we have to do is transition deploy/site to a new deployment which can be achieved simply by changing its underlying container image. Let us use the Apache HTTPD image from Docker Hub:

```
$ kubectl set image deploy/site site=httpd
deployment.extensions/site image updated
```

If we go back to the window where the sample client is running, we will see that soon we will be greeted by the Apache HTTPD server rather than Nginx:

```
Server: nginx/1.17.1
Server: nginx/1.17.1
Server: nginx/1.17.1
Server: Apache/2.4.39 (Unix)
Server: Apache/2.4.39 (Unix)
Server: Apache/2.4.39 (Unix)
...
```

It also interesting to leave another, parallel, window open to monitor Pod activity using the `kubectl get pod -w` command so that we can observe how new Pods are being spun up and old ones are being terminated.

Pods' Endpoints

In most cases, the role of the Service controller is that of providing a single endpoint for two or more Pods. However, in certain circumstances, we may want to identify which are the specific endpoints of those Pods that the Service controller has selected.

The `kubectl get endpoints/<SERVICE-NAME>` command displays up to three endpoints directly on the screen, and it is used for immediate debugging purposes. For example:

```
$ kubectl get endpoints/nginx -o wide
NAME     ENDPOINTS
nginx    10.4.0.6:80,10.4.1.6:80,10.4.2.6:80
```

The same information can be retrieved using the kubectl describe service/<SERVICE-NAME> command as we have seen before. If we want a more programmatic approach that allows us to react to changes in the number and value of the endpoints, we can use a JSONPath query instead. For example:

```
$ kubectl get endpoints/nginx -o jsonpath \
    --template= "{.subsets[*].addresses[*].ip}"
10.4.0.6 10.4.1.6 10.4.2.6
```

Management Recap

As we have seen, services may be created imperatively using the kubectl expose command or declaratively using a manifest file. Services are listed using the kubectl get services command and deleted using the kubectl delete service/<SERVICE-NAME> command. For example:

```
$ kubectl get services
NAME        TYPE         CLUSTER-IP   EXTERNAL-IP
kubernetes  ClusterIP    10.7.240.1   <none>
nginx       LoadBalancer 10.7.241.102 35.186.156.253
```

```
$ kubectl delete services/nginx
service "nginx" deleted
```

Please note that when a service is deleted, the underlying Deployment will still keep running since they both have separate life cycles.

Summary

In this chapter, we learned that the Service controller helps create a layer of indirection between service consumers and Pods in order to facilitate instrumenting properties such as self-healing, canary releases, load balancing, graceful startup and shutdown, and zero-downtime deployments.

We covered specific connectivity use cases such as Pod-to-Pod, LAN-to-Pod, and Internet-to-Pod and saw that the latter is of special usefulness because it allows to reach our applications using a public IP address.

We also explained how to use full DNS records to disambiguate clashing services across different namespaces and the need to name ports when more than one is declared.

CHAPTER 5

ConfigMap and Secrets

One key principle of cloud native applications is that of *externalized configuration*. In the Twelve-Factor App methodology, this architectural property is best described by factor III. A relevant passage within this factor, at `https://12factor.net/config`, reads:

> *The twelve-factor app stores config in environment variables (often shortened to env vars or env). Env vars are easy to change between deploys without changing any code; unlike config files, there is little chance of them being checked into the code repo accidentally; and unlike custom config files, or other config mechanisms such as Java System Properties, they are a language- and OS-agnostic standard.*

Containers within Pods typically run a regular Linux distribution—such as Alpine—which means that Kubernetes can assume the presence of a shell and environment variables, unlike a low-level virtualization platform that is agnostic to the operating system.

As suggested by the Twelve-Factor App passage for factor III, nearly all programming languages can access environment variables, so this is certainly a universal and portable approach to pass configuration details into applications. Kubernetes is not limited to simply populating environment variables though; it can also make configuration available through a virtual

E. Garbarino, *Beginning Kubernetes on the Google Cloud Platform*,
https://doi.org/10.1007/978-1-4842-5491-2_5

file system. It also has a few other tricks under its sleeve such as the ability to parse key/value pairs from files and obfuscate sensitive data.

This chapter is organized into two broad parts. The first part covers the setting of environment variables in a manual fashion and, then, their automated population using the ConfigMap object through a variety of approaches: literal values, hardcoded values in the manifest, and data loaded from files. Special attention is also given to the storage of complex configuration data both in plain text and binary forms and the exposure of configuration variables using a virtual file system—which facilitates live configuration updates.

The second part focuses on Secrets which is ConfigMap's sister capability: it supports nearly all the same functionality as ConfigMap except that it is more appropriate for passwords and other types of sensitive data. Within Secrets, the special case of Docker Registry credentials—required when pulling images from a Docker Registry that requires authentication—and the storage of TLS certificates and keys are also treated.

Setting Environment Variables Manually

In an imperative scenario, environment variables can be defined by adding the `--env=<NAME>=<VALUE>` flag to the `kubectl run` command. For example, let us say that we have a variable called `mysql_host` and another one called `ldap_host`, whose values are `mysql1.company.com` and `ldap1.company.com`, respectively; we would apply these properties in this way:

```
$ kubectl run my-pod --rm -i --image=alpine \
    --restart=Never \
    --env=mysql_host=mysql1.company.com \
    --env=ldap_host=ldap1.company.com \
    -- printenv | grep host
mysql_host=mysql1.company.com
ldap_host=ldap1.company.com
```

Note in the last argument that we run the printenv command and grep the variables that contain the keyword host in them; the result includes the two variables we had passed using the --env=<NAME>=<VALUE> flag.

The declarative version requires that we set the pod.spec.containers.env property within a Pod manifest:

```
# podHardCodedEnv.yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  restartPolicy: Never
  containers:
    - name: alpine
      image: alpine
      args: ["sh","-c","printenv | grep host"]
      env:
        - name: mysql_host
          value: mysql1.company.com
        - name: ldap_host
          value: ldap1.company.com
```

There will not be any terminal output when running the command kubectl apply -f podHardCodedEnv.yaml. This is because the printenv and grep commands declared in the manifest's args attribute will exit successfully and the apply command does not print out the Pod's standard output; we therefore need to consult its logs:

```
$ kubectl logs pod/my-pod
mysql_host=mysql1.company.com
ldap_host=ldap1.company.com
```

Even though we have managed to externalize the `mysql_host` and `ldap_host` in the presented example, we still have a relatively inflexible system in which a new version of the Pod manifest would need to be produced for every new environment configuration. Declaring environment variables inside a Pod manifest is only appropriate when they are constants that apply across all environments. In the next section, we will learn how to decouple configuration from the Pod manifest by using the ConfigMap object.

Storing Configuration Properties in Kubernetes

Kubernetes provides the ConfigMap object for the purpose of storing global configuration properties that are decoupled from the details (namely configuration manifests) of individual workloads such as monolithic Pods, Deployments, Jobs, and so on. A basic ConfigMap object is composed of a top-level name—The ConfigMap "name" itself—and a set of key/value pairs. Normally, a new ConfigMap name is used for every set of closely related properties. Likewise, a given ConfigMap may be applicable to multiple applications; for example, the same MySQL credentials may be used both by a Java and a .NET containerized applications. This approach facilitates the centralized management of common configuration settings.

A ConfigMap is created, in an imperative fashion, by using the `kubectl create configmap <NAME>` command and adding as many `--from-literal=<KEY>=<VALUE>` flags as the number of properties that we have. For example, the following command creates a ConfigMap called `data-sources` with the `mysql_host=mysql1.company.com` and `ldap_host=ldap1.company.com` properties:

```
$ kubectl create configmap data-sources \
    --from-literal=mysql_host=mysql1.company.com \
    --from-literal=ldap_host=ldap1.company.com
configmap/data-sources created
```

This command is equivalent to the declarative version shown next. It is applied using the `kubectl apply -f simpleconfigmap.yaml` command:

```
# simpleconfigmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: data-sources
data:
  mysql_host: mysql1.company.com
  ldap_host: ldap1.company.com
```

Now that we have created a ConfigMap object, we can quickly check its state using the `kubectl describe` command:

```
$ kubectl describe configmap/data-sources
...
Data
====
ldap_host:
----
ldap1.company.com
mysql_host:
----
mysql1.company.com
```

At this point, we have managed to use the ConfigMap object to store configuration data represented as key/value pairs, but how do we extract the values back? A programmatic approach involves using a JSONPath query and storing the results in environment variables:

```
$ mysql_host=$(kubectl get configmap/data-sources \
    -o jsonpath --template="{.data.mysql_host}")
$ ldap_host=$(kubectl get configmap/data-sources \
    -o jsonpath --template="{.data.ldap_host}")
```

We can then use the assigned variables to pass configuration values to a Pod as follows:

```
$ kubectl run my-pod --rm -i --image=alpine \
    --restart=Never \
    --env=mysql_host=$mysql_host \
    --env=ldap_host=$ldap_host \
    -- printenv | grep host
mysql_host=mysql1.company.com
ldap_host=ldap1.company.com
```

Although this approach works—in that it decouples configuration from a Pod's manifest—it requires that we query the ConfigMap relevant object each time that we run a Kubernetes object that takes configuration settings. In the next section, we will see how we can make this process more efficient.

Applying Configuration Automatically

Pods' containers can be made aware that their environment variables (and values) are found in an existing ConfigMap so that there is no need to specify the environment variables manually one by one. Specifying the desired ConfigMap in the Pod manifest can be achieved by using the pod.spec.containers.envFrom.configMapRef.name property as shown in the podWithConfigMapReference.yaml manifest:

```
# podWithConfigMapReference.yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  restartPolicy: Never
  containers:
    - name: alpine
      image: alpine
      args: ["sh","-c","printenv | grep host"]
      envFrom:
      - configMapRef:
          name: data-sources
```

To run this Pod manifest, starting from scratch—assuming we run kubectl delete all --all beforehand—we would first set up a ConfigMap by typing kubectl apply -f simpleconfigmap.yaml and then simply run the Pod using kubectl apply -f podWithConfigMapReference.yaml without the need of interacting with the ConfigMap directly:

```
$ kubectl apply -f simpleconfigmap.yaml
configmap/data-sources created
```

```
$ kubectl apply -f podWithConfigMapReference.yaml
pod/my-pod created
```

The results can be observed by checking my-pod's logs:

```
$ kubectl logs pod/my-pod
mysql_host=mysql1.company.com
ldap_host=ldap1.company.com
```

This is perhaps the easiest way to let a Pod's container retrieve configuration data automatically from a ConfigMap, but it has the side effect that it is indiscriminate; it will set up all the environment keys and values declared in the ConfigMap, regardless of whether they are relevant or not.

Let us suppose that there is a special key called `secret_host` whose value is `hushhush.company.com`:

```
# selectiveProperties.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: data-sources
data:
  mysql_host: mysql1.company.com
  ldap_host: ldap1.company.com
  secret_host: hushhush.company.com
```

Now we want to define `my-pod` again in such a way that it retrieves only `mysql_host` and `ldap_host` from the `data-sources` ConfigMap *but not* `secret_host`. In this case, we use the syntax similar to that applicable to hardcoded values; we create an array of items under `pod.spec.containers.env` and also use `name` to name the keys except than rather than hardcoding the value using `value`, and we reference the ConfigMap's applicable key by creating a `valueFrom` object as follows:

```
...
spec:
  containers:
    - name: ...
      env:
        - name: mysql_host
          valueFrom:
            configMapKeyRef:
              name: data-sources
              key: mysql_host
```

Again, note in this code snippet that rather than value, we use valueFrom and that we set up a configMapKeyRef object underneath composed of two attributes: name to reference the ConfigMap and key to reference the desired key.

The complete, final Pod manifest named podManifest.yaml is presented here:

```
# podManifest.yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  restartPolicy: Never
  containers:
    - name: alpine
      image: alpine
      args: ["sh","-c","printenv | grep host"]
      env:
        - name: mysql_host
          valueFrom:
            configMapKeyRef:
              name: data-sources
              key: mysql_host
        - name: ldap_host
          valueFrom:
            configMapKeyRef:
              name: data-sources
              key: ldap_host
```

There are no explicit references to secret_host, and therefore only the mysql_host and ldap_host values will be set. Given that we have defined a new version of the data-sources ConfigMap, we will first clean up the environment by typing kubectl delete all --all, apply the new ConfigMap, and then run the my-pod again:

```
$ kubectl apply -f selectiveProperties.yaml
configmap/data-sources configured

$ kubectl apply -f podManifest.yaml
pod/my-pod created
```

As expected, the secret_host key has not been extracted from the ConfigMap when checking the my-pod's logs:

```
$ kubectl logs pod/my-pod
mysql_host=mysql1.company.com
ldap_host=ldap1.company.com
```

Passing a ConfigMap's Values to a Pod's Startup Arguments

Sometimes, we want to use the ConfigMap data directly as command arguments rather than having containerized applications check environment variables explicitly as we did in the previous sections using printenv. To achieve this, first we have to assign the desired ConfigMap data to environment variables either using the pod.spec.containers.env or pod.spec.containers.envFrom attributes, as covered in previous sections. Once this setup is completed, we can refer to said variables anywhere in a Pod manifest using the $(ENV_VARIABLE_KEY) syntax.

For example, let us say that we wanted to create a Pod whose only purpose is to greet the mysql_host using the echo command. To implement this requirement, we reference the mysql_host variable in the command's argument by referencing the `$(mysql_host)` variable query statement:

```
# podManifestWithArgVariables.yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  restartPolicy: Never
  containers:
    - name: alpine
      image: alpine
      args:
      - /bin/sh
      - -c
      - echo Hello $(mysql_host)
      envFrom:
      - configMapRef:
          name: data-sources
```

Before applying this Pod manifest, we must ensure that we delete any other running Pods—by running `kubectl delete pod --all` but that we leave the ConfigMap named `data-sources` in place:

```
$ kubectl apply -f podManifestWithArgVariables.yaml
pod/my-pod created
```

```
$ kubectl logs my-pod
Hello mysql1.company.com
```

As expected, the variable `$(mysql_host)` was resolved to its value which is `mysql1.company.com`.

Loading a ConfigMap's Properties from a File

Until now, we have seen how to define key/value pairs directly as flags to the `kubectl create configmap` command or within a ConfigMap's manifest. It is often the case that configuration files are stored in an external system such as Git; if this is the case, we do not want to rely on shell scripting sorcery to parse and convert such files. Luckily, we can import files directly into a ConfigMap with the added benefit of being able to express key/value pairs using a simple `<KEY>=<VALUE>` syntax similar to Java's `.properties` and Microsoft's/Python's `.ini` file types. Let us consider the example file called `data-sources.properties`:

```
# data-sources.properties
mysql_host=mysql1.company.com
ldap_host=ldap1.company.com
```

Once this file is saved as `data-sources.properties`, we can reference it by adding the `--from-env=<FILE-NAME>` flag when running the `kubectl create configmap` command. For example:

```
$ kubectl create configmap data-sources \
    --from-env-file=data-sources.properties
configmap/data-sources created
```

Please note that since this is a `create` command as opposed to an `apply` one, we may first need to delete any previously declared ConfigMap objects with the same name by issuing the `kubectl delete configmap/data-sources` command.

Storing Large Files in a ConfigMap

Whether some applications require just a set of simple key/value pairs for configuration, some others may consume large documents often in XML, YAML, or JSON formats. A good example is the XML-based configuration file that the Spring framework uses to define “beans” in Java applications—prior to the emergence of Spring Boot, which relies primarily on annotations rather than on a fat external configuration file.

The ConfigMap service is not limited to storing just simple key/value pairs; a value for a key may be a long text document including line feeds as well. Long text documents can be defined within a regular ConfigMap itself or referenced as an external file.

For example, let us suppose that we need to include an address record in XML alongside our data source configuration. Such a record would be included in the ConfigMap manifest named `configMapLongText.yaml` as follows:

```
# configMapLongText.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: data-sources
data:
  mysql_host: mysql1.company.com
  ldap_host: ldap1.company.com
  address: |
    <address>
       <number>88</number>
       <street>Wood Street</street>
       <city>London</city>
       <postcode>EC2V 7RS</postcode>
    </address>
```

Let us apply the manifest:

```
$ kubectl apply -f configMapLongText.yaml
configmap/data-sources created
```

Now we can use kubectl describe configmap/data-sources to confirm that the XML-based address value has been stored effectively—in addition to mysql_host and ldap_host:

```
$ kubectl describe configmap/data-sources
...
Data
====
address:
----
<address>
   <number>88</number>
   <street>Wood Street</street>
   <city>London</city>
   <postcode>EC2V 7RS</postcode>
</address>
...
```

The other alternative, and more flexible approach, is that of referencing an external file. Let us assume that the address is stored in a file called address.xml:

```
<address>
  <number>88</number>
  <street>Wood Street</street>
  <city>London</city>
  <postcode>EC2V 7RS</postcode>
</address>
```

To reference this file, we just have to add the --from-file=<FILE> flag when using the kubectl create configmap command. For example:

```
$ kubectl create configmap data-sources \
    --from-file=address.xml
configmap "data-sources" created
```

This is equivalent to the previously seen approach except for the fact that the key for the address value has now become the file name itself (address.xml) as we can see when examining the result with the kubectl describe command:

```
$ kubectl describe configmap/data-sources
...
Data
====
address.xml:
----
<address>
  <number>88</number>
  <street>Wood Street</street>
  <city>London</city>
  <postcode>EC2V 7RS</postcode>
</address>
```

It is worth considering that file paths will be converted into keys exclusive of parent folders. For example, /tmp/address.xml and /home/ernie/address.xml will both be converted to a key named address.xml. If both were to be referenced through separate --from-file directives, a key collision will be reported.

Note also that, in the interest of succinctness, we have not applied the literal values for mysql_host and ldap_host. If we wanted an exact imperative equivalent to the declarative form, we should have added a couple of --from-literal flags to include those properties too.

So far, we have learned how to store long text files, but what about retrieving the contents so that they can be used by Pods? Environment variables are not convenient and neither originally intended to store long blocks of text with multiple line feeds. Yet, we can still retrieve multiline text by asking echo to parse line feeds using the -e flag. For example, assuming that we had applied the address XML file seen in the last example, we can retrieve it as follows:

```
# Assuming we are inside a Pod's container
$ echo -e $address > /tmp/address.xml
$ cat /tmp/address.xml
<address>
   <number>88</number>
   <street>Wood Street</street>
   <city>London</city>
   <postcode>EC2V 7RS</postcode>
</address>
```

Even though this trick allows us the retrieval of a relatively simple XML document, polluting a Pod's set of environment variables with multiline text is undesirable, and it also has a major limitation: the data is immutable and cannot longer be changed once the Pod has been created. In the next section, we will explore a more convenient approach to getting long text documents into a Pod's container.

Live ConfigMap Updates

ConfigMaps are typically defined in declarative form. The difference between `kubectl create` and `kubectl apply` is that the latter refreshes (overrides) the state of existing matching instances—if any. Whenever we apply a new ConfigMap using `kubectl apply -f <FILE>,` we effectively update any matching ConfigMap instance and its configuration key/value pairs, but *this does not mean that the Pods that are bound to the refreshed ConfigMaps get updated*. This is because the method we have seen so far, to propagate configuration data, is via environment variables that are only set once—when the Pod's container is created.

Making ConfigMap data available to Pods through environment variables (and/or command arguments) has two limitations. The first is that it is rather inconvenient for long, multiple-line text. The second, and most fundamental one, is that environment variables, once set, cannot be updated anymore unless the Pod is restarted in order to restart the underlying Linux process as well. Kubernetes has a solution that solves these two problems in one stroke; it can make ConfigMap properties available as files inside the Pod's container so that keys appear as file names and values as their contents, and, on top of this, it will refresh said files whenever the underlying ConfigMap is updated. The ConfigMap object does truly allow us to have the cake and eat it too.

Let us have a look at how this "configuration as files" solution works. First of all, let us consider again the manifest for our `data-sources` ConfigMap which includes a multiple-line property called address:

```
# configMapLongText.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: data-sources
data:
  mysql_host: mysql1.company.com
  ldap_host: ldap1.company.com
  address: |
    <address>
       <number>88</number>
       <street>Wood Street</street>
       <city>London</city>
       <postcode>EC2V 7RS</postcode>
    </address>
```

Making the data-sources ConfigMap available to a Pod as a file system involves two steps. First, we have to define a volume name under pod.spec.volumes in the Pod manifest. Under this attribute, we specify the volume's name, in our case, my-volume, and the ConfigMap's name to which said volume will be linked, data-sources:

```
...
volumes:
  - name: my-volume
    configMap:
      name: data-sources
```

The second step is *mounting the volume* by referencing it using the name we had chosen (my-volume) under pod.spec.containers.volumeMounts. We also have to specify the path under which we want the volume to be mounted:

```
...
volumeMounts:
  - name: my-volume
    mountPath: /var/config
...
```

Before we combine the volume definition and volume mount into the final Pod manifest, we also want to include a script to inspect the resulting directory and file structure by issuing a ls -l /var/config command. We also want to view the contents of a specific key, address, by issuing a cat /var/config/address command.

We also said that files are refreshed automatically whenever the underlying ConfigMap gets updated; we can observe this behavior by monitoring /var/config/address for changes by using the inotifywait command. The resulting script is as follows:

```
apk update;
apk add inotify-tools;
ls -l /var/config;
while true;
do cat /var/config/address;
   inotifywait -q -e modify /var/config/address;
done
```

The script works in the following manner: the first two apk commands install inotifywait which is part of the inotify-tools package, then it displays the files found in /var/config, and, finally, it enters an infinite loop in which it displays the contents of /var/config/address when the file is modified.

The resulting Pod manifest, called podManifestVolume.yaml, including the provided volume and volumeMount declarations plus the scripts seen earlier, is presented here:

```
# podManifestVolume.yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  restartPolicy: Never
  containers:
    - name: alpine
      image: alpine
      args:
        - sh
        - -c
        - >
          apk update;
          apk add inotify-tools;
          ls -l /var/config;
          while true;
          do cat /var/config/address;
          inotifywait -q
          -e modify /var/config/address;
          done
      volumeMounts:
        - name: my-volume
          mountPath: /var/config
  volumes:
    - name: my-volume
      configMap:
        name: data-sources
```

We will now apply configMapLongText.yaml (the ConfigMap) and podManifestVolume.yaml (the manifest defined earlier):

```
$ kubectl apply -f configMapLongText.yaml -f podManifestVolume.
yaml
configmap/data-sources created
pod/my-pod created
```

The results of ls -la /var/config and cat /var/config/address are shown by checking my-pod's logs:

```
$ kubectl logs -f pod/my-pod
...
lrwxrwxrwx 1 14 Jul 8 address -> ..data/address
lrwxrwxrwx 1 16 Jul 8 ldap_host -> ..data/ldap_host
lrwxrwxrwx 1 17 Jul 8 mysql_host -> ..data/mysql_host
<address>
```

```
   <number>88</number>
   <street>Wood Street</street>
   <city>London</city>
   <postcode>EC2V 7RS</postcode>
</address>
```

Let us examine the resulting output. The `ls -l /var/config` command shows that every ConfigMap key (`address`, `ldap_host`, `mysql_host`) is represented as a file. The second command, `cat /var/config/address`, shows that the value of every key has now become the files' content; in this case, `address` contains an XML file.

We can now observe how the "configuration as files" feature is useful to propagate configuration changes. First, we will define a new version of `data-sources`, named `configMapLongText_changed.yaml`, which includes a changed value for the `address` key:

```
# configMapLongText_changed.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: data-sources
data:
  mysql_host: mysql1.company.com
  ldap_host: ldap1.company.com
  address: |
    <address>
       <number>33</number>
       <street>Canada Square</street>
       <city>London</city>
       <postcode>E14 5LB</postcode>
    </address>
```

Before applying this manifest, we must ensure that we leave the window in which we had launched the `kubectl logs -f pod/my-pod` running in parallel and that we write the following command in a new one:

```
$ kubectl apply -f configMapLongText_changed.yaml
configmap/data-sources configured
```

After a few seconds, we will notice that the window where `kubectl logs -f pod/my-pod` is running shows the new address declared in `configMapLongText_changed.yaml`:

```
...
<address>
   <number>33</number>
   <street>Canada Square</street>
   <city>London</city>
   <postcode>E14 5LB</postcode>
</address>
```

As we have seen in this section, serving configuration as files has the advantage of allowing the inclusion of long text files and making it possible for a running application to detect new configuration changes. This does not mean that the use of environment variables is an inferior solution. Even in the case where configuration details are volatile, using environment variables can still be a good idea combined with a canary testing approach where only a subset of Pods—new ones—get the new changes as the old ones are progressively retired.

Storing Binary Data

The ConfigMap object is designed to store text data since we normally retrieve its content using a text-friendly interface like environment variables in the Linux shell and JSON and YAML output when using the `kubectl get configmap/<NAME>` command together with the `-o json` and `-o yaml` flags, respectively.

Since a key's value can appear as a file's content when using volume, it seems like a practical approach to also store BLOB (Binary Large OBject) data, but the intuition here is misleading for the reasons what have given earlier. The solution to this problem is to encode and decode whatever binary files we are interested using an ASCII encoding mechanism like base64. For example, supposing that we wanted to store the contents of an image called `logo.png` on a ConfigMap called `binary`, we would issue the following two commands:

```
$ base64 logo.png > /tmp/logo.base64
$ kubectl create configmap binary \
    --from-file=/tmp/logo.base64
```

Then, from within a Pod, assuming that the `binary` ConfigMap is mounted under `/var/config`, we would obtain the original image back as follows:

```
$ base64 -d /var/config/logo.base64 > /tmp/logo.png
```

Naturally, inside a programming language such as Python or Java, we would rather use a native library rather than a shell command as shown in this example. Note also that whereas base64 provides some level of obfuscation, it is not a form of encryption. We will discuss this topic further in the next section.

Secrets

The ConfigMap object is intended for clear-text, nonsensitive data that often originates in a centralized SCM. For passwords, and other sensitive data, the Secret object should be used instead. The Secret object is, in most cases, a "drop-in" replacement for the ConfigMap—when running in generic mode—for all imperative and declarative use cases except for the fact that clear-text data is meant to be coded in base64 and is automatically decoded when made available through environment variables and volumes.

The security capabilities associated with the Secret object are being constantly improved. At the time of writing, encryption for secrets at rest is supported (https://kubernetes.io/docs/tasks/administer-cluster/encrypt-data/), and design measures are taken to prevent Pods from accessing secrets that are not meant to be shared with them. Having said this, the level of security provided by the Secret object should not be considered appropriate to safeguard credentials from Kubernetes cluster administrators with root access.

We will now look at the key differences between how to store sensitive information using the Secret object as opposed to the ConfigMap one.

Difference Between ConfigMap and Secret Objects

The command `kubectl create secret generic <NAME>` is analogous to the `kubectl create configmap <NAME>` one. It takes three flags just like its ConfigMap counterpart: `--from-literal` for in-place values, `--from-env-file` for files containing multiple key/value pairs, and `--from-file` for large data files.

We will now contemplate each of the aforementioned use cases, with the intention of storing the `mysql_user=ernie` and `mysql_pass=HushHush` credentials. Please note that all three of the presented versions are equivalent and use the same name, so we must run `kubectl delete secrets/my-secrets` if we get an error such as `Error from server (AlreadyExists)` when running all examples straight one after another:

Use Case 1: `--from-literal` values:

```
$ kubectl create secret generic my-secrets \
    --from-literal=mysql_user=ernie \
    --from-literal=mysql_pass=HushHush
secret/my-secrets created
```

Use Case 2: `--from-env` and a file called `mysql.properties`:

```
# secrets/mysql.properties
mysql_user=ernie
mysql_pass=HushHush
$ kubectl create secret generic my-secrets \
    --from-env-file=secrets/mysql.properties
secret/my-secrets created
```

Use Case 3: `--from-file`:

```
$ echo -n ernie > mysql_user
$ echo -n HushHush > mysql_pass
$ kubectl create secret generic my-secrets \
    --from-file=mysql_user --from-file=mysql_pass
secret/my-secrets created
```

Using a declarative manifest, we first need to encode our values manually to base64 as follows:

```
$ echo -n ernie | base64
ZXJuaWU=
$ echo -n HushHush | base64
SHVzaEh1c2g=
```

Then, we can use these base64 encoded values within a manifest:

```
# secrets/secretManifest.yaml
apiVersion: v1
kind: Secret
```

```
metadata:
  name: my-secrets
data:
  mysql_user: ZXJuaWU=
  mysql_pass: SHVzaEh1c2g=
```

As usual, when working with a declarative manifest rather than the imperative form, we simply apply the manifest file using `kubectl apply -f <FILE>` command:

```
$ kubectl apply -f secrets/secretManifest.yaml
secret/my-secrets unchanged
```

In total, we have seen four different ways of defining the same set of credentials; the first three using the `--from-literal`, `--from-env-file`, and `--from-file` flags applicable to the `kubectl create secret generic` command and the last using a manifest file. In all cases, the resulting object named `my-secrets` is the same—barring metadata information and some other minor details:

```
$ kubectl get secret/my-secrets -o yaml
apiVersion: v1
data:
  mysql_pass: SHVzaEh1c2g=
  mysql_user: ZXJuaWU=
kind: Secret
...
```

The `kubectl describe` command is also helpful, but it will not reveal the base64 values; only their length:

```
$ kubectl describe secret/my-secrets
...
Data
====
mysql_user:  5 bytes
mysql_pass:  8 bytes
```

Reading Properties from Secrets

Secret properties are made available in Pod manifests either as environment variables or as volume mounts in a similar way as ConfigMap. In most cases, the overall syntax remains the same, except for the fact that we use the keyword `secret` where `configMap` would have applied.

In the following examples, we assume that the `my-secrets` Secret has been already created and that it contains the `mysql_user` and `mysql_pass` keys and values.

Let us begin with the `envForm` approach which is the simplest approach to extract secrets since it simply projects all key/value pairs declared in the secret object as environment variables using the `pod.spec.containers.envFrom` declaration. This is achieved exactly as we do in the case of a ConfigMap except that we have to replace `configMapRef` with `secretRef`:

```
# secrets/podManifestFromEnv.yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  restartPolicy: Never
  containers:
    - name: alpine
      image: alpine
      args: ["sh","-c","printenv | grep mysql"]
      envFrom:
      - secretRef:
          name: my-secrets
```

Applying the manifest and then checking the Pod's logs reveal the secret's values. This demonstrates that Kubernetes decodes the value back back from base64 to plain text before setting up the environment variables inside the container runtime:

```
$ kubectl apply -f secrets/podManifestFromEnv.yaml
pod/my-pod created
```

```
$ kubectl logs pod/my-pod
mysql_user=ernie
mysql_pass=HushHush
```

The other, slightly more ardous—but safer—approach is specifying environment variables one by one and selecting specific properties. This is exactly the same approach as the ConfigMap counterpart except that we replace `configMapKeyRef` with `secretKeyRef`:

```
# secrets/podManifesSelectedEnvs.yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  restartPolicy: Never
  containers:
    - name: alpine
      image: alpine
      args: ["sh","-c","printenv | grep mysql"]
      env:
        - name: mysql_user
          valueFrom:
            secretKeyRef:
              name: my-secrets
              key: mysql_user
        - name: mysql_pass
          valueFrom:
            secretKeyRef:
              name: my-secrets
              key: mysql_pass
```

The result of applying this manifest is exactly the same as the previous example:

```
$ kubectl apply -f \
    secrets/podManifestSelectedEnvs.yaml
pod/my-pod created
```

```
$ kubectl logs pod/my-pod
mysql_user=ernie
mysql_pass=HushHush
```

Let us now turn our attention to volumes. Again, the workflow is the same as in the case of ConfigMap objects. We first have to declare a volume under `pod.spec.volumes` and then mount it under a given containers at `pod.spec.containers.volumeMounts`. Only the first part—the definition of the volume—is different in respect to ConfigMap objects. Here there are two changes: `configMap` must be replaced with `secret` and `name` with `secretName`:

```
# secrets/podManifestVolume.yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
```

```
spec:
  restartPolicy: Never
  containers:
    - name: alpine
      image: alpine
      args: ["sh","-c","ls -l /var/config"]
      volumeMounts:
        - name: my-volume
          mountPath: /var/config
  volumes:
    - name: my-volume
      secret: # rather than configMap
        secretName: my-secrets # rather than name
```

Once the manifest is applied, the secret properties can be found as files under /var/config. We have kept the same mount point as in the previous ConfigMap example since we wanted to highlight the similarities.

We have opted for a simple script this time which simply lists the contents of the /var/config directory upon running the Pod:

```
$ kubectl apply -f secrets/podManifestVolume.yaml
pod/my-pod created
```

```
$ kubectl logs pod/my-pod
total 0
lrwxrwxrwx 1 17 Jul 8 mysql_pass -> ..data/mysql_pass
lrwxrwxrwx 1 17 Jul 8 mysql_user -> ..data/mysql_user
```

The property of change propagation is still present in Secrets and works in the same manner as in ConfigMaps. We do not reproduce the example here for succinctness.

Docker Registry Credentials

So far, we have seen the use of the Secret object using its so called *generic* mode. The handling of Docker Registry credentials is one of its purposely designed extensions that helps pulling Docker images without the need—and lack of safety—of specifying credentials explicitly in a Pod manifest.

In the examples found in this text, such as those involving Alpine or Nginx images, we have always been dealing with public Docker registries (such as Docker Hub), so there is no need to be concerned with credentials. However, whenever a private Docker repository is required, we need to supply the correct username, password, and e-mail address before we can pull an image.

The process is fairly simple since it only involves creating a secret object that holds the Docker Registry's credentials and we just need to reference said object in the applicable Pod manifest.

A Docker Registry secret is created using the `kubectl create docker-registry <NAME>` command along with specific flags for each credential component:

- `--docker-server=<HOST>`: The server's host
- `--docker-username=<USER_ID>`: The username
- `--docker-password=<USER_PASS>`: The password, unencrypted
- `--docker-email=<USER_EMAIL>`: The user's e-mail address

The following is an example of creating a secret for a Docker Hub's private repository named `docker-hub-secret`:

```
$ kubectl create secret docker-registry docker-hub-secret \
    --docker-server=docker.io \
    --docker-username=egarbarino \
    --docker-password=HushHush \
    --docker-email=antispam@garba.org
secret/docker-hub-secret created
```

Now, we can refer to docker-hub-secret from within a Pod manifest under pod.spec.imagePullSecrets.name. In the next example, we are referring to an image stored in Docker Hub located at docker.io/egarbarino/hello-image:

```
# secrets-docker/podFromPrivate.yaml
apiVersion: v1
kind: Pod
metadata:
  name: hello-app
spec:
  containers:
    - name: hello-app
      image: docker.io/egarbarino/hello-image
  imagePullSecrets:
    - name: docker-hub-secret
$ kubectl apply -f secrets-docker/podFromPrivate.yaml
pod/hello-app created
```

The hello-app Docker image is a Flask (Python) application that listens on port 80. If the Docker credentials are successful, we should be able to connect to it by establishing a tunnel between our localhost and the hello-app Pod:

```
$ kubectl port-forward pod/hello-app 8888:80 \
    > /dev/null &
[1] 5052
```

```
$ curl http://localhost:8888
Hello World | Host: hello-app | Hits: 1
```

The reader should supply their own Docker Hub credentials—the author's password is obviously not `HushHush`. It is also worth noting that the Docker Registry credentials are stored using simply base64 encoding which is not a form of encryption, merely *obfuscation*. The credentials can be retrieved by enquiring the secret object and decoding the result:

```
$ kubectl get secret/docker-hub-secret \
    -o jsonpath \
    --template="{.data.\.dockerconfigjson}" \
    | base64 -d
{
   "auths":{
      "docker.io":{
         "username":"egarbarino",
         "password":"HushHush",
         "email":"antispam@garba.org",
         "auth":"ZWdhcmJhcmlubzpUZXNOaW5nJDEyMw=="
      }
   }
}
```

TLS Public Key Pair

In addition to generic (user-defined) secrets and Docker Registry credentials, the Secret object has also a special provision to store TLS public/key pairs so that they can be referenced by objects such as *Ingress* (`https://kubernetes.io/docs/concepts/services-networking/ingress/`), a layer 7 (http/https) proxy. Please note that the Ingress controller is still in beta at the time of writing and it is not covered in this text.

A public/private key pair is stored using the `kubectl create secret tls <NAME>` command and the following two flags:

- `--cert=<FILE>`: PEM-encoded public key certificate. It typically has a `.crt` extension.
- `--key=<FILE>`: Private key. It typically has a `.key` extension.

Assuming we have the files `tls.crt` and `tls.key` sitting in the `secrets-tls` directory, the following command will store them in the Secret object:

```
$ kubectl create secret tls my-tls-secret \
    --cert=secrets-tls/tls.crt \
    --key=secrets-tls/tls.key
secret/my-tls-secret created
```

The resulting object is no different from a generic or docker-registry Secret. The files are encoded using base64 and can easily be retrieved and decoded by querying the resulting Secret object. In the next example, we retrieve the contents, decode them, and compare them to the originals; the `diff` command emits no output which means that both files are identical:

```
$ kubectl get secret/my-tls-secret \
    --output="jsonpath={.data.tls\.crt}" \
    | base64 -d > /tmp/recovered.crt
$ kubectl get secret/my-tls-secret \
    --output="jsonpath={.data.tls\.key}" \
    | base64 -d > /tmp/recovered.key

$ diff secrets-tls/tls.crt /tmp/recovered.crt
$ diff secrets-tls/tls.key /tmp/recovered.key
```

Management Recap

Both the regular ConfigMap and Secret objects respond to the typical `kubectl get <RESOURCE-TYPE>/<OBJECT-IDENTIFIER>` and `kubectl delete <RESOURCE-TYPE>/<OBJECT-IDENTIFIER>` commands, for listing and deleting, respectively.

This is an example of listing existing ConfigMap objects and deleting the one named `data-sources`:

```
$ kubectl get configmap
NAME                    DATA      AGE
data-sources            1         58s
language-translations   1         34s

$ kubectl delete configmap/data-sources
configmap "data-sources" deleted
```

Similarly, this is an example of listing existing Secret objects and deleting the one named `my-secrets`:

```
$ kubectl get secret
NAME              TYPE                              DATA
*-token-c4bdr     kubernetes.io/service-*-token     3
docker-hub-secret kubernetes.io/dockerconfigjson    1
my-secrets        Opaque                            2
my-tls-secret     kubernetes.io/tls                 2

$ kubectl delete secret/my-secrets
secret "my-secrets" deleted
```

Summary

This chapter showed how to externalize configuration—so that it is not hard-coded into applications—by setting it manually using environment variables flags and automatically via the ConfigMap and Secret objects.

We learned that the ConfigMap and Secret objects are similar. They both help populating configuration properties (using flags, manifest files, and external files containing key/value pairs) as well as injecting said properties into Pods' containers (projecting all data as environment variables, selecting specific ones, making variables available as a virtual file system). We also explored how to deal with long files in text and binary form and how to produce live configuration updates.

Finally, we saw that the Secrets object has also the special ability of storing Docker Registry credentials that are useful for pulling Docker images from private repositories and for storing TLS keys that can be ingested by TLS-capable objects such as the Ingress controller.

CHAPTER 6

Jobs

Batch processing differs from persistent applications—such as web servers—in that programs are expected to complete and terminate once they achieve their goal. Typical examples of batch processing include database migration scripts, video encoding jobs, and Extract-Load-Transform (ETL) runbooks. Similarly to the ReplicaSet controller, Kubernetes has a dedicated *Job* controller that manages Pods for the purpose of running batch-oriented workloads.

Instrumenting batch processes using Jobs is relatively straightforward as we will soon learn in this short chapter. In the first section, we will learn the concepts of *completions* and *parallelism* which are the two fundamental variables that determine the dynamics of a Job. Then, we will explore the three fundamental types of batch processes that may be implemented using the Job controller: *single batch processes*, *completion count–based batch processes*, and *externally coordinated batch processes.* Finally, toward the end, we will look at the typical management tasks associated with Job handling: determining when a Job has completed, configuring jobs with appropriate time-out thresholds, and deleting them without disposing of their results.

E. Garbarino, *Beginning Kubernetes on the Google Cloud Platform*,
https://doi.org/10.1007/978-1-4842-5491-2_6

Completions and Parallelism

Kubernetes relies on the standard Unix exit code to tell whether a Pod has finished its job successfully or not. If a Pod's container entry process finishes by returning the exit code `0`, the Pod is assumed to have *completed* its goal successfully. Otherwise, if the code is nonzero, the Pod is assumed to have failed.

The Job controller uses two key concepts to orchestrate a Job: *completions* and *parallelism*. Completions, specified using the `job.spec.completions` attribute, determine the number of times that a Job must be run and exit with a success exit code—in other words, `0`. Parallelism, specified using the `job.spec.parallelism` attribute, instead, sets the number of Jobs that may be run in parallel—in other words, concurrently.

The combination of values for these two attributes will determine the number of *unique Pods* that will be created to achieve a given number of completions as well as the number of *parallel Pods* (Pods running concurrently) that may be spun up. Table 6-1 provides the results for a set of sample combinations.

Table 6-1. *The effects of various completions and parallelism permutations*

`completions`	`parallelism`	**Unique Pods**	**Parallel Pods**
Unset	Unset	1	1
1	Unset	1	1
1	1	1	1
3	1	3	1
3	3	3	3
1	3	1	1
Unset	3	3	3

There is no need to spend much effort in working out the combinations presented on Table 6-1 for now. We will learn the appropriate use of `completions` and `parallelism` in the context of practical use cases in the following sections.

Batch Process Types Overview

Batch processes may be categorized into three types: single batch processes, completion count–based batch processes, and externally coordinated batch processes:

- **Single Batch Process:** Running a Pod successfully just once is sufficient to finish the job.
- **Completion Count–Based Batch Processes:** An *n* number of Pods must complete successfully to finish the job—in parallel, in sequence, or a combination thereof.
- **Externally Coordinated Batch Processes:** A pool of worker Pods works on a centrally coordinated job. The number of required completions is not known in advance.

As explained in the previous section, the definition of *successful* is that a Pod's container's process terminates producing an exit status code of value `0`.

Single Batch Processes

A single batch process is implemented using a Pod that is scheduled to run successfully just once. Such a Job may be run both in an imperative and declarative manner. A single Job is created using the `kubectl create job`

<NAME> --image=<IMAGE> command. Let us consider a Bash script that computes the two times table:

```
for i in $(seq 10)
  do echo $(($i*2))
done
```

We can run this script as a Job, in an imperative fashion, or by passing it as the argument to the alpine's sh command:

```
$ kubectl create job two-times --image=alpine \
    -- sh -c \
    "for i in \$(seq 10);do echo \$((\$i*2));done"
job.batch/two-times created
```

Note Jobs may also be creating using the traditional kubectl run <NAME> --image=<IMAGE> --restart=OnFailure command. This traditional form is now deprecated; when creating Jobs using kubectl run, adding the flag --restart=OnFailure is essential; if omitted, a Deployment will be created instead.

The results can be obtained by running the kubectl logs -l job-name=<NAME> command which will match the Pods (only one in this case) with the job name's label. The Job controller will add the value job-name and set it to its name so that it is easy to identify the parent-child relationship between Jobs and Pods:

```
$ kubectl logs -l job-name=two-times
2
4
6
```

```
8
10
12
14
16
18
20
```

We have just run our first Job. As simple as this sequence of even numbers may seem, it should be understood that the workload instrumented as a Job could be as simple as this example but also as complex as code running queries against a SQL database or training a machine learning algorithm.

One aspect to keep in mind is that single batch processes are not "fire and forget"; they leave resources lurking around and must be cleaned up manually as we will see further on. In fact, the same Job name cannot be used twice unless we delete the first instance first.

Note We may need to add the `--tail=-1` flag to display all results whenever it seems that we are missing rows. When using a label selector, as in the case of `two-times.yaml`, the number of lines is restricted to 10.

Listing the available Jobs using the `kubectl get jobs` command will reveal, when considering the value under the `COMPLETIONS` column, `1/1`, that one completion was expected, and one was actually archived:

```
$ kubectl get jobs
NAME        COMPLETIONS   DURATION   AGE
two-times   1/1           3s         9m6sm
```

When we type kubectl get pods, the Job-controlled Pods can be identified by the job's prefix, two-times in this case:

```
$ kubectl get pods
NAME             READY   STATUS      RESTARTS   AGE
two-times-4lrp8  0/1     Completed   0          8m45s
```

The declarative equivalent for this job is presented next, and it is run by using the kubectl apply -f two-times.yaml command:

```
# two-times.yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: two-times
spec:
  template:
    spec:
      containers:
      - command:
        - /bin/sh
        - -c
        - for i in $(seq 10);do echo $(($i*2));done
        image: alpine
        name: two-times
      restartPolicy: Never
```

Note that the job.spec.completions and job.spec.paralellism attributes are absent; they both will be given a default value of 1.

Also bear in mind that neither the Job controller nor its completed Pod will be removed unless we explicitly delete it using the kubectl delete job/<NAME> command. This is important to keep in mind if we want to run this manifest right after trying the imperative form with the same job name.

The two-times.yaml Job is an example of a simple script that is guaranteed to succeed. What if the Job where to fail? For example, consider the following script which unequivocally prints the hostname, and the date, which exists with 1, a nonsuccess exit code:

```
echo $HOSTNAME failed on $(date) ; exit 1
```

The way in which the Job controller would react to running this script will depend primarily on two aspects:

- The job.spec.backoffLimit attribute (which is set to 6 by default) will determine how many times the Job controller will try to run the Pod before it gives up.
- The job.spec.template.spec.restartPolicy attribute, which could be either Never or OnFailure, will determine whether new Pods will be spun up, or not, for every retry. If set to the former, the Job controller will spin up a new Pod for every attempt while not disposing of failed Pods. If set to the latter, instead, the Job controller will restart the same Pod until it succeeds; however, because failed Pods are reused—by *restarting* them—the output they had produced as a result of a failure is lost.

Deciding between a restartPolicy value of Never and OnFailure depends on what compromise is more acceptable to us. Never is usually the most sensible option since it does not dispose of failed Pods' output and it allows us to troubleshoot what went wrong with them; however; leaving failed Pods around takes up more resources. An industrial Job solution should ideally save valuable data to a persistent storage medium—for example, an attached volume as demonstrated in Chapter 2.

Let us now look at each use case to get more intuition into the side effects of each of them. The unlucky-never.yaml manifest presented next sets backoffLimit attribute to 3 and the restartPolicy attribute to Never:

```
# unlucky-never.yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: unlucky
spec:
  backoffLimit: 3
  template:
    spec:
      restartPolicy: Never
      containers:
      - command:
        - /bin/sh
        - -c
        - echo $HOSTNAME failed on $(date) ; exit 1
        name: unlucky
        image: alpine
```

We will run unlucky-never.yaml by issuing the kubectl apply -f unlucky-never.yaml command, but first, we will open a separate window in which we will run the kubectl get pods -w to see what Pods the Job controller creates, as it reacts to the failures produced by exit 1:

```
$ kubectl get pods -w
unlucky-6qm98 0/1   Pending             0        0s
unlucky-6qm98 0/1   ContainerCreating   0        0s
unlucky-6qm98 0/1   Error               0        1s
unlucky-sxj97 0/1   Pending             0        0s
unlucky-sxj97 0/1   ContainerCreating   0        0s
```

```
unlucky-sxj97 1/1   Running             0        0s
unlucky-sxj97 0/1   Error               0        1s
unlucky-f5h9c 0/1   Pending             0        0s
unlucky-f5h9c 0/1   ContainerCreating   0        0s
unlucky-f5h9c 0/1   Error               0        0s
```

As it can appreciated, three new Pods were created: `unlucky-6qm98`, `unlucky-sxj97`, and `unlucky-f5h9c`. All have ended with an error, but why? Let us check their logs:

```
$ kubectl logs -l job-name=unlucky
unlucky-6qm98 failed on Sun Jul 14 15:45:18 UTC 2019
unlucky-f5h9c failed on Sun Jul 14 15:45:30 UTC 2019
unlucky-sxj97 failed on Sun Jul 14 15:45:20 UTC 2019
```

This is the advantage of setting the `restartPolicy` attribute to `Never`. As seen earlier, the logs for failed Pods are preserved, which allow us to diagnose the nature of the error. One last useful check is via the `kubectl describe job/<NAME>` command. Next, we see that no Pods are currently running, that zero have succeeded, and that three have failed:

```
$ kubectl describe job/unlucky | grep Statuses
Pods Statuses:  0 Running / 0 Succeeded / 3 Failed
```

Setting the `restartPolicy` attribute to `OnFailure` results in a rather different behavior. Let us go through the same drill again but using a new manifest, named `unlucky-onFailure.yaml`, in which the only change is the aforementioned attribute:

```
# unlucky-onFailure.yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: unlucky
```

```
spec:
  backoffLimit: 3
  template:
    spec:
      restartPolicy: OnFailure
      containers:
      - command:
        - /bin/sh
        - -c
        - echo $HOSTNAME failed on $(date) ; exit 1
        name: unlucky
        image: alpine
```

Before applying the manifest by issuing the `kubectl apply -f unlucky-onFailure.yaml` command, we will follow, in a separate terminal window, the results of `kubectl get pods -w`, as we did before.

```
$ kubectl get pods -w
NAME          READY STATUS            RESTARTS AGE
unlucky-fgtq4 0/1   Pending           0        0s
unlucky-fgtq4 0/1   ContainerCreating 0        0s
unlucky-fgtq4 0/1   Error             0        0s
unlucky-fgtq4 0/1   Error             1        1s
unlucky-fgtq4 0/1   CrashLoopBackOff  1        2s
unlucky-fgtq4 0/1   Error             2        17s
unlucky-fgtq4 0/1   CrashLoopBackOff  2        30s
unlucky-fgtq4 1/1   Running           3        41s
unlucky-fgtq4 1/1   Terminating       3        41s
unlucky-fgtq4 0/1   Terminating       3        42s
```

In contrast to `unlucky-never.yaml` whose `restartPolicy` was `Never`, we see two key differences here: there is only one Pod, `unlucky-fgtq4`, which is restarted three times, as opposed to three different Pods, and the

Pod is terminated at the end as opposed to ending in an `Error` status. The fundamental side effect is that the logs are deleted and we are, therefore, unable to diagnose the problem:

```
$ kubectl logs -l job-name=unlucky
# nothing
```

Another worthwhile difference is that `kubectl describe job/unlucky` command will claim that only one Pod has failed. This is true; only one Pod has failed indeed—although it was restarted three times, as per the `backoffLimit` setting:

```
$ kubectl describe job/unlucky | grep Statuses
Pods Statuses:  0 Running / 0 Succeeded / 1 Failed
```

Completion Count–Based Batch Process

The concept of running a Pod once, two, or more times—and ending with a success exit code—is known as a *completion*. In the case of a single batch process, as demonstrated in the previous section, the value of `job.spec.completions` is `1` by default; instead, a completion count–based batch process generally has the `completions` attribute set to a value greater or equal than two.

In this use case, Pods run *independently* with no awareness of one another. In other words, each Pod runs in isolation in respect to the results and outcomes of other Pods. How the code running within Pods decide which data to work on and how to avoid duplicate work—as well as where to save the results—are all implementation concerns that escape the Job's control capabilities. Typical solutions to implementing a shared state are an external queue, database, or a shared file system. Since processes are independent, they may be run in parallel; the number of processes that can run in parallel is specified using the `job.spec.parallelism` property.

With a combination of job.spec.completions and job.spec.parallelism, we can control how many times a process must be run and how many of them will be run in parallel to accelerate the overarching batch processing time.

To internalize the instrumentation of multiple independence processes, we need an example of a process that is independent from other instances, but that, at the same time, collects—in principle—different results. For this purpose, we have designed a Bash script that checks the current time and reports a success if the current second is even but a failure if it is odd:

```
n=$(date +%S)
if [ $((n%2)) -eq 0 ]
then
  echo SUCCESS: $n
  exit 0
else
  echo FAILURE: $n
  exit 1
fi
```

As an overarching goal, we set out to collect six samples of seconds that are even; this means that we set job.spec.completions to 6. We also want to speed up the process by running two Pods in parallel, so we set job.spec.parallelism to 2. The end result is the following manifest, named even-seconds.yaml:

```
# even-seconds.yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: even-seconds
```

```
spec:
  completions: 6
  parallelism: 2
  template:
    spec:
      restartPolicy: Never
      containers:
     - command:
       - /bin/sh
       - -c
       - >
         n=$(date +%S);
         if [ $((n%2)) -eq 0 ];
         then
           echo SUCCESS: $n;
           exit 0;
         else
           echo FAILURE: $n;
           exit 1;
         fi
        name: even-seconds
        image: alpine
```

To appreciate the dynamics of this example, it is convenient to set up three terminal windows (or tabs/panels) as follows:

- **Terminal Window 1**: To observe Job activity by running `kubectl get job -w`
- **Terminal Window 2**: To observe Pod activity by running `kubectl get pods -w`
- **Terminal Window 3**: To run `even-seconds.yaml` by issuing the `apply -f even-seconds.yaml` command.

An interesting summary is obtained by running the kubectl describe job/even-seconds command within the row prefixed with Pod Statuses which is, in turn, obtained from the values under job.status—populated at runtime.

Let us apply even-seconds.yaml and observe the Job's behavior in terms of its orchestrated Pods:

```
# Keep this running before running `kubectl apply`
$ while true; do kubectl describe job/even-seconds \
    | grep Statuses ; sleep 1 ; done
Pods Statuses:  2 Running / 0 Succeeded / 0 Failed
Pods Statuses:  2 Running / 0 Succeeded / 1 Failed
Pods Statuses:  2 Running / 0 Succeeded / 2 Failed
Pods Statuses:  2 Running / 1 Succeeded / 3 Failed
Pods Statuses:  2 Running / 3 Succeeded / 3 Failed
Pods Statuses:  1 Running / 5 Succeeded / 3 Failed
Pods Statuses:  0 Running / 6 Succeeded / 3 Failed
```

The resulting output is not deterministic; the number of failures and restarts, as well as the actual collected results, will depend on a myriad of factors: the container's startup time, the current time, the CPU speed, etc.

Having stated that the reader may obtain different results, let us analyze the output. In the beginning, the Job controller spins up two Pods, none of which have neither succeeded nor failed. Then, in the second row, one Pod fails, but since parallelism is set to 2, another Pod is spun up so that there are always two running in parallel. Halfway through the process, the Job controller starts registering progressively more succeeded Pods. In the antepenultimate row, only one Pod is running since five have already succeeded and there is only one left to go. Finally, in the last row, the last Pod succeeds which completes our intended number of completions: six.

We skip the output from `kubectl get pods -w` and leave it as an exercise for the reader (it will align with the earlier output showing multiple Pods in different states: `ContainerCreating`, `Error`, `Completed`, etc.). It is interesting to find out whether our overarching goal of obtaining six samples of even seconds has been achieved. Let us check the logs and see:

```
$ kubectl logs -l job-name=even-seconds \
    | grep SUCCESS
SUCCESS: 34
SUCCESS: 32
SUCCESS: 34
SUCCESS: 30
SUCCESS: 32
SUCCESS: 36
```

This is exactly what we had set out to achieve. What about the failed Pods? Had we set the `restartPolicy` attribute to `OnFailure,` it would have been tricky to find out, but since we have purposely chosen `Never`, we can retrieve the output from failed Jobs and confirm that the failures were due to the sampling of odd seconds:

```
$ kubectl logs -l job-name=even-seconds \
    | grep FAILURE
FAILURE: 27
FAILURE: 27
FAILURE: 29
```

Before moving into the third use case, one aspect worthy of attention, when setting both the `job.spec.completions` and `job.spec.paralellism` attributes, is that the Job controller will never instantiate more parallel Pods than the number of expected (and/or remaining) completions. For example, if we define `completions: 2` and `parallelism: 5`, it is as though we had set `parallelism` to `2`. Likewise, the number of parallel running Pods will never be greater than the number of pending completions.

Externally Coordinated Batch Process

A batch process is said to be externally coordinated when there is a controlling mechanism that tells each Pod whether there are units of work left to be completed. In the most basic form, this is implemented as a queue: the controlling process (the producer) inserts tasks into a queue that are then retrieved by one or more worker processes (the consumers).

In this scenario, the Job controller assumes that multiple Pods are working against the same objective and, that, whenever a Pod reports successful completion, the overarching batch goal is complete, and no further Pod runs are required. Imagine a team of three people—Mary, Jane, and Joe—working for a removal company in the process of loading furniture into a van. When Jane takes the last piece of furniture into the van, say a chair, not only Jane's job is done but so is Mary's and Joe's; they will all report that the job is done.

Configuring a Job to cater for this use case involves setting the `job.spec.parallelism` attribute to the desired number of parallel workers but leaving the `job.spec.completions` attribute unset. In essence, we only define the quantity of worker processes that there will be in the pool. This is the key aspect that differentiates externally coordinated batch processes from single and completion count–based ones.

To demonstrate this use case, we first need to set up some form of controlling queue mechanism. To make sure that we focus on the learning objective at hand—how to configure an externally coordinated Job—we will avoid introducing a large queue or Pub/Sub solution such as RabbitMQ; instead, we will define a simple process that listens in port 1080 and provides a new integer on each network request (starting from one):

```
i=1
while true; do
  echo -n $i | nc -l -p 1080
  i=$(($i+1))
done
```

This script works exactly like those red ticket dispensers found in hospitals and post offices where the first customer gets ticket 1, the second customer gets ticket 2, and so on. The difference is that in our shell-based version, customers get a new number by opening port 1080 with the nc command, as opposed to pulling a ticket from a dispenser. Please also note that we use the Alpine's distribution of the Netcat (nc) command to set up a dummy TCP server; it is worth to be aware that Netcat implementations tend to be rather fragmented across operating systems in terms of the number and type of flags available.

Before we create a Job to consume the tickets from the queue, let us first see the script in action so that we get acquainted with its operation. The provided shell script is launched as a Pod using the kubectl run command and exposed as a Service—using the --expose flag—so that it can be accessed it using the queue hostname from other Pods:

```
$ kubectl run queue --image=alpine \
    --restart=Never \
    --port=1080 \
    --expose \
    -- sh -c \
      "i=1;while true;do echo -n \$i \
      | nc -l -p 1080; i=\$((\$i+1));done"
service/queue created
pod/queue created
```

We can test the queue Pod manually as shown in the next example, but we have to remember to restart the deployment's Pod before running the rest of the examples so that the counter starts from 1 again. Alternatively, a script called `startQueue.sh` is provided in the chapter's source folder which deletes any existing running queue and starts a new one:

```
$ kubectl run test --rm -ti --image=alpine \
    --restart=Never -- sh
If you don't see a command prompt, try pressing enter.
/ # nc -w 1 queue 1080
1
/ # nc -w 1 queue 1080
2
/ # nc -w 1 queue 1080
3
/ # nc -w 1 queue 1080
...
```

Please note that the host named `queue` referred by the `nc` command is the DNS entry created by the Service controller as a result of adding the flag `--expose` when launching the queue Pod in the previous example. Now we have a central process to coordinate the work of multiple Pods. As simple as it may be, the queue service provides a unique task—represented by a new integer—to each consumer. Let us now define a consumer process whose job is simply to take an integer from the queue service and multiply it by two. When the number is 101 or greater, the overarching goal is considered achieved, and the script can claim victory by returning `0`—the successful exit status code:

```
while true
do n=$(nc -w 1 queue 1080)
   if [ $(($n+0)) -ge 101 ]
   then
     exit 0
   else r=$(($n*2))
     echo -en "$r\n"
     sleep 1
   fi
done
```

If this script were to run on its own, it would produce a sequence of numbers `2, 4, 6, ...` until reaching `200`. Any failure, say, in accessing `queue` on port 1080 would result in a nonzero exit code. Let us now embed the shell script in the container's compartment so that we can run it using the `alpine` image:

```
# multiplier.yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: multiplier
spec:
  parallelism: 3
  template:
    spec:
      restartPolicy: Never
      containers:
      - command:
        - /bin/sh
        - -c
        - >
          while true;
          do n=$(nc -w 1 queue 1080);
            if [ $(($n+0)) -ge 101 ];
            then
              exit 0;
            else r=$(($n*2))
              echo -en "$r\n";
              sleep 1;
            fi;
          done
        name: multiplier
        image: alpine
```

We now run the job by executing the kubectl apply -f <FILE> command:

```
$ kubectl apply -f multiplier.yaml
job "multiplier" created
```

We can observe the Job's behavior by running the kubectl get pods -w and kubectl get jobs -w commands on separate windows or tabs as suggested earlier in this chapter. We will see that three Pods will be instantiated and that, after a few seconds, their status will transition from Running to Complete. This will appear to happen almost at once since they will all start obtaining a number equal or greater than 101 pretty much at the same time:

```
$ kubectl get pods
NAME                READY   STATUS    RESTARTS   AGE
multiplier-7zdm7    1/1     Running   0          7s
multiplier-gdjn2    1/1     Running   0          7s
multiplier-k9fz8    1/1     Running   0          7s
queue               1/1     Running   0          50s

# A few seconds later...
$ kubectl get pods
NAME                READY   STATUS      RESTARTS   AGE
multiplier-7zdm7    0/1     Completed   0          36s
multiplier-gdjn2    0/1     Completed   0          36s
```

```
multiplier-k9fz8   0/1     Completed   0          36s
queue              1/1     Running     0          79s
```

The combined result of all three Pods should be a total of 100 numbers ranging from 2 to 200. We can check whether the job was completely successful by counting the number of lines that have been produced and inspecting the entire list itself:

```
$ kubectl logs -l job-name=multiplier --tail=-1 | wc
    100     100     347

$ kubectl logs -l job-name=multiplier --tail=-1 \
   | sort -g
2
4
6
...
196
198
200
```

Note The `--tail=-1` flag is necessary since the use of the label selector `-l` sets the tail limit to 10.

If we wanted a more formal proof of success, we can also sum the list of even numbers between 2 and 200 and prove that the sum of the combined logs is the same:

```
$ n=0;for i in $(seq 100);do \
    n=$(($n+($i*2)));done;echo $n
10100
```

```
$ n=0; \
  list=$(kubectl logs -l job-name=multiplier \
    --tail=-1); \
    for i in $list;do n=$(($n+$i));done;echo $n
10100
```

The combination of the queue service running in port 1080 that produces a sequence of integer numbers, plus the multiplier Job that reads those numbers and multiply them by two, show the basis of how highly scalable, parallelizable batch processes may be instrumented using the Job controller. Each Pod instance works on multiplying a single integer, which is rather random, depending of which Pods hits the queue service first; but the aggregated result of all independent calculations results in a complete list of even numbers between 2 and 200.

Note If the multiplier Job finds some errors in trying to access the queue TCP server, the count provided by the wc may be larger than 100 since it will include errors too.

Waiting Until a Job Completes

There are numerous ways of checking whether a Job has completed in an ad hoc manner. We can look at the number of successful completions using kubectl get jobs, or at the state of the jobs' Pods using kubectl get pod. However, if we want to integrate this check into a programmatic scenario, we need to interrogate the Job object directly. An easy way to tell whether a job has finished is by querying the job.status.completionTime attribute, which is only populated with the time when the associated job finishes.

As an example, the following shell expression waits until the multiplier job finishes by repeatedly checking until the job.status.completionTime attribute has become non-empty:

```
$ until [ ! -z $(kubectl get job/multiplier \
    -o jsonpath \
    --template="{.status.completionTime}") ]; \
    do sleep 1 ; echo waiting ... ; done ; echo done
```

Timing Out Stuck Jobs

As a general rule, it is a good idea to keep failing Jobs running, whenever said failures are because dependencies are not yet available. For example, in the case of the multiplier Job that we have used to demonstrate the use case of externally coordinated batch processes, the Pod controller will keep restarting the Pods until the queue service becomes available—provided that the number of retries is not greater than the backoffLimit. This behavior provides increased decoupling and reliability.

In certain cases, though, we have precise expectations about the maximum time that a Job may remain in a failed, incomplete state until the job is aborted. The way to achieve this is by setting the job.spec.activeDeadlineSeconds property to the desired number of seconds.

For example, let us take the same multiplier Job that we had used before and set the job.spec.activeDeadlineSeconds value to 10:

```
# multiplier-timeout.yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: multiplier
spec:
  activeDeadlineSeconds: 10
...
```

If we run the Job assuming that we had not started the *queue* service—we can clean up the environment by running `kubectl delete all --all`—and we watch the value of `job.status`, we will see that the Job will eventually be canceled:

```
$ kubectl apply -f multiplier-timeout.yaml
job "multiplier" created
```

```
$ kubectl get job/multiplier -o yaml -w \
    | grep -e   "^status:" -A 10
status:
  active: 3
  failed: 1
  startTime: "2019-07-17T22:49:37Z"
--
status:
  conditions:
 - lastProbeTime: "2019-07-17T22:49:47Z"
    lastTransitionTime: "2019-07-17T22:49:47Z"
    message: Job was active longer
             than specified deadline
    reason: DeadlineExceeded
    status: "True"
    type: Failed
  failed: 4
  startTime: "2019-07-17T22:49:37Z"
```

As it can be seen in the presented output, the Job was ended due to a `DeadlineExceeded` condition exactly ten seconds after the time indicated by the `.status.startTime` property.

Another way of timing out a Job is by setting the `job.spec.backoffLimit` attribute, which is 6 by default. This attribute defines how many times the Job controller should create a new Pod once it finishes with a terminal error. We have to bear in mind, though, that the wait time for each retry is, by default, exponentially higher than the previous one (10 seconds, 20 seconds, 40 seconds, etc.)

Finally, the activeDeadlineSeconds attribute sets a time-out for the overall Job's duration regardless of whether it is found in a failed state or not. A Job that is going through various successful completions with not one single failed Pod will also be aborted if the set deadline is reached.

Management Recap

As any other regular Kubernetes object, Job objects can be listed using the kubectl get jobs command and deleted using the kubectl delete job/<NAME> command. Similarly to the other controllers, like the ReplicaSet one, the delete command has a *cascading* effect in the sense that it will also delete the Pods referred by the Job's label selector. If we wanted to delete only the Job by itself, we would need to use the --cascade=false flag. For example, in the command sequence presented next, we run the two-times.yaml Job used to demonstrate Completion count–based batch processes, we delete the Job, and then, finally, we get the results produced by the Pods:

```
# Create a Job
$ kubectl apply -f two-times.yaml
job.batch/two-times created

# Delete the Job with but not its Pods
$ kubectl delete job/two-times --cascade=false
job.batch "two-times" deleted

# Confirm that the Job is effectively deleted
$ kubectl get jobs
No resources found.
```

```
# Extract logs from the two-times Pods
$ kubectl logs -l job-name=two-times | wc
     10      10      26
```

Summary

In this chapter, we learned that a Job is a Pod controller, similar to the ReplicaSet one, with the difference that Jobs are expected to terminate at some point in time. We saw that the fundamental unit of work in a Job is a completion, which takes place when a Pod exists with a status code `0`, and parallelism, which allows scaling batch processing throughput by multiplying the number of concurrently working Pods.

We explored the three fundamental types of batch processing use cases: single batch processes, completion count–based batch processes, and externally coordinated batch processes. We highlighted that the key difference between externally coordinated batch processes, and single and completion count–based ones was that the former's success criteria depend on a mechanism external to the Job controller—typically a queue.

Finally, we looked at regular management tasks such as monitoring a Job until it completes, timing out stuck jobs, and deleting them while preserving their results.

CHAPTER 7

CronJobs

Some tasks may need to be run periodically at regular intervals; for example, we may want to compress and archive logs every week or create backups every month. Bare Pods could be used to instrument said tasks, but this approach would require the administrator to set up a scheduling system outside of the Kubernetes cluster itself.

The need to schedule recurrent tasks brought the Kubernetes team to design a distinct controller modeled after the traditional *cron* utility found in Unix-like operating systems. Unsurprisingly, the controller is called *CronJob*, and it uses the same scheduling specification format as the *crontab* file; for example, `*/5 * * * *` specifies that the job will be run every five minutes. In terms of implementation, the CronJob object is similar to the Deployment controller in the sense that it does not control Pods directly; it creates a regular Job object which is, in turn, responsible for managing Pods—The Deployment controller uses a ReplicaSet controller for this purpose.

A key advantage of Kubernetes' out-of-the-box CronJob controller is that unlike its Unix cousin, it does not require a "pet" server that needs to be managed and nursed back to health when it fails.

This chapter begins with the introduction of a simple CronJob that can be launched both imperatively and interactively. Then, we look at the scheduling of recurring tasks in which we describe the crontab string's syntax. Following this, we cover the setting up of one-off tasks; how to

E. Garbarino, *Beginning Kubernetes on the Google Cloud Platform*,
https://doi.org/10.1007/978-1-4842-5491-2_7

increase the Job's history; the interaction among the CronJob controller, Jobs, and Pods; and the task of suspending and resuming active CronJobs.

Toward the end, we explain Job concurrency, which determines how overlapping Jobs are treated depending on the specified setting as well as to how the CronJob's "catch-up" behavior may be altered when it comes to skipped iterations. Finally, we recap on the CronJob's life cycle management tasks.

A Simple CronJob

A basic CronJob type can be created both imperatively, using the `kubectl run <NAME> --restart=Never --schedule=<STRING> --image=<URI>` command, and declaratively, using a manifest file. The first two flags signal the `kubectl run` command that a CronJob is desired, as opposed to a Pod or a Deployment. We will first look at the imperative version.

The default scheduling interval—and the lowest timer resolution—is one minute, and it is represented using the crontab string by five consecutive asterisks: `* * * * * *`. The string is passed to the `kubectl run` command using the `--schedule=<STRING>` flag. We will learn about the crontab string format in the next section. In the following CronJob, called `simple`, we also specify the `alpine` image and pass a shell command to print the date and the Pod's hostname:

```
$ kubectl run simple \
    --restart=Never \
    --schedule="* * * * *" \
    --image=alpine \
    -- /bin/sh -c \
    "echo Executed at \$(date) on Pod \$HOSTNAME"
cronjob.batch/simple created
```

Note Kubernetes is in the process of deprecating the creation of CronJobs using the `kubectl run` command. Kubernetes version v1.14 has introduced the `kubectl create cronjob <NAME>` command, which will be the de facto imperative CronJob creation approach from this version onward. This command takes the same flags as the `kubectl run` equivalent barring the `--restart` flag. For example:

```
$ kubectl create cronjob simple \
    --schedule="* * * * *" \
    --image=alpine \
    -- /bin/sh -c \
    "echo Executed at \$(date) on Pod \$HOSTNAME"
```

Right after we create the CronJob object—or even before—we can start watching the Job and Pod activity using the `kubectl get cronjobs -w`, `kubectl get jobs -w`, and `kubectl get pod -w` commands on different terminal windows or tabs; this is so because three different kinds of objects will be created—eventually: a CronJob, a set of Jobs, and a set of Pods, respectively. We will see all of these object types at play in the next examples.

Let us first see what the results are in the first few seconds after creating the CronJob object, named `simple`, shown earlier:

```
$ kubectl get cronjobs -w
NAME   SCHEDULE  SUSPEND  ACTIVE  LAST SCHEDULE   AGE
simple * * * * * False    0       <none>          0s
```

```
$ kubectl get jobs -w
$ # nothing
```

```
$ kubectl get pods -w
$ # nothing
```

As we can see in the resulting output, the CronJob's ACTIVE value is 0, and there are neither Job nor Pod objects running. As soon as the clock used by the CronJob controller turns to the next minute, we will see that ACTIVE—briefly—turns to 1 and that a new Job is created along with a child Pod:

```
$ kubectl get cronjobs -w
NAME   SCHEDULE  SUSPEND  ACTIVE  LAST SCHEDULE  AGE
simple * * * * * False    1       6s             29s
```

```
$ kubectl get jobs -w
NAME                DESIRED   SUCCESSFUL   AGE
simple-1520847060   1         0            0s
simple-1520847060   1         1            1s
```

```
$ kubectl get pods -w
NAME                    READY STATUS
simple-1520847060-xrnlh 0/1   Pending
simple-1520847060-xrnlh 0/1   ContainerCreating
simple-1520847060-xrnlh 0/1   Completed
```

This process will repeat every minute, indefinitely. We can check the results of our scheduled Jobs using kubectl logs <POD-NAME>. Next, we check the first Pod that was run shown in the kubectl get pods -w output, plus the second one:

```
$ kubectl logs simple-1520847060-xrnlh
Executed at Fri Mar 9 10:02:04 UTC 2019 on Pod simple-
1520847060-xrnlh
```

```
$ kubectl logs simple-1520847120-5fh5k
Executed at Fri Mar 9 10:03:04 UTC 2019 on Pod simple-
1520847120-5fh5k
```

By comparing the Pods' echoed time, 10:02:04 and 10:03:04, we can see that their execution is one minute apart—this may not necessarily be always second—exact since startup times may vary. If we keep the kubectl get pod -w running, we will see that a new Pod, with a different name, will be created and run every single minute.

The declarative version is presented next. Please note that at the time of writing, the API is still in beta—hence the apiVersion: batch/v1beta1 property—but it is likely that it will become final after publishing, so the reader may eventually want to try apiVersion: v1, instead:

```
# simple.yaml
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: simple
spec:
  schedule: "* * * * *"
  jobTemplate:
    spec:
      template:
       spec:
          containers:
          - name: simple
            image: alpine
            args:
            - /bin/sh
            - -c
            - >-
              echo Executed at $(date)
              on Pod $HOSTNAME
          restartPolicy: OnFailure
```

As any other manifest file, the `kubectl apply -f <FILE>` command is used. For example:

```
# clean up the environment first

$ kubectl apply -f simple.yaml
cronjob.batch/simple created
```

Setting Up Recurring Tasks

The frequency and timing details of a task implemented using the CronJob object are defined using the `cronjob.spec.schedule` property (or `--schedule` when using `kubectl run`) whose value is a crontab string.

The use of this format is bliss for former administrators of Unix-like operating systems, but it may be met with suspicion by those who expected a more intuitive and user-friendly approach in today's world of JSON and YAML. The use of this seemingly archaic format will not present a problem, though, as we will illustrate it in enough detail.

The crontab string format has five components—minute, hour, day of the month, month, and day of the week, separated by space. This means that the lowest resolution is one full minute; tasks can't be scheduled to run, say, every 15 or 30 seconds.

Table 7-1. *Crontab's components and valid range of digit-wise values*

Minute	Hour	Day/Month	Month	Day/Week
0-59	0-23	1-31	1-12	0-6

Each component takes either a specific value or an expression:

- `Digits`: Specific value respecting the valid range for each date component. For example, `30 15 1 1 *` will set the task to run at 15:30 on the first of January every year. The valid range of digits for each component is shown in Table 7-1.
- `*`: Any value. For example, `* * * * *` means that the task will run every minute, hour, day in the month, month, and day of the week.
- `,`: Value list separator. We use this to include multiple values. For example, `0,30 * * * *` means "every half an hour" since it specifies minute 0 and minute 30.
- `-`: Range of values. Sometimes writing a list of values is too tedious, so we may rather specify a range. For example, `* 0-12 * * *` means that the task will run every minute but only between 00:00 and 12:00 (AM).
- `/`: Step values. If the task runs at regular intervals, for example, every five minutes, or every two hours, rather than specifying the exact moments using a value list separator, we can simply step through. For example, for said settings we would use `*/5 * * * *` and `0 */2 * * *`, respectively.

The easiest way to reason about the crontab string format is to take the default string value, `* * * * *`, as the baseline, and alter it to make it less granular, according to the requirement at hand. Following this train of thought, Table 7-2 shows a sequence of sample crontab values; the macro is a keyword that can be used in lieu of the component-based string.

Table 7-2. Sample simple crontab values

String	Macro	Meaning
`* * * * *`	n/a	Every minute
`0 * * * *`	`@hourly`	Every hour (at the start of the hour)
`0 0 * * *`	`@daily`	Every day (at 00:00)
`0 0 * * 0`	`@weekly`	Every first day of the week (Sunday, at 00:00)
`0 0 1 * *`	`@monthly`	Every first day of the month (at 00:00)
`0 0 1 1 *`	`@yearly`	Every first day of the year (at 00:00)

Based on the sample values presented on Table 7-2, we can add further refinement by making the digit-wise components less regular; examples are shown in Table 7-3.

Table 7-3. Refinement of digit-wise cycles

String	Meaning
`*/15 * * * *`	Run every quarter of an hour
`0 0-5 * * *`	Run every hour between 00:00AM and 05:00AM
`0 0 1,15 * *`	Run only on the 1st and 15th of each Month
`0 0 * * 0-5/2`	Run from Sunday to Friday, every two days
`0 0 1 1,7 *`	Run only on the first of January and July
`15 2 5 1 *`	Every 5th of January at 02:15AM

Please note that the set of macros such as `@hourly` is a feature that appeared in newer cron implementations in Unix-like operating systems and whose support in Kubernetes appears to be partial. For example, `@reboot` macro is not implemented, so the author recommends using the traditional string format whenever possible.

There is also a handy site, https://crontab.guru/, that allows understanding and validating crontab string format combinations so that they result in effective intervals. This site is useful to answer pressing doubts such as is my *crontab* string formatted correctly? Will my CronJob run at the expected time?

Setting Up One-Off Tasks

The *crontab* format adopted by the Job controller is the traditional rather than the extended one, which means that it does not accept a year component. The implication is that the most granular event, such as `15 2 5 1 *` (5th of January at 02:15AM), will make the associated Job run every single year, indefinitely. In Unix-like operating systems, one-off tasks are usually run using the `at` utility rather than `cron`. In Kubernetes, we don't need a separate controller, but we require some external process to dispose of the CronJob after it has been run, so that it does not repeat itself the next year. Alternatively, the Pod itself can check whether it is running on the scheduled, intended year.

Jobs History

If we run the simple CronJob example described in the previous section and keep repeatedly running `kubectl get pods` (e.g., using the Linux `watch` command), we will never see more than three Pods. For example:

```
$ kubectl get pods
NAME                      READY STATUS    RESTARTS AGE
simple-1520677440-swmj6 0/1   Completed 0        2m
simple-1520677500-qx4xc 0/1   Completed 0        1m
simple-1520677560-dcpp6 0/1   Completed 0        9s
```

The moment that the `simple` CronJob is due to be run again, the oldest Pod—`simple-1520677440-swmj6`—will be disposed of by the CronJob controller. This behavior is usually desirable, because it ensures that the Kubernetes cluster does not run out of resources, but, as a consequence, it means that we will lose all information and logs about the Pods that are disposed.

Fortunately, this behavior can be tweaked using the `cronjob.spec.successfulJobsHistoryLimit` and the `cronjob.spec.failedJobsHistoryLimit` attributes. The default value for both properties is `3`. A value of `0` means that the Pods will be disposed of immediately after they complete, whereas a positive value specifies the exact number of Pods that will be kept available for examination—including log extraction.

In most cases, though, if the result of a CronJob is of uttermost importance, it is best to save said result to a more permanent storage mechanism, or an external logging system, rather than STDOUT.

Interacting with a CronJob's Jobs and Pods

A typical annoyance when working with CronJob objects is that locating its resulting Jobs and Pods—for troubleshooting purposes—can be tedious. Jobs are given a random name which is then used for `pod.metadata.label.job-name` label in Pods. This doesn't help when asking the question "Please give me all the jobs or Pods that match a given CronJob." The solution is adding a label, manually, to the Pod template; such a label will also appear in the Job object, so there is no need to define two separate labels. To keep in line with the same label convention used by the Job object when labeling Pods, the author recommends naming the label name as `cronjob-name`.

Taking the simple.yaml manifest used in the previous sections, applying the suggested cronjob-name label and saving it as simple-label.yaml results in the following manifest:

```
# simple-label.yaml
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: simple
spec:
  schedule: "* * * * *"
  jobTemplate:
    spec:
      template:
       metadata:
         labels:
           cronjob-name: simple
       spec:
          containers:
          - name: simple
            image: alpine
            args:
            - /bin/sh
            - -c
            - >-
              echo Executed at $(date)
              on Pod $HOSTNAME
          restartPolicy: OnFailure
```

We can now interact with the CronJob's Jobs and their Pods using the label selector flag `-l` in a convenient way. Next, we apply the `simple-label.yaml` manifest, let it run for over three minutes, list its Jobs, and then get its Pods' logs:

```
# clean up environment first

$ kubectl apply -f simple-label.yaml

# wait > 3 minutes

$ kubectl get jobs -l cronjob-name=simple
NAME                DESIRED   SUCCESSFUL   AGE
simple-1520977740   1         1            2m
simple-1520977800   1         1            1m
simple-1520977860   1         1            17s

$ kubectl logs -l cronjob-name=simple
Executed at Tue Mar 13 21:49:04 UTC 2019 on Pod simple-
1520977740-qcmr8
Executed at Tue Mar 13 21:50:04 UTC 2019 on Pod simple-
1520977800-jqwl8
Executed at Tue Mar 13 21:51:04 UTC 2019 on Pod simple-
1520977860-bcszj
```

Suspending a CronJob

CronJobs may be suspended at any time. They can also be launched in a suspended mode and only made active at a specific time; for example, we may have a number of network diagnosis tools that run every minute during debugging sessions, but we would like to disable the CronJob rather than deleting it. Whether a CronJob is in a suspended state or not is controlled using the `cronjob.spec.suspend` attribute which is set to `false` by default.

Suspending and resuming a live CronJob involve either editing the manifest live using kubectl edit cronjob/<NAME> command or using the kubectl patch command. Assuming the simple CronJob is still running, the following command will suspend it:

```
$ kubectl patch cronjob/simple \
    --patch '{"spec" : { "suspend" : true }}'
cronjob "simple" patched
```

We can validate whether the kubectl patch command has been successful by ensuring that the value of the SUSPEND column is True when running the kubectl get jobs command:

```
$ kubectl get cronjob
NAME     SCHEDULE    SUSPEND   ACTIVE   LAST SCHEDULE
simple   * * * * *   True      0        3m17s
```

Resuming a suspended CronJob is simply a matter of setting the suspend property back to false:

```
$ kubectl patch cronjob/simple \
    --patch '{"spec" : { "suspend" : false }}'
cronjob "simple" patched
```

Please note that given the asynchronous nature of the patching process, as well as that of the status reported when typing kubectl get cronjob, there might be a few temporary discrepancies between the effective CronJob's status and that observed.

Job Concurrency

What happens if a Job has not completed yet by the time the next scheduled run event is reached? This is a good question; the way in which the CronJob controller will react to this scenario depends on the value of the `cronjob.spec.concurrencyPolicy` attribute:

- `Allow` (default value): The CronJob will simply spin up a new Job and let the previous one keep running in parallel.
- `Forbid`: The CronJob controller will wait until the currently running Job completes before spinning up a new one.
- `Replace`: the CronJob controller will terminate the currently running Job and start a new one.

Let us now look at each `concurrencyPolicy` use case in detail, starting with the default `Allow` value. For this purpose, we will alter the `simple.yaml` CronJob manifest by replacing the shell script with a 150 seconds wait state (using the `sleep 150` command) and printing the timestamp and hostname before and after said `sleep` statement:

```
echo $(date) on Pod $HOSTNAME - Start
sleep 150
echo $(date) on Pod $HOSTNAME - Finish
```

We will then save the new manifest as `long-allow.yaml` which, embedding the presented script, results in the following file:

```
# long-allow.yaml
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: long
```

```
spec:
  schedule: "* * * * *"
  jobTemplate:
    spec:
      template:
        metadata:
          labels:
            cronjob-name: long
        spec:
          containers:
           - name: long
             image: alpine
             args:
             - /bin/sh
             - -c
             - echo $(date) on Pod $HOSTNAME - Start;
               sleep 150;
               echo $(date) on Pod $HOSTNAME - Finish
          restartPolicy: OnFailure
```

Applying the manifest using `kubectl apply -f long-allow.yaml` and waiting approximately over three minutes result in the following log output:

```
$ kubectl logs -l cronjob-name=long | sort -g
22:17:07 on Pod long-1520979420-t62wq - Start
22:18:07 on Pod long-1520979480-kwqm8 - Start
22:19:07 on Pod long-1520979540-mh5c4 - Start
22:19:37 on Pod long-1520979420-t62wq - Finish
...
```

As we can see here, the Pods `t62wq`, `kwqm8`, and `mh5c4` have started in sequence right after the turn of every minute. The first Pod, `t62wq`, has completed exactly 2 minutes and 30 seconds after it has started. At this time, `kwqm8` and `mh5c4` are still running in parallel since they haven't produced a `Finish` message yet.

This behavior, of overlapping start and finish Job times, may not be what we want; for example, it may lead to Node resource consumption escalating out of control. It is possible that jobs are meant to run in sequence and that a new iteration is only allowed once the previous one has finished. In this case, we set the `cronjob.spec.concurrencyPolicy` property to `Forbid`.

To observe the behavior of setting the `concurrencyPolicy` value to `Forbid`, we will modify the CronJob manifest as follows:

```
# long-forbid.yaml
...
spec:
  concurrencyPolicy: Forbid # New attribute
  schedule: "* * * * *"
...
```

We will then save the new manifest as `long-forbid.yaml` and apply it by issuing the `kubectl apply -f long-forbid.yaml` command—cleaning up the environment first, and not forgetting that we must wait a few minutes before the logs are populated:

```
$ kubectl logs -l cronjob-name=long | sort -g
22:39:10 on Pod long-1520980740-647m6 - Start
22:41:40 on Pod long-1520980740-647m6 - Finish
22:41:50 on Pod long-1520980860-d6dfb - Start
22:44:20 on Pod long-1520980860-d6dfb - Finish
```

As seen here, the execution of Jobs is now in perfect sequential order. The problem—if a problem at all—of overlapping messages seems to be solved now, but if we pay closer attention, we will see that the Jobs *no longer run exactly at the turn of every minute*. The reason why is that whenever the `cronjob.spec.concurrencyPolicy` attribute is set to `Forbid`, the CronJob object will wait until the current Job completes before initiating a new one.

The side effect of using the `Forbid` value is that Jobs may be skipped altogether if they take significantly longer than the crontab string interval. For example, let us suppose that a backup is scheduled to run every hour using the `0 * * * *` crontab string. If the backup Job takes, say, six hours, only 4 backups may be produced during the day rather than 24.

If we don't want Jobs to be run in parallel but we want to avoid missing scheduled "run slots" as well, then the only solution is to terminate whatever current Job is running and start a new one at the scheduled event. This is exactly what setting the `cronjob.spec.concurrencyPolicy` attribute to `Replace` achieves. Let us modify the manifest again to set this value and save it as `long-replace.yaml`:

```
# long-replace.yaml
...
spec:
  concurrencyPolicy: Replace # New attribute
  schedule: "* * * * *"
...
```

As usual, we clean up the environment first, apply the manifest by issuing the `kubectl apply -f long-replace.yaml` command, and wait a few minutes for the logs to populate:

```
$ kubectl logs -l cronjob-name=long | sort -g
23:37:07 on Pod long-1520984220-phrqc - Start
23:38:07 on Pod long-1520984280-vm67d - Start
...
```

As it can be appreciated by observing the resulting output, the `Replace` concurrency setting does enforce the timely start of the Job as per the crontab string but with two rather radical side effects. The first is that the currently running job is brutally terminated. This is why we don't see the `Finish` sentence printed on the logs. The second one is that, given that the running Jobs are terminated rather than allowed to complete, we have limited time to query their logs before the Jobs are deleted in a short time period. The `Replace` setting is, therefore, only useful for Jobs that are considered "stuck" when they haven't completed by the time the next scheduled event is reached.

In other words, the behavior resulting from setting the `concurrencyPolicy` to `Replace` is only applicable when the workloads being performed, by the underlying Pods, are of an idempotent nature; they can safely be interrupted in mid-flight without causing data loss or corruption, regardless of their current computation's state or their pending output. It follows that if said Pods happen to have something important to tell the world, then a more persistent backing service other than STDOUT is recommended.

To conclude this section, Table 7-4 summarizes the main behaviors that are associated with each of the values (`Allow`, `Forbid`, and `Replace`) for the `cronjob.spec.concurrencyPolicy` attribute.

Table 7-4. *CronJob behavior for each* `concurrencyPolicy` *value*

Behavior	`Allow`	`Forbid`	`Replace`
Multiple Jobs may run in parallel	Yes	No	No
Overlapping of Jobs' results	Yes	No	No
Timely execution of scheduled events	Yes	No	Yes
Abrupt termination of running Job	No	No	Yes

Catching Up with Missed Scheduled Events

As we have seen in the previous section, whenever multiple events are missed, the CronJob controller will generally try to catch up with one—but not all—of the missed events. The ability to run a Job associated with missed scheduled event is not magical; it is actually determined by the `cronjob.spec.startingDeadlineSeconds` property. When this property is left unspecified, there is no deadline.

Say that we had configured a 25-minute-lasting CronJob to run on minutes 0 and 1 (`0,1 * * * *`) and that we had also set the `cronjob.spec.concurrencyPolicy` attribute to `Forbid`. In this case, the first instance will run exactly at minute 0, but a second instance will still be run at minute 25 even though it is far away from the intended scheduled second minute "slot."

If we happen to assign a discrete positive value to the `cronjob.spec.startingDeadlineSeconds` attribute, then a "catch-up" run may not take place once the intended run iteration is reached. For example, if we set this property to `300` seconds (five minutes), the second run will certainly not happen since the CronJob controller will wait five minutes after minute `1`, and then, if by that time the previous Job has not yet completed, it will just give up. This behavior, although seemingly problematic, prevents the situation whereby a Job is queued up indefinitely, potentially leading to escalating resource consumption in the long run.

Management Recap

The current list of running CronJobs is obtained by issuing the `kubectl get cronjobs` command, whereas a specific CronJob is queried using either `kubectl describe cronjob/<NAME>` or `kubectl get cronjob/<NAME>` adding the `-o json` or `-o yaml` flags to obtain further details in a structured format.

As explained in the previous sections, it is convenient to provide a label to the Pod specification within a CronJob manifest so that it is easy to match a CronJob's dependent Jobs and Pods using the `-l` (match label) flag.

When a CronJob is deleted using the `kubectl delete cronjob/<NAME>` command, all of its dependent running and completed Jobs and Pods will be deleted as well:

```
$ kubectl delete cronjob/long
job "long-1521468900" deleted
pod "long-1521468900-k6mkz" deleted
cronjob "long" deleted
```

If we want to delete the CronJob but leave currently running Jobs and Pod undisturbed, so that they can finish their activities, we can use the `--cascade=false` flag. For instance:

```
$ kubectl delete cronjob/long --cascade=false
cronjob.batch "long" deleted

# After a few seconds...
$ kubectl get cronjobs
No resources found.

$ kubectl get jobs
NAME              DESIRED   SUCCESSFUL   AGE
long-1521468540   1         0            45s

$ kubectl get pods
NAME                   READY  STATUS  RESTARTS   AGE
long-1521468540-68dqd  1/1   Running  0          53s
```

Summary

In this chapter, we learned that the CronJob controller is similar to the Deployment object, in that it controls a secondary controller, a Job, which then in turn controls Pods; this relationship is illustrated in Figure 1-2 in Chapter 1. We observed that the CronJob controller uses the same crontab string format as that used by the familiar Unix cron utility and that the shortest interval is a minute, while the largest one is a year. We also pointed out that the advantage of the CronJob's controller over its Unix cousin was that it reduced complexity (and potential failures) by avoiding the necessity of maintaining an extra "pet" server just for hosting the cron utility.

We paid special attention to Job concurrency which determines how overlapping Jobs—whenever a new Job is meant to start when the previous one has not finished yet—are treated. We saw that the default value for the `cronjob.spec.concurrencyPolicy` attribute, `Allow`, simply let new Jobs be created and "pile up" in parallel to existing ones, whereas `Forbid` makes the controller wait until the previous Job has finished. `Replace`, the third, and last possible value, takes a radical approach; it simply abruptly terminates the previous running Job before starting a new one.

At the end, we learned how to tweak the way in which the CronJob catches up with missed iterations—when setting the concurrency policy to `Forbid`—using the `cronjob.spec.startingDeadlineSeconds` attribute.

CHAPTER 8

DaemonSets

The DaemonSet controller ensures that every Node runs a single instance of a Pod. This is useful for uniform, horizontal services that need to be deployed at the Node level, such as log collectors, caching agents, proxies, or any other kind of system-level capability. But why would, a distributed system such as Kubernetes, promote "tight coupling" between Pods and "boxes" as a feature? Because of performance advantages—and outright necessity sometimes. Pods deployed within the same Node can share the local network interface as well as the local file system; the benefit is significantly lower latency penalties than those that apply in the case of "off-board" network interactions.

In a way, a DaemonSet treats Nodes in the way that Pods treat containers: while Pods ensure that two or more containers are collocated together, DaemonSets guarantee that daemons—implemented also as Pods—are always locally available in every Node so that consumer Pods a can reach them.

This short chapter is organized as follows; first we will explore the two broad connectivity use cases for DaemonSets (TCP and the file system). Then, we will learn how to target specific Nodes using labels. At the end, we will describe the difference in the update strategy between Deployments and DaemonSets and why the `maxSurge` attribute is not available in the latter.

E. Garbarino, *Beginning Kubernetes on the Google Cloud Platform*,
https://doi.org/10.1007/978-1-4842-5491-2_8

TCP-Based Daemons

A TCP-based Daemon is a regular Pod managed by a DaemonSet controller whereby the services are accessed via TCP. The difference is that because every Daemon-controlled Pod is guaranteed to be deployed in every Node, there is no need for service discovery mechanism since client Pods can simply establish a connection to the local Node they run on. We will see how all of this works in a moment.

First, let us define a simple Node-level service using the Netcat `nc` command: a log collector that listens on port 6666 and appends all log requests to a file called `/var/node_log`:

```
nc -lk -p 6666 -e sh -c "cat >> /var/node_log"
```

The next step is to wrap our shell-based log collector in a Pod template which is, in turn, embedded in a DaemonSet manifest. Similarly to a Deployment, the label and label selector must match:

```
# logDaemon.yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: logd
spec:
  selector:
    matchLabels:
      name: logd
  template:
    metadata:
      labels:
        name: logd
    spec:
     containers:
      - name: logd
        image: alpine
        args:
        - /bin/sh
        - -c
        - >-
          nc -lk -p 6666 -e
          sh -c "cat >> /var/node_log"
        ports:
        - containerPort: 6666
          hostPort: 6666
```

After applying the `logDaemon.yaml` manifest, we will have a cheap, homebrew log collection service deployed in every Kubernetes Node:

```
$ kubectl apply -f logDaemon.yaml
daemonset.apps/logd created
```

Since the DaemonSet should create exactly one Pod per worker Node, the number of Pods in the Kubernetes cluster and the number of DaemonSet-controlled Pods should match:

```
$ kubectl get nodes
NAME                   STATUS   AGE
gke-*-ab1848a0-ngbp    Ready    20m
gke-*-ab1848a0-pv2g    Ready    20m
gke-*-ab1848a0-s9z7    Ready    20m
```

```
$ kubectl get pods -l name=logd -o wide
NAME       STATUS   AGE  NODE
logd-95vnl Running  11m  gke-*-s9z7
logd-ck495 Running  11m  gke-*-pv2g
logd-zttf4 Running  11m  gke-*-ngbp
```

Now that the log collector DaemonSet-controlled Pods are available in every Node, we will create a sample client to test it. The following shell script produces a greeting every 15 seconds which is sent to a TCP service running on $HOST_IP on port 6666:

```
while true
do echo $(date) - Greetings from $HOSTNAME |
   nc $HOST_IP 6666
   sleep 15
done
```

We have three distinct parameters in this script: the port number, 6666, which is hard-coded; the $HOSTNAME variable, which is populated automatically by the Pod controller—and it is accessible from within the container—and $HOST_IP which is a user-defined variable. Neither the hostname nor the IP address of the Node on which a Pod runs is made known to the Pod explicitly. This creates a new problem to solve that requires the use of the *Downward API*.

The Downward API allows to query fields from a Pod's object and make them available to the same Pod's containers as environment variables. In this particular case, we are interested in the pod.status.hostIP attribute. In order to "inject" this attribute's value into the HOST_IP environment variable, we first declare the variable's name using the name attribute and then reference the desired property from the Pod's object using the valueFrom.fieldRef.fieldPath attribute—all under the pod.spec.containers.env compartment:

```
...
env:
  - name: HOST_IP
    valueFrom:
      fieldRef:
        fieldPath: status.hostIP
...
```

Having defined a sample client, and the extra configuration required to inject the Node's IP, we now combine the sample client's shell script and the Downward API query, to populate HOST_IP in a single Deployment manifest named logDaemonClient.yaml:

```
# logDaemonClient.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: client
spec:
  replicas: 7
  selector:
    matchLabels:
      name: client
  template:
    metadata:
      labels:
        name: client
    spec:
      containers:
      - name: client
        image: alpine
        env:
          - name: HOST_IP
            valueFrom:
              fieldRef:
                fieldPath: status.hostIP
        args:
        - /bin/sh
        - -c
        - >-
```

```
          while true;
          do echo $(date) -
          Greetings from $HOSTNAME |
          nc $HOST_IP 6666;
          sleep 15;
          done
```

Applying `logDaemonClient.yaml` will result in the creation of multiple replicas (seven in total) which will land on different, assorted Nodes, as it can be observed by running the `kubectl get pods` command using the `-o wide` flag:

```
$ kubectl apply -f logDaemonClient.yaml
deployment.apps/client created
```

```
$ kubectl get pods -l name=client -o wide
NAME            IP          NODE
client-*-5hn9p  10.28.2.15  gke-*-ngbp
client-*-74ssw  10.28.0.14  gke-*-pv2g
client-*-h5fmm  10.28.2.16  gke-*-ngbp
client-*-rjgz8  10.28.1.14  gke-*-s9z7
client-*-tgk7r  10.28.0.13  gke-*-pv2g
client-*-twk5p  10.28.2.14  gke-*-ngbp
client-*-wg6th  10.28.1.15  gke-*-s9z7
```

If we pick a random DaemonSet-controlled Pod and query its logs, we will see that they originate in the same Node as the Deployment-controlled client Pods:

```
$ kubectl exec logd-8cgrb -- cat /var/node_log
08:58:56 - Greetings from client-*-tgk7r
08:58:57 - Greetings from client-*-74ssw
08:59:11 - Greetings from client-*-tgk7r
08:59:12 - Greetings from client-*-74ssw
```

```
08:59:26 - Greetings from client-*-tgk7r
08:59:27 - Greetings from client-*-74ssw
...
```

For reference, these are the Nodes in which the DaemonSet-controlled Pods have landed:

```
$ kubectl get pods -l name=logd -o wide
NAME       IP         NODE
logd-8cgrb 10.28.0.12 gke-*-pv2g
logd-m5z4m 10.28.1.13 gke-*-s9z7
logd-zd9z9 10.28.2.13 gke-*-ngbp
```

Note that the Pods `logd-8cgrb`, `client-*-tgk7r`, and `client-*-74ssw` are all deployed on the same Node named `gke-*-pv2g`:

```
$ kubectl get pod/logd-8cgrb -o jsonpath \
    --template="{.spec.nodeName}"
gke-*-pv2g

$ kubectl get pod/client-5cbbb8f78-tgk7r \
    -o jsonpath --template="{.spec.nodeName}"
gke-*-pv2g

$ kubectl get pod/client-5cbbb8f78-74ssw \
    -o jsonpath --template="{.spec.nodeName}"
gke-*-pv2g
```

To recap, to set up a general TCP-based DaemonSet solution, we need to define a DaemonSet manifest to deploy the daemon itself, and then, for consuming Pods, we require the use of the Downward API to inject the address of the Node on which the Pod runs. The use of the Downward API involves querying specific Pod's object properties and making them available through environment variables to the Pod.

File System–Based Daemons

In the previous section, we have contemplated the case of a TCP-based DaemonSet which is characterized by the fact that clients access it directly using the node's IP address—injected through the Downward API—as opposed to the Service object. Pods deployed using the DaemonSet controller have also another means to communicate with one another, when deployed on the same Node: the file system.

Let us consider the case of a daemon that creates a *tarball* out of all logs found in /var/log every 60 seconds using the following shell script:

```
while true
do tar czf \
   /var/log/all-logs-`date +%F`.tar.gz /var/log/*.log
   sleep 60
done
```

The manifest for a file system–based DaemonSet requires us that we specify a volume (the description of the driver and directory available across the Pod) and a volume mount (the binding of the volume to a file path inside the applicable container).

For the volume, we specify a volume called logdir that points to the Node's /var/log under the pod.spec.volumes attribute:

```
# at pod.spec
volumes:
- name: logdir
  hostPath:
    path: /var/log
```

Then, we reference the logdir volume under the pod.spec.containers.volumeMounts compartment and establish that it will be mounted under the /var/log path inside of our container:

```
# at pod.spec.containers
volumeMounts:
- name: logdir
  mountPath: /var/log
```

Finally, we assemble the presented two definitions into a DaemonSet manifest called logCompressor.yaml:

```
# logCompressor.yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: logcd
spec:
  selector:
    matchLabels:
      name: logcd
  template:
    metadata:
      labels:
        name: logcd
    spec:
      volumes:
      - name: logdir
        hostPath:
          path: /var/log
      containers:
      - name: logcd
        image: alpine
        volumeMounts:
        - name: logdir
          mountPath: /var/log
        args:
        - /bin/sh
        - -c
        - >-
          while true;
          do tar czf
          /var/log/all-logs-`date +%F`.tar.gz
          /var/log/*.log;
          sleep 60;
          done
```

After applying `logCompressor.yaml`, we can query a random Pod to tell whether a tarball file has been created in its allocated Node:

```
# clean up the environment first
$ kubectl apply -f logCompressor.yaml
daemonset.apps/logcd created

$ kubectl get pods
NAME           READY     STATUS    RESTARTS   AGE
logcd-gdxc7    1/1       Running   0          0s
logcd-krf2r    1/1       Running   0          0s
logcd-rd9mb    1/1       Running   0          0s

$ kubectl exec logcd-gdxc7 \
    -- find /var/log -name "*.gz"
/var/log/all-logs-2019-04-26.tar.gz
```

Now that our file system–based DaemonSet is up and running, let us proceed with modifying the client so that it sends its output to `/var/log/$HOSTNAME.log` rather than the TCP port 6666:

```
# logCompressorClient.yaml
apiVersion: apps/v1
kind: Deployment
```

```
metadata:
  name: client2
spec:
  replicas: 7
  selector:
    matchLabels:
      app: client2
  template:
    metadata:
      labels:
        app: client2
    spec:
      volumes:
      - name: logdir
        hostPath:
          path: /var/log
      containers:
      - name: client2
        image: alpine
        volumeMounts:
         - name: logdir
           mountPath: /var/log
        args:
        - /bin/sh
        - -c
        - >
          while true;
          do echo $(date) -
          Greetings from $HOSTNAME >> \
          /var/log/$HOSTNAME.log;
          sleep 15;
          done
```

If we look closely, we will see that logCompressorClient.yaml includes the same volumes and volumeMounts compartments as logCompressor.yaml. This is because both the DaemonSet and its clients require the details of the file system they both share.

Once logCompressorClient.yaml is applied, we can wait a few minutes and prove whether the resulting tarball in each host—created by the DaemonSet—includes the log files generated by the Deployment:

```
$ kubectl apply -f logCompressorClient.yaml
deployment.apps/client2 created

# after a few minutes...

$ kubectl exec logcd-gdxc7 -- \
    tar -tf /var/log/all-logs-2019-04-26.tar.gz

var/log/cloud-init.log
var/log/kube-proxy.log
var/log/client2-5549f6854-c9mz2.log
var/log/client2-5549f6854-dvs4f.log
var/log/client2-5549f6854-lhgxx.log
var/log/client2-5549f6854-m29gb.log
var/log/client2-5549f6854-nl6nx.log
var/log/client2-5549f6854-trcgz.log
```

The files following the $HOSTNAME.log naming convention such as client2-5549f6854-c9mz2.log are indeed those generated by logCompressorClient.yaml.

Note A real-world log solution would rarely rely on the Node's file system but on a Cloud-supported external drive (such as Google Cloud Platform Persistent Disk) instead. Conversely, a CronJob (see

Chapter 7), as opposed to a shell `sleep` statement, would typically be more suitable than a DaemonSet given that it provides global scheduling capabilities.

Daemons That Run on Specific Nodes Only

In some advanced scenarios, not all Nodes in a Kubernetes cluster may be homogenous, disposable virtual machines, underpinned by commodity hardware. Some servers may have special capabilities such as graphical processing units (GPUs) and fast solid-state drive (SSD)—or even use a different operating system. Alternatively, we may want to establish a hard rather than logical segregation between environments; in other words, we may want to isolate environments at the Node level rather than at the object level—as we typically do when using namespaces. This is the use case that we will contemplate in this section.

Segregating DaemonSets, so that they land on specific Nodes, requires us that we apply a label to each Node that we want to be able to differentiate. Let us begin by listing the Nodes we currently have:

```
$ kubectl get nodes
NAME          STATUS   AGE
gke-*-809q    Ready    1h
gke-*-dvzf    Ready    1h
gke-*-v0v7    Ready    1h
```

We can now apply a label to each of the listed Nodes. The labeling approach relies on the `kubectl label <RESOURCE-TYPE>/<OBJECT-IDENTIFIER>` command that we have explored in Chapter 2. We will designate the first Node, `gke-*-809q`, as `prod` (production), and `gke-*-dvzf`, `gke-*-v0v7` as dev (development):

```
$ kubectl label node/gke-*-809q env=prod
node "gke-*-809q" labeled
```

```
$ kubectl label node/gke-*-dvzf env=dev
node "gke-*-dvzf" labeled

$ kubectl label node/gke-*-v0v7 env=dev
node "gke-*-v0v7" labeled
```

Then, we can check the value of each Node’s label using the `kubectl get nodes` command together with the `-L env` flag which shows an extra column named `ENV`:

```
$ kubectl get nodes -L env
NAME         STATUS  AGE ENV
gke-*-809q   Ready   1h  prod
gke-*-dvzf   Ready   1h  dev
gke-*-v0v7   Ready   1h  dev
```

Now, all we have to do is take the `logCompressor.yaml` manifest shown in the previous section and add `pod.spec.nodeSelector` attribute to the Pod template:

```
# at spec.template.spec
spec:
  nodeselector:
    env: prod
```

If we save the new manifest as `logCompressorProd.yaml` and apply it, the result will be that the DaemonSet’s Pod will only be deployed to the Node whose label is `prod`:

```
# clean up the environment first
$ kubectl apply -f logCompressorProd.yaml
daemonset.apps/logcd configured

$ kubectl get pods -o wide
NAME        READY  STATUS    NODE
logcd-4rps8 1/1    Running   gke-*-809q
```

Note Please note that the choice of Nodes labeled as `prod` (production) and `dev` (development) is merely for didactic purposes. Actual environments that segregate SDLC phases are typically achieving using namespaces and sometimes outright distinct Kubernetes cluster instances.

Update Strategy

Updating an existing DaemonSet does not work in exactly the same way as updating a Deployment, when the aim is to produce a zero-downtime update. In a typical zero-downtime Deployment update, one or more extra Pods are ramped up—controlled using the `deployment.spec.strategy.rollingUpdate.maxSurge` attribute—with the purpose of always having at least one additional Pod available before terminating an old one. This process is aided by the Service controller, which can take Pods in and out of the load balancer as the migration progresses. In the case of a DaemonSet, the `maxSurge` attribute is not available; we will see why.

While the location—the exact landing Node —of the Pods controlled by a regular Deployment controller is fairly inconsequential, the DaemonSet controller, instead, has the contractual objective of ensuring the availability of exactly one Pod per Node. The number of both minimum and maximum "replicas" is, therefore, the total number of Nodes in the cluster—barring the case of using special Node selectors as seen in the last section. Furthermore, Pods deployed by the DaemonSet controller are typically accessed locally using the file system or a Node-level TCP port rather than via a Service controller-managed proxy and DNS entry. In a nutshell, a DaemonSet's Pod is a Node-level singleton rather than an anonymous member of a swarm of scalable objects. DaemonSets implement system-level workloads that take a more foundational role and higher priority than other more ephemeral applications like, for instance, web servers.

Let us imagine for a second what would be the consequence of, hypothetically speaking, setting a DaemonSet's maxSurge attribute to 1. If this were possible, a number of Pods greater than the total number of Nodes in the cluster could exist for a while, during a DaemonSet update process. For example, in a Kubernetes cluster of three Nodes, a maxSurge of 1 would allow the presence of four Nodes during a DaemonSet update. The logical consequence is that additional Pod(s) would land on a Node in which there is already an existing Pod in operation; this violates the principle of what a DaemonSet is meant to guarantee: the presence of exactly one Pod per Node. The conclusion is that updating a DaemonSet (e.g., selecting a new image) will involve some natural downtime at least at the local Node level.

The DaemonSet manifest allows two types of update strategies: OnDelete and RollingUpdate. The first instructs the DaemonSet controller to wait until each Pod is *manually deleted* before the controller can replace it with a new Pod based on the template included in the new manifest. The second one operates similarly to the Deployment controller's declaration for rolling updates, except that there is no maxSurge property, only maxUnavailable. The default update strategy is in fact RollingUpdate, with a maxUnavailablity value of 1:

```
# at daemonset.spec (default)
updateStrategy:
 type: RollingUpdate
 rollingUpdate:
   maxUnavailable: 1
```

This default configuration results in one Node being updated at a time whenever an update is produced. For example, if we run again the logCompressor.yaml manifest presented earlier in this chapter and change its default image to busybox—we use alpine by default—we will see that the DaemonSet controller will take one Node at a time, terminate its running Pod, deploy the new one, and only then move onto the next one:

```
$ kubectl get pods -o wide
NAME        READY STATUS  IP         NODE
logcd-k6kb4 1/1   Running 10.28.0.10 gke-*-h7b4
logcd-mtpnp 1/1   Running 10.28.2.12 gke-*-10gx
logcd-pgztn 1/1   Running 10.28.1.10 gke-*-lnxh
```

```
$ kubectl set image ds/logcd logcd=busybox
daemonset.extensions/logcd image updated
```

```
$ kubectl get pods -o wide -w
NAME        READY STATUS      IP         NODE
logcd-k6kb4 1/1   Running     10.28.0.10 gke-*-h7b4
logcd-mtpnp 1/1   Running     10.28.2.12 gke-*-10gx
logcd-pgztn 1/1   Running     10.28.1.10 gke-*-lnxh
logcd-pgztn 1/1   Terminating 10.28.1.10 gke-*-lnxh
logcd-57tzz 0/1   Pending     <none>     gke-*-lnxh
logcd-57tzz 1/1   Running     10.28.1.11 gke-*-lnxh
logcd-k6kb4 1/1   Terminating 10.28.0.10 gke-*-h7b4
...
```

In the output from kubectl get pods, we can see that that there are, initially, three Nodes and that right after the kubectl set image command is issued, the Pod in the Node gke-*-lnxh is terminated, and a new one is created before a different Node, gke-*-h7b4, is selected by the DaemonSet controller to apply the update process again.

Management Recap

The number of live DaemonSet controllers is listed using the `kubectl get daemonsets` command. For example, after applying the `logCompressor.yaml` manifest used as an example in this chapter, the result would be as follows:

```
$ kubectl get daemonsets
NAME  DESIRED CURRENT READY UP-TO-DATE AVAILABLE
logcd 3       3       3     3          3
```

If a specific Node selector is not specified—which is the case in the shown output—the number under the `DESIRED` column should match the total number of Nodes in the Kubernetes cluster. Please also note that a column named `NODE SELECTOR` is displayed (omitted in the presented output for space restrictions) which indicates whether the DaemonSet is bound to a particular Node.

For further interrogation on a specific DaemonSet, the `kubectl get daemonset/<NAME>` or `kubectl describe daemonset/<NAME>` commands may be used.

Deletion works as with any other Kubernetes workload controller. The default delete command, `kubectl delete daemonset/<NAME>`, will delete all of the DaemonSet's dependent Pods unless the flag `--cascade=false` is applied:

```
$ kubectl delete ds/logcd
daemonset.extensions "logcd" deleted

$ kubectl get pods
NAME         READY  STATUS
logcd-xgvm9  1/1    Terminating
logcd-z79xb  1/1    Terminating
logcd-5r5mn  1/1    Terminating
```

Summary

In this chapter, we learned that the DaemonSet controller serves to deploy a single Pod in every Node of a Kubernetes cluster so that they can be accessed locally using TCP or the file system—both typically faster than other types of *off-board* network access. Log aggregators, as exemplified in this chapter, were used as a pedagogic example; more industrial use cases include health monitoring agents and service mesh proxies—or so-called *sidecars*.

Although DaemonSets are similar to Deployments, we saw that a key difference is that they are not intended for zero-downtime update scenarios because Pods are terminated before being replaced by a new version; this behavior may temporarily disrupt local Pods that access the DaemonSet's Pods at the Node level using either the TCP loopback device or the file system. As such, clients to DaemonSet's Pods must be designed in a fault-tolerant manner if they are expected to withstand the live update of DaemonSet's Pods.

CHAPTER 9

StatefulSets

The Twelve-Factor App methodology is, arguably, one of the most widespread set of principles for cloud native applications. Factor IV called "Backing Services" says that backing services should be treated as attached resources. One of the passages, at `https://12factor.net/backing-services`, reads:

> *The code for a twelve-factor app makes no distinction between local and third party services. To the app, both are attached resources, accessed via a URL or other locator/credentials stored in the config. A deploy of the twelve-factor app should be able to swap out a local MySQL database with one managed by a third party (such as Amazon RDS) without any changes to the app's code. Likewise, a local SMTP server could be swapped with a third-party SMTP service (such as Postmark) without code changes. In both cases, only the resource handle in the config needs to change.*

The Twelve-Factor App methodology is silent on what are the best practices to deliver *our own* backing services since it presumes that applications will land on a stateless PaaS such as Heroku, Cloud Foundry, or Google App Engine—all fully managed services. What if we need to implement our own backing services rather than relying on, say, Google Bigtable? The Kubernetes answer to implement backing services is the StatefulSet controller.

Backing services have different dynamics than stateless Twelve-Factor Applications. Scaling is not a trivial matter; scaling "down" may result in the loss of data, and scaling up may result in the inappropriate replication

E. Garbarino, *Beginning Kubernetes on the Google Cloud Platform*,
https://doi.org/10.1007/978-1-4842-5491-2_9

or resharding of the existing cluster. Some backing services are not meant to scale at all—at least in an automatic fashion.

Backing services also vary greatly in how they achieve high scalability. Some use a supervisor-worker (master-slave) strategy (e.g., MySQL and MongoDB), whereas some others have a multi-master architecture instead, for example, Memcached and Cassandra. The StatefulSet controller in Kubernetes cannot make broad, sweeping assumptions about the nature of each data store; it, therefore, focuses on low-level, primitive properties—such as stable network identity—that may, selectively, assist in their implementation depending on the discrete problem or required property at hand.

In this chapter, we will build a primitive key/value data store from scratch, which will serve to internalize the principles of implementing StatefulSets without the risk of leaving details out that may not happen to apply to a single particular product, such as MySQL or MongoDB. We will be enriching said primitive key/value data store as we progress through the chapter. In the first sections, we will introduce the principles of sequential Pod creation, stable network identity, and the use of a headless service—the latter is key in the publishing of backing services. Then we will look at the Pod life cycle events that we can instrument to implement graceful startup and shutdown functionality. Finally, we will show how to implement storage-based persistence which is what *statefulness* is ultimately all about.

A Primitive Key/Value Store

What we are about to see may be considered a poor man's Memcached or BerkleyDB. This key/value store performs only three tasks: it saves key/value pairs, it looks up and retrieves a value by a unique key, and it lists all existing keys. The keys are saved as regular files on the file system whereby the file name is the key and its contents are the value. There is no input validation, delete functions, and security measures of any kind.

The three aforementioned functions (*save, load,* and *allKeys*, respectively) are implemented as HTTP services using the Flask framework in Python 3:

```
#!/usr/bin/python3
# server.py
from flask import Flask
import os
import sys

if len(sys.argv) < 3:
  print("server.py PORT DATA_DIR")
  sys.exit(1)

app     = Flask(__name__)
port    = sys.argv[1]
dataDir = sys.argv[2] + '/'

@app.route('/save/<key>/<word>')
def save(key, word):
  with open(dataDir + key, 'w') as f:
       f.write(word)
  return word

@app.route('/load/<key>')
def load(key):
  try:
    with open(dataDir + key) as f:
      return f.read()
  except FileNotFoundError:
      return "_key_not_found_"

@app.route('/allKeys')
def allKeys():
  keys = ".join(map(lambda x: x + ",",
         filter(lambda f:
                os.path.isfile(dataDir+'/'+f),
                os.listdir(dataDir)))).rstrip(',')
  return keys

if __name__ == '__main__':
  app.run(host='0.0.0.0', port=port)
```

Please note that the actual file server.py found under this chapter's folder has extra features (code in other words) that have been omitted in the presented listing. Said omitted features help deal with graceful startup and shutdown and will be discussed in a few sections ahead.

To experiment with the server locally, we can first install Flask and then launch the server by passing the port number and data directory as arguments:

```
$ sudo pip3 install Flask
$ mkdir -p /tmp/data
$ ./server.py 1080 /tmp/data
```

Once the server is up and running, we can "play" with it by inserting and retrieving some key/value pairs:

```
$ curl http://localhost:1080/save/title/Sapiens
Sapiens
$ curl http://localhost:1080/save/author/Yuval
Yuval
$ curl http://localhost:1080/allKeys
author,title
$ curl http://localhost:1080/load/title
Sapiens
$ curl http://localhost:1080/load/author
Yuval
```

Minimal StatefulSet Manifest

In the previous section, we have presented a key/value store HTTP-based server written in Python that we will now run using the StatefulSet controller.

A minimal StatefulSet manifest is, for the most part, just like a Deployment one: it allows defining the number of replicas, a Pod template with one or more containers, and so on:

```
# wip/server.yaml
# Minimal manifest for running server.py
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: server
spec:
  selector:
    matchLabels:
      app: server
  serviceName: server
  replicas: 3
  template:
    metadata:
      labels:
        app: server
    spec:
      containers:
      - name: server
        image: python:alpine
        args:
        - bin/sh
        - -c
    - >-
     pip install flask;
     python -u /var/scripts/server.py 80
     /var/data
    ports:
    - containerPort: 80
    volumeMounts:
     - name: scripts
      mountPath: /var/scripts
     - name: data
      mountPath: /var/data
   volumes:
    - name: scripts
     configMap:
      name: scripts
    - name: data
     emptyDir:
      medium: Memory
```

Rather than containerizing `server.py`, we use `python:alpine`, an off-the-shelf Docker image that contains the Python 3 interpreter. The file `server.py` must be "uploaded" as a ConfigMap named `scripts` which is set up as follows:

```
#!/bin/sh
# wip/configmap.sh
kubectl delete configmap scripts \
 --ignore-not-found=true
kubectl create configmap scripts \
 --from-file=../server.py
```

Also, note the use of a volume called data that is set up as a volume of type emptyData using RAM memory. This means that our key/value store works, for now, as an in-memory cache rather than a persistent store that survives server crashes. We will elaborate more on this aspect soon.

Now, we have everything we need to run our key/value store as a StatefulSet:

```
$ cd wip
$ ./configmap.sh
$ kubectl apply -f server.yaml
statefulset.apps/server created
```

When we list the resulting Pods with kubectl get pods, we can notice that unlike Deployments, the Pod names follow a sequential order, starting from 0, rather than having a random suffix. We will discuss the sequential Pod creation property in the next section:

```
$ kubectl get pods
NAME       READY     STATUS    RESTARTS   AGE
server-0   1/1       Running   0          5s
server-1   1/1       Running   0          0s
server-2   0/1       Pending   0          0s
```

To prove that the key/value store server is working property, we can establish a proxy with one of the Pods, set up a key/value pair, and then retrieve its value:

```
$ kubectl port-forward server-0 1080:80
Forwarding from 127.0.0.1:1080 -> 80
...

# Set a key/value pair
$ curl http://localhost:1080/save/title/Sapiens
Sapiens
```

```
# Retrieve the value for the title key
$ curl http://localhost:1080/load/title
Sapiens
```

If `kubectl port-forward` reports an error such as `bind: address already in use`, it means that we had left `server.py` running on port 1080 or that some other process is using this port. The reader may change the port in the source code if it happens to be permanently allocated to some other application.

Sequential Pod Creation

By default, the Deployment controller creates all Pods in parallel (barring upgrades in which `.maxSurge` constraints may apply) to speed up the process. The StatefulSet controller, instead, creates Pods in sequence, starting from `0` up to the defined number of replicas minus one. This behavior is controlled by the `statefulset.spec.podManagementPolicy` attribute whose default value is `OrderedReady`. The other possible value is `Parallel`, which produces the same behavior as the Deployment and ReplicaSet controllers.

We can see sequential Pod creation in action by running `kubectl get pods -w` before applying `kubectl apply -f server.yaml` in the previous section:

```
$ kubectl get pods -w
NAME      READY  STATUS
server-0  0/1    Pending
server-0  0/1    ContainerCreating
server-0  1/1    Running
server-1  0/1    Pending
server-1  0/1    ContainerCreating
server-1  1/1    Running
```

```
server-2   0/1    Pending
server-2   0/1    ContainerCreating
server-2   1/1    Running
```

Why is sequential Pod creation important? Because backing services often have semantics that rely on solid assumptions about which exact Pods have been previously created and what Pods will be created next:

- In a backing store based on the supervisor-worker paradigm, such as MongoDB or MySQL, the worker pods pod-1 and pod-2 might expect pod-0 (the supervisor) to be defined first so that they can register with it.
- A Pod creation sequence may involve scaling up an existing cluster in which data may be copied from, say, pod-0, to pod-1, pod-2, and pod-3.
- A pod deletion sequence may require unregistering the worker one by one.

As mentioned in the last point, the property of sequential Pod creation also works in reverse: the Pod with the highest index is always terminated first. For example, reducing the key/store cluster to 0 replicas by issuing `kubectl scale statefulset/server --replicas=0` results in the following behavior:

```
$ kubectl get pods -w
NAME       READY     STATUS
server-0   1/1       Running
server-1   1/1       Running
server-2   1/1       Running
server-2   1/1       Terminating
server-2   0/1       Terminating
server-1   1/1       Terminating
```

```
server-1   0/1       Terminating
server-0   1/1       Terminating
server-0   0/1       Terminating
```

Stable Network Identity

In a stateless application, the specific identity and location of each replica is ephemeral. As long as we can reach the load balancer, it doesn't matter which specific replica serves our request. This is not necessarily the case for a backing service, such as our primitive key/value store or the likes of Memcached. The way in which many multi-master stores solve the problem of scale, without creating a central contention point, is by making each client (or equivalent proxy agent) aware of every single server host so that the client itself decides where to store and retrieve the data.

It follows that, in the case of StatefulSets, depending on the data store's horizontal scaling strategy, it may be crucial for clients to know exactly the Pods to which they have saved data so that following scaling, restarts, and failures events, the load request always matches the original Pod used for original save request.

For example, if we set the key `title` on Pod `server-0`, we know that we can come back later and retrieve it from exactly the same Pod. Instead, if the Pod were to be managed by the regular Deployment controller, a Pod would be given a random name such as `server-1539155708-55dqs` or `server-1539155708-pkj2w`. Even if the client could remember such random names, there is no guarantee that the stored key/value pair will survive a deletion or scale up/down event.

The property of stable network identity is of fundamental importance to apply sharding mechanisms that allow to scale the data across multiple compute and data resources. Sharding means that a given data set is broken down into chunks, each of which ends up in different servers. The criteria for breaking a data set into chunks may be vary depending on the type of data at hand and the fields or attributes that are more evenly distributed; for example, for a contact entity, the name and surname are a good attributes, whereas the gender is not.

Let us suppose that our data set consists of the keys a, b, c, and d. How do we evenly distribute each letter across our three servers? The simplest solution is to apply modulo arithmetic. By getting the ASCII decimal code for each letter and getting the modulo for the number of servers, we obtain a cheap sharding solution, as shown in Table 9-1.

Table 9-1. *Applying the modulo operator to ASCII letters*

Key	Decimal	Modulo	Server
a	97	97 % 3 = 1	server-1
b	98	98 % 3 = 2	server-2
c	99	99 % 3 = 0	server-0
d	100	100 % 3 = 1	server-1

The actual hashing algorithm in a production-grade backing store like Cassandra or Memached will be more elaborate and use a consistent hashing algorithm—so that the keys, in the aggregate, do not land on different servers when new servers are added or removed—but the fundamentals remain the same.

The key insight here is that clients require servers to have a stable network identity since they will allocate a subset of keys to each of them. This is precisely one of the key features that distinguish StatefulSets from Deployments.

Headless Service

Stable network identity is not very useful if the server Pods cannot be reached by clients in the first place. Client access to StatefulSet-controlled Pods is different than that applicable to Deployment-controlled Pods because load balancing across random Pod instances is inappropriate; clients need to access discrete Pods directly. However, the specific Node and IP address in which a Pod will land can only be determined at runtime, so a discovery mechanism is still necessary.

The solution used in the case of StatefulSets still lies in the Service controller (Chapter 4), except that it is configured to provide a so-called *headless service*. A headless service provides just a DNS entry and proxy rather than a load balancer, and it is set up using a regular Service manifest except that `service.spec.clusterIP` attribute is set to `None`:

```
# wip/service.yaml
apiVersion: v1
kind: Service
metadata:
  name: server
  labels:
    app: server
spec:
  ports:
  - port: 80
  clusterIP: None
  selector:
    app: server
```

Let us apply the service.yaml manifest as follows:

```
# Assume wip/server.yaml has been applied first

$ cd wip
$ kubectl apply -f service.yaml
service/server created
```

The DNS entry created for the server service will provide a DNS SRV record for every Pod that is running and ready. We can use the nslookup command to obtain such DNS SRV records:

```
$ kubectl run --image=alpine --restart=Never \
    --rm -i test \
    -- nslookup server

10.36.1.7 server-0.server.default.svc.cluster.local
10.36.1.8 server-2.server.default.svc.cluster.local
10.36.2.7 server-1.server.default.svc.cluster.local
```

A Smart Client for Our Primitive Key/Value Store

So far, we have managed to run multiple replicas of a key/value store using the StatefulSet controller and make each Pod endpoint reachable by an interested consumer application using the headless service. We have also explained before that scalability is managed by clients, not by servers themselves—in the multi-master paradigm, we have chosen to explore in this chapter. Now, let us create a smart client that allows us to get further insight into the behavior of a StatefulSet.

We will describe the key aspects of our client, before presenting the entire source code at the end. The first thing that the client needs to understand is the exact set of servers with which it will interact and whether it should run in read-only mode or not:

```
if len(sys.argv) < 2:
  print('client.py SRV_1[:PORT],SRV_2[:PORT],' +
        '... [readonly]')
  sys.exit(1)

# Process input arguments
servers  = sys.argv[1].split(',')
readonly = (True if len(sys.argv) >= 3
            and sys.argv[2] == 'readonly' else False)
```

A typical invocation for a three-replica server would be as follows—do not run this example; it is for illustration only:

```
./server.py \
  server-0.server,server-1.server,server-2.server
```

Our client saves a set of keys consisting of the English alphabet across a defined number of servers, now stored in the servers variable. When the client starts, it will first print the complete alphabet underlined by a dotted line and the server number that has been selected to store each key by obtaining the modulo for the ASCII code of each key:

```
# Print alphabet and selected server for each letter
sn = len(servers)
print(' ' * 20 + string.ascii_lowercase)
print(' ' * 20 + '-' * 26)
print(' ' * 20 + ".join(
                   map(lambda c: str(ord(c) % sn),
                   string.ascii_lowercase)))
print(' ' * 20 + '-' * 26)
```

This code snippet would produce the following output if three servers are defined as arguments, which in turn become a list in the variable `servers`:

```
abcdefghijklmnopqrstuvwxyz
--------------------------
12012012012012012012012012
--------------------------
```

The meaning of having each letter of the alphabet vertically aligned with the digits 0, 1, or 2 is that the keys will be distributed as indicated in Table 9-2.

Table 9-2. *Alphabet letters and assigned servers*

Keys	Server
`[c,f,i,l,o,r,u,x]`	server-0.server
`[a,d,g,j,m,p,s,v,y]`	server-1.server
`[b,e,h,k,n,q,t,w,z]`	server-2.server

Now, following this initialization, the client will run in a loop checking whether each key is found in the matching server, and if not, it will try to insert it, unless it is running in read-only mode:

```
# Iterate through the alphabet repeatedly
while True:
  print(str(datetime.datetime.now())[:19] + ' ',
        end=")
  hits = 0
  for c in string.ascii_lowercase:
    server = servers[ord(c) % sn]
    try:
      r = curl('http://' + server + '/load/' + c)
      # Key found and value match
      if r == c:
        hits = hits + 1
        print('h',end=")
      # Key not found
      elif r == '_key_not_found_':
        if readonly:
          print('m',end=")
        else:
          # Save Key/Value (not read only)
          r = curl('http://' + server +
                   '/save/' + c + '/' + c)
          print('w',end=")
      # Value mismatch
      else:
        print('x',end=")
    except urllib.error.HTTPError as e:
          print(str(e.getcode())[0],end=")
    except urllib.error.URLError as e:
          print('.',end=")
  print(' | hits = {} ({:.0f}%)'
        .format(hits,hits/0.26))
  time.sleep(2)
```

The outcome of each key will be displayed using a single status letter. The following is an example of running the client for the first time, assuming that the StatefulSet and its headless Service are up and running:

```
      abcdefghijklmnopqrstuvwxyz
      --------------------------
      12012012012012012012012012
      --------------------------
03:51 wwwwwwwwwwwwwwwwwwwwwwwwww | hits = 0 (0%)
03:53 hhhhhhhhhhhhhhhhhhhhhhhhhh | hits = 26 (100%)
03:55 hhhhhhhhhhhhhhhhhhhhhhhhhh | hits = 26 (100%)
...
```

The meaning of each letter, under each of the alphabet's letters, including `w` and `h` shown in this output, is presented in Table 9-3.

Table 9-3. *Meaning of each status letter under an alphabet's letter column*

Letter	Description
`w`	Write: The key was not found in the server, so it was saved.
`h`	Hit: The key was found in the server.
`m`	Miss: The key was not found, and it will not be saved (read only).
`x`	Exception: The key was found, but it does not match its value.
`.`	Server inaccessible or network failure.
`0-9`	A HTTP server error was returned. For example, 503 would be 5.

After having examined each code snippet closely, we now present the complete client Python script:

```
#!/usr/bin/python3
# client.py
import string
import time
import sys
import urllib.request
import urllib.error
import datetime

if len(sys.argv) < 2:
  print('client.py SRV_1[:PORT],SRV_2[:PORT],' +
        '... [readonly]')
  sys.exit(1)

# Process input arguments
servers  = sys.argv[1].split(',')
readonly = (True if len(sys.argv) >= 3
            and sys.argv[2] == 'readonly' else False)

# Remove boilerplate from HTTP calls
def curl(url):
  return urllib.request.urlopen(url).read().decode()

# Print alphabet and selected server for each letter
sn = len(servers)
print(' ' * 20 + string.ascii_lowercase)
print(' ' * 20 + '-' * 26)
print(' ' * 20 + ".join(
                   map(lambda c: str(ord(c) % sn),
                       string.ascii_lowercase)))
print(' ' * 20 + '-' * 26)

# Iterate through the alphabet repeatedly
while True:
  print(str(datetime.datetime.now())[:19] + ' ',
        end=")
  hits = 0
  for c in string.ascii_lowercase:
    server = servers[ord(c) % sn]
    try:
      r = curl('http://' + server + '/load/' + c)
      # Key found and value match
      if r == c:
        hits = hits + 1
        print('h',end=")
      # Key not found
      elif r == '_key_not_found_':
        if readonly:
          print('m',end=")
        else:
          # Save Key/Value (not read only)
          r = curl('http://' + server +
                   '/save/' + c + '/' + c)
          print('w',end=")
      # Value mismatch
      else:
        print('x',end=")
    except urllib.error.HTTPError as e:
          print(str(e.getcode())[0],end=")
    except urllib.error.URLError as e:
          print('.',end=")
  print(' | hits = {} ({:.0f}%)'
        .format(hits,hits/0.26))
  time.sleep(2)
```

In addition to `server.py`, which we had defined earlier, we must also add `client.py` to the ConfigMap named `scripts`. As a result, we define a new file called `wip/configmap2.sh`:

```
#!/bin/sh
# wip/configmap2.sh
kubectl delete configmap scripts \
  --ignore-not-found=true
kubectl create configmap scripts \
  --from-file=server.py --from-file=../client.py
```

Finally, we need a Pod manifest to run the client and provide it with the intended StatefulSet Pod names:

```
# client.yaml
apiVersion: v1
kind: Pod
metadata:
  name: client
spec:
  restartPolicy: Never
  containers:
    - name: client
      image: python:alpine
      args:
      - bin/sh
      - -c
      - "python -u /var/scripts/client.py
        server-0.server,server-1.server,\
        server-2.server"
      volumeMounts:
        - name: scripts
          mountPath: /var/scripts
  volumes:
    - name: scripts
      configMap:
        name: scripts
```

As a final experiment, it is interesting to reset our environment from scratch, launch the client first, and observe its initial behavior *in the absence* of the `servers` StatefulSet:

```
# Clean up the environment first

$ cd wip
$ ./configmap2.sh
configmap "scripts" deleted
configmap/scripts created

# Run the client
$ kubectl apply -f ../client.yaml
pod/client created

# Query the client's logs
$ kubectl logs -f client

      abcdefghijklmnopqrstuvwxyz
      --------------------------
      12012012012012012012012012
      --------------------------
00:21 .......................... | hits = 0 (0%)
00:24 .......................... | hits = 0 (0%)
00:26 .......................... | hits = 0 (0%)
00:28 .......................... | hits = 0 (0%)
```

The `.` (dot) character means that there are no servers accessible for any of the keys. Let us go ahead and start the server and its associated headless service while watching the client's log on a separate window:

```
# Note: we are still under the wip directory

$ kubectl apply -f server.yaml
statefulset.apps/server created

$ kubectl apply -f service.yaml
service/server created
```

The client will automatically start saving keys (`w`) to each server that comes up:

```
      abcdefghijklmnopqrstuvwxyz
      --------------------------
      12012012012012012012012012
      --------------------------
...   ...                         ...
04:14 .......................... | hits = 0 (0%)
04:18 .......................... | hits = 0 (0%)
04:21 .......................... | hits = 0 (0%)
04:23 w..w..w..w..w..w..w..w..w. | hits = 0 (0%)
04:25 h..h..h..h..h..h..h..h..h. | hits = 9 (35%)
04:27 h.wh.wh.wh.wh.wh.wh.wh.wh. | hits = 9 (35%)
04:29 hwhhwhhwhhwhhwhhwhhwhhwhhw | hits = 17 (65%)
04:31 hhhhhhhhhhhhhhhhhhhhhhhhhh | hits = 26 (100%)
04:33 hhhhhhhhhhhhhhhhhhhhhhhhhh | hits = 26 (100%)
```

This is how to interpret what went on, line by line:

```
04:23 w..w..w..w..w..w..w..w..w. | hits = 0 (0%)
```

Server `server-1.server` is the first to come up which results in the keys `[a,d,g,j,m,p,s,v,y]` being saved (`w`) to it. Servers `server-0.server` and `server-2.server` are not yet accessible:

```
04:25 h..h..h..h..h..h..h..h..h. | hits = 9 (35%)
```

Server server-1.server already contains the keys [a,d,g,j,m,p,s,v,y] which result in a hit (h). Servers server-0.server and server-2.server are not yet accessible.

```
04:27 h.wh.wh.wh.wh.wh.wh.wh.wh. | hits = 9 (35%)
```

Now server server-2.server has come up, and the keys [b,e,h,k,n,q,t,w,z] have been saved to it. Only server-0.server remains inaccessible now:

```
04:29 hwhhwhhwhhwhhwhhwhhwhhwhhw | hits = 17 (65%)
```

Server server-0 has finally come up, and the keys [c,f,i,l,o,r,u,x] have been saved to it:

```
04:31 hhhhhhhhhhhhhhhhhhhhhhhhhh | hits = 26 (100%)
```

The key set is now distributed across all three servers.

Please note that this example seems to violate the principle of sequential Pod creation. If we watch the Pod creation behavior, using kubectl get pods -w, we will see that the principle still applies. However, if the Pods start fast enough, a combination of the time at which the readiness probe becomes active, plus DNS caching, may result in a seemingly unorderly behavior from a client perspective. If there is a requirement for clients themselves to experience guaranteed sequentially ordered Pod creation, then we need to add some delay to allow for the readiness probe to kick in and for DNS caches to be flushed and/or refreshed.

Controlling the Creation and Termination of a Backing Store

As we have explained in the introduction of this chapter, the StatefulSet controller can't make broad assumptions about the specific nature of a backing store since each high scaling and availability paradigm together with the choices taken by each product's technical design and constraints (e.g., MySQL vs. MongoDB) result in an overwhelming number of possibilities. For example:

- Registering a worker with its supervisor
- Registering replicas with a controller such as ZooKeeper
- Replicating the master's data set with the workers
- Waiting for the last member of the cluster to be up and running before marking the antecedent replicas as *ready*
- Leader reelection (and publication of election results to clients)
- Recalculating hashes

Having said this, we can reason about the life cycle of a StatefulSet and understand what *opportunities* the Kubernetes administrator has to exert control. The main cooking ingredients that the StatefulSet controller gives us to exert said control are

1. The guarantee that Pod creation and termination is orderly and that their identity is predictable. For example, if the value of `$HOSTNAME` is `server-2`, we can expect that `server-0`, and `server-1` would have been created first.
2. The chance of reaching other Pods in the same set using the headless service: this is connected to the first point; if we are inside `server-2`, the headless server will have published the DNS entries to connect to `server-0` and `sever-1`.

3. The opportunity to run one or more bespoke initialization containers before the main containers are run. These are defined at `statefulset.spec.template.spec.initContainers` in the StatefulSet manifest. For example, we may want to use the initialization container to import a SQL script from an external source before the official backing store's container is run. The official backing store's Docker image may be vendor provided, and it might not a good idea to "pollute it" with custom code.

4. The opportunity to run commands when each of the main containers initializes and when they are targeted for termination using the *life cycle hooks* such as `postStart` and `preStop` declared at `statefulset.spec.template.spec.containers.lifecycle`. We will use this feature for our primitive key/value store in the sections ahead.

5. The opportunity to catch the SIGTERM Linux signal which is sent to each of the first process, of each of the main containers, upon termination. When containers receive the SIGTERM signal, they can run graceful shutdown code for the amount of seconds defined by the `statefulset.spec.template.spec.terminationGracePeriodSeconds` property.

6. The Pod's default liveness and readiness probes (see Chapter 2) which allow deciding when a given Pod should be made available to the world.

Order of Pod Life Cycle Events

In the last section, we have seen that there are multiple opportunities to run code when creating and destroying a backing store's Pods, but we have not discussed when it is appropriate to apply each of the various options. For example, should we set up the initial tables for a MySQL database using *Init Containers* or the *PostStart hook*? To answer these questions, it is necessary to understand the order of events that take place throughout a Pod's life cycle. For this purpose, we will first cover the *creation* life cycle and, then, the *termination* one.

Note This section assumes a variety of general concepts that may be new to the reader:

- **Grace Period**: The time that a process is allowed to run graceful shutdown tasks before it is forcefully terminated.
- **Graceful Startup and Shutdown**: The execution of "tear up" and "tear down" (respectively) tasks that help minimize disruption and prevent leaving systems, processes, or data in an inconsistent state.
- **Hook**: A placeholder that allows the insertion of a trigger, script, or other code execution mechanism.
- **Sigkill**: A signal that is sent to a process to cause it to terminate immediately. This signal cannot be normally caught or ignored.

- **SIGTERM**: A signal that is sent to a process to request its termination. This signal can be caught and interpreted or ignored by the process. It is helpful as part of a graceful shutdown strategy to implement process-level cleanup code.
- **Steady State**: A state when its characterizing variables are unchanging in time.

The creation life cycle involves the launching of a Pod (either because the StatefulSet is being created for the first time or due to a scaling event). This means taking a Pod from nonexistence to a *Running* status. Table 9-4 shows the order (*C0…C4, S*) of the most relevant life cycle events where the ***C*** stands for Creation and ***S*** for Steady State. The Steady State is that in which the Pod is not incurring life cycles changes connected neither with startup nor shutdown processes. In the second row, P stands for Pending, whereas R stands for Running.

Table 9-4. *StatefulSet-controlled Pod creation life cycle events*

Description	**C0**	**C1**	**C2**	**C3**	**C4**	**S**
Pod Status	P	P	R	R	R	R
Init Container Ran		●				
Main Container Ran			●			
PostStart Hook Run			●			
Liveness Probe Run				●		
Readiness Probe Run				●		
This Endpoint Published					●	
N-1 Endpoints Published	●	●	●	●	●	●
N+1 Endpoints Published						●

Note that Table 9-4 represents a rough guideline and some critical considerations apply:

- The *Pod Status* is the official Pod phase as provided by the `pod.status.phase` property. While Pending (P) and Running (R) are meant to be the official phases, the `kubectl get pod` command may show an intermediate state such as *Init* between *C0* and *C1* and *PodInitializing* between *C1* and *C2*.
- If the *Init Container* fails by quitting with a nonzero exit code, the main container will not be executed.
- The *Main Container* and the command associated with the *PostStart Hook* run in parallel, and Kubernetes does not guarantee which one will be run first.
- The *Liveness and Readiness Probes* start running sometime before the main container(s) start.
- The applicable *Pod's endpoint* is *published* by the Service controller sometime after the internal readiness probe is positive, but DNS time-to-live (TTL) settings (both at the server and client sides) and network propagation issues may delay the visibility of the Pod by other replicas and clients.
- The *N-1 Endpoints* (e.g., `server-0.server` and `server-1.server` if the reference Pod N is `server-2`) are *supposed* to be accessible because `server-2` is only initialized once `server-1` becomes *Ready*; however, they may have failed by the time they are queried or may be temporarily inaccessible. For this reason, bullet proof code should always ping and probe a dependent Pod and not make the blind assumption that it is necessarily up and running because of the sequential Pod creation guarantee.

- The *N+2 Endpoints* (e.g., `server-3.server` and `server-4.server` if the reference Pod N is `server-2`) will only be initialized after the current Pod becomes Ready. Therefore, if certain code needs to wait for future Pods to become available, they must be run as a main container and have a DNS probing/ping loop.

Let us now look at the termination Pod life cycle (Table 9-5) which starts at *T1* whenever a StatefulSet is scaled down or an individual Pod is deleted. The first column, ***S***, represents the Pod's steady state before the termination event is triggered.

Table 9-5. *StatefulSet-controlled Pod terminarion life cycle events*

Description	S	T1	T2	T3
Pod Status	R	T	T	T
Main Container Running	●	●	●	
PreStop Hook Run		●		
Grace Period Started		●		
Grace Period Ended				●
Main Container SIGTERM			●	●
Main Container SIGKILL				
This Endpoint Published	●			
N-1 Endpoints Published	●	●	●	●
N+1 Endpoints Published				

Again, rather noteworthy considerations apply when considering Table 9-5:

- The termination event at *T1* is “brutal” and does not leave as much graceful shutdown scope as one might need. In particular, the terminated Pod is knocked off of the headless service immediately; hence why the ● (dot) character on *This Endpoint Published* disappears at *T1* itself. While the underlying application may still remain accessible for those clients that have established by TCP connection beforehand, those that happen to query the DNS service, or endpoint controller, right after T1, will “lose sight” of the terminated Pod.
- The *PreStop Hook* and the *Grace Period* will start together at *T1*. The *SIGTERM* signal will be received by the main container at *T2,* after the PreStop hook completes but *before* the *Grace Period* ends. Whatever graceful shutdown code is required must complete within the allotted time as per the value of the `terminationGracePeriodSeconds` attribute.
- At *T3*, when the *Grace Period* ends, the main containers will be killed with no further chances for recovery or running mitigating code.

Implementing Graceful Shutdown Using Pod Life Cycle Hooks

In the last two sections, we have discussed the variety of options that are available to control the life cycle of a StatefulSet which is, in turn, the aggregate of the individual life cycle of each of its constituent Pods. Now, in this section, we are going back to our lab-orientated workflow and add a simple form of graceful shutdown to our primitive key/value store.

Our graceful shutdown consists of returning a 503 HTTP error whenever a Pod is terminating, as opposed to letting clients simply time out. Even though a Pod is knocked off the headless service DNS as soon as the termination event is acknowledged, clients that remember the IP address (and/or that have already established a TCP connection) before the DNS entry was last checked can exhibit erratic behavior if Pods time out all of a sudden without notice. As simple as this solution may appear, it may help a smart client to "back off" for a while, implement a *circuit breaker* pattern, and/or access the data from a different source.

For this purpose, we will check the presence of a file named _shutting_down_ before servicing any HTTP request:

```
if os.path.isfile(dataDir + '_shutting_down_'):
  return "_shutting_down_", 503
```

The preceding `if` statement, for indicating whether services are about to shut down, is now at the top of every Flask HTTP function in the updated `server.py` script. Please note that for brevity and to avoid distractions, the code listing of `server.py` presented earlier in this chapter does not show these two lines of code after the `save()`, `load()`, and `allKeys()` functions.

Now that we have a cheap graceful shutdown mechanism on the client side, we need to implement it on the server one. What we need to do here is create a `_shutting_down_` file as soon as a termination event is received and delete it when the Pod is started. The objective is to use the presence or absence of presence of this file to signify whether the server is about to shut down or not, respectively. To implement the creation and delete of this file, we will use the `preStop` and `postStart` pod life cycle hooks (see Tables 9-4 and 9-5), respectively:

```
# server.yaml
...
spec:
  template:
    spec:
      containers:
      - name: server
        lifecycle:
          postStart:
            exec:
              command:
              - /bin/sh
              - -c
              - rm -f /var/data/_shutting_down_
          preStop:
            exec:
              command:
              - /bin/sh
              - -c
              - touch /var/data/_shutting_down_
...
```

This snippet is included in the *new and final* `server.yaml` manifest, located directly under the chapter's root directory rather than `wip/`, in which we also set `terminationGracePeriodSeconds` to `10` so that it takes less time to observe termination behavior (the default is 30 seconds). For simplicity, we also append the Service manifest in a single YAML file, using the `---` (triple hyphen) YAML notation, so that we can create the final server with one command stroke only:

```
# Memory-based key/value store
# server.yaml
---
apiVersion: v1
kind: Service
metadata:
  name: server
  labels:
    app: server
spec:
  ports:
  - port: 80
  clusterIP: None
  selector:
    app: server
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: server
spec:
  selector:
    matchLabels:
      app: server
  serviceName: server
  replicas: 3
  template:
    metadata:
      labels:
        app: server
    spec:
      terminationGracePeriodSeconds: 10
      containers:
      - name: server
        image: python:alpine
        args:
        - bin/sh
        - -c
        - >-
          pip install flask;
          python -u /var/scripts/server.py 80
          /var/data
        ports:
        - containerPort: 80
        volumeMounts:
          - name: scripts
            mountPath: /var/scripts
          - name: data
            mountPath: /var/data
        lifecycle:
          postStart:
            exec:
              command:
              - /bin/sh
              - -c
              - rm -f /var/data/_shutting_down_
          preStop:
            exec:
              command:
              - /bin/sh
              - -c
              - touch /var/data/_shutting_down_
      volumes:
        - name: scripts
          configMap:
            name: scripts
        - name: data
          emptyDir:
            medium: Memory
```

Observing StatefulSet Failures

In the last section, we have implemented a graceful shutdown capability in `server.yaml` and `server.py`, by taking advantage of the `postStart` and `preStop` life cycle hooks. In this section, we will see this capability in action. Let us start by changing the current working directory to the chapter's root directory and running the newly defined files:

```
# clean up the environment first

$ ./configmap.sh
configmap/scripts created

$ kubectl apply -f server.yaml
service/server created
statefulset.apps/server created

$ kubectl apply -f client.yaml
pod/client created
```

Now that both the server and client objects have been created, we can tail the client's logs again:

```
$ kubectl logs -f client
      abcdefghijklmnopqrstuvwxyz
      --------------------------
      12012012012012012012012012
      --------------------------
24:41 wwwwwwwwwwwwwwwwwwwwwwwwww | hits = 0 (0%)
24:43 hhhhhhhhhhhhhhhhhhhhhhhhhh | hits = 26 (100%)
24:45 hhhhhhhhhhhhhhhhhhhhhhhhhh | hits = 26 (100%)
24:47 hhhhhhhhhhhhhhhhhhhhhhhhhh | hits = 26 (100%)
...
```

While keeping `kubectl logs -f client` running on a separate terminal window, we can now see the effects of deleting a Pod from the StatefulSet by issuing the `kubectl delete pod/server-1` command:

```
34:45 hhhhhhhhhhhhhhhhhhhhhhhhhh | hits = 26 (100%)
34:47 5hh5hh5hh5hh5hh5hh5hh5hh5h | hits = 17 (65%)
34:49 5hh5hh5hh5hh5hh5hh5hh5hh5h | hits = 17 (65%)
34:51 5hh5hh5hh5hh5hh5hh5hh5hh5h | hits = 17 (65%)
34:53 .hh.hh.hh.hh.hh.hh.hh.hh.h | hits = 17 (65%)
34:55 .hh.hh.hh.hh.hh.hh.hh.hh.h | hits = 17 (65%)
34:57 .hh.hh.hh.hh.hh.hh.hh.hh.h | hits = 17 (65%)
35:00 .hh.hh.hh.hh.hh.hh.hh.hh.h | hits = 17 (65%)
35:02 .hh.hh.hh.hh.hh.hh.hh.hh.h | hits = 17 (65%)
35:04 whhwhhwhhwhhwhhwhhwhhwhhwh | hits = 17 (65%)
35:06 hhhhhhhhhhhhhhhhhhhhhhhhhh | hits = 26 (100%)
```

In this log, we see that the client shows a digit `5` (in lieu of `503`) for the server we have just deleted between seconds `34:47` and `34:51`. This is the period during which both the server and the client can implement code to disengage in a non-impactful manner. Between `34:53` and `35:02`, the client is unable to reach server 1, as implied by the `.` (dot), until it finally manages to save the keys again at `35:04` as indicated by `w` (write). The client finally reports `h` (hit) on all servers at second at `35:06`.

In case the reader is wondering why the deleted server comes back from the dead automatically after a while, it is because the StatefulSet controller's responsibility is ensuring that the runtime specification matches the state declared in the manifest. Given that we have deleted a Pod, the StatefulSet control has taken corrective actions so that the number of declared replicas and number effective running replicas match.

Scaling Up and Down

The magic of zero-downtime scaling in the case of stateless Pods managed by the Deployment controller is more difficult to achieve in the case of StatefulSets. We will first discuss *scaling up* and then *scaling down*, since each have their own challenges.

A scale up event results in the appearance of new Pods that existing clients may not be aware of, unless they check the DNS SRV records frequently and update themselves in turn. However, this would upset the hashing algorithm and result in a cascading number of misses in a backing store like our primitive key/value one.

A scale down event is more destructive than a scale up one, not simply because we are reducing the number of data-holding replicas but because we only have the graceful period to do whatever data repartitioning is necessary before Kubernetes terminates our Pods in a mercifulness manner, based on the life cycle technical contract.

The conclusion is that, if our aim is to reduce disruption to a bare minimum, we need to put extra thought and steps before a `kubectl scale` command is issued against a StatefulSet.

The challenge that we have is, essentially, that we have to recalculate the key hashes whenever the number of servers changes, regardless of whether it is a scale up or down event. Table 9-6 presents the resulting selected servers for 2 and 3, replicas, for the keys `a`, `b`, `c`, and `d` as an example.

Table 9-6. *The effects of applying modulo 2 and 3 to a, b, c, and d*

Key	Dec	Modulo	N = 2	N = 3
a	97	97 % N	server-1	server-1
b	98	98 % N	server-0	server-2
c	99	99 % N	server-1	server-0
d	100	99 % N	server-0	server-1

Hence, if the number of servers are scaled down from three replicas down to two, we first should first distribute the key value pairs stored in servers 0-2 to only servers 0-1, and vice versa, if scaling up from two replicas up to three. We have created the simplest possible program to capture this process in a script called `rebalance.py`. This Python script takes an AS-IS server list and a TO-BE server list to perform the necessary repartition of key/value pairs:

```
#!/usr/bin/python3
# rebalance.py
import sys
import urllib.request

if len(sys.argv) < 3:
  print('rebalance.py AS_IS_SRV_1[:PORT],' +
        'AS_IS_SRV_2[:PORT]... ' +
        'TO_BE_SRV_1[:PORT],TO_BE_SRV_2[:PORT],...')
  sys.exit(1)

# Process arguments
as_is_servers = sys.argv[1].split(',')
to_be_servers = sys.argv[2].split(',')
```

```
# Remove boilerplate from HTTP calls
def curl(url):
  return urllib.request.urlopen(url).read().decode()

# Copy key/vale pairs from AS IS to TO BE servers
urls = []
for server in as_is_servers:
  keys = curl('http://' + server +
              '/allKeys').split(',')
  print(server + ': ' + str(keys))
  for key in keys:
    print(key + '=',end=")
    value = curl('http://' + server +
                 '/load/' + key)
    sn = ord(key) % len(to_be_servers)
    target_server = to_be_servers[sn]
    print(value + ' ' + server +
          '->' + target_server)
    urls.append('http://' + target_server +
                '/save/' + key + '/' + value)
for url in urls:
  print(url,end=")
  print(' ' + curl(url))
```

Like in the case of server.py and client.py, the script is uploaded using a ConfigMap:

```
#!/bin/sh
# configmap.sh
kubectl delete configmap scripts \
  --ignore-not-found=true
kubectl create configmap scripts \
  --from-file=server.py \
  --from-file=client.py --from-file=rebalance.py
```

As explained a few moments ago, scaling is not trivial and needs to be carefully controlled; therefore, we will contemplate a Pod manifest to perform a scale down migration from three to two replicas named `rebalance-down.yaml`:

```
# rebalance-down.yaml
# Reduce key/store cluster to 2 replicas from 3
apiVersion: v1
kind: Pod
metadata:
  name: rebalance
spec:
  restartPolicy: Never
  containers:
  - name: rebalance
    image: python:alpine
    args:
    - bin/sh
    - -c
    - "python -u /var/scripts/rebalance.py
      server-0.server,server-1.server,\
      server-2.server
      server-0.server,server-1.server"
    volumeMounts:
      - name: scripts
        mountPath: /var/scripts
  volumes:
  - name: scripts
    configMap:
      name: scripts
```

Likewise, we have also defined rebalance-up.yaml to scale up from two to three replicas:

```
# rebalance-up.yaml
# Scale key/store cluster to 3 replicas from 2
apiVersion: v1
kind: Pod
metadata:
  name: rebalance
spec:
  restartPolicy: Never
  containers:
  - name: rebalance
    image: python:alpine
    args:
    - bin/sh
    - -c
    - "python -u /var/scripts/rebalance.py
      server-0.server,server-1.server
      server-0.server,server-1.server,\
      server-2.server"
    volumeMounts:
      - name: scripts
        mountPath: /var/scripts
  volumes:
  - name: scripts
    configMap:
      name: scripts
```

Now that we have defined our rebalancing script and our manifest to scale up and down our cluster, we can clean up the environment and deploy the server and client again so that we do not have interference from previous examples in our Kubernetes cluster:

```
# clean up the environment first

$ ./configmap.sh
configmap/scripts created

$ kubectl apply -f server.yaml
service/server created
statefulset.apps/server created

$ kubectl apply -f client.yaml
pod/client created
```

Scaling Down

In the last section, we have discussed the fact that scaling is not trivial and that bluntly issuing a `kubectl scale` command may cause unnecessary disruption. In this section, we will see how to scale down our primitive key/value store in an orderly fashion.

We will go through the following steps:

1. Run the new TO-BE client (targeting two replicas) in read-only mode—so that we can observe the results of the migration.

2. Stop the read/write three-replica client Pod.

3. Migrate the key/value pairs from the three Pod cluster to the two Pod one.

4. Scale down the three-replica cluster down to two replicas.

Let us get started by defining a Pod manifest called client-ro-2.yaml to run client.py targeting only server-0 and server-1 in read-only mode:

```
# client-ro-2.yaml
apiVersion: v1
kind: Pod
metadata:
  name: client-ro-2
spec:
  restartPolicy: Never
  containers:
    - name: client-ro-2
      image: python:alpine
      args:
      - bin/sh
      - -c
      - >
        python -u /var/scripts/client.py
        server-0.server,server-1.server readonly
      volumeMounts:
        - name: scripts
          mountPath: /var/scripts
  volumes:
    - name: scripts
      configMap:
        name: scripts
```

Let us now apply it and follow its logs:

```
$ kubectl apply -f client-ro-2.yaml
pod/client-ro-2 created
```

```
$ kubectl logs -f client-ro-2

      abcdefghijklmnopqrstuvwxyz
      --------------------------
      10101010101010101010101010
      --------------------------
33:33 hmmmmhhmmmmhhmmmmhhmmmmhhm | hits = 9 (35%)
33:35 hmmmmhhmmmmhhmmmmhhmmmmhhm | hits = 9 (35%)
33:37 hmmmmhhmmmmhhmmmmhhmmmmhhm | hits = 9 (35%)
```

Note that in a real-world scenario, we would only run the client targeting the new smaller StatefulSet after the repartitioning is complete. However, here we do it sooner to observe the effect of the migration in real time. Note also that only servers `1` and `0` are now included and that most key lookups result in misses.

Now comes the delicate part which is reading the keys and recalculating the new hash for the smaller, two-replica cluster which is the job of the `reblance-down.yaml` manifest, which in turn executes `rebalance.py`. Before we do this, we must stop the client Pod from performing writes to the *soon to be deprecated,* three-replica cluster first:

```
$ kubectl delete --grace-period=1 pod/client
pod "client" deleted
```

```
$ kubectl apply -f rebalance-down.
yaml                            pod/rebalance created
```

```
$ kubectl logs -f rebalance
server-0.server:
['x', 'u', 'r', 'o', 'l', 'i', 'f', 'c']
```

```
x=x server-0.server->server-0.server
u=u server-0.server->server-1.server
...
server-1.server:
['y', 'v', 's', 'p', 'm', 'j', 'g', 'd', 'a']
y=y server-1.server->server-1.server
v=v server-1.server->server-0.server
...
server-2.server:
['z', 'w', 't', 'q', 'n', 'k', 'h', 'e', 'b']
z=z server-2.server->server-0.server
w=w server-2.server->server-1.server
...
```

By the time that pod/rebalance completes, the window running kubectl log -f client-ro-2 will show hits for the entire key set:

```
28:45 hmmmmhhmmmmhhmmmmhhmmmmhhm | hits = 9 (35%)
28:47 hmmmmhhmmmmhhmmmmhhmmmmhhm | hits = 9 (35%)
28:49 hmmmmhhmmmmhhmmmmhhmmmmhhm | hits = 9 (35%)
28:51 hmhhhhhhhhhhhhhhhhhhhhhhhh | hits = 25 (96%)
28:53 hhhhhhhhhhhhhhhhhhhhhhhhhh | hits = 26 (100%)
28:55 hhhhhhhhhhhhhhhhhhhhhhhhhh | hits = 26 (100%)
28:57 hhhhhhhhhhhhhhhhhhhhhhhhhh | hits = 26 (100%)
```

At this point, we can scale down the cluster to two replicas, after which, it is possible to launch clients in read/write mode against the, *now smaller*, two-replica cluster:

```
$ kubectl scale statefulset/server --replicas=2
statefulset.apps/server scaled
```

Scaling Up

In the last section, we had just scaled down our cluster down to two replicas from three. To observe a scaling up operation in action, we will work in a similar fashion as before, by first launching a read-only client targeting three replicas defined in client-ro-3.yaml:

```
# client-ro-3.yaml
apiVersion: v1
kind: Pod
metadata:
  name: client-ro-3
spec:
  restartPolicy: Never
  containers:
    - name: client-ro-3
      image: python:alpine
      args:
      - bin/sh
      - -c
      - "python -u /var/scripts/client.py
        server-0.server,server-1.server,\
        server-2.server readonly"
      volumeMounts:
        - name: scripts
          mountPath: /var/scripts
  volumes:
    - name: scripts
      configMap:
        name: scripts
```

When we run this client, we expect to see a failure on `server-2`, denoted by `.` (dot). This is exactly what we expect because `server-2` is not running yet:

```
$ kubectl apply -f client-ro-3.yaml
pod/client-ro-3 created
```

```
$ kubectl logs -f client-ro-3

      abcdefghijklmnopqrstuvwxyz
      --------------------------
      12012012012012012012012012
      --------------------------
43:57 h.hh.hh.hh.hh.hh.hh.hh.hh. | hits = 17 (65%)
43:59 h.hh.hh.hh.hh.hh.hh.hh.hh. | hits = 17 (65%)
44:01 h.hh.hh.hh.hh.hh.hh.hh.hh. | hits = 17 (65%)
44:03 h.hh.hh.hh.hh.hh.hh.hh.hh. | hits = 17 (65%)
```

Since write-enabled clients are not expected to be aware of a new replica (`client.yaml` is not assumed to be running), it is safe to scale the cluster up to three replicas without further ado:

```
$ kubectl scale statefulset/server --replicas=3
statefulset.apps/server scaled
```

Right after, we should observe on the window running `kubectl logs -f client-ro-3` that the server failures denoted by a `.` (dot) turn into a miss denoted by a letter `m`:

```
...
49:54 h.hh.hh.hh.hh.hh.hh.hh.hh. | hits = 17 (65%)
49:56 h.hh.hh.hh.hh.hh.hh.hh.hh. | hits = 17 (65%)
49:58 h.hh.hh.hh.hh.hh.hh.hh.hh. | hits = 17 (65%)
50:00 hmhhmhhmhhmhhmhhmhhmhhmhhm | hits = 17 (65%)
50:02 hmhhmhhmhhmhhmhhmhhmhhmhhm | hits = 17 (65%)
50:06 hmhhmhhmhhmhhmhhmhhmhhmhhm | hits = 17 (65%)
...
```

So far, so good; we can now repartition the key/value pairs to a three-replica cluster by applying `rebalance-up.yaml`:

```
$ kubectl delete pod/rebalance
pod "rebalance" deleted

$ kubectl apply -f rebalance-up.yaml
pod/rebalance created
```

Whenever the terminal window in which we are following `client-ro-3`'s logs shows the misses turning into hits, we would have successfully scaled up the cluster:

```
53:24 hmhhmhhmhhmhhmhhmhhmhhmhhm | hits = 17 (65%)
53:26 hmhhmhhmhhmhhmhhmhhhhhmhhh | hits = 19 (73%)
53:28 hhhhhhhhhhhhhhhhhhhhhhhhhh | hits = 26 (100%)
53:30 hhhhhhhhhhhhhhhhhhhhhhhhhh | hits = 26 (100%)
53:32 hhhhhhhhhhhhhhhhhhhhhhhhhh | hits = 26 (100%)
```

At this point, it is safe to enable read/write clients against the three-replica cluster.

Note Kubernetes refers to the StatefulSet's Pods as *replicas* in the sense that we use the `replicas` attribute and replica-semantics when using the `kubectl scale` command. However, StatefulSet's replicas are not necessarily verbatim stateless copies as they typically are in the case of Deployments. From a logical perspective, given that we are using each StatefulSet's Pod instance to store a subset of a data set, it helps to think of them as "partitions."

Conclusions on Scaling Up and Down Operations

The scaling up and down operations demonstrated, in a primitive and rather manual fashion, the problem of rehashing keys whenever the cluster size changes and data chunks must be rearranged into a different number of servers.

Most advanced off-the-shelf backing stores such as MongoDB shield the users from the kind of manual operations that we have just observed by implementing asynchronous replication algorithms. Why bother then? Because as simplistic as our primitive key/value store may be, it helps getting intuition into the trade-offs that off-the-shelf solutions take to provide the illusion of nearly zero-downtime scaling.

In this regard, our primitive key/value store is scalable but not highly available, not because the data is stored in memory (which we will address before the end of this chapter) but because a single replica failure results in the inaccessibility of data. Cassandra, for instance, is not only scalable using an approach similar to the hashing scheme used in this text, but it is also highly available: it may be configured so that the same data is written to two or more Nodes before it is considered "persisted."

The complication introduced by backing stores that are potentially highly available (in addition to highly scalable) is that they introduce the challenge of handling eventual consistency. For example, we could easily modify our client so that a key is saved to two or more replicas using a replication scheme such as *`(totalReplicaCount + 1) % totalReplicaCount`*. However, whenever two keys are read and their values differ, the client needs to make a call as to which one to consider. In the case of our primitive key/value store, we could easily modify it, to provide a timestamp, so that the client can take the most recent one as valid.

Proper Statefulness: Persistence to Disk

Until now, we have treated our key/value store as an in-memory cache that loses data as soon as a replica sneezes. This was on purpose because all of the Statefulness properties we have discussed so far are relatively orthogonal to disk persistence:

- Stable network identity
- Ordered Pod creation
- Service discovery and headless services
- Scaling strategies

Furthermore, we have decided to make the difference between RAM and disk storage a matter of simply replacing the underlying implementation of the volume `data`. All of the examples shown so far rely on the RAM-based file system, as follows:

```
# server.yaml
    ...
    volumes:
      - name: data
        emptyDir:
          medium: Memory
```

The path of low resistance to disk persistence would be the use of the `hostPath` volume type. The `hostPath` volume type allows storing data directly on the node's file system; however, this approach is problematic, since it only works if Pods are always scheduled to run on the same Nodes. Another issue is that the Node-level storage is not meant to be persistent: even if Pods were to be scheduled to the same Nodes, the Node's file system is not guaranteed to survive a Node crash or restart.

As the reader may have guessed, what we need is *attached network storage* whose life cycle is independent from that of the Kubernetes worker Nodes. In GCP, this is simply a matter of creating a "Persistent Disk" which can be achieved by issuing a single command:

```
$ gcloud compute disks create my-disk --size=1GB
Created
NAME     ZONE            SIZE_GB  TYPE         STATUS
my-disk  europe-west2-a  1        pd-standard  READY
```

We can then associate the `data` volume with the GCP disk named `my-disk` in the StatefulSet manifest:

```
# server.yaml
    ...
    volumes:
      - name: data
        gcePersistentDisk:
          pdName: my-disk
          fsType: ext4
```

The limitation with the preceding approach is that only one single Pod can have read/write access to `my-disk`. All other Pods may only have read-only access. If we try the preceding by altering the `volumes` declaration in `server.yaml`, we will see that only `server-0` will start successfully but `server-1` will fail (and thus `server-2` will not be scheduled). This is because only one Pod (`server-0`) can have read/write access to the Persistent Disk `my-disk`.

The multi-master scheme employed by our key/value store requires all replicas (and thus, Pods) to have full read/write access. Moreover, the point of a scalable system is that data is spread across multiple disks rather than stored in a single central disk that is accessed by multiple servers. Ideally, we would need to create a separate disk for each replica. Something along the lines of:

```
# Example only, don't run these commands
$ gcloud compute disks create my-disk-server-0
$ gcloud compute disks create my-disk-server-1
$ gcloud compute disks create my-disk-server-2
```

Such an approach would involve planning in advance the number of disks required for a given number of replicas. What if Kubernetes could somewhat issue the preceding `gcloud compute disks create` commands (or use the underlying API), on our behalf, for each Pod that requires persistent storage? Good news, it can! Welcome to Persistent Volume Claims.

Persistent Volume Claims

Persistent Volume Claims can be understood as a mechanism to allow Kubernetes to create disks on demand, based on the identity of each Pod, so that if a Pod crashes or gets rescheduled, a 1:1 link (*bound* in the Kubernetes jargon) is maintained between each Pod and its associated volume. No matter on which Node a Pod "wakes up," Kubernetes will always attach its corresponding *bound* volume, for `server-0`, `disk-0`; for `server-1`, `disk-1`; etc.

While Kubernetes may be running in an environment in which there are multiple storage arrays capable of granting volumes such as `disk-0`, and `disk-1`, such storage arrays also vary from cloud vendor to cloud vendor. In AWS, for instance, this block storage capability is called Amazon Elastic Storage (EBS) and is implemented differently than GCP Persistent Disk. The question is, how does Kubernetes know whom to ask for a volume? Well, the storage array (or equivalent) capability is incarnated in Kubernetes as an object called a *StorageClass*.

A Persistent Volume Claim is performed against a given StorageClass. Google Kubernetes Engine (GKE) provides an out-of-the-box StorageClass called `standard`:

```
$ kubectl get storageclass
NAME                 PROVISIONER            AGE
standard (default)   kubernetes.io/gce-pd   40m
```

```
$ kubectl describe storageclass/standard
Name:                  standard
IsDefaultClass:        Yes
Annotations:           storageclass.*.kubernetes.io/*
Provisioner:           kubernetes.io/gce-pd
Parameters:            type=pd-standard
AllowVolumeExpansion:  <unset>
MountOptions:          <none>
ReclaimPolicy:         Delete
VolumeBindingMode:     Immediate
Events:                <none>
```

The `standard` StorageClass is the one used—unless otherwise specified—whenever GKE disks (*volumes* in Kubernetes) are created on demand. A StorageClass has a sort of driver that allows Kubernetes to orchestrate the creation of disks (or block devices in general) without requiring the administrator to issue manual commands to an external storage interface such as `gcloud compute disk create`.

It takes little extra code to have our StatefulSet manifest request disks to the standard StorageClass. It is a matter of creating the following entry under `statefulset.spec`:

```
# server-disk.yaml
    ...
    volumeClaimTemplates:
      - metadata:
          name: data
```

```
        spec:
          accessModes: ["ReadWriteOnce"]
          resources:
            requests:
              storage: 1Gi
```

In this manifest snippet, we are naming the volume data, requesting a 1GB persistent disk and setting the access mode to ReadWriteOnce, which means that no other replicas will have exclusive read/write access over its provided volume.

As mentioned before, the Persistent Volume Claim can be understood as a mechanism for creating disks on our behalf with the cloud providers' storage interface, in our case, GCP. Let us now see that this is indeed true. We will modify the original server.yaml file to include the above Persistent Volume Claim and remove the old emptyDir volume definition. The new resulting file is called server-disk.yaml. Let us clean up our environment again and launch our newly defined, disk-based server:

```
# clean up the environment first

$ ./configmap.sh
configmap "scripts" deleted
configmap/scripts created

$ kubectl apply -f server-disk.yaml
service/server created
statefulset.apps/server created
```

After a few seconds, we can check that three volumes (one for each replica) have been created by running kubectl get pv. Please note that some details have been removed and/or abbreviated for brevity:

```
$ kubectl get pv
NAME           CAP STATUS CLAIM
pvc-42339fcc-* 1Gi Bound  default/data-server-0
pvc-4cb81b93-* 1Gi Bound  default/data-server-1
pvc-59792e64-* 1Gi Bound  default/data-server-2
```

We can now also observe that GCP sees the same volumes as proper Google Cloud Persistent Disks as though we had created them manually:

```
$ gcloud compute disks list
...
gke-my-cluster-f8fca-pvc-42339fcc-*
gke-my-cluster-f8fca-pvc-4cb81b93-*
gke-my-cluster-f8fca-pvc-59792e64-*
```

Not only we now have three discrete volumes, one for each of the servers (`server-0`, `server-1`, `server-2`), but we also have the volume claims themselves as first class Kubernetes citizens:

```
$ kubectl get pvc
NAME          STATUS VOLUME         CAP MODE
data-server-0 Bound  pvc-42339fcc-* 1Gi RWO
data-server-1 Bound  pvc-4cb81b93-* 1Gi RWO
data-server-2 Bound  pvc-59792e64-* 1Gi RWO
```

The key feature of Persistent Volume Claims is that the life cycle of Pods is independent from that of volumes. In other words, we can delete Pods, let them crash, scale the StatefulSet—or even brutally delete it. No matter what happens, the associated volume with every Pod will be reattached again. Let us launch our test client against so that we can prove that this is indeed the case:

```
$ kubectl apply -f client.yaml
pod/client created

$ kubectl logs -f client

      abcdefghijklmnopqrstuvwxyz
      --------------------------
      12012012012012012012012012
      --------------------------
53:25 wwwwwwwwwwwwwwwwwwwwwwwwww | hits = 0 (0%)
53:27 hhhhhhhhhhhhhhhhhhhhhhhhhh | hits = 26 (100%)
53:29 hhhhhhhhhhhhhhhhhhhhhhhhhh | hits = 26 (100%)
53:31 hhhhhhhhhhhhhhhhhhhhhhhhhh | hits = 26 (100%)
```

At second 25 earlier, the keys have been written once to the three replicas. Now that the keys are backed up by proper persistent store, we should see server failures but never a write (letter w) again. We will start by deleting server-2 by issuing the kubectl delete pod/server-2 command to see whether this is indeed true:

```
57:04 hhhhhhhhhhhhhhhhhhhhhhhhhh | hits = 26 (100%)
57:06 h5hh5hh5hh5hh5hh5hh5hh5hh5 | hits = 17 (65%)
57:08 h5hh5hh5hh5hh5hh5hh5hh5hh5 | hits = 17 (65%)
57:10 h5hh5hh5hh5hh5hh5hh5hh5hh5 | hits = 17 (65%)
57:12 h5hh5hh5hh5hh5hh5hh5hh5hh5 | hits = 17 (65%)
57:15 h5hh5hh5hh5hh5hh5hh5hh5hh5 | hits = 17 (65%)
57:17 h.hhhhhhhhhhhhhhhhhhhhhhhh | hits = 25 (96%)
58:25 hhhhhhhhhhhhhhhhhhhhhhhhhh | hits = 26 (100%)
```

Here we see that between 57:06 and 57:15, server-2 is shutting down (the five digit stands for HTTP error 503). Then the server becomes inaccessible for a while at 57:17 and then comes back online again. Note that there are no m (miss) nor w (write) letters since server-2 has never lost any data.

Let us now do something more radical and delete the StatefulSet itself by issuing the `kubectl delete statefulset/server` command:

```
26:03 hhhhhhhhhhhhhhhhhhhhhhhhhh | hits = 26 (100%)
26:06 55555555555555555555555555 | hits = 0 (0%)
26:08 55555555555555555555555555 | hits = 0 (0%)
26:10 55555555555555555555555555 | hits = 0 (0%)
26:12 55555555555555555555555555 | hits = 0 (0%)
26:14 55555555555555555555555555 | hits = 0 (0%)
26:16 .......................... | hits = 0 (0%)
27:25 .......................... | hits = 0 (0%)
27:27 .......................... | hits = 0 (0%)
```

Note that all servers go in shutting down mode and then become inaccessible as denoted by `.` (dot). After we confirm that all Pods have been terminated, we start the StatefulSet again by issuing the `kubectl apply -f server-disk.yaml` command:

```
28:05 .......................... | hits = 0 (0%)
28:07 .......................... | hits = 0 (0%)
28:10 ..h..h..h..h..h..h..h..h.. | hits = 8 (31%)
28:12 ..h..h..h..h..h..h..h..h.. | hits = 8 (31%)
28:14 ..h..h..h..h..h..h..h..h.. | hits = 8 (31%)
28:16 ..h..h..h..h..h..h..h..h.. | hits = 8 (31%)
28:18 ..h..h..h..h..h..h..h..h.. | hits = 8 (31%)
28:20 h.hh.hh.hh.hh.hh.hh.hh.hh. | hits = 17 (65%)
28:22 h.hh.hh.hh.hh.hh.hh.hh.hh. | hits = 17 (65%)
28:25 h.hh.hh.hh.hh.hh.hh.hh.hh. | hits = 17 (65%)
28:27 h.hh.hh.hh.hh.hh.hh.hh.hh. | hits = 17 (65%)
28:29 h.hh.hh.hh.hh.hh.hh.hh.hh. | hits = 17 (65%)
28:31 hhhhhhhhhhhhhhhhhhhhhhhhhh | hits = 26 (100%)
28:33 hhhhhhhhhhhhhhhhhhhhhhhhhh | hits = 26 (100%)
28:35 hhhhhhhhhhhhhhhhhhhhhhhhhh | hits = 26 (100%)
```

Here we see that the servers come progressively online and that, when they do, the client registers `h` (hit) which means that the key has been successfully retrieved and there was no need to write it again. Please note that in this example, we are abstracting away from the possibility of corrupted files on disks due to partial writes, unclosed file handles, or other application-level malfunctions.

Summary

In this chapter, we implemented a key/value data store backing service from scratch using StatefulSets that helped us observe the key properties that this controller type guarantees such as sequential Pod creation and stable network identity. The latter, stable network identity is fundamental to expose Pods by using a headless service and by having consistent persistence, both features covered in this chapter as well.

We also examined Pod life cycle events and their relevance in the setting up of clusters and managing scaling events (as well as in launching stateful clusters for the first time). The takeaway for the reader is that ad hoc scaling (using the `kubectl scale`) command is difficult when considering data partioning and replication aspects; extra steps—running scripts and/or administration procedures—are often required both before and after a scale event.

Index

E. Garbarino, *Beginning Kubernetes on the Google Cloud Platform*,
https://doi.org/10.1007/978-1-4842-5491-2