O'REILLY®

Compliments of MESOSPHERE

Kubernetes Up & Running

DIVE INTO THE FUTURE OF INFRASTRUCTURE

Brendan Burns,
Kelsey Hightower & Joe Beda

Kubernetes: Up and Running

Dive into the Future of Infrastructure

This Excerpt contains Chapters 1, 2, 10, 13, and 14 of the book *Kubernetes: Up and Running*. The complete book is available at *oreilly.com* and through other retailers.

Kelsey Hightower, Brendan Burns, and Joe Beda

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Kubernetes: Up and Running

by Kelsey Hightower, Brendan Burns, and Joe Beda

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com/safari*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Angela Rufino
Production Editor: Melanie Yarbrough
Copyeditor: Christina Edwards
Proofreader: Rachel Head

Indexer: Kevin Broccoli
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

September 2017: First Edition

Revision History for the First Edition

2017-09-05: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491935675* for release details.

While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-491-93567-5

[LSI]

Table of Contents

Foreword

The future of infrastructure is programmable and data-centric.

In our always-connected economy, customers are expecting highly customized, data-informed, real-time interactions...and not just from the technology elite like Google, Apple, and Amazon. From banks to cruise lines, healthcare to manufacturing, auto makers to retailers, just about every company I work with today is trying to stay competitive by continuously improving their digital experiences with customers and by tapping data and insights to make those experiences compelling and valuable.

This new generation of data-intensive, highly dynamic applications and services require two related infrastructure capabilities. First is container orchestration, so that new ideas can quickly get developed, packaged, and shipped as containerized microservices. Second is data services orchestration, to power the information backbone of the app, because data now needs to be a first-class citizen, not just something that gets passed around. Both sets of technologies are distributed systems, and for many businesses this has meant complex and siloed infrastructures or clouds.

Apache Mesos (the technology on which our company was founded) was conceived to pool and automate both these types of services. While it's broadly known that Mesos works well in automating data services like Apache Kafka (a message queue), Apache Cassandra (a distributed database), and Apache Spark (an analytics engine), what's less well known is how Mesos has also served as a platform for a wide range of container orchestration engines: Aurora (built by Twitter), Jarvis (used at Apple), Titus (from Netflix), and Marathon (created by us at Mesosphere). But no container orchestrator until now has been met with the kind of acclaim and adoption that we have all seen with Kubernetes.

As an early contributor to Kubernetes, we were thrilled this September to share that we've added 100% pure upstream Kubernetes to Mesosphere's DC/OS distributed computing platform. DC/OS allows you to to deploy both Kubernetes and dozens of open source big data services with push-button ease, and stitches together your entire datacenter and cloud instances into a single set of compute resources to simplify operations and scale-out. Kubernetes is an amazing tool, and we are excited to bring easy on-premise deployment and integration with stateful services to the community of users.

Mesosphere is proud to sponsor *Kubernetes: Up and Running*. This book lays out how Kubernetes is architected, and how its tools and APIs can be used to improve the development, delivery and maintenance of modern distributed applications. We hope you enjoy the book, and that it helps you bring your own breakthrough applications to life.

— Tobi Knaup,
Chief Technology Officer, Mesosphere

CHAPTER 1
Introduction

Kubernetes is an open source orchestrator for deploying containerized applications. Kubernetes was originally developed by Google, inspired by a decade of experience deploying scalable, reliable systems in containers via application-oriented APIs.[1]

But Kubernetes is much more than simply exporting technology developed at Google. Kubernetes has grown to be the product of a rich and growing open source community. This means that Kubernetes is a product that is suited not just to the needs of internet-scale companies but to cloud-native developers of all scales, from a cluster of Raspberry Pi computers to a warehouse full of the latest machines. Kubernetes provides the software necessary to successfully build and deploy reliable, scalable distributed systems.

You may be wondering what we mean when we say "reliable, scalable distributed systems." More and more services are delivered over the network via APIs. These APIs are often delivered by a *distributed system*, the various pieces that implement the API running on different machines, connected via the network and coordinating their actions via network communication. Because we rely on these APIs increasingly for all aspects of our daily lives (e.g., finding directions to the nearest hospital), these systems must be highly *reliable*. They cannot fail, even if a part of the system crashes or otherwise fails. Likewise, they must maintain *availability* even during software rollouts or other maintenance events. Finally, because more and more of the world is coming online and using such services, they must be highly *scalable* so that they can grow their capacity to keep up with ever-increasing usage without radical redesign of the distributed system that implements the services.

[1] Brendan Burns et al., "Borg, Omega, and Kubernetes: Lessons Learned from Three Container-Management Systems over a Decade," *ACM Queue* 14 (2016): 70–93, available at *http://bit.ly/2vIrL4S*.

Depending on when and why you have come to hold this book in your hands, you may have varying degrees of experience with containers, distributed systems, and Kubernetes. Regardless of what your experience is, we believe this book will enable you to make the most of your use of Kubernetes.

There are many reasons why people come to use containers and container APIs like Kubernetes, but we believe they effectively all can be traced back to one of these benefits:

- Velocity
- Scaling (of both software and teams)
- Abstracting your infrastructure
- Efficiency

In the following sections we describe how Kubernetes can help provide each of these benefits.

Velocity

Velocity is the key component in nearly all software development today. The changing nature of software from boxed software shipped on CDs to web-based services that change every few hours means that the difference between you and your competitors is often the speed with which you can develop and deploy new components and features.

It is important to note, however, that this velocity is not defined in terms of simply raw speed. While your users are always looking for iterative improvement, they are more interested in a highly reliable service. Once upon a time, it was OK for a service to be down for maintenance at midnight every night. But today, our users expect constant uptime, even if the software they are running is changing constantly.

Consequently, velocity is measured not in terms of the raw number of features you can ship per hour or day, but rather in terms of the number of things you can ship while maintaining a highly available service.

In this way, containers and Kubernetes can provide the tools that you need to move quickly, while staying available. The core concepts that enable this are immutability, declarative configuration, and online self-healing systems. These ideas all interrelate to radically improve the speed with which you can reliably deploy software.

The Value of Immutability

Containers and Kubernetes encourage developers to build distributed systems that adhere to the principles of immutable infrastructure. With immutable infrastructure, once an artifact is created in the system it does not change via user modifications.

Traditionally, computers and software systems have been treated as *mutable* infrastructure. With mutable infrastructure, changes are applied as incremental updates to an existing system. A system upgrade via the `apt-get update` tool is a good example of an update to a mutable system. Running `apt` sequentially downloads any updated binaries, copies them on top of older binaries, and makes incremental updates to configuration files. With a mutable system, the current state of the infrastructure is not represented as a single artifact, but rather an accumulation of incremental updates and changes. On many systems these incremental updates come from not just system upgrades but operator modifications as well.

In contrast, in an immutable system, rather than a series of incremental updates and changes, an entirely new, complete image is built, where the update simply replaces the entire image with the newer image in a single operation. There are no incremental changes. As you can imagine, this is a significant shift from the more traditional world of configuration management.

To make this more concrete in the world of containers, consider two different ways to upgrade your software:

1. You can log into a container, run a command to download your new software, kill the old server, and start the new one.
2. You can build a new container image, push it to a container registry, kill the existing container, and start a new one.

At first blush, these two approaches might seem largely indistinguishable. So what is it about the act of building a new container that improves reliability?

The key differentiation is the artifact that you create, and the record of how you created it. These records make it easy to understand exactly the differences in some new version and, if something goes wrong, determine what has changed and how to fix it.

Additionally, building a new image rather than modifying an existing one means the old image is still around, and can quickly be used for a rollback if an error occurs. In contrast, once you copy your new binary over an existing binary, such rollback is nearly impossible.

Immutable container images are at the core of everything that you will build in Kubernetes. It is possible to imperatively change running containers, but this is an antipattern to be used only in extreme cases where there are no other options (e.g., if it is the only way to temporarily repair a mission-critical production system). And even then, the changes must also be recorded through a declarative configuration update at some later time, after the fire is out.

Declarative Configuration

Immutability extends beyond containers running in your cluster to the way you describe your application to Kubernetes. Everything in Kubernetes is a *declarative configuration object* that represents the desired state of the system. It is Kubernetes's job to ensure that the actual state of the world matches this desired state.

Much like mutable versus immutable infrastructure, declarative configuration is an alternative to *imperative* configuration, where the state of the world is defined by the execution of a series of instructions rather than a declaration of the desired state of the world. While imperative commands define actions, declarative configurations define state.

To understand these two approaches, consider the task of producing three replicas of a piece of software. With an imperative approach, the configuration would say: "run A, run B, and run C." The corresponding declarative configuration would be "replicas equals three."

Because it describes the state of the world, declarative configuration does not have to be executed to be understood. Its impact is concretely declared. Since the effects of declarative configuration can be understood before they are executed, declarative configuration is far less error-prone. Further, the traditional tools of software development, such as source control, code review, and unit testing, can be used in declarative configuration in ways that are impossible for imperative instructions.

The combination of declarative state stored in a version control system and Kubernetes's ability to make reality match this declarative state makes rollback of a change trivially easy. It is simply restating the previous declarative state of the system. With imperative systems this is usually impossible, since while the imperative instructions describe how to get you from point *A* to point *B*, they rarely include the reverse instructions that can get you back.

Self-Healing Systems

Kubernetes is an online, self-healing system. When it receives a desired state configuration, it does not simply take actions to make the current state match the desired state a single time. It continuously takes actions to ensure that the current state matches the desired state. This means that not only will Kubernetes initialize your system, but it will guard it against any failures or perturbations that might destabilize your system and affect reliability.

A more traditional operator repair involves a manual series of mitigation steps, or human intervention performed in response to some sort of alert. Imperative repair like this is more expensive (since it generally requires an on-call operator to be available to enact the repair). It is also generally slower, since a human must often wake up and log in to respond. Furthermore, it is less reliable since the imperative series of repair operations suffer from all of the problems of imperative management described in the previous section. Self-healing systems like Kubernetes both reduce the burden on operators and improve the overall reliability of the system by performing reliable repairs more quickly.

As a concrete example of this self-healing behavior, if you assert a desired state of three replicas to Kubernetes, it does not just create three replicas—it continuously ensures that there are exactly three replicas. If you manually create a fourth replica Kubernetes will destroy one to bring the number back to three. If you manually destroy a replica, Kubernetes will create one to again return you to the desired state.

Online self-healing systems improve developer velocity because the time and energy you might otherwise have spent on operations and maintenance can instead be spent on developing and testing new features.

Scaling Your Service and Your Teams

As your product grows, its inevitable that you will need to scale both your software and the teams that develop it. Fortunately, Kubernetes can help with both of these goals. Kubernetes achieves scalability by favoring *decoupled* architectures.

Decoupling

In a decoupled architecture each component is separated from other components by defined APIs and service load balancers. APIs and load balancers isolate each piece of the system from the others. APIs provide a buffer between implementer and consumer, and load balancers provide a buffer between running instances of each service.

Decoupling components via load balancers makes it easy to scale the programs that make up your service, because increasing the size (and therefore the capacity) of the program can be done without adjusting or reconfiguring any of the other layers of your service.

Decoupling servers via APIs makes it easier to scale the development teams because each team can focus on a single, smaller *microservice* with a comprehensible surface area. Crisp APIs between microservices limit the amount of cross-team communication overhead required to build and deploy software. This communication overhead is often the major restricting factor when scaling teams.

Easy Scaling for Applications and Clusters

Concretely, when you need to scale your service, the immutable, declarative nature of Kubernetes makes this scaling trivial to implement. Because your containers are immutable, and the number of replicas is simply a number in a declarative config, scaling your service upward is simply a matter of changing a number in a configuration file, asserting this new declarative state to Kubernetes, and letting it take care of the rest. Alternately, you can set up autoscaling and simply let Kubernetes take care of it for you.

Of course, that sort of scaling assumes that there are resources available in your cluster to consume. Sometimes you actually need to scale up the cluster itself. Here again, Kubernetes makes this task easier. Because each machine in a cluster is entirely identical to every other machine, and the applications themselves are decoupled from the details of the machine by containers, adding additional resources to the cluster is simply a matter of imaging a new machine and joining it into the cluster. This can be accomplished via a few simple commands or via a prebaked machine image.

One of the challenges of scaling machine resources is predicting their use. If you are running on physical infrastructure, the time to obtain a new machine is measured in days or weeks. On both physical and cloud infrastructure, predicting future costs is difficult because it is hard to predict the growth and scaling needs of specific applications.

Kubernetes can simplify forecasting future compute costs. To understand why this is true, consider scaling up three teams, A, B, and C. Historically you have seen that each team's growth is highly variable and thus hard to predict. If you are provisioning individual machines for each service, you have no choice but to forecast based on the maximum expected growth for each service, since machines dedicated to one team cannot be used for another team. If instead you use Kubernetes to decouple the teams from the specific machines they are using, you can forecast growth based on the aggregate growth of all three services. Combining three variable growth rates into a single growth rate reduces statistical noise and produces a more reliable forecast of expected growth. Furthermore, decoupling the teams from specific machines means that teams can share fractional parts of each other's machines, reducing even further the overheads associated with forecasting growth of computing resources.

Scaling Development Teams with Microservices

As noted in a variety of research, the ideal team size is the "two-pizza team," or roughly six to eight people, because this group size often results in good knowledge sharing, fast decision making, and a common sense of purpose. Larger teams tend to suffer from hierarchy, poor visibility, and infighting, which hinder agility and success.

However, many projects require significantly more resources to be successful and achieve their goals. Consequently, there is a tension between the ideal team size for agility and the necessary team size for the product's end goals.

The common solution to this tension has been the development of decoupled, service-oriented teams that each build a single microservice. Each small team is responsible for the design and delivery of a service that is consumed by other small teams. The aggregation of all of these services ultimately provides the implementation of the overall product's surface area.

Kubernetes provides numerous abstractions and APIs that make it easier to build these decoupled microservice architectures.

- Pods, or groups of containers, can group together container images developed by different teams into a single deployable unit.
- Kubernetes services provide load balancing, naming, and discovery to isolate one microservice from another.
- Namespaces provide isolation and access control, so that each microservice can control the degree to which other services interact with it.
- Ingress objects provide an easy-to-use frontend that can combine multiple microservices into a single externalized API surface area.

Finally, decoupling the application container image and machine means that different microservices can colocate on the same machine without interfering with each other, reducing the overhead and cost of microservice architectures. The health-checking and rollout features of Kubernetes guarantee a consistent approach to application rollout and reliability that ensures that a proliferation of microservice teams does not also result in a proliferation of different approaches to service production lifecycle and operations.

Separation of Concerns for Consistency and Scaling

In addition to the consistency that Kubernetes brings to operations, the decoupling and separation of concerns produced by the Kubernetes stack lead to significantly greater consistency for the lower levels of your infrastructure. This enables your operations function to scale to managing many machines with a single small, focused team. We have talked at length about the decoupling of application container and machine/operating system (OS), but an important aspect of this decoupling is that the container orchestration API becomes a crisp contract that separates the responsibilities of the application operator from the cluster orchestration operator. We call this the "not my monkey, not my circus" line. The application developer relies on the service-level agreement (SLA) delivered by the container orchestration API, without worrying about the details of how this SLA is achieved. Likewise, the container orchestration API reliability engineer focuses on delivering the orchestration API's SLA without worrying about the applications that are running on top of it.

This decoupling of concerns means that a small team running a Kubernetes cluster can be responsible for supporting hundreds or even thousands of teams running applications within that cluster (Figure 1-1). Likewise, a small team can be responsible for tens (or more) of clusters running around the world. It's important to note that the same decoupling of containers and OS enables the OS reliability engineers to focus on the SLA of the individual machine's OS. This becomes another line of separate responsibility, with the Kubernetes operators relying on the OS SLA, and the OS operators worrying solely about delivering that SLA. Again, this enables you to scale a small team of OS experts to a fleet of thousands of machines.

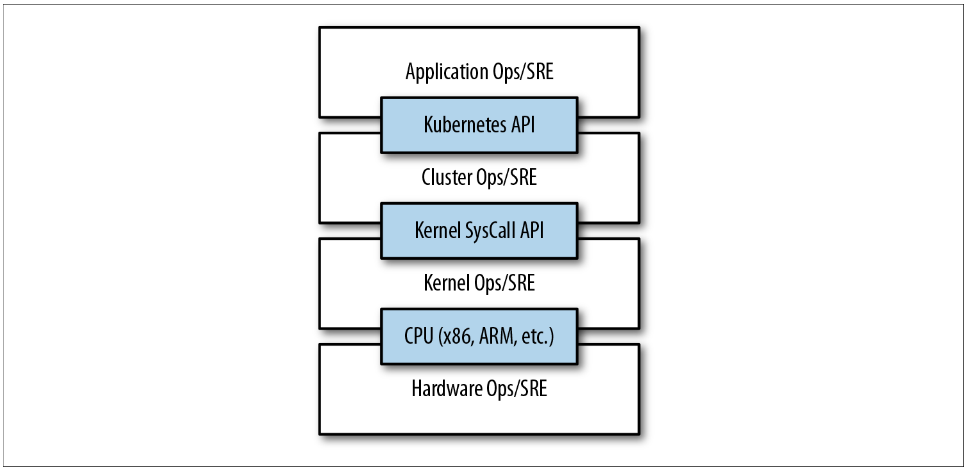

Figure 1-1. An illustration of how different operations teams are decoupled using APIs

Of course, devoting even a small team to managing an OS is beyond the scale of many organizations. In these environments, a managed Kubernetes-as-a-Service (KaaS) provided by a public cloud provider is a great option.

At the time of writing, you can use managed KaaS on Microsoft Azure, with Azure Container Service, as well as on the Google Cloud Platform via the Google Container Engine (GCE). There is no equivalent service available on Amazon Web Services (AWS), though the `kops` project provides tools for easy installation and management of Kubernetes on AWS.

The decision of whether to use KaaS or manage it yourself is one each user needs to make based on the skills and demands of their situation. Often for small organizations, KaaS provides an easy-to-use solution that enables them to focus their time and energy on building the software to support their work rather than managing a cluster. For a larger organization that can afford a dedicated team for managing its Kubernetes cluster, it may make sense to manage it yourself since it enables greater flexibility in terms of cluster capabilities and operations.

Abstracting Your Infrastructure

The goal of the public cloud is to provide easy-to-use, self-service infrastructure for developers to consume. However, too often cloud APIs are oriented around mirroring the infrastructure that IT expects, not the concepts (e.g., "virtual machines" instead of "applications") that developers want to consume. Additionally, in many cases the cloud comes with particular details in implementation or services that are specific to the cloud provider. Consuming these APIs directly makes it difficult to run your application in multiple environments, or spread between cloud and physical environments.

The move to application-oriented container APIs like Kubernetes has two concrete benefits. First, as we described previously, it separates developers from specific machines. This not only makes the machine-oriented IT role easier, since machines can simply be added in aggregate to scale the cluster, but in the context of the cloud it also enables a high degree of portability since developers are consuming a higher-level API that is implemented in terms of the specific cloud infrastructure APIs.

When your developers build their applications in terms of container images and deploy them in terms of portable Kubernetes APIs, transferring your application between environments, or even running in hybrid environments, is simply a matter of sending the declarative config to a new cluster. Kubernetes has a number of plug-ins that can abstract you from a particular cloud. For example, Kubernetes services know how to create load balancers on all major public clouds as well as several different private and physical infrastructures. Likewise, Kubernetes `PersistentVolumes` and `PersistentVolumeClaims` can be used to abstract your applications away from specific storage implementations. Of course, to achieve this portability you need to avoid cloud-managed services (e.g., Amazon's DynamoDB or Google's Cloud Spanner), which means that you will be forced to deploy and manage open source storage solutions like Cassandra, MySQL, or MongoDB.

Putting it all together, building on top of Kubernetes's application-oriented abstractions ensures that the effort that you put into building, deploying, and managing your application is truly portable across a wide variety of environments.

Efficiency

In addition to the developer and IT management benefits that containers and Kubernetes provide, there is also a concrete economic benefit to the abstraction. Because developers no longer think in terms of machines, their applications can be colocated on the same machines without impacting the applications themselves. This means that tasks from multiple users can be packed tightly onto fewer machines.

Efficiency can be measured by the ratio of the useful work performed by a machine or process to the total amount of energy spent doing so. When it comes to deploying and managing applications, many of the available tools and processes (e.g., bash scripts, `apt` updates, or imperative configuration management) are somewhat inefficient. When discussing efficiency it's often helpful to think of both the cost of running a server and the human cost required to manage it.

Running a server incurs a cost based on power usage, cooling requirements, data center space, and raw compute power. Once a server is racked and powered on (or clicked and spun up), the meter literally starts running. Any idle CPU time is money wasted. Thus, it becomes part of the system administrator's responsibilities to keep utilization at acceptable levels, which requires ongoing management. This is where containers and the Kubernetes workflow come in. Kubernetes provides tools that automate the distribution of applications across a cluster of machines, ensuring higher levels of utilization than are possible with traditional tooling.

A further increase in efficiency comes from the fact that a developer's test environment can be quickly and cheaply created as a set of containers running in a personal view of a shared Kubernetes cluster (using a feature called *namespaces*). In the past, turning up a test cluster for a developer might have meant turning up three machines. With Kubernetes it is simple to have all developers share a single test cluster, aggregating their usage onto a much smaller set of machines. Reducing the overall number of machines used in turn drives up the efficiency of each system: since more of the resources (CPU, RAM, etc.) on each individual machine are used, the overall cost of each container becomes much lower.

Reducing the cost of development instances in your stack enables development practices that might previously have been cost-prohibitive. For example, with your application deployed via Kubernetes it becomes conceivable to deploy and test every single commit contributed by every developer throughout your entire stack.

When the cost of each deployment is measured in terms of a small number of containers, rather than multiple complete virtual machines (VMs), the cost you incur for such testing is dramatically lower. Returning to the original value of Kubernetes, this increased testing also increases velocity, since you have both strong signals as to the reliability of your code as well as the granularity of detail required to quickly identify where a problem may have been introduced.

Summary

Kubernetes was built to radically change the way that applications are built and deployed in the cloud. Fundamentally, it was designed to give developers more velocity, efficiency, and agility. We hope the preceding sections have given you an idea of why you should deploy your applications using Kubernetes. Now that you are convinced of that, the following chapters will teach you *how* to deploy your application.

CHAPTER 2
Creating and Running Containers

Kubernetes is a platform for creating, deploying, and managing distributed applications. These applications come in many different shapes and sizes, but ultimately, they are all comprised of one or more *applications* that run on individual machines. These applications accept input, manipulate data, and then return the results. Before we can even consider building a distributed system, we must first consider how to build the *application container images* that make up the pieces of our distributed system.

Applications are typically comprised of a language runtime, libraries, and your source code. In many cases your application relies on external libraries such as `libc` and `libssl`. These external libraries are generally shipped as shared components in the OS that you have installed on a particular machine.

Problems occur when an application developed on a programmer's laptop has a dependency on a shared library that isn't available when the program is rolled out to the production OS. Even when the development and production environments share the exact same version of the OS, problems can occur when developers forget to include dependent asset files inside a package that they deploy to production.

A program can only execute successfully if it can be reliably deployed onto the machine where it should run. Too often the state of the art for deployment involves running imperative scripts, which inevitably have twisty and Byzantine failure cases.

Finally, traditional methods of running multiple applications on a single machine require that all of these programs share the same versions of shared libraries on the system. If the different applications are developed by different teams or organizations, these shared dependencies add needless complexity and coupling between these teams.

In Chapter 1, we argued strongly for the value of immutable images and infrastructure. It turns out that this is exactly the value provided by the container image. As we will see, it easily solves all the problems of dependency management and encapsulation just described.

When working with applications it's often helpful to package them in a way that makes it easy to share them with others. Docker, the default container runtime engine, makes it easy to package an application and push it to a remote registry where it can later be pulled by others.

In this chapter we are going to work with a simple example application that we built for this book to help show this workflow in action. You can find the application on GitHub.

Container images bundle an application and its dependencies, under a root filesystem, into a single artifact. The most popular container image format is the Docker image format, the primary image format supported by Kubernetes. Docker images also include additional metadata used by a container runtime to start a running application instance based on the contents of the container image.

This chapter covers the following topics:

- How to package an application using the Docker image format
- How to start an application using the Docker container runtime

Container Images

For nearly everyone, their first interaction with any container technology is with a container image. A *container image* is a binary package that encapsulates all of the files necessary to run an application inside of an OS container. Depending on how you first experiment with containers, you will either build a container image from your local filesystem or download a preexisting image from a *container registry*. In either case, once the container image is present on your computer, you can run that image to produce a running application inside an OS container.

The Docker Image Format

The most popular and widespread container image format is the Docker image format, which was developed by the Docker open source project for packaging, distributing, and running containers using the `docker` command. Subsequently work has begun by Docker, Inc., and others to standardize the container image format via the Open Container Image (OCI) project. While the OCI set of standards have recently (as of mid-2017) been released as a 1.0 standard, adoption of these standards is still very early. The Docker image format continues to be the de facto standard, and is made up of a series of filesystem layers. Each layer adds, removes, or modifies files from the preceding layer in the filesystem. This is an example of an *overlay* filesystem. There are a variety of different concrete implementations of such filesystems, including `aufs`, `overlay`, and `overlay2`.

Container Layering

Container images are constructed of a series of filesystem layers, where each layer inherits and modifies the layers that came before it. To help explain this in detail, let's build some containers. Note that for correctness the ordering of the layers should be bottom up, but for ease of understanding we take the opposite approach:

```
.
└── container A: a base operating system only, such as Debian
    └── container B: build upon #A, by adding Ruby v2.1.10
    └── container C: build upon #A, by adding Golang v1.6
```

At this point we have three containers: A, B, and C. B and C are *forked* from A and share nothing besides the base container's files. Taking it further, we can build on top of B by adding Rails (version 4.2.6). We may also want to support a legacy application that requires an older version of Rails (e.g., version 3.2.x). We can build a container image to support that application based on B also, planning to someday migrate the app to v4:

```
. (continuing from above)
└── container B: build upon #A, by adding Ruby v2.1.10
    └── container D: build upon #B, by adding Rails v4.2.6
    └── container E: build upon #B, by adding Rails v3.2.x
```

Conceptually, each container image layer builds upon a previous one. Each parent reference is a pointer. While the example here is a simple set of containers, other real-world containers can be part of a larger and extensive directed acyclic graph.

Container images are typically combined with a container configuration file, which provides instructions on how to set up the container environment and execute an application entrypoint. The container configuration often includes information on how to set up networking, namespace isolation, resource constraints (cgroups), and what `syscall` restrictions should be placed on a running container instance. The container root filesystem and configuration file are typically bundled using the Docker image format.

Containers fall into two main categories:

- System containers
- Application containers

System containers seek to mimic virtual machines and often run a full boot process. They often include a set of system services typically found in a VM, such as ssh, cron, and syslog.

Application containers differ from system containers in that they commonly run a single application. While running a single application per container might seem like an unnecessary constraint, it provides the perfect level of granularity for composing scalable applications, and is a design philosophy that is leveraged heavily by pods.

Building Application Images with Docker

In general, container orchestration systems like Kubernetes are focused on building and deploying distributed systems made up of application containers. Consequently, we will focus on application containers for the remainder of this chapter.

Dockerfiles

A Dockerfile can be used to automate the creation of a Docker container image. The following example describes the steps required to build the kuard (Kubernetes up and running) image, which is both secure and lightweight in terms of size:

```
FROM alpine
MAINTAINER Kelsey Hightower <kelsey.hightower@kuar.io>
COPY bin/kuard /kuard
ENTRYPOINT ["/kuard"]
```

This text can be stored in a text file, typically named *Dockerfile*, and used to create a Docker image.

Run the following command to create the kuard Docker image:

```
$ docker build -t kuard-amd64:1 .
```

We have chosen to build on top of Alpine, an extremely minimal Linux distribution. Consequently, the final image should check in at around 6 MB, which is drastically smaller than many publicly available images that tend to be built on top of more complete OS versions such as Debian.

At this point our kuard image lives in the local Docker registry where the image was built and is only accessible to a single machine. The true power of Docker comes from the ability to share images across thousands of machines and the broader Docker community.

Image Security

When it comes to security there are no shortcuts. When building images that will ultimately run in a production Kubernetes cluster, be sure to follow best practices for packaging and distributing applications. For example, don't build containers with passwords baked in—and this includes not just in the final layer, but any layers in the image. One of the counterintuitive problems introduced by container layers is that deleting a file in one layer doesn't delete that file from preceding layers. It still takes up space and it can be accessed by anyone with the right tools—an enterprising attacker can simply create an image that only consists of the layers that contain the password.

Secrets and images should *never* be mixed. If you do so, you will be hacked, and you will bring shame to your entire company or department. We all want to be on TV someday, but there are better ways to go about that.

Optimizing Image Sizes

There are several gotchas that come when people begin to experiment with container images that lead to overly large images. The first thing to remember is that files that are removed by subsequent layers in the system are actually still present in the images; they're just inaccessible. Consider the following situation:

```
.
└── layer A: contains a large file named 'BigFile'
    └── layer B: removes 'BigFile'
        └── layer C: builds on B, by adding a static binary
```

You might think that *BigFile* is no longer present in this image. After all, when you run the image, it is no longer accessible. But in fact it is still present in layer A, which means that whenever you push or pull the image, *BigFile* is still transmitted through the network, even if you can no longer access it.

Another pitfall that people fall into revolves around image caching and building. Remember that each layer is an independent delta from the layer below it. Every time you change a layer, it changes every layer that comes after it. Changing the preceding layers means that they need to be rebuilt, repushed, and repulled to deploy your image to development.

To understand this more fully, consider two images:

```
.
└── layer A: contains a base OS
    └── layer B: adds source code server.js
        └── layer C: installs the 'node' package
```

versus:

```
.
└── layer A: contains a base OS
    └── layer B: installs the 'node' package
        └── layer C: adds source code server.js
```

It seems obvious that both of these images will behave identically, and indeed the first time they are pulled they do. However, consider what happens when *server.js* changes. In one case, it is only the change that needs to be pulled or pushed, but in the other case, both *server.js* and the layer providing the `node` package need to be pulled and pushed, since the `node` layer is dependent on the *server.js* layer. In general, you want to order your layers from least likely to change to most likely to change in order to optimize the image size for pushing and pulling.

Storing Images in a Remote Registry

What good is a container image if it's only available on a single machine?

Kubernetes relies on the fact that images described in a pod manifest are available across every machine in the cluster. One option for getting this image to all machines in the cluster would be to export the `kuard` image and import it on every other machine in the Kubernetes cluster. We can't think of anything more tedious than managing Docker images this way. The process of manually importing and exporting Docker images has human error written all over it. Just say no!

The standard within the Docker community is to store Docker images in a remote registry. There are tons of options when it comes to Docker registries, and what you choose will be largely based on your needs in terms of security requirements and collaboration features.

Generally speaking the first choice you need to make regarding a registry is whether to use a private or a public registry. Public registries allow anyone to download images stored in the registry, while private registries require authentication to download images. In choosing public versus private, it's helpful to consider your use case.

Public registries are great for sharing images with the world, because they allow for easy, unauthenticated use of the container images. You can easily distribute your software as a container image and have confidence that users everywhere will have the exact same experience.

In contrast, a private repository is best for storing your applications that are private to your service and that you don't want the world to use.

Regardless, to push an image, you need to authenticate to the registry. You can generally do this with the `docker login` command, though there are some differences for certain registries. In the examples here we are pushing to the Google Cloud Platform registry, called the Google Container Registry (GCR). For new users hosting publicly readable images, the Docker Hub is a great place to start.

Once you are logged in, you can tag the `kuard` image by prepending the target Docker registry:

```
$ docker tag kuard-amd64:1 gcr.io/kuar-demo/kuard-amd64:1
```

Then you can push the kuard image:

```
$ docker push gcr.io/kuar-demo/kuard-amd64:1
```

Now that the kuard image is available on a remote registry, it's time to deploy it using Docker. Because we pushed it to the public Docker registry, it will be available everywhere without authentication.

The Docker Container Runtime

Kubernetes provides an API for describing an application deployment, but relies on a container runtime to set up an application container using the container-specific APIs native to the target OS. On a Linux system that means configuring cgroups and namespaces.

The default container runtime used by Kubernetes is Docker. Docker provides an API for creating application containers on Linux and Windows systems.

Running Containers with Docker

The Docker CLI tool can be used to deploy containers. To deploy a container from the gcr.io/kuar-demo/kuard-amd64:1 image, run the following command:

```
$ docker run -d --name kuard \
  --publish 8080:8080 \
  gcr.io/kuar-demo/kuard-amd64:1
```

This command starts the kuard database and maps ports 8080 on your local machine to 8080 in the container. This is because each container gets its own IP address, so listening on *localhost* inside the container doesn't cause you to listen on your machine. Without the port forwarding, connections will be inaccessible to your machine.

Exploring the kuard Application

kuard exposes a simple web interface, which can be loaded by pointing your browser at *http://localhost:8080* or via the command line:

```
$ curl http://localhost:8080
```

kuard also exposes a number of interesting functions that we will explore later on in this book.

Limiting Resource Usage

Docker provides the ability to limit the amount of resources used by applications by exposing the underlying cgroup technology provided by the Linux kernel.

Limiting memory resources

One of the key benefits to running applications within a container is the ability to restrict resource utilization. This allows multiple applications to coexist on the same hardware and ensures fair usage.

To limit `kuard` to 200 MB of memory and 1 GB of swap space, use the `--memory` and `--memory-swap` flags with the `docker run` command.

Stop and remove the current `kuard` container:

```
$ docker stop kuard
$ docker rm kuard
```

Then start another `kuard` container using the appropriate flags to limit memory usage:

```
$ docker run -d --name kuard \
  --publish 8080:8080 \
  --memory 200m \
  --memory-swap 1G \
  gcr.io/kuar-demo/kuard-amd64:1
```

Limiting CPU resources

Another critical resource on a machine is the CPU. Restrict CPU utilization using the `--cpu-shares` flag with the `docker run` command:

```
$ docker run -d --name kuard \
  --publish 8080:8080 \
  --memory 200m \
  --memory-swap 1G \
  --cpu-shares 1024 \
  gcr.io/kuar-demo/kuard-amd64:1
```

Cleanup

Once you are done building an image, you can delete it with the `docker rmi` command:

```
docker rmi <tag-name>
```

or

```
docker rmi <image-id>
```

Images can either be deleted via their tag name (e.g., `gcr.io/kuar-demo/kuard-amd64:1`) or via their image ID. As with all ID values in the `docker` tool, the image ID can be shortened as long as it remains unique. Generally only three or four characters of the ID are necessary.

It's important to note that unless you explicitly delete an image it will live on your system forever, *even* if you build a new image with an identical name. Building this new image simply moves the tag to the new image; it doesn't delete or replace the old image.

Consequently, as you iterate while you are creating a new image, you will often create many, many different images that end up taking up unnecessary space on your computer.

To see the images currently on your machine, you can use the `docker images` command. You can then delete tags you are no longer using.

A slightly more sophisticated approach is to set up a `cron` job to run an image garbage collector. For example, the `docker-gc` tool is a commonly used image garbage collector that can easily run as a recurring `cron` job, once per day or once per hour, depending on how many images you are creating.

Summary

Application containers provide a clean abstraction for applications, and when packaged in the Docker image format, applications become easy to build, deploy, and distribute. Containers also provide isolation between applications running on the same machine, which helps avoid dependency conflicts. The ability to mount external directories means we can run not only stateless applications in a container, but also applications like `influxdb` that generate lots of data.

CHAPTER 3

Jobs

So far we have focused on long-running processes such as databases and web applications. These types of workloads run until either they are upgraded or the service is no longer needed. While long-running processes make up the large majority of workloads that run on a Kubernetes cluster, there is often a need to run short-lived, one-off tasks. The `Job` object is made for handling these types of tasks.

A Job creates Pods that run until successful termination (i.e., exit with 0). In contrast, a regular Pod will continually restart regardless of its exit code. Jobs are useful for things you only want to do once, such as database migrations or batch jobs. If run as a regular Pod, your database migration task would run in a loop, continually repopulating the database after every exit.

In this chapter we explore the most common Job patterns afforded by Kubernetes. We will also leverage these patterns in real-life scenarios.

The Job Object

The `Job` object is responsible for creating and managing pods defined in a template in the Job specification. These pods generally run until successful completion. The `Job` object coordinates running a number of pods in parallel.

If the Pod fails before a successful termination, the Job controller will create a new Pod based on the Pod template in the Job specification. Given that Pods have to be scheduled, there is a chance that your Job will not execute if the required resources are not found by the scheduler. Also, due to the nature of distributed systems there is a small chance, during certain failure scenarios, that duplicate pods will be created for a specific task.

Job Patterns

Jobs are designed to manage batch-like workloads where work items are processed by one or more Pods. By default each Job runs a single Pod once until successful termination. This Job pattern is defined by two primary attributes of a Job, namely the number of Job completions and the number of Pods to run in parallel. In the case of the "run once until completion" pattern, the `completions` and `parallelism` parameters are set to `1`.

Table 3-1 highlights Job patterns based on the combination of `completions` and `parallelism` for a Job configuration.

Table 3-1. Job patterns

Type	Use case	Behavior	completions	parallelism
One shot	Database migrations	A single pod running once until successful termination	1	1
Parallel fixed completions	Multiple pods processing a set of work in parallel	One or more Pods running one or more times until reaching a fixed completion count	1+	1+
Work queue: parallel Jobs	Multiple pods processing from a centralized work queue	One or more Pods running once until successful termination	1	2+

One Shot

One-shot Jobs provide a way to run a single Pod once until successful termination. While this may sound like an easy task, there is some work involved in pulling this off. First, a Pod must be created and submitted to the Kubernetes API. This is done using a Pod template defined in the Job configuration. Once a Job is up and running, the Pod backing the Job must be monitored for successful termination. A Job can fail for any number of reasons including an application error, an uncaught exception during runtime, or a node failure before the Job has a chance to complete. In all cases the Job controller is responsible for recreating the Pod until a successful termination occurs.

There are multiple ways to create a one-shot Job in Kubernetes. The easiest is to use the `kubectl` command-line tool:

```
$ kubectl run -i oneshot \
  --image=gcr.io/kuar-demo/kuard-amd64:1 \
  --restart=OnFailure \
  -- --keygen-enable \
     --keygen-exit-on-complete \
     --keygen-num-to-gen 10

...
```

```
(ID 0) Workload starting
(ID 0 1/10) Item done: SHA256:nAsUsG54XoKRkJwyN+OShkUPKew3mwq7OCc
(ID 0 2/10) Item done: SHA256:HVKX1ANns6SgF/er1lyo+ZCdnB8geFGt0/8
(ID 0 3/10) Item done: SHA256:irjCLRov3mTT0P0JfsvUyhKRQ1TdGR8H1jg
(ID 0 4/10) Item done: SHA256:nbQAIVY/yrhmEGk3Ui2sAHuxb/o6mYO0qRk
(ID 0 5/10) Item done: SHA256:CCpBoXNlXOMQvR2v38yqimXGAa/w2Tym+aI
(ID 0 6/10) Item done: SHA256:wEY2TTIDz4ATjcr1iimxavCzZzNjRmbOQp8
(ID 0 7/10) Item done: SHA256:t3JSrCt7sQweBgqG5CrbMoBulwk4lfDWiTI
(ID 0 8/10) Item done: SHA256:E84/Vze7KKyjCh9OZh02MkXJGoty9PhaCec
(ID 0 9/10) Item done: SHA256:UOmYex79qqbI1MhcIfG4hDnGKonlsij2k3s
(ID 0 10/10) Item done: SHA256:WCR8wIGOFag84Bsa8f/9QHuKqF+0mEnCADY
(ID 0) Workload exiting
```

There are some things to note here:

- The `-i` option to `kubectl` indicates that this is an interactive command. `kubectl` will wait until the Job is running and then show the log output from the first (and in this case only) pod in the Job.
- `--restart=OnFailure` is the option that tells `kubectl` to create a `Job` object.
- All of the options after `--` are command-line arguments to the container image. These instruct our test server (`kuard`) to generate 10 4,096-bit SSH keys and then exit.
- Your output may not match this exactly. `kubectl` often misses the first couple of lines of output with the `-i` option.

After the Job has completed, the Job object and related Pod are still around. This is so that you can inspect the log output. Note that this Job won't show up in `kubectl get jobs` unless you pass the `-a` flag. Without this flag `kubectl` hides completed Jobs. Delete the Job before continuing:

```
$ kubectl delete jobs oneshot
```

The other option for creating a one-shot Job is using a configuration file, as shown in Example 3-1.

Example 3-1. job-oneshot.yaml

```
apiVersion: batch/v1
kind: Job
metadata:
  name: oneshot
  labels:
    chapter: jobs
spec:
  template:
    metadata:
      labels:
        chapter: jobs
```

```
spec:
  containers:
  - name: kuard
    image: gcr.io/kuar-demo/kuard-amd64:1
    imagePullPolicy: Always
    args:
    - "--keygen-enable"
    - "--keygen-exit-on-complete"
    - "--keygen-num-to-gen=10"
  restartPolicy: OnFailure
```

Submit the job using the `kubectl apply` command:

```
$ kubectl apply -f job-oneshot.yaml
job "oneshot" created
```

Then `describe` the `oneshot` job:

```
$ kubectl describe jobs oneshot

Name:           oneshot
Namespace:      default
Image(s):       gcr.io/kuar-demo/kuard-amd64:1
Selector:       controller-uid=cf87484b-e664-11e6-8222-42010a8a007b
Parallelism:    1
Completions:    1
Start Time:     Sun, 29 Jan 2017 12:52:13 -0800
Labels:         Job=oneshot
Pods Statuses:  0 Running / 1 Succeeded / 0 Failed
No volumes.
Events:
  ... Reason             Message
  ... ------             -------
  ... SuccessfulCreate   Created pod: oneshot-4kfdt
```

You can view the results of the Job by looking at the logs of the pod that was created:

```
$ kubectl logs oneshot-4kfdt

...
Serving on :8080
(ID 0) Workload starting
(ID 0 1/10) Item done: SHA256:+r6b4W81DbEjxMcD3LHjU+EIGnLEzbpxITKn8IqhkPI
(ID 0 2/10) Item done: SHA256:mzHewajaY1KA8VluSLOnNMk9fDE5zdn7vvBS5Ne8AxM
(ID 0 3/10) Item done: SHA256:TRtEQHfflJmwkqnNyGgQm/IvXNykSBIg8c03h0g3onE
(ID 0 4/10) Item done: SHA256:tSwPYH/J347il/mgqTxRRdeZcOazEtgZlA8A3/HWbro
(ID 0 5/10) Item done: SHA256:IP8XtguJ6GbWwLHqjKecVfdS96B17nnO21I/TNc1j9k
(ID 0 6/10) Item done: SHA256:ZfNxdQvuST/6ZzEVkyxdRG98p73c/5TM99SEbPeRWfc
(ID 0 7/10) Item done: SHA256:tH+CNl/IUl/HUuKdMsq2XEmDQ8oAvmhMO6Iwj8ZEOj0
(ID 0 8/10) Item done: SHA256:3GfsUaALVEHQcGNLBOu4Qd1zqqqJ8j738i5r+I5XwVI
(ID 0 9/10) Item done: SHA256:5wV4L/xEiHSJXwLUT2fHf0SCKM2g3XH3sVtNbgskCXw
(ID 0 10/10) Item done: SHA256:bPqqOonwSbjzLqe9ZuVRmZkz+DBjaNTZ9HwmQhbdWLI
(ID 0) Workload exiting
```

Congratulations, your job has run successfully!

> You may have noticed that we didn't specify any labels when creating the Job object. Like with other controllers (DaemonSet, ReplicaSets, deployments, etc.) that use labels to identify a set of Pods, unexpected behaviors can happen if a pod is reused across objects.
>
> Because Jobs have a finite beginning and ending, it is common for users to create many of them. This makes picking unique labels more difficult and more critical. For this reason, the Job object will automatically pick a unique label and use it to identify the pods it creates. In advanced scenarios (such as swapping out a running Job without killing the pods it is managing) users can choose to turn off this automatic behavior and manually specify labels and selectors.

Pod failure

We just saw how a Job can complete successfully. But what happens if something fails? Let's try that out and see what happens.

Let's modify the arguments to kuard in our configuration file to cause it to fail out with a nonzero exit code after generating three keys, as shown in Example 3-2.

Example 3-2. job-oneshot-failure1.yaml

```
...
spec:
  template:
    spec:
      containers:
        ...
        args:
        - "--keygen-enable"
        - "--keygen-exit-on-complete"
        - "--keygen-exit-code=1"
        - "--keygen-num-to-gen=3"
...
```

Now launch this with kubectl apply -f jobs-oneshot-failure1.yaml. Let it run for a bit and then look at the pod status:

```
$ kubectl get pod -a -l job-name=oneshot

NAME            READY     STATUS             RESTARTS   AGE
oneshot-3ddk0   0/1       CrashLoopBackOff   4          3m
```

Here we see that the same Pod has restarted four times. Kubernetes is in CrashLoop BackOff for this Pod. It is not uncommon to have a bug someplace that causes a program to crash as soon as it starts. In that case, Kubernetes will wait a bit before restarting the pod to avoid a crash loop eating resources on the node. This is all handled local to the node by the `kubelet` without the Job being involved at all.

Kill the Job (`kubectl delete jobs oneshot`), and let's try something else. Modify the config file again and change the `restartPolicy` from `OnFailure` to `Never`. Launch this with `kubectl apply -f jobs-oneshot-failure2.yaml`.

If we let this run for a bit and then look at related pods we'll find something interesting:

```
$ kubectl get pod -l job-name=oneshot -a

NAME            READY     STATUS    RESTARTS   AGE oneshot-0wm49   0/1
Error     0          1m oneshot-6h9s2   0/1       Error     0          39s
oneshot-hkzw0   1/1       Running   0          6s oneshot-k5swz   0/1
Error     0          28s oneshot-m1rdw   0/1       Error     0          19s
oneshot-x157b   0/1       Error     0          57s
```

What we see is that we have multiple pods here that have errored out. By setting `restartPolicy: Never` we are telling the `kubelet` not to restart the Pod on failure, but rather just declare the Pod as failed. The `Job` object then notices and creates a replacement Pod. If you aren't careful, this'll create a lot of "junk" in your cluster. For this reason, we suggest you use `restartPolicy: OnFailure` so failed Pods are rerun in place.

Clean this up with `kubectl delete jobs oneshot`.

So far we've seen a program fail by exiting with a nonzero exit code. But workers can fail in other ways. Specifically, they can get stuck and not make any forward progress. To help cover this case, you can use liveness probes with Jobs. If the liveness probe policy determines that a Pod is dead, it'll be restarted/replaced for you.

Parallelism

Generating keys can be slow. Let's start a bunch of workers together to make key generation faster. We're going to use a combination of the `completions` and `parallelism` parameters. Our goal is to generate 100 keys by having 10 runs of `kuard` with each run generating 10 keys. But we don't want to swamp our cluster, so we'll limit ourselves to only five pods at a time.

This translates to setting `completions` to `10` and `parallelism` to `5`. The config is shown in Example 3-2.

Example 3-3. job-parallel.yaml

```
apiVersion: batch/v1
kind: Job
metadata:
  name: parallel
  labels:
    chapter: jobs
spec:
  parallelism: 5
  completions: 10
  template:
    metadata:
      labels:
        chapter: jobs
    spec:
      containers:
      - name: kuard
        image: gcr.io/kuar-demo/kuard-amd64:1
        imagePullPolicy: Always
        args:
        - "--keygen-enable"
        - "--keygen-exit-on-complete"
        - "--keygen-num-to-gen=10"
      restartPolicy: OnFailure
```

Start it up:

```
$ kubectl apply -f job-parallel.yaml
job "parallel" created
```

Now watch as the pods come up, do their thing, and exit. New pods are created until 10 have completed altogether. Here we use the `--watch` flag to have `kubectl` stay around and list changes as they happen:

```
$ kubectl get pods -w
NAME             READY     STATUS    RESTARTS   AGE
parallel-55tlv   1/1       Running   0          5s
parallel-5s7s9   1/1       Running   0          5s
parallel-jp7bj   1/1       Running   0          5s
parallel-lssmn   1/1       Running   0          5s
parallel-qxcxp   1/1       Running   0          5s
NAME             READY     STATUS      RESTARTS   AGE
parallel-jp7bj   0/1       Completed   0          26s
parallel-tzp9n   0/1       Pending   0         0s
parallel-tzp9n   0/1       Pending   0         0s
parallel-tzp9n   0/1       ContainerCreating   0         1s
parallel-tzp9n   1/1       Running   0         1s
parallel-tzp9n   0/1       Completed   0         48s
parallel-x1kmr   0/1       Pending   0         0s
parallel-x1kmr   0/1       Pending   0         0s
parallel-x1kmr   0/1       ContainerCreating   0         0s
```

```
parallel-x1kmr   1/1       Running   0          1s
parallel-5s7s9   0/1       Completed   0          1m
parallel-tprfj   0/1       Pending   0          0s
parallel-tprfj   0/1       Pending   0          0s
parallel-tprfj   0/1       ContainerCreating   0          0s
parallel-tprfj   1/1       Running   0          2s
parallel-x1kmr   0/1       Completed   0          52s
parallel-bgvz5   0/1       Pending   0          0s
parallel-bgvz5   0/1       Pending   0          0s
parallel-bgvz5   0/1       ContainerCreating   0          0s
parallel-bgvz5   1/1       Running   0          2s
parallel-qxcxp   0/1       Completed   0          2m
parallel-xplw2   0/1       Pending   0          1s
parallel-xplw2   0/1       Pending   0          1s
parallel-xplw2   0/1       ContainerCreating   0          1s
parallel-xplw2   1/1       Running   0          3s
parallel-bgvz5   0/1       Completed   0          40s
parallel-55tlv   0/1       Completed   0          2m
parallel-lssmn   0/1       Completed   0          2m
```

Feel free to poke around at the completed Jobs and check out their logs to see the fingerprints of the keys they generated. Clean up by deleting the finished Job object with kubectl delete job parallel.

Work Queues

A common use case for Jobs is to process work from a work queue. In this scenario, some task creates a number of work items and publishes them to a work queue. A worker Job can be run to process each work item until the work queue is empty (Figure 3-1).

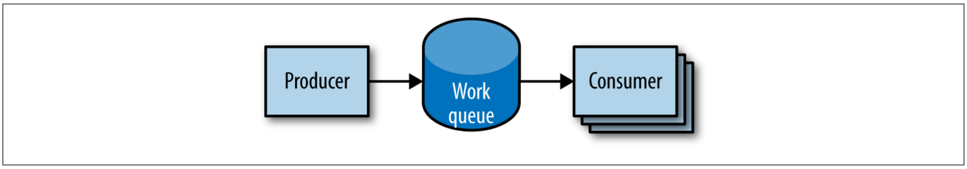

Figure 3-1. Parallel jobs

Starting a work queue

We start by launching a centralized work queue service. kuard has a simple memory-based work queue system built in. We will start an instance of kuard to act as a coordinator for all the work to be done.

Create a simple ReplicaSet to manage a singleton work queue daemon. We are using a ReplicaSet to ensure that a new Pod will get created in the face of machine failure, as shown in Example 3-4.

Example 3-4. rs-queue.yaml

```
apiVersion: extensions/v1beta1
kind: ReplicaSet
metadata:
  labels:
    app: work-queue
    component: queue
    chapter: jobs
  name: queue
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: work-queue
        component: queue
        chapter: jobs
    spec:
      containers:
      - name: queue
        image: "gcr.io/kuar-demo/kuard-amd64:1"
        imagePullPolicy: Always
```

Run the work queue with the following command:

```
$ kubectl apply -f rs-queue.yaml
```

At this point the work queue daemon should be up and running. Let's use port forwarding to connect to it. Leave this command running in a terminal window:

```
$ QUEUE_POD=$(kubectl get pods -l app=work-queue,component=queue \
    -o jsonpath='{.items[0].metadata.name}')
$ kubectl port-forward $QUEUE_POD 8080:8080
Forwarding from 127.0.0.1:8080 -> 8080
Forwarding from [::1]:8080 -> 8080
```

You can open your browser to *http://localhost:8080* and see the `kuard` interface. Switch to the "MemQ Server" tab to keep an eye on what is going on.

With the work queue server in place, we should expose it using a service. This will make it easy for producers and consumers to locate the work queue via DNS, as Example 3-5 shows.

Example 3-5. service-queue.yaml

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: work-queue
    component: queue
    chapter: jobs
  name: queue
spec:
  ports:
  - port: 8080
    protocol: TCP
    targetPort: 8080
  selector:
    app: work-queue
    component: queue
```

Create the queue service with `kubectl`:

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

Loading up the queue

We are now ready to put a bunch of work items in the queue. For the sake of simplicity we'll just use `curl` to drive the API for the work queue server and insert a bunch of work items. `curl` will communicate to the work queue through the `kubectl port-forward` we set up earlier, as shown in Example 3-6.

Example 3-6. load-queue.sh

```
# Create a work queue called 'keygen'
curl -X PUT localhost:8080/memq/server/queues/keygen

# Create 100 work items and load up the queue.
for i in work-item-{0..99}; do
  curl -X POST localhost:8080/memq/server/queues/keygen/enqueue \
    -d "$i"
done
```

Run these commands, and you should see 100 JSON objects output to your terminal with a unique message identifier for each work item. You can confirm the status of the queue by looking at the "MemQ Server" tab in the UI, or you can ask the work queue API directly:

```
$ curl 127.0.0.1:8080/memq/server/stats
{
    "kind": "stats",
    "queues": [
        {
            "depth": 100,
            "dequeued": 0,
            "drained": 0,
            "enqueued": 100,
            "name": "keygen"
        }
        ]
    }
```

Now we are ready to kick off a Job to consume the work queue until it's empty.

Creating the consumer job

This is where things get interesting! kuard is also able to act in consumer mode. Here we set it up to draw work items from the work queue, create a key, and then exit once the queue is empty, as shown in Example 3-7.

Example 3-7. job-consumers.yaml

```
apiVersion: batch/v1
kind: Job
metadata:
  labels:
    app: message-queue
    component: consumer
    chapter: jobs
  name: consumers
spec:
  parallelism: 5
  template:
    metadata:
      labels:
        app: message-queue
        component: consumer
        chapter: jobs
    spec:
      containers:
      - name: worker
        image: "gcr.io/kuar-demo/kuard-amd64:1"
        imagePullPolicy: Always
        args:
        - "--keygen-enable"
        - "--keygen-exit-on-complete"
        - "--keygen-memq-server=http://queue:8080/memq/server"
        - "--keygen-memq-queue=keygen"
      restartPolicy: OnFailure
```

We are telling the Job to start up five pods in parallel. As the completions parameter is unset, we put the Job into a worker pool mode. Once the first pod exits with a zero exit code, the Job will start winding down and will not start any new Pods. This means that none of the workers should exit until the work is done and they are all in the process of finishing up.

Create the consumers Job:

```
$ kubectl apply -f job-consumers.yaml
job "consumers" created
```

Once the Job has been created you can view the pods backing the Job:

```
$ kubectl get pods
NAME              READY     STATUS    RESTARTS   AGE
queue-43s87       1/1       Running   0          5m
consumers-6wjxc   1/1       Running   0          2m
consumers-7l5mh   1/1       Running   0          2m
consumers-hvz42   1/1       Running   0          2m
consumers-pc8hr   1/1       Running   0          2m
consumers-w20cc   1/1       Running   0          2m
```

Note there are five pods running in parallel. These pods will continue to run until the work queue is empty. You can watch as it happens in the UI on the work queue server. As the queue empties, the consumer pods will exit cleanly and the `consumers` Job will be considered complete.

Cleaning up

Using labels we can clean up all of the stuff we created in this section:

```
$ kubectl delete rs,svc,job -l chapter=jobs
```

Summary

On a single cluster, Kubernetes can handle both long-running workloads such as web applications and short-lived workloads such as batch jobs. The Job abstraction allows you to model batch job patterns ranging from simple one-time tasks to parallel jobs that process many items until work has been exhausted.

Jobs are a low-level primitive and can be used directly for simple workloads. However, Kubernetes is built from the ground up to be extensible by higher-level objects. Jobs are no exception; they can easily be used by higher-level orchestration systems to take on more complex tasks.

CHAPTER 4

Integrating Storage Solutions and Kubernetes

In many cases decoupling state from applications and building your microservices to be as stateless as possible results in maximally reliable, manageable systems.

However, nearly every system that has any complexity has state in the system somewhere, from the records in a database to the index shards that serve results for a web search engine. At some point you have to have data stored somewhere.

Integrating this data with containers and container orchestration solutions is often the most complicated aspect of building a distributed system. This complexity largely stems from the fact that the move to containerized architectures is also a move toward decoupled, immutable, and declarative application development. These patterns are relatively easy to apply to stateless web applications, but even "cloud-native" storage solutions like Cassandra or MongoDB involve some sort of manual or imperative steps to set up a reliable, replicated solution.

As an example of this, consider setting up a ReplicaSet in MongoDB, which involves deploying the Mongo daemon and then running an imperative command to identify the leader, as well as the participants in the Mongo cluster. Of course, these steps can be scripted, but in a containerized world it is difficult to see how to integrate such commands into a deployment. Likewise, even getting DNS-resolvable names for individual containers in a replicated set of containers is challenging.

Additional complexity comes from the fact that there is data gravity. Most containerized systems aren't built in a vacuum; they are usually adapted from existing systems deployed onto VMs, and these systems likely include data that has to be imported or migrated.

Finally, evolution to the cloud means that many times storage is actually an externalized cloud service, and in that context it can never really exist inside of the Kubernetes cluster.

This chapter covers a variety of approaches for integrating storage into containerized microservices in Kubernetes. First, we cover how to import existing external storage solutions (either cloud services or running on VMs) into Kubernetes. Next, we explore how to run reliable singletons inside of Kubernetes that enable you to have an environment that largely matches the VMs where you previously deployed storage solutions. Finally we cover StatefulSets, which are still under development but represent the future of stateful workloads in Kubernetes.

Importing External Services

In many cases, you have an existing machine running in your network that has some sort of database running on it. In this situation you may not want to immediately move that database into containers and Kubernetes. Perhaps it is run by a different team, or you are doing a gradual move, or the task of migrating the data is simply more trouble than it's worth.

Regardless of the reasons for staying put, this legacy server and service are not going to move into Kubernetes, but nonetheless it is still worthwhile to represent this server in Kubernetes. When you do this, you get to take advantage of all of the built-in naming and service discovery primitives provided by Kubernetes. Additionally, this enables you to configure all your applications so that it looks like the database that is running on a machine somewhere is actually a Kubernetes service. This means that it is trivial to replace it with a database that is a Kubernetes service. For example, in production, you may rely on your legacy database that is running on a machine, but for continuous testing you may deploy a test database as a transient container. Since it is created and destroyed for each test run, data persistence isn't important in the continuous testing case. Representing both databases as Kubernetes services enables you to maintain identical configurations in both testing and production. High fidelity between test and production ensures that passing tests will lead to successful deployment in production.

To see concretely how you maintain high fidelity between development and production, remember that all Kubernetes objects are deployed into *namespaces*. Imagine that we have `test` and `product` namespaces defined. The test service is imported using an object like:

```
kind: Service
metadata:
  name: my-database
  # note 'test' namespace here
  namespace: test
...
```

The production service looks the same, except it uses a different namespace:

```
kind: Service
metadata:
  name: my-database
  # note 'prod' namespace here
  namespace: prod
...
```

When you deploy a Pod into the `test` namespace and it looks up the service named `my-database`, it will receive a pointer to `my-database.test.svc.cluster.internal`, which in turn points to the test database. In contrast, when a Pod deployed in the `prod` namespace looks up the same name (`my-database`) it will receive a pointer to `my-database.prod.svc.cluster.internal`, which is the production database. Thus, the same service name, in two different namespaces, resolves to two different services. For more details on how this works.

> The following techniques all use database or other storage services, but these approaches can be used equally well with other services that aren't running inside your Kubernetes cluster.

Services Without Selectors

When we first introduced services, we talked at length about label queries and how they were used to identify the dynamic set of Pods that were the backends for a particular service. With external services, however, there is no such label query. Instead, you generally have a DNS name that points to the specific server running the database. For our example, let's assume that this server is named `database.company.com`. To import this external database service into Kubernetes, we start by creating a service without a Pod selector that references the DNS name of the database server (Example 4-1).

Example 4-1. dns-service.yaml

```
kind: Service
apiVersion: v1
metadata:
  name: external-database
spec:
  type: ExternalName
  externalName: "database.company.com
```

When a typical Kubernetes service is created, an IP address is also created and the Kubernetes DNS service is populated with an A record that points to that IP address. When you create a service of type `ExternalName`, the Kubernetes DNS service is instead populated with a CNAME record that points to the external name you specified (`database.company.com` in this case). When an application in the cluster does a DNS lookup for the hostname `external-database.svc.default.cluster`, the DNS protocol aliases that name to "database.company.com." This then resolves to the IP address of your external database server. In this way, all containers in Kubernetes believe that they are talking to a service that is backed with other containers, when in fact they are being redirected to the external database.

Note that this is not restricted to databases you are running on your own infrastructure. Many cloud databases and other services provide you with a DNS name to use when accessing the database (e.g., `my-database.databases.cloudprovider.com`). You can use this DNS name as the `externalName`. This imports the cloud-provided database into the namespace of your Kubernetes cluster.

Sometimes, however, you don't have a DNS address for an external database service, just an IP address. In such cases, it is still possible to import this server as a Kubernetes service, but the operation is a little different. First, you create a `Service` without a label selector, but also without the `ExternalName` type we used before (Example 4-2).

Example 4-2. external-ip-service.yaml

```
kind: Service
apiVersion: v1
metadata:
  name: external-ip-database
```

At this point, Kubernetes will allocate a virtual IP address for this service and populate an A record for it. However, because there is no selector for the service, there will be no endpoints populated for the load balancer to redirect traffic to.

Given that this is an external service, the user is responsible for populating the endpoints manually with an `Endpoints` resource (Example 4-3).

Example 4-3. external-ip-endpoints.yaml

```
kind: Endpoints
apiVersion: v1
metadata:
  name: external-ip-database
subsets:
  - addresses:
    - ip: 192.168.0.1
```

```
ports:
- port: 3306
```

If you have more than one IP address for redundancy, you can repeat them in the `addresses` array. Once the endpoints are populated, the load balancer will start redirecting traffic from your Kubernetes service to the IP address endpoint(s).

> Because the user has assumed responsibility for keeping the IP address of the server up to date, you need to either ensure that it never changes or make sure that some automated process updates the `Endpoints` record.

Limitations of External Services: Health Checking

External services in Kubernetes have one significant restriction: they do not perform any health checking. The user is responsible for ensuring that the endpoint or DNS name supplied to Kubernetes is as reliable as necessary for the application.

Running Reliable Singletons

The challenge of running storage solutions in Kubernetes is often that primitives like ReplicaSet expect that every container is identical and replaceable, but for most storage solutions this isn't the case. One option to address this is to use Kubernetes primitives, but not attempt to replicate the storage. Instead, simply run a single Pod that runs the database or other storage solution. In this way the challenges of running replicated storage in Kubernetes don't occur, since there is no replication.

At first blush, this might seem to run counter to the principles of building reliable distributed systems, but in general, it is no less reliable than running your database or storage infrastructure on a single virtual or physical machine, which is how many people currently have built their systems. Indeed, in reality, if you structure the system properly the only thing you are sacrificing is potential downtime for upgrades or in case of machine failure. While for large-scale or mission-critical systems this may not be acceptable, for many smaller-scale applications this kind of limited downtime is a reasonable trade-off for the reduced complexity. If this is not true for you, feel free to skip this section and either import existing services as described in the previous section, or move on to Kubernetes-native `StatefulSets`, described in the following section. For everyone else, we'll review how to build reliable singletons for data storage.

Running a MySQL Singleton

In this section, we'll describe how to run a reliable singleton instance of the MySQL database as a Pod in Kubernetes, and how to expose that singleton to other applications in the cluster.

To do this, we are going to create three basic objects:

- A persistent volume to manage the lifespan of the on-disk storage independently from the lifespan of the running MySQL application
- A MySQL Pod that will run the MySQL application
- A service that will expose this Pod to other containers in the cluster

We described persistent volumes, but a quick review makes sense. A persistent volume is a storage location that has a lifetime independent of any Pod or container. This is very useful in the case of persistent storage solutions where the on-disk representation of a database should survive even if the containers running the database application crash, or move to different machines. If the application moves to a different machine, the volume should move with it, and data should be preserved. Separating the data storage out as a persistent volume makes this possible. To begin, we'll create a persistent volume for our MySQL database to use.

This example uses NFS for maximum portability, but Kubernetes supports many different persistent volume drive types. For example, there are persistent volume drivers for all major public cloud providers, as well as many private cloud providers. To use these solutions, simply replace `nfs` with the appropriate cloud provider volume type (e.g., `azure`, `awsElasticBlockStore`, or `gcePersistentDisk`). In all cases, this change is all you need. Kubernetes knows how to create the appropriate storage disk in the respective cloud provider. This is a great example of how Kubernetes simplifies the development of reliable distributed systems.

Here's the example persistent volume object (Example 4-4).

Example 4-4. nfs-volume.yaml

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: database
  labels:
    volume: my-volume
spec:
  capacity:
    storage: 1Gi
  nfs:
    server: 192.168.0.1
    path: "/exports"
```

This defines an NFS persistent volume object with 1 GB of storage space.

We can create this persistent volume as usual with:

```
    $ kubectl apply -f nfs-volume.yaml
```

Now that we have a persistent volume created, we need to claim that persistent volume for our Pod. We do this with a `PersistentVolumeClaim` object (Example 4-5).

Example 4-5. nfs-volume-claim.yaml

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: database
spec:
  resources:
    requests:
      storage: 1Gi
  selector:
    matchLabels:
      volume: my-volume
```

The `selector` field uses labels to find the matching volume we defined previously.

This kind of indirection may seem overly complicated, but it has a purpose—it serves to isolate our Pod definition from our storage definition. You can declare volumes directly inside a Pod specification, but this locks that Pod specification to a particular volume provider (e.g., a specific public or private cloud). By using volume claims, you can keep your Pod specifications cloud-agnostic; simply create different volumes, specific to the cloud, and use a `PersistentVolumeClaim` to bind them together.

Now that we've claimed our volume, we can use a ReplicaSet to construct our singleton Pod. It might seem odd that we are using a ReplicaSet to manage a single Pod, but it is necessary for reliability. Remember that once scheduled to a machine, a bare Pod is bound to that machine forever. If the machine fails, then any Pods that are on that machine that are not being managed by a higher-level controller like a ReplicaSet vanish along with the machine and are not rescheduled elsewhere. Consequently, to ensure that our database Pod is rescheduled in the presence of machine failures, we use the higher-level ReplicaSet controller, with a replica size of one, to manage our database (Example 4-6).

Example 4-6. mysql-replicaset.yaml

```
apiVersion: extensions/v1beta1
kind: ReplicaSet
metadata:
  name: mysql
  # labels so that we can bind a Service to this Pod
  labels:
    app: mysql
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mysql
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
      - name: database
        image: mysql
        resources:
          requests:
            cpu: 1
            memory: 2Gi
        env:
        # Environment variables are not a best practice for security,
        # but we're using them here for brevity in the example.
        # See Chapter 11 for better options.
        - name: MYSQL_ROOT_PASSWORD
          value: some-password-here
        livenessProbe:
          tcpSocket:
            port: 3306
        ports:
        - containerPort: 3306
        volumeMounts:
          - name: database
            # /var/lib/mysql is where MySQL stores its databases
            mountPath: "/var/lib/mysql"
      volumes:
      - name: database
        persistentVolumeClaim:
          claimName: database
```

Once we create the `ReplicaSet` it will in turn create a Pod running MySQL using the persistent disk we originally created. The final step is to expose this as a Kubernetes service (Example 4-7).

Example 4-7. mysql-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: mysql
spec:
  ports:
  - port: 3306
    protocol: TCP
  selector:
    app: mysql
```

Now we have a reliable singleton MySQL instance running in our cluster and exposed as a service named `mysql`, which we can access at the full domain name `mysql.svc.default.cluster`.

Similar instructions can be used for a variety of data stores, and if your needs are simple and you can survive limited downtime in the face of a machine failure or a need to upgrade the database software, a reliable singleton may be the right approach to storage for your application.

Dynamic Volume Provisioning

Many clusters also include *dynamic volume provisioning*. With dynamic volume provisioning, the cluster operator creates one or more `StorageClass` objects. Here's a default storage class that automatically provisions disk objects on the Microsoft Azure platform (Example 4-8).

Example 4-8. storageclass.yaml

```
apiVersion: storage.k8s.io/v1beta1
kind: StorageClass
metadata:
  name: default
  annotations:
    storageclass.beta.kubernetes.io/is-default-class: "true"
  labels:
    kubernetes.io/cluster-service: "true"
provisioner: kubernetes.io/azure-disk
```

Once a storage class has been created for a cluster, you can refer to this storage class in your persistent volume claim, rather than referring to any specific persistent volume. When the dynamic provisioner sees this storage claim, it uses the appropriate volume driver to create the volume and bind it to your persistent volume claim.

Here's an example of a `PersistentVolumeClaim` that uses the `default` storage class we just defined to claim a newly created persistent volume (Example 4-9).

Example 4-9. dynamic-volume-claim.yaml

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: my-claim
  annotations:
    volume.beta.kubernetes.io/storage-class: default
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
```

The `volume.beta.kubernetes.io/storage-class` annotation is what links this claim back up to the storage class we created.

Persistent volumes are great for traditional applications that require storage, but if you need to develop high-availability, scalable storage in a Kubernetes-native fashion, the newly released StatefulSet object can be used. With this in mind, we'll describe how to deploy MongoDB using StatefulSets in the next section.

Kubernetes-Native Storage with StatefulSets

When Kubernetes was first developed, there was a heavy emphasis on homogeneity for all replicas in a replicated set. In this design, no replica had an individual identity or configuration. It was up to the individual application developer to determine a design that could establish this identity for the application.

While this approach provides a great deal of isolation for the orchestration system, it also makes it quite difficult to develop stateful applications. After significant input from the community and a great deal of experimentation with various existing stateful applications, StatefulSets were introduced into Kubernetes in version 1.5.

> Because StatefulSets are a beta feature, it's possible that the API will change before it becomes an official Kubernetes API. The StatefulSet API has had a lot of input and is generally considered fairly stable, but the beta status should be considered before taking on StatefulSets. In many cases the previously outlined patterns for stateful applications may serve you better in the near term.

Properties of StatefulSets

StatefulSets are replicated groups of Pods similar to ReplicaSets, but unlike a ReplicaSet, they have certain unique properties:

- Each replica gets a persistent hostname with a unique index (e.g., `database-0`, `database-1`, etc.).
- Each replica is created in order from lowest to highest index, and creation will block until the Pod at the previous index is healthy and available. This also applies to scaling up.
- When deleted, each replica will be deleted in order from highest to lowest. This also applies to scaling down the number of replicas.

Manually Replicated MongoDB with StatefulSets

In this section, we'll deploy a replicated MongoDB cluster. For now, the replication setup itself will be done manually to give you a feel for how StatefulSets work. Eventually we will automate this setup as well.

To start, we'll create a replicated set of three MongoDB Pods using a StatefulSet object (Example 4-10).

Example 4-10. mongo-simple.yaml

```
apiVersion: apps/v1beta1
kind: StatefulSet
metadata:
  name: mongo
spec:
  serviceName: "mongo"
  replicas: 3
  template:
    metadata:
      labels:
        app: mongo
    spec:
      containers:
      - name: mongodb
        image: mongo:3.4.1
        command:
        - mongod
        - --replSet
        - rs0
        ports:
        - containerPort: 27017
          name: peer
```

As you can see, the definition is similar to the ReplicaSet definition from previous sections. The only changes are the `apiVersion` and `kind` fields. Create the StatefulSet:

```
$ kubectl apply -f mongo-simple.yaml
```

Once created, the differences between a ReplicaSet and a StatefulSet become apparent. Run `kubectl get pods` and you will likely see:

```
NAME      READY     STATUS            RESTARTS   AGE
mongo-0   1/1       Running           0          1m
mongo-1   0/1       ContainerCreating 0          10s
```

There are two important differences between this and what you would see with a ReplicaSet. The first is that each replicated Pod has a numeric index (`0`, `1`, …), instead of the random suffix that is added by the ReplicaSet controller. The second is that the Pods are being slowly created in order, not all at once as they would be with a ReplicaSet.

Once the StatefulSet is created, we also need to create a "headless" service to manage the DNS entries for the StatefulSet. In Kubernetes a service is called "headless" if it doesn't have a cluster virtual IP address. Since with StatefulSets each Pod has a unique identity, it doesn't really make sense to have a load-balancing IP address for the replicated service. You can create a headless service using `clusterIP: None` in the service specification (Example 4-11).

Example 4-11. mongo-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: mongo
spec:
  ports:
  - port: 27017
    name: peer
  clusterIP: None
  selector:
    app: mongo
```

Once you create that service, there are usually four DNS entries that are populated. As usual, `mongo.default.svc.cluster.local` is created, but unlike with a standard service, doing a DNS lookup on this hostname provides all the addresses in the StatefulSet. In addition, entries are created for `mongo-0.mongo.default.svc.cluster.local` as well as `mongo-1.mongo` and `mongo-2.mongo`. Each of these resolves to the specific IP address of the replica index in the StatefulSet. Thus, with StatefulSets you get well-defined, persistent names for each replica in the set. This is often very useful when you are configuring a replicated storage solution. You can see these DNS entries in action by running commands in one of the Mongo replicas:

```
$ kubectl exec mongo-0 bash ping mongo-1.mongo
```

Next, we're going to manually set up Mongo replication using these per-Pod hostnames.

We'll choose mongo-0.mongo to be our initial primary. Run the mongo tool in that Pod:

```
$ kubectl exec -it mongo-0 mongo
> rs.initiate( {
  _id: "rs0",
  members:[ { _id: 0, host: "mongo-0.mongo:27017" } ]
 });
 OK
```

This command tells mongodb to initiate the ReplicaSet rs0 with mongo-0.mongo as the primary replica.

The rs0 name is arbitrary. You can use whatever you'd like, but you'll need to change it in the *mongo.yaml* StatefulSet definition as well.

Once you have initiated the Mongo ReplicaSet, you can add the remaining replicas by running the following commands in the mongo tool on the mongo-0.mongo Pod:

```
$ kubectl exec -it mongo-0 mongo
> rs.add("mongo-1.mongo:27017");
> rs.add("mongo-2.mongo:27017");
```

As you can see, we are using the replica-specific DNS names to add them as replicas in our Mongo cluster. At this point, we're done. Our replicated MongoDB is up and running. But it's really not as automated as we'd like it to be. In the next section, we'll see how to use scripts to automate the setup.

Automating MongoDB Cluster Creation

To automate the deployment of our StatefulSet-based MongoDB cluster, we're going to add an additional container to our Pods to perform the initialization.

To configure this Pod without having to build a new Docker image, we're going to use a ConfigMap to add a script into the existing MongoDB image. Here's the container we're adding:

```
...
      - name: init-mongo
        image: mongo:3.4.1
        command:
        - bash
        - /config/init.sh
        volumeMounts:
        - name: config
          mountPath: /config
      volumes:
      - name: config
```

```
          configMap:
            name: "mongo-init"
```

Note that it is mounting a ConfigMap volume whose name is `mongo-init`. This ConfigMap holds a script that performs our initialization. First, the script determines whether it is running on `mongo-0` or not. If it is on `mongo-0`, it creates the ReplicaSet using the same command we ran imperatively previously. If it is on a different Mongo replica, it waits until the ReplicaSet exists, and then it registers itself as a member of that ReplicaSet.

Example 4-12 has the complete ConfigMap object.

Example 4-12. mongo-configmap.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: mongo-init
data:
  init.sh: |
    #!/bin/bash

    # Need to wait for the readiness health check to pass so that the
    # mongo names resolve. This is kind of wonky.
    until ping -c 1 ${HOSTNAME}.mongo; do
      echo "waiting for DNS (${HOSTNAME}.mongo)..."
      sleep 2
    done

    until /usr/bin/mongo --eval 'printjson(db.serverStatus())'; do
      echo "connecting to local mongo..."
      sleep 2
    done
    echo "connected to local."

    HOST=mongo-0.mongo:27017

    until /usr/bin/mongo --host=${HOST} --eval 'printjson(db.serverStatus())'; do
      echo "connecting to remote mongo..."
      sleep 2
    done
    echo "connected to remote."

    if [[ "${HOSTNAME}" != 'mongo-0' ]]; then
      until /usr/bin/mongo --host=${HOST} --eval="printjson(rs.status())" \
            | grep -v "no replset config has been received"; do
        echo "waiting for replication set initialization"
        sleep 2
      done
      echo "adding self to mongo-0"
      /usr/bin/mongo --host=${HOST} \
         --eval="printjson(rs.add('${HOSTNAME}.mongo'))"
    fi

    if [[ "${HOSTNAME}" == 'mongo-0' ]]; then
      echo "initializing replica set"
      /usr/bin/mongo --eval="printjson(rs.initiate(\
          {'_id': 'rs0', 'members': [{'_id': 0, \
           'host': 'mongo-0.mongo:27017'}]}))"
    fi
    echo "initialized"

    while true; do
      sleep 3600
    done
```

This script currently sleeps forever after initializing the cluster. Every container in a Pod has to have the same `RestartPolicy`. Since we want our main Mongo container to be restarted, we need to have our initialization container run forever too, or else Kubernetes might think our Mongo Pod is unhealthy.

Putting it all together, here is the complete StatefulSet that uses the ConfigMap in Example 4-13.

Example 4-13. mongo.yaml

```
apiVersion: apps/v1beta1
kind: StatefulSet
metadata:
  name: mongo
spec:
  serviceName: "mongo"
  replicas: 3
  template:
    metadata:
      labels:
        app: mongo
    spec:
      containers:
      - name: mongodb
        image: mongo:3.4.1
        command:
        - mongod
        - --replSet
        - rs0
        ports:
        - containerPort: 27017
          name: web
      # This container initializes the mongodb server, then sleeps.
      - name: init-mongo
```

```
          image: mongo:3.4.1
          command:
          - bash
          - /config/init.sh
          volumeMounts:
          - name: config
            mountPath: /config
        volumes:
        - name: config
          configMap:
            name: "mongo-init"
```

Given all of these files, you can create a Mongo cluster with:

```
$ kubectl apply -f mongo-config-map.yaml
$ kubectl apply -f mongo-service.yaml
$ kubectl apply -f mongo.yaml
```

Or if you want, you can combine them all into a single YAML file where the individual objects are separated by `---`. Ensure that you keep the same ordering, since the StatefulSet definition relies on the ConfigMap definition existing.

Persistent Volumes and StatefulSets

For persistent storage, you need to mount a persistent volume into the */data/db* directory. In the Pod template, you need to update it to mount a persistent volume claim to that directory:

```
...
        volumeMounts:
        - name: database
          mountPath: /data/db
```

While this approach is similar to the one we saw with reliable singletons, because the StatefulSet replicates more than one Pod you cannot simply reference a persistent volume claim. Instead, you need to add a *persistent volume claim template*. You can think of the claim template as being identical to the Pod template, but instead of creating Pods, it creates volume claims. You need to add the following onto the bottom of your StatefulSet definition:

```
  volumeClaimTemplates:
  - metadata:
      name: database
      annotations:
        volume.alpha.kubernetes.io/storage-class: anything
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 100Gi
```

When you add a volume claim template to a StatefulSet definition, each time the StatefulSet controller creates a Pod that is part of the StatefulSet it will create a persistent volume claim based on this template as part of that Pod.

In order for these replicated persistent volumes to work correctly, you either need to have autoprovisioning set up for persistent volumes, or you need to prepopulate a collection of persistent volume objects for the StatefulSet controller to draw from. If there are no claims that can be created, the StatefulSet controller will not be able to create the corresponding Pods.

One Final Thing: Readiness Probes

The final piece in productionizing our MongoDB cluster is to add liveness checks to our Mongo-serving containers. The liveness probe is used to determine if a container is operating correctly. For the liveness checks, we can use the `mongo` tool itself by adding the following to the Pod template in the StatefulSet object:

```
...
 livenessProbe:
   exec:
     command:
       - /usr/bin/mongo
       - --eval
       - db.serverStatus()
     initialDelaySeconds: 10
     timeoutSeconds: 10
 ...
```

Summary

Once we have combined StatefulSets, persistent volume claims, and liveness probing, we have a hardened, scalable cloud-native MongoDB installation running on Kubernetes. While this example dealt with MongoDB, the steps for creating StatefulSets to manage other storage solutions are quite similar and similar patterns can be followed.

CHAPTER 5
Deploying Real-World Applications

The previous chapters described a variety of API objects that are available in a Kubernetes cluster and ways in which those objects can best be used to construct reliable distributed systems. However, none of the preceding chapters really discussed how you might use the objects in practice to deploy a complete, real-world application. That is the focus of this chapter.

We'll take a look at three real-world applications:

- Parse, an open source API server for mobile applications
- Ghost, a blogging and content management platform
- Redis, a lightweight, performant key/value store

These complete examples should give you a better idea of how to structure your own deployments using Kubernetes.

Parse

The Parse server is a cloud API dedicated to providing easy-to-use storage for mobile applications. It provides a variety of different client libraries that make it easy to integrate with Android, iOS, and other mobile platforms. Parse was purchased by Facebook in 2013 and subsequently shut down. Fortunately for us, a compatible server was open sourced by the core Parse team and is available for us to use. This section describes how to set up Parse in Kubernetes.

Prerequisites

Parse uses MongoDB cluster for its storage. Chapter 4 described how to set up a replicated MongoDB using Kubernetes `StatefulSets`. This section assumes you have a three-replica Mongo cluster running in Kubernetes with the names `mongo-0.mongo`, `mongo-1.mongo`, and `mongo-2.mongo`.

These instructions also assume that you have a Docker login; if you don't have one, you can get one for free at *https://docker.com*.

Finally, we assume you have a Kubernetes cluster deployed and the `kubectl` tool properly configured.

Building the parse-server

The open source `parse-server` comes with a *Dockerfile* by default, for easy containerization. First, clone the Parse repository:

```
$ git clone https://github.com/ParsePlatform/parse-server
```

Then move into that directory and build the image:

```
$ cd parse-server
$ docker build -t ${DOCKER_USER}/parse-server .
```

Finally, push that image up to the Docker hub:

```
$ docker push ${DOCKER_USER}/parse-server
```

Deploying the parse-server

Once you have the container image built, deploying the `parse-server` into your cluster is fairly straightforward. Parse looks for three environment variables when being configured:

`APPLICATION_ID`
: An identifier for authorizing your application

`MASTER_KEY`
: An identifier that authorizes the master (root) user

`DATABASE_URI`
: The URI for your MongoDB cluster

Putting this all together, you can deploy Parse as a Kubernetes Deployment using the YAML file in Example 5-1.

Example 5-1. parse.yaml

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: parse-server
  namespace: default
```

```
spec:
  replicas: 1
  template:
    metadata:
      labels:
        run: parse-server
    spec:
      containers:
      - name: parse-server
        image: ${DOCKER_USER}/parse-server
        env:
        - name: DATABASE_URI
          value: "mongodb://mongo-0.mongo:27017,\
            mongo-1.mongo:27017,mongo-2.mongo\
            :27017/dev?replicaSet=rs0"
        - name: APP_ID
          value: my-app-id
        - name: MASTER_KEY
          value: my-master-key
```

Testing Parse

To test your deployment, you need to expose it as a Kubernetes service. You can do that using the service definition in Example 5-2.

Example 5-2. parse-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: parse-server
  namespace: default
spec:
  ports:
  - port: 1337
    protocol: TCP
    targetPort: 1337
  selector:
    run: parse-server
```

Now your Parse server is up and running and ready to receive requests from your mobile applications. Of course, in any real application you are likely going to want to secure the connection with HTTPS. You can see the `parse-server` GitHub page for more details on such a configuration.

Ghost

Ghost is a popular blogging engine with a clean interface written in JavaScript. It can either use a file-based SQLite database or MySQL for storage.

Configuring Ghost

Ghost is configured with a simple JavaScript file that describes the server. We will store this file as a configuration map. A simple development configuration for Ghost looks like Example 5-3.

Example 5-3. ghost-config.js

```
var path = require('path'),
    config;

config = {
    development: {
        url: 'http://localhost:2368',
        database: {
            client: 'sqlite3',
            connection: {
                filename: path.join(process.env.GHOST_CONTENT,
                                    '/data/ghost-dev.db')
            },
            debug: false
        },
        server: {
            host: '0.0.0.0',
            port: '2368'
        },
        paths: {
            contentPath: path.join(process.env.GHOST_CONTENT, '/')
        }
    }
};

module.exports = config;
```

Once you have this configuration file saved to *config.js*, you can create a Kubernetes ConfigMap object using:

```
$ kubectl apply cm --from-file ghost-config.js ghost-config
```

This creates a ConfigMap that is named `ghost-config`. As with the Parse example, we will mount this configuration file as a volume inside of our container. We will deploy Ghost as a `Deployment` object, which defines this volume mount as part of the Pod template (Example 5-4).

Example 5-4. ghost.yaml

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: ghost
```

```
spec:
  replicas: 1
  selector:
    matchLabels:
      run: ghost
  template:
    metadata:
      labels:
        run: ghost
    spec:
      containers:
      - image: ghost
        name: ghost
        command:
        - sh
        - -c
        - cp /ghost-config/config.js /var/lib/ghost/config.js
          && /entrypoint.sh npm start
        volumeMounts:
        - mountPath: /ghost-config
          name: config
      volumes:
      - name: config
        configMap:
          defaultMode: 420
          name: ghost-config
```

One thing to note here is that we are copying the *config.js* file from a different location into the location where Ghost expects to find it, since the ConfigMap can only mount directories, not individual files. Ghost expects other files that are not in that ConfigMap to be present in its directory, and thus we cannot simply mount the entire ConfigMap into */var/lib/ghost*.

You can run this with:

```
$ kubectl apply -f ghost.yaml
```

Once the pod is up and running, you can expose it as a service with:

```
$ kubectl expose deployments ghost --port=2368
```

Once the service is exposed, you can use the `kubectl proxy` command to access the Ghost server:

```
$ kubectl proxy
```

Then visit *http://localhost:8001/api/v1/namespaces/default/services/ghost/proxy/* in your web browser to begin interacting with Ghost.

Ghost + MySQL

Of course, this example isn't very scalable, or even reliable, since the contents of the blog are stored in a local file inside the container. A more scalable approach is to store the blog's data in a MySQL database.

To do this, first modify *config.js* to include:

```
...
database: {
   client: 'mysql',
   connection: {
     host     : 'mysql',
     user     : 'root',
     password : 'root',
     database : 'ghost_db',
     charset  : 'utf8'
   }
 },
...
```

Next, create a new `ghost-config` ConfigMap object:

```
$ kubectl create configmap ghost-config-mysql --from-file config.js
```

Then update the Ghost deployment to change the name of the ConfigMap mounted from `config-map` to `config-map-mysql`:

```
...
      - configMap:
          name: ghost-config-mysql
...
```

Using the instructions from "Kubernetes-Native Storage with StatefulSets" on page 44, deploy a MySQL server in your Kubernetes cluster. Make sure that it has a service named `mysql` defined as well.

You will need to create the database in the MySQL database:

```
$ kubectl exec -it mysql-zzmlw -- mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
...

mysql> create database ghost_db;
...
```

Finally, perform a rollout to deploy this new configuration.

```
$ kubectl apply -f ghost.yaml
```

Because your Ghost server is now decoupled from its database, you can scale up your Ghost server and it will continue to share the data across all replicas.

Edit *ghost.yaml* to set `spec.replicas` to `3`, then run:

```
$ kubectl apply -f ghost.yaml
```

Your ghost installation is now scaled up to three replicas.

Redis

Redis is a popular in-memory key/value store, with numerous additional features. It's an interesting application to deploy because it is a good example of the value of the Kubernetes Pod abstraction. This is because a reliable Redis installation actually is two programs working together. The first is `redis-server`, which implements the key/value store, and the other is `redis-sentinel`, which implements health checking and failover for a replicated Redis cluster.

When Redis is deployed in a replicated manner, there is a single master server that can be used for both read and write operations. Additionally, there are other replica servers that duplicate the data written to the master and can be used for load-balancing read operations. Any of these replicas can fail over to become the master if the original master fails. This failover is performed by the Redis sentinel. In our deployment, both a Redis server and a Redis sentinel are colocated in the same file.

Configuring Redis

As before, we're going to use Kubernetes ConfigMaps to configure our Redis installation. Redis needs separate configurations for the master and slave replicas. To configure the master, create a file named *master.conf* that contains the code in Example 5-5.

Example 5-5. master.conf

```
bind 0.0.0.0
port 6379

dir /redis-data
```

This directs Redis to bind to all network interfaces on port 6379 (the default Redis port) and store its files in the */redis-data* directory.

The slave configuration is identical, but it adds a single `slaveof` directive. Create a file named *slave.conf* that contains what's in Example 5-6.

Example 5-6. slave.conf

```
bind 0.0.0.0
port 6379

dir .
```

```
slaveof redis-0.redis 6379
```

Notice that we are using `redis-0.redis` for the name of the master. We will set up this name using a service and a StatefulSet.

We also need a configuration for the Redis sentinel. Create a file named *sentinel.conf* with the contents of Example 5-7.

Example 5-7. sentinel.conf

```
bind 0.0.0.0
port 26379

sentinel monitor redis redis-0.redis 6379 2
sentinel parallel-syncs redis 1
sentinel down-after-milliseconds redis 10000
sentinel failover-timeout redis 20000
```

Now that we have all of our configuration files, we need to create a couple of simple wrapper scripts to use in our StatefulSet deployment.

The first script simply looks at the hostname for the Pod and determines whether this is the master or a slave, and launches Redis with the appropriate configuration. Create a file named *init.sh* containing the code in Example 5-8.

Example 5-8. init.sh

```
#!/bin/bash
if [[ ${HOSTNAME} == 'redis-0' ]]; then
  redis-server /redis-config/master.conf
else
  redis-server /redis-config/slave.conf
fi
```

The other script is for the sentinel. In this case it is necessary because we need to wait for the `redis-0.redis` DNS name to become available. Create a script named *sentinel.sh* containing the code in Example 5-9.

Example 5-9. sentinel.sh

```
#!/bin/bash
while ! ping -c 1 redis-0.redis; do
  echo 'Waiting for server'
  sleep 1
done

redis-sentinel /redis-config/sentinel.conf
```

Now we need to package all of these files up into a ConfigMap object. You can do this with a single command line:

```
$ kubectl create configmap \
  --from-file=slave.conf=./slave.conf \
  --from-file=master.conf=./master.conf \
  --from-file=sentinel.conf=./sentinel.conf \
  --from-file=init.sh=./init.sh \
  --from-file=sentinel.sh=./sentinel.sh \
  redis-config
```

Creating a Redis Service

The next step in deploying Redis is to create a Kubernetes service that will provide naming and discovery for the Redis replicas (e.g., `redis-0.redis`). To do this, we create a service without a cluster IP address (Example 5-10).

Example 5-10. redis-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: redis
spec:
  ports:
  - port: 6379
    name: peer
  clusterIP: None
  selector:
    app: redis
```

You can create this service with `kubectl apply -f redis-service.yaml`. Don't worry that the Pods for the service don't exist yet. Kubernetes doesn't care; it will add the right names when the Pods are created.

Deploying Redis

We're ready to deploy our Redis cluster. To do this we're going to use a StatefulSet. We introduced StatefulSets in "Manually Replicated MongoDB with StatefulSets" on page 45, when we discussed our MongoDB installation. StatefulSets provide indexing (e.g., `redis-0.redis`) as well as ordered creation and deletion semantics (`redis-0` will always be created before `redis-1`, and so on). They're quite useful for stateful applications like Redis, but honestly, they basically look like Kubernetes `Deployments`. For our Redis cluster, here's what the StatefulSet looks like Example 5-11.

Example 5-11. redis.yaml

```
apiVersion: apps/v1beta1
kind: StatefulSet
metadata:
  name: redis
spec:
  replicas: 3
  serviceName: redis
  template:
    metadata:
      labels:
        app: redis
    spec:
      containers:
      - command: [sh, -c, source /redis-config/init.sh ]
        image: redis:3.2.7-alpine
        name: redis
        ports:
        - containerPort: 6379
          name: redis
        volumeMounts:
        - mountPath: /redis-config
          name: config
        - mountPath: /redis-data
          name: data
      - command: [sh, -c, source /redis-config/sentinel.sh]
        image: redis:3.2.7-alpine
        name: sentinel
        volumeMounts:
        - mountPath: /redis-config
          name: config
      volumes:
      - configMap:
          defaultMode: 420
          name: redis-config
        name: config
      - emptyDir:
        name: data
```

You can see that there are two containers in this Pod. One runs the *init.sh* script that we created and the main Redis server, and the other is the sentinel that monitors the servers.

You can also note that there are two volumes defined in the Pod. One is the volume that uses our ConfigMap to configure the two Redis applications, and the other is a simple `emptyDir` volume that is mapped into the Redis server container to hold the application data so that it survives a container restart. For a more reliable Redis installation this could be a network-attached disk, as discussed in Chapter 4.

Now that we've defined our Redis cluster, we can create it using:

```
$ kubectl apply -f redis.yaml
```

Playing with Our Redis Cluster

To demonstrate that we've actually successfully created a Redis cluster, we can perform some tests.

First, we can determine which server the Redis sentinel believes is the master. To do this, we can run the `redis-cli` command in one of the pods:

```
$ kubectl exec redis-2 -c redis \
  -- redis-cli -p 26379 sentinel get-master-addr-by-name redis
```

This should print out the IP address of the `redis-0` pod. You can confirm this using `kubectl get pods -o wide`.

Next, we'll confirm that the replication is actually working.

To do this, first try to read the value `foo` from one of the replicas:

```
$ kubectl exec redis-2 -c redis -- redis-cli -p 6379 get foo
```

You should see no data in the response.

Next, try to write that data to a replica:

```
$ kubectl exec redis-2 -c redis -- redis-cli -p 6379 set foo 10
READONLY You can't write against a read only slave.
```

You can't write to a replica, because it's read-only. Let's try the same command against `redis-0`, which is the master:

```
$ kubectl exec redis-0 -c redis -- redis-cli -p 6379 set foo 10
OK
```

Now try the original read from a replica:

```
$ kubectl exec redis-2 -c redis -- redis-cli -p 6379 get foo
10
```

This shows that our cluster is set up correctly, and data is replicating between masters and slaves.

Summary

In the preceding sections we described how to deploy a variety of applications using assorted Kubernetes concepts. We saw how to put together service-based naming and discovery to deploy web frontends like Ghost as well as API servers like Parse, and we saw how Pod abstraction makes it easy to deploy the components that make up a reliable Redis cluster. Regardless of whether you will actually deploy these applications to production, the examples demonstrated patterns that you can repeat to manage your applications using Kubernetes. We hope that seeing the concepts we described in previous chapters come to life in real-world examples helps you better understand how to make Kubernetes work for you.

About the Authors

Kelsey Hightower has worn every hat possible throughout his career in tech, and enjoys leadership roles focused on making things happen and shipping software. Kelsey is a strong open source advocate focused on building simple tools that make people smile. When he is not slinging Go code, you can catch him giving technical workshops covering everything from programming to system administration.

Joe Beda started his career at Microsoft working on Internet Explorer (he was young and naive). Throughout 7 years at Microsoft and 10 at Google, Joe has worked on GUI frameworks, real-time voice and chat, telephony, machine learning for ads, and cloud computing. Most notably, while at Google, Joe started the Google Compute Engine and, along with Brendan and Craig McLuckie, created Kubernetes. Joe is now CTO of Heptio, a startup he founded along with Craig. Joe proudly calls Seattle home.

Brendan Burns began his career with a brief stint in the software industry followed by a PhD in Robotics focused on motion planning for human-like robot arms. This was followed by a brief stint as a professor of computer science. Eventually, he returned to Seattle and joined Google, where he worked on web search infrastructure with a special focus on low-latency indexing. While at Google, he created the Kubernetes project with Joe and Craig McLuckie. Brendan is currently a Director of Engineering at Microsoft Azure.

Colophon

The animal on the cover of *Kubernetes: Up and Running* is a bottlenose dolphin (*Tursiops truncatus*).

Bottlenose dolphins live in groups typically of 10–30 members, called pods, but group size varies from single individuals to more than 1,000. Dolphins often work as a team to harvest fish schools, but they also hunt individually. Dolphins search for prey primarily using echolocation, which is similar to sonar.

The Bottlenose dolphin is found in most tropical to temperate oceans; its color is grey, with the shade of grey varying among populations; it can be bluish-grey, brownish-grey, or even nearly black, and is often darker on the back from the rostrum to behind the dorsal fin. Bottlenose dolphins have the largest brain to body mass ratio of any mammal on Earth, sharing close ratios with those of humans and other great apes, which more than likely attributes to their incredibly high intelligence and emotional intelligence.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to *animals.oreilly.com*.

The cover image is from *British Quadrapeds*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.