O'REILLY®

Kubernetes

Scheduling the Future at Cloud Scale

David K. Rensin

Kubernetes

Scheduling the Future at Cloud Scale

David K. Rensin

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Kubernetes
by David Rensin

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://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com .

Editor: Brian Anderson
Production Editor: Matt Hacker
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

June 2015: First Edition

Revision History for the First Edition
2015-06-19: First Release
2015-09-25: Second Release

978-1-491-93188-2

[LSI]

Table of Contents

In The Beginning **1**
Introduction 1
Who I Am 2
Who I Think You Are 3
The Problem 3

Go Big or Go Home! **5**
Introducing Kubernetes—Scaling through Scheduling 5
Applications vs. Services 6
The Master and Its Minions 7
Pods 10
Volumes 12
From Bricks to House 14

Organize, Grow, and Go **15**
Better Living through Labels, Annotations, and Selectors 15
Replication Controllers 18
Services 21
Health Checking 27
Moving On 30

Here, There, and Everywhere **31**
Starting Small with Your Local Machine 32
Bare Metal 33
Virtual Metal (IaaS on a Public Cloud) 33
Other Configurations 34
Fully Managed 35

A Word about Multi-Cloud Deployments 36
Getting Started with Some Examples 36
Where to Go for More 36

In The Beginning…

Cloud computing has come a long way.

Just a few years ago there was a raging religious debate about whether people and projects would migrate en masse to public cloud infrastructures. Thanks to the success of providers like AWS, Google, and Microsoft, that debate is largely over.

Introduction

In the "early days" (three years ago), managing a web-scale application meant doing a lot of tooling on your own. You had to manage your own VM images, instance fleets, load balancers, and more. It got complicated fast. Then, orchestration tools like Chef, Puppet, Ansible, and Salt caught up to the problem and things got a little bit easier.

A little later (approximately two years ago) people started to really feel the pain of managing their applications at the VM layer. Even under the best circumstances it takes a brand new virtual machine at least a couple of minutes to spin up, get recognized by a load balancer, and begin handling traffic. That's a lot faster than ordering and installing new hardware, but not quite as fast as we expect our systems to respond.

Then came Docker.

Just In Case…

If you have no idea what containers are or how Docker helped make them popular, you should stop reading this paper right now and go here.

So now the problem of VM spin-up times and image versioning has been seriously mitigated. All should be right with the world, right? Wrong.

Containers are lightweight and awesome, but they aren't full VMs. That means that they need a lot of orchestration to run efficiently and resiliently. Their execution needs to be scheduled and managed. When they die (and they do), they need to be seamlessly replaced and re-balanced.

This is a non-trivial problem.

In this book, I will introduce you to one of the solutions to this challenge—Kubernetes. It's not the only way to skin this cat, but getting a good grasp on what it is and how it works will arm you with the information you need to make good choices later.

Who I Am

Full disclosure: I work for Google.

Specifically, I am the Director of Global Cloud Support and Services. As you might imagine, I very definitely have a bias towards the things my employer uses and/or invented, and it would be pretty silly for me to pretend otherwise.

That said, I used to work at their biggest competitor—AWS—and before that, I wrote a book for O'Reilly on Cloud Computing, so I do have *some* perspective.

I'll do my best to write in an evenhanded way, but it's unlikely I'll be able to completely stamp out my biases for the sake of perfectly objective prose. I promise to keep the preachy bits to a minimum and keep the text as non-denominational as I can muster.

If you're so inclined, you can see my full bio here.

Finally, you should know that the words you read are completely my own. This paper does not reflect the views of Google, my family, friends, pets, or anyone I now know or might meet in the future. I speak for myself and nobody else. I own these words.

So that's me. Let's chat a little about you…

Who I Think You Are

For you to get the most out of this book, I need you to have accomplished the following basic things:

1. Spun up at least three instances in somebody's public cloud infrastructure—it doesn't matter whose. (Bonus points points if you've deployed behind a load balancer.)
2. Have read and digested the basics about Docker and containers.
3. Have created at least one local container—just to play with.

If any of those things are not true, you should probably wait to read this paper until they are. If you don't, then you risk confusion.

The Problem

Containers are really lightweight. That makes them super flexible and fast. However, they are designed to be short-lived and fragile. I know it seems odd to talk about system components that are *designed* to not be particularly resilient, but there's a good reason for it.

Instead of making each small computing component of a system bullet-proof, you can actually make the whole system a lot *more* stable by assuming each compute unit *is* going to fail and designing your overall process to handle it.

All the scheduling and orchestration systems gaining mindshare now— Kubernetes or others—are designed first and foremost with this principle in mind. They will kill and re-deploy a container in a cluster if it even *thinks* about misbehaving!

This is probably the thing people have the hardest time with when they make the jump from VM-backed instances to containers. You just can't have the same expectation for isolation or resiliency with a container as you do for a full-fledged virtual machine.

The comparison I like to make is between a commercial passenger airplane and the Apollo Lunar Module (LM).

An airplane is meant to fly multiple times a day and ferry hundreds of people long distances. It's made to withstand big changes in altitude, the failure of at least one of its engines, and seriously violent

winds. Discovery Channel documentaries notwithstanding, it takes a *lot* to make a properly maintained commercial passenger jet fail.

The LM, on the other hand, was basically made of tin foil and balsa wood. It was optimized for weight and not much else. Little things could (and did during design and construction) easily destroy the thing. That was OK, though. It was meant to operate in a near vacuum and under very specific conditions. It could afford to be lightweight and fragile because it only operated under very orchestrated conditions.

Any of this sound familiar?

VMs are a lot like commercial passenger jets. They contain full operating systems—including firewalls and other protective systems—and can be super resilient. Containers, on the other hand, are like the LM. They're optimized for weight and therefore are a lot less forgiving.

In the real world, individual containers fail a lot more than individual virtual machines. To compensate for this, containers have to be run in managed clusters that are heavily scheduled and orchestrated. The environment has to detect a container failure and be prepared to replace it immediately. The environment has to make sure that containers are spread reasonably evenly across physical machines (so as to lessen the effect of a machine failure on the system) and manage overall network and memory resources for the cluster.

It's a big job and well beyond the abilities of normal IT orchestration tools like Chef, Puppet, etc….

Go Big or Go Home!

If having to manage virtual machines gets cumbersome at scale, it probably won't come as a surprise to you that it was a problem Google hit pretty early on—nearly ten years ago, in fact. If you've ever had to manage more than a few dozen VMs, this will be familiar to you. Now imagine the problems when managing and coordinating *millions* of VMs.

At that scale, you start to re-think the problem entirely, and that's exactly what happened. If your plan for scale was to have a staggeringly large fleet of identical things that could be interchanged at a moment's notice, then did it really matter if any one of them failed? Just mark it as bad, clean it up, and replace it.

Using that lens, the challenge shifts from configuration management to orchestration, scheduling, and isolation. A failure of one computing unit cannot take down another (isolation), resources should be reasonably well balanced geographically to distribute load (orchestration), and you need to detect and replace failures near instantaneously (scheduling).

Introducing Kubernetes—Scaling through Scheduling

Pretty early on, engineers working at companies with similar scaling problems started playing around with smaller units of deployment using cgroups and kernel namespaces to create process separation. The net result of these efforts over time became what we commonly refer to as containers.

Google necessarily had to create a lot of orchestration and scheduling software to handle isolation, load balancing, and placement. That system is called Borg, and it schedules and launches approximately 7,000 containers a *second* on any given day.

With the initial release of Docker in March of 2013, Google decided it was finally time to take the most useful (and externalizable) bits of the Borg cluster management system, package them up and publish them via Open Source.

Kubernetes was born. (You can browse the source code here.)

Applications vs. Services

It is regularly said that in the new world of containers we should be thinking in terms of services (and sometimes *micro-services*) instead of *applications*. That sentiment is often confusing to a newcomer, so let me try to ground it a little for you. At first this discussion might seem a little off topic. It isn't. I promise.

Danger—Religion Ahead!

To begin with, I need to acknowledge that the line between the two concepts can sometimes get blurry, and people occasionally get religious in the way they argue over it. I'm not trying to pick a fight over philosophy, but it's important to give a newcomer some frame of reference. If you happen to be a more experienced developer and already have well-formed opinions that differ from mine, please know that I'm not *trying* to provoke you.

A service is a process that:

1. is designed to do a small number of things (often just one).
2. has no user interface and is invoked solely via some kind of API.

An *application*, on the other hand, is pretty much the opposite of that. It has a user interface (even if it's just a command line) and often performs lots of different tasks. It can also expose an API, but that's just bonus points in my book.

It has become increasingly common for applications to call several services behind the scenes. The web UI you interact with at *https://www.google.com* actually calls several services behind the scenes.

Where it starts to go off the rails is when people refer to the web page you open in your browser as a web *application*. That's not necessarily wrong so much as it's just too confusing. Let me try to be more precise.

Your web browser is an application. It has a user interface and does lots of different things. When you tell it to open a web page it connects to a web server. It then asks the web server to do some stuff via the HTTP protocol.

The web server has no user interface, only does a limited number of things, and can only be interacted with via an API (HTTP in this example). Therefore, in our discussion, the web server is really a *service*—not an *application*.

This may seem a little too pedantic for this conversation, but it's actually kind of important. A Kubernetes cluster does not manage a fleet of applications. It manages a cluster of services. You might run an application (often your web browser) that communicates with these services, but the two concepts should not be confused.

A service running in a container managed by Kubernetes is designed to do a very small number of discrete things. As you design your overall system, you should keep that in mind. I've seen a lot of well meaning websites fall over because they made their services do too much. That stems from not keeping this distinction in mind when they designed things.

If your services are small and of limited purpose, then they can more easily be scheduled and re-arranged as your load demands. Otherwise, the dependencies become too much to manage and either your scale or your stability suffers.

The Master and Its Minions

At the end of the day, all cloud infrastructures resolve down to physical machines—lots and lots of machines that sit in lots and lots of data centers scattered all around the world. For the sake of explanation, here's a simplified (but still useful) view of the basic Kubernetes layout.

Bunches of machines sit networked together in lots of data centers. Each of those machines is hosting one or more Docker containers. Those worker machines are called *nodes*.

NOTE

Nodes used to be called *minions* and you will sometimes still see them referred to in this way. I happen to think they should have kept that name because I like whimsical things, but I digress…

Other machines run special coordinating software that schedule containers on the nodes. These machines are called masters. Collections of *masters* and nodes are known as *clusters*.

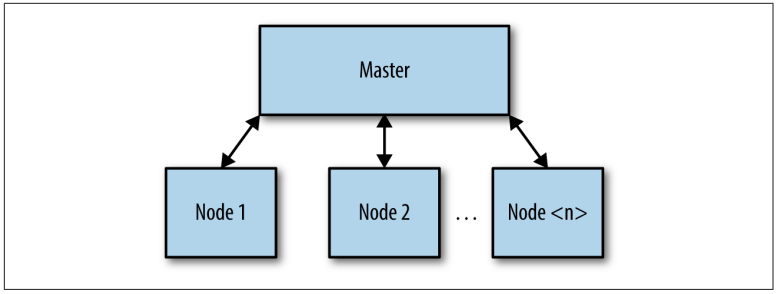

Figure 2-1. The Basic Kubernetes Layout

That's the simple view. Now let me get a little more specific.

Masters and nodes are defined by which software components they run.

The Master runs three main items:

1. **API Server**—nearly all the components on the master and nodes accomplish their respective tasks by making API calls. These are handled by the *API Server* running on the master.
2. **Etcd**—*Etcd* is a service whose job is to keep and replicate the current configuration and run state of the cluster. It is implemented as a lightweight distributed key-value store and was developed inside the CoreOS project.
3. **Scheduler and Controller Manager**—These processes schedule containers (actually, pods—but more on them later) onto target

nodes. They also make sure that the correct numbers of these things are running at all times.

A node usually runs three important processes:

1. **Kubelet**—A special background process (daemon that runs on each node whose job is to respond to commands from the master to create, destroy, and monitor the containers on that host.
2. **Proxy**—This is a simple network proxy that's used to separate the IP address of a target container from the name of the service it provides. (I'll cover this in depth a little later.)
3. **cAdvisor** (optional)—http://bit.ly/1izYGLi[Container Advisor (cAdvisor)] is a special daemon that collects, aggregates, processes, and exports information about running containers. This information includes information about resource isolation, historical usage, and key network statistics.

These various parts can be distributed across different machines for scale or all run on the same host for simplicity. The key difference between a master and a node comes down to who's running which set of processes.

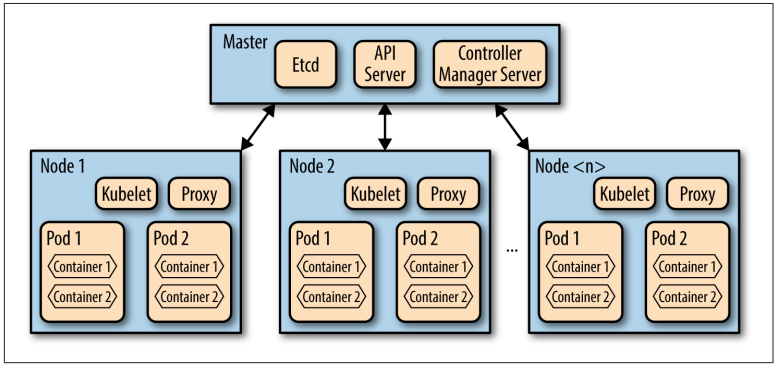

Figure 2-2. The Expanded Kubernetes Layout

If you've read ahead in the Kubernetes documentation, you might be tempted to point out that I glossed over some bits—particularly on the master. You're right, I did. That was on purpose. Right now, the important thing is to get you up to speed on the basics. I'll fill in some of the finer details a little later.

At this point in your reading I am assuming you have some basic familiarity with containers and have created a least one simple one with Docker. If that's **not** the case, you should stop here and head over to the main Docker site and run through the basic tutorial.

NOTE

I have taken great care to keep this text "code free." As a developer, I love program code, but the purpose of this book is to introduce the concepts and structure of Kubernetes. It's *not* meant to be a how-to guide to setting up a cluster.

For a good introduction to the kinds of configuration files used for this, you should look here.

That said, I will very occasionally sprinkle in a few lines of sample configuration to illustrate a point. These will be written in YAML because that's the format Kubernetes expects for its configurations.

Pods

A *pod* is a collection of containers and volumes that are bundled and scheduled together because they share a common resource—usually a filesystem or IP address.

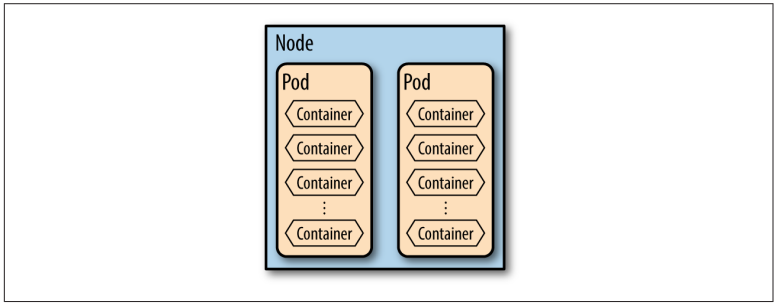

Figure 2-3. How Pods Fit in the Picture

Kubernetes introduces some simplifications with pods vs. normal Docker. In the standard Docker configuration, each container gets its own IP address. Kubernetes simplifies this scheme by assigning a *shared* IP address to the pod. The containers in the pod all share the same address and communicate with one another via *localhost*. In this way, you can think of a pod a little like a VM because it basically emulates a logical host to the containers in it.

This is a very important optimization. Kubernetes schedules and orchestrates things at the *pod level*, not the container level. That means if you have several containers running in the same pod they have to be managed together. This concept—known as *shared fate*—is a key underpinning of any clustering system.

At this point you might be thinking that things would be easier if you just ran processes that need to talk to each other in the same container.

You *can* do it, but I really *wouldn't*. It's a bad idea.

If you do, you undercut a lot of what Kubernetes has to offer. Specifically:

1. **Management Transparency**—If you are running more than one process in a container, then **you** are responsible for monitoring and managing the resources each uses. It is entirely possible that one misbehaved process can starve the others within the container, and it will be up to you to detect and fix that. On the other hand, if you separate your logical units of work into separate containers, Kubernetes can manage that for you, which will make things easier to debug and fix.
2. **Deployment and Maintenance**—Individual containers can be rebuilt and redeployed by you whenever you make a software change. That decoupling of deployment dependencies will make your development and testing faster. It also makes it super easy to rollback in case there's a problem.
3. **Focus**—If Kubernetes is handling your process and resource management, then your containers can be lighter. You can focus on your code instead of your overhead.

Another key concept in any clustering system—including Kubernetes—is *lack of durability*. Pods are **not** durable things, and you shouldn't count on them to be. From time to time (as the overall health of the cluster demands), the master scheduler may choose to *evict* a pod from its host. That's a polite way of saying that it will delete the pod and bring up a new copy on another node.

You are responsible for preserving the state of your application.

That's not as hard as it may seem. It just takes a small adjustment to your planning. Instead of storing your state in memory in some

non-durable way, you should think about using a shared data store like Redis, Memcached, Cassandra, etc.

That's the architecture cloud vendors have been preaching for years to people trying to build super-scalable systems—even with more long-lived things like VMs—so this ought not come as a huge surprise.

There is some discussion in the Kubernetes community about trying to add *migration* to the system. In that case, the current running state (including memory) would be saved and moved from one node to another when an eviction occurs. Google introduced something similar recently called *live migration* to its managed VM offering (Google Compute Engine), but at the time of this writing, no such mechanism exists in Kubernetes.

Sharing and preserving state between the containers in your pod, however, has an even easier solution: *volumes*.

Volumes

Those of you who have played with more than the basics of Docker will already be familiar with Docker volumes. In Docker, a *volume* is a virtual filesystem that your container can see and use.

An easy example of when to use a volume is if you are running a web server that has to have ready access to some static content. The easy way to do that is to create a volume for the container and prepopulate it with the needed content. That way, every time a new container is started it has access to a local copy of the content.

So far, that seems pretty straightforward.

Kubernetes also has volumes, but they behave differently. A Kubernetes volume is defined at the *pod level*—not the container level. This solves a couple of key problems.

1. **Durability**—Containers die and are reborn all the time. If a volume is tied to a container, it will also go away when the container dies. If you've been using that space to write temporary files, you're out of luck. If the volume is bound to the *pod*, on the other hand, then the data will survive the death and rebirth of any container in that pod. That solves one headache.

2. **Communication**—Since volumes exist at the pod level, any container in the pod can see and use them. That makes moving temporary data between containers super easy.

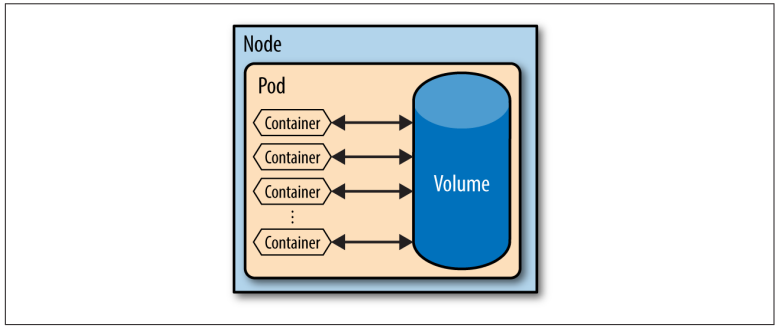

Figure 2-4. Containers Sharing Storage

Because they share the same generic name—*volume*—it's important to always be clear when discussing storage. Instead of saying "I have a volume that has…," be sure to say something like "I have a container volume," or "I have a pod volume." That will make talking to other people (and getting help) a little easier.

Kubernetes currently supports a handful of different pod volume types—with many more in various stages of development in the community. Here are the three most popular types.

EmptyDir

The most commonly used type is *EmptyDir*.

This type of volume is bound to the pod and is initially always empty when it's first created. (Hence the name!) Since the volume is bound to the pod, it only exists for the life of the pod. When the pod is evicted, the contents of the volume are lost.

For the life of the pod, every container in the pod can read and write to this volume—which makes sharing temporary data really easy. As you can imagine, however, it's important to be diligent and store data that needs to live more permanently some other way.

In general, this type of storage is known as *ephemeral*. Storage whose contents survive the life of its host is known as *persistent*.

Network File System (NFS)

Recently, Kubernetes added the ability to mount an NFS volume at the pod level. That was a particularly welcome enhancement because it meant that containers could store and retrieve important file-based data—like logs—easily and persistently, since NFS volumes exists beyond the life of the pod.

GCEPersistentDisk (PD)

Google Cloud Platform (GCP) has a managed Kubernetes offering named GKE. If you are using Kubernetes via GKE, then you have the option of creating a durable network-attached storage volume called a *persistent disk* (PD) that can also be mounted as a volume on a pod. You can think of a PD as a managed NFS service. GCP will take care of all the lifecycle and process bits and you just worry about managing your data. They are long-lived and will survive as long as you want them to.

From Bricks to House

Those are the basic building blocks of your cluster. Now it's time to talk about how these things assemble to create scale, flexibility, and stability.

Organize, Grow, and Go

Once you start creating pods, you'll quickly discover how important it is to organize them. As your clusters grow in size and scope, you'll need to use this organization to manage things effectively. More than that, however, you will need a way to find pods that have been created for a specific purpose and route requests and data to them. In an environment where things are being created and destroyed with some frequency, that's harder than you think!

Better Living through Labels, Annotations, and Selectors

Kubernetes provides two basic ways to document your infrastructure—*labels* and *annotations*.

Labels

A *label* is a key/value pair that you assign to a Kubernetes object (a pod in this case). You can use pretty well any name you like for your label, as long as you follow some basic naming rules. In this case, the label will decorate a pod and will be part of the *pod.yaml* file you might create to define your pods and containers.

Let's use an easy example to demonstrate. Suppose you wanted to identify a pod as being part of the front-end tier of your application. You might create a label named *tier* and assign it a value of *frontend* —like so:

```
"labels": {

"tier": "frontend"
```

```
}
```

The text "tier" is the key, and the text "frontend" is the value.

Keys are a combination of zero or more *prefixes* followed by a "/" character followed by a name string. The prefix and slash are optional. Two examples:

```
"application.game.awesome-game/tier"

"tier"
```

In the first case, "application.game.awesome-game" is the prefix, and "tier" is the name. In the second example there is no prefix.

You have to make sure that your labels conform to the same rules as regular DNS entries—known as DNS Labels.

The **prefix** part of the key can be one or more DNS Labels separated by "." characters. The total length of the prefix (including dots) cannot exceed 253 characters.

Values have the same rules but cannot be any longer than 63 characters.

Neither keys nor values may contain spaces.

Um…That Seems a Little "In the Weeds"

I'm embarrassed to tell you how many times I've tried to figure out why a certain request didn't get properly routed to the right pod only to discover that my label was too long or had an invalid character. Accordingly, I would be remiss if didn't at least *try* to keep you from suffering the same pain!

Label Selectors

Labels are queryable—which makes them especially useful in organizing things. The mechanism for this query is a label selector.

Heads Up!

You will live and die by your label selectors. Pay close attention here!

A label selector is a string that identifies which labels you are trying to match.

There are two kinds of label selectors—*equality-based* and *set-based*.

An equality-based test is just a "IS/IS NOT" test. For example:

```
tier = frontend
```

will return all pods that have a label with the key "tier" and the value "frontend". On the other hand, if we wanted to get all the pods that were **not** in the frontend tier, we would say:

```
tier != frontend
```

You can also combine requirements with commas like so:

```
tier != frontend, game = super-shooter-2
```

This would return all pods that were part of the game named "super-shooter-2" but were not in its front end tier.

Set-based tests, on the other hand, are of the "IN/NOT IN" variety. For example:

```
environment in (production, qa)
tier notin (frontend, backend)
partition
```

The first test returns pods that have the "environment" label and a value of either "production" or "qa". The next test returns all the pods **not** in the front end or back end tiers. Finally, the third test will return all pods that have the "partition" label—no matter what value it contains.

Like equality-based tests, these can also be combined with commas to perform an AND operation like so:

```
environment in (production, qa), tier notin (frontend, back-
end), partition
```

This test returns all pods that are in either the production or qa environment, also not in either the front end or back end tiers, and have a partition label of some kind.

Annotations

Annotations are bits of useful information you might want to store about a pod (or cluster, node, etc.) that you will not have to query

against. They are also key/value pairs and have the same rules as labels.

Examples of things you might put there are the pager contact, the build date, or a pointer to more information someplace else—like a URL.

Labels are used to store identifying information about a thing that you might need to query against. Annotations are used to store other arbitrary information that would be handy to have close but won't need to be filtered or searched.

It Might Be Boring, but…

I know that labeling and annotating bits of cluster infrastructure is nobody's idea of a hootenanny. You need to do it, though. Really. Label selectors are the central means of routing and orchestration, so you need to have good labeling hygiene to make things work well.

If you don't, your requests will probably never get routed correctly to your pods!

If you don't take the time upfront to label and annotate at least the big pieces, you will regret it dearly when it comes time to run your clusters day-to-day. You don't have to write War and Peace, but you need to write *something*.

Replication Controllers

If you're building an application that needs to handle a lot of load or have a lot of availability, you clearly are going to want more than one pod running at a time. This is no different than having multiple data centers (back in the days when we all ran our own hardware) or having multiple VMs running behind some kind of load-balancing service. It's just simple common sense.

Multiple copies of a pod are called *replicas*. They exist to provide scale and fault-tolerance to your cluster.

The process that manages these replicas is the *replication controller*. Specifically, the job of the replication controller is to make sure that the correct number of replicas are running at all times. That's its prime directive. Anything else it may do is to serve that end.

This means that the controller will create or destroy replicas as it needs to in order to maintain that objective.

The way the controller does this is by following a set of rules you define in a *pod template*. The pod template is a specific definition you provide about the desired state of the cluster. You specify which images will be used to startup each pod, how many replicas there will be, and other state-related things. (The *pod.yaml* file I referenced back in the **Labels** section is an example of this template.)

The Gestalt of a Replication Controller

The whole replication scheme in Kubernetes is designed to be *loosely coupled*. What I mean by that is that you don't actually tell a controller which pods you want it to control. Instead, you define the label selector it will use. Pods that match that query will be managed by the controller.

In addition, if you kill a replication controller it will **not** delete the replicas it has under management. (For that, you have to explicitly set the controller's *replicas* field to 0.)

This design has a number of interesting benefits:

- First, if you want to remove a pod from service for debugging—but not delete it—you just need to change its label so that it no longer matches the label selector used by the controller. That will lower the number of replicas managed by the controller by 1, so it will automatically start a new replica to compensate.
- Next, you can change the label selector used by the controller to cause it to assume control over an entirely different fleet in real-time.
- Finally, you can kill a replication controller and start a new one in its place and your pods will be none-the-wiser.

Nothing Lasts Forever

Replication controllers *will* die. Count on it. So plan accordingly!

You can usually count on a replication controller to be more long-lived than any one pod (and certainly more than any one container), but you should not think of them as invincible. They aren't. A well-

designed cluster has at least two controllers running at all times so that they can avoid any *single points of failure (SPOF)*.

That's a good thing, because SPOFs are the death of any high-availability system.

Scheduling != Scaling

Since replication controllers are concerned only with making sure that the correct numbers of pods are running at any one time, they are great for availability. All your replication controllers and clusters would have to fail (or be unreachable owing to network failure) for your service to be unavailable. That sounds pretty good, right?

What you may have noticed, however, is that we haven't talked about how you dynamically size your cluster as your load increases or decreases. With just the components we've discussed so far, you will always have a fixed-size cluster. That might be highly available, but it won't be very cost effective during off-peak hours.

For that, you will need something sitting in front of your cluster that understands your business logic for scaling—usually a load balancer or traffic shaper. The basic idea is that requests coming from your users will hit a front end load balancer and be directed to your cluster. As the traffic increases to the load balancer—commonly measured in *queries per second* (QPS)—the balancer will change the replicas value in the controller and more pods will be created. As the load abates, the reverse will happen.

There are some finer details I'm glossing over, but I'll cover them in more depth when we look at services. For those of you who want to dig into some really interesting reading about how a controller decides to evict and replace a pod, I recommend this article.

The Best Time to Use a Replication Controller Is…

Always! (Really).

Replication controllers are *always* a good idea—even for the simplest configurations.

Even if you have a situation where you only need to run one container in one pod, it's still a good idea to use a replication controller because when that pod unexpectedly dies (and it will from time to time) you want a new one to automatically take its place. It's a pretty

simple matter to create a controller with a replica count of 1 and then just let it run.

Replication controllers also make zero-downtime rolling updates much easier. Once upon a time, nearly every system went down for at least a few minutes once and a while for "scheduled maintenance." That's completely unacceptable now.

We expect our services to be available 24/7/365. Realistically, every service goes down at least a little bit every year so most really rock solid services aim for "five 9s" of uptime—99.999%. That means a service can only be unavailable .001% of the time—a mere *5.26 minutes a year*.

Good luck scheduling your maintenance in *that* window!

Replication controllers let us do cost-effective rolling updates. We start by bringing up a new controller with 1 updated replica and then removing 1 replica from the old controller. We keep doing this +1/-1 dance until the new controller has the number of replicas we need and the old controller is empty. Then we just delete the old controller.

If we're careful, we can make sure that the total number of replicas across both controllers never exceeds the capacity we wanted to pay for. It's an exceptionally cost-effective and safe way to roll out (and roll back) new code without having any scheduled downtime.

Services

Now you have a bunch of pods running your code in a distributed cluster. You have a couple of replication controllers alive to manage things, so life should be good.

Well…Almost…

The replication controller is only concerned about making sure the right number of replicas is constantly running. Everything else is up to you. Particularly, it doesn't care if your public-facing application is easily findable by your users. Since it will evict and create pods as it sees fit, there's no guarantee that the IP addresses of your pods will stay constant—in fact, they almost certainly will not.

That's going to break a lot of things.

For example, if you're application is multi-tiered, then unplanned IP address changes in your backend may make it impossible for your frontend to connect. Similarly, a load balancer sitting in front of your frontend tier won't know where to route new traffic as your pods die and get new IP addresses.

The way Kubernetes solves this is through services.

A *service* is a long-lived, well-known endpoint that points to a set of pods in your cluster. It consists of three things—an external IP address (known as a *portal*, or sometimes a *portal IP*), a port, and a label selector.

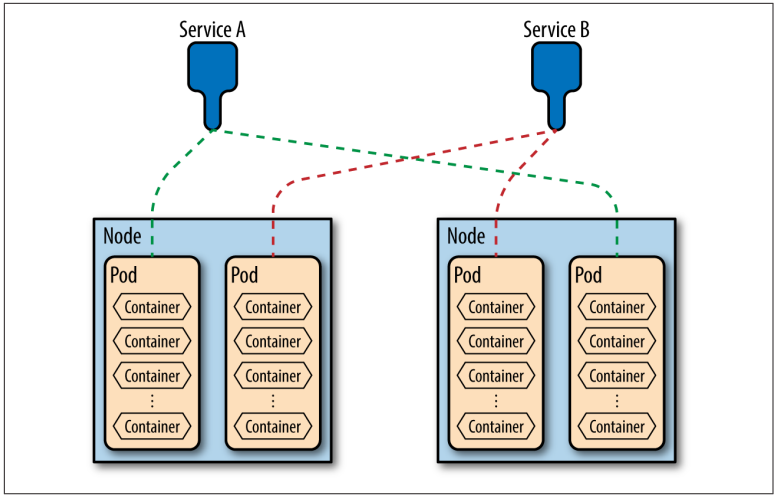

Figure 3-1. Services Hide Orchestration

The service is exposed via a small proxy process. When a request comes in for an endpoint you want to expose, the *service proxy* decides which pod to route it to via a label selector. Just like with a replication controller, the use of a label selector lets us keep fluid which pods will service which request.

Since pods will be created and evicted with unknown frequency, the service proxy acts as a thin lookup service to figure out how to handle requests. The service proxy is therefore nothing more than a tuple that maps a portal, port, and label selector. It's a kind of dictionary for your traffic, not unlike DNS.

The Life of a Client Request

There are enough moving parts to this diagram that now's a good time to talk about how they work together. Let's suppose you have a mobile device that is going to connect to some application API running in your cluster via REST over HTTPS. Here's how that goes:

1. The client looks up your endpoint via DNS and attempts a connection
2. More likely than not, that endpoint is some kind of frontend load balancer. This load balancer figures out which cluster it wants to route the request to and then sends the request along to the portal IP for the requested service.
3. The proxy service uses a label selector to decide which pods are available to send the request to and then forwards the query on to be serviced.

It's a pretty straightforward workflow, but its design has some interesting and useful features.

First, there's no guarantee that the pod that serviced one request will service the next one—even if it's very close in time or from the same client. The consequence of that is that you have to make sure your pods don't keep state ephemerally.

Second, there's no guarantee that the pod that serviced the request will even exist when the next request comes in. It's entirely possible that it will be evicted for some reason and replaced by the replication controller. That's completely invisible to your user because when that change happens the evicted pod will no longer match the service label selector and the new one will.

In practice, this happens in less than a second. I've personally measured this de-registration / eviction / replacement / registration cycle and found it to take on the order of 300 milliseconds. Compare that to replacing a running VM instance behind a load balancer. *That* process is almost always on the order of *minutes*.

Lastly, you can tinker with which pods service which requests just by playing with the label selector or changing labels on individual pods. If you're wondering why you'd want to do that, imagine trying to A/B test a new version of your web service in real-time using simple DNS.

You also might be wondering how a service proxy decides *which* pod is going to service the request if more than one matches the label selector. As of this writing, the answer is that it uses simple round-robin routing. There are efforts in progress in the community to have pods expose other run-state information to the service proxy and for the proxy to use that information to make business-based routing decisions, but that's still a little ways off.

Of course, these advantages don't just benefit your end clients. Your pods will benefit as well. Suppose you have a frontend pod that needs to connect to a backend pod. Knowing that the IP address of your backend pod can change pretty much anytime, it's a good idea to have your backend expose itself as a service to which only your frontend can connect.

The analogy is having frontend VMs connect to backend VMs via DNS instead of fixed IPs.

That's the best practice, and you should keep it in mind as we discuss some of the fine print around services.

A Few of the Finer Points about Integration with Legacy Stuff

Everything you just read is always true if you use the defaults. Like most systems, however, Kubernetes lets you tweak things for your specific edge cases. The most common of these edge cases is when you need your cluster to talk to some legacy backend like an older production database.

To do that, we have to talk a little bit about how different services find one another—from static IP address maps all the way to fully clustered DNS.

Selector-less Services

It *is* possible to have services that *do not* use label selectors. When you define your service you *can* just give it a set of static IPs for the backend processes you want it to represent. Of course, that removes one of the key advantages of using services in the first place, so you're probably wondering why you would ever do such a thing.

Sometimes you will have non-Kubernetes backend things you need your pods to know about and connect to. Perhaps you will need your pods to connect to some legacy backend database that is run-

ning in some other infrastructure. In that case you have a choice. You could:

1. Put the IP address (or DNS name) of the legacy backend in each pod, or
2. Create a service that doesn't route to a Kubernetes pod, but to your other legacy service.

Far and away, (2) is your better choice.

1. It fits seamlessly into your regular architecture—which makes change management easier. If the IP address of the legacy backend changes, you don't have to re-deploy pods. You just change the service configuration.
2. You can have the frontend tier in one cluster easily point to the backend tier in another cluster just by changing the label selector for the service. In certain high-availability (HA) situations, you might need to do this as a fallback until you get things working correctly with your primary backend tier.
3. DNS is slow (minutes), so relying on it will seriously degrade your responsiveness. Lots of software caches DNS entries, so the problem gets even worse.

Service Discovery with Environment Variables

When a pod wants to consume another service, it needs a way to do a lookup and figure out how to connect.

Kubernetes provides two such mechanisms—environment variable and DNS.

When a pod exposes a service on a node, Kubernetes creates a set of environment variables on that node to describe the new service. That way, other pods on the same node can consume it easily.

As you can imagine, managing discovery via environment variables is not super scalable, so Kubernetes gives us a second way to do it: Cluster DNS.

Cluster DNS

In a perfect world, there would be a resilient service that could let any pod discover all the services in the cluster. That way, different

tiers could talk to each other without having to worry about IP addresses and other fragile schemes.

That's where cluster DNS comes in.

You can configure your cluster to schedule a pod and service that expose DNS. When new pods are created, they are told about this service and will use it for lookups—which is pretty handy.

These DNS pods contains three special containers:

1. Etcd—Which will store all the actual lookup information
2. SkyDns—A special DNS server written to read from etcd. You can find out more about it here.
3. Kube2sky—A Kubernetes-specific program that watches the master for any changes to the list of services and then publishes the information into etcd. SkyDns will then pick it up.

You can instructions on how to configure this for yourself here.

Exposing Your Services to the World

OK!

Now your services can find each other. At some point, however, you will probably want to expose some of the services in your cluster to the rest of the world. For this, you have three basic choices: Direct Access, DIY Load Balancing, and Managed Hosting.

Option #1: Direct Access

The simplest thing for you to do is to configure your firewall to pass traffic from the outside world to the portal IP of your service. The proxy on that node will then pick which container should service the request.

The problem, of course, is that this strategy is not particularly fault tolerant. You are limited to just one pod to service the request.

Option #2: DIY Load Balancing

The next thing you might try is to put a load balancer in front of your cluster and populate it with the portal IPs of your service. That way, you can have multiple pods available to service requests. A common way to do this is to just setup instances of the super popular HAProxy software to handle this.

That's better, to be sure, but there's still a fair amount of configuration and maintenance you will need to do—especially if you want to dynamically size your load balancer fleet under load.

A really good getting-started tutorial on doing this with HAProxy can be found here. If you're planning on deploying Kubernetes on bare metal (as opposed to in a public cloud) and want to roll your own load balancing, then I would definitely read that doc.

Option #3: Managed Hosting

All the major cloud providers that support Kubernetes also provide a pretty easy way to scale out your load. When you define your service, you can include a flag named *CreateExternalLoadBalancer* and set its value to *true*.

When you do this, the cloud provider will automatically add the portal IPs for your service to a fleet of load balancers that it creates on your behalf. The mechanics of this will vary from provider to provider.

You can find documentation about how to do this on Google's managed Kubernetes offering (GKE) here.

Health Checking

Do you write perfect code? Yeah. Me neither.

One of the great things about Kubernetes is that it will evict degraded pods and replace them so that it can make sure you always have a system performing reliably at capacity. Sometimes it can do this for you automatically, but sometimes you'll need to provide some hints.

Low-Level Process Checking

You get this for free in Kubernetes. The Kubelet running on each node will talk to the Docker runtime to make sure that the containers in your pods are responding. If they aren't, they will be killed and replaced.

The problem, of course, is that you have no ability to finesse what it means for a container to be considered healthy. In this case, only a bare minimum of checking is occurring—e.g., whether the container process is still running.

That's a pretty low bar. Your code could be completely hung and non-responsive and still pass that test. For a reliable production system, we need more.

Automatic Application Level Checking

The next level of sophistication we can employ to test the health of our deployment is *automatic health checking*. Kubernetes supports some simple probes that it will run on your behalf to determine the health of your pods.

When you configure the Kubelet for your nodes, you can ask it to perform one of three types of health checks.

TCP Socket

For this check you tell the Kubelet which TCP port you want to probe and how long it should take to connect. If the Kubelet cannot open a socket to that port on your pod in the allotted time period, it will restart the pod.

HTTP GET

If your pod is serving HTTP traffic, a simple health check you can configure is to ask the Kubelet to periodically attempt an HTTP GET from a specific URL. For the pod to register as healthy, that URL fetch must:

1. Return a status code between 200 and 399
2. Return before the timeout interval expires

Container Exec

Finally, your pod might not already be serving HTTP, and perhaps a simple socket probe is not enough. In that case, you can configure the Kubelet to periodically launch a command line inside the containers in your pod. If that command exits with a status code of 0 (the normal "OK" code for a Unix process) then the pod will be marked as healthy.

Configuring Automatic Health Checks

The following is a snippet from a pod configuration that enables a simple HTTP health check. The Kubelet will periodically probe the

URL */_status/healthz* on port *8080*. As long as that fetch returns a code between 200-399, everything will be marked healthy.

```
livenessProbe:

  # turn on application health checking

  enabled: true

  type: http

  # length of time to wait for a pod to initialize

  # after pod startup, before applying health checking

  initialDelaySeconds: 30

  # an http probe

  httpGet:

    path: /_status/healthz

    port: 8080
```

Health check configuration is set in the *livenessProbe* section.

One interesting thing to notice is the *initialDelaySeconds* setting. In this example, the Kubelet will wait 30 seconds after the pod starts before probing for health. This gives your code time to initialize and start your listening threads before the first health check. Otherwise, your pods would never be considered healthy because they would always fail the first check!

Manual Application Level Checking

As your business logic grows in scope, so will the complexity of what you might consider "healthy" or "unhealthy." It won't be long before you won't be able to simply use the automatic health checks to maintain availability and performance.

For that, you're going to want to implement some business rule driven manual health checks.

The basic idea is this:

1. You run a special pod in your cluster designed to probe your other pods and take the results they give you and decide if they're operating correctly.
2. If a pod looks unhealthy, you change one of its labels so that it no longer matches the label selector the replication controller is testing against.
3. The controller will detect that the number of required pods is less than the value it requires and will start a replacement pod.
4. Your health check code can then decide whether or not it wants to delete the malfunctioning pod or simply keep it out of service for further debugging.

If this seems familiar to you, it's because this process is very similar to the one I introduced earlier when we discussed rolling updates.

Moving On

That covers the *what* and *how* parts of the picture. You know *what* the pieces are and *how* they fit together. Now it's time to move on to *where* they will all run.

Here, There, and Everywhere

So here we are, 30 pages or so later, and you now have a solid understanding of what Kubernetes is and how it works. By this point in your reading I hope you've started to form an opinion about whether or not Kubernetes is a technology that makes sense to you right now.

In my opinion, it's clearly the direction the world is heading, but you might think it's a little too bleeding edge to invest in right this second. That is only the first of two important decisions you have to make.

Once you've decided to keep going, the next question you have to answer is this: *do I roll my own or use someone's managed offering*?

You have three basic choices:

1. Use physical servers you own (or will buy/rent) and install Kubernetes from scratch. Let's call this option the *bare metal* option. You can take this route if you have these servers in your office or you rent them in a CoLo. It doesn't matter. The key thing is that you will be dealing with physical machines.
2. Use virtual machines from a public cloud provider and install Kubernetes on them from scratch. This has the obvious advantage of not needing to buy physical hardware, but is very different than the *bare metal* option, because there are important changes to your configuration and operation. Let's call this the *virtual metal* option.
3. Use one of the managed offerings from the major cloud providers. This route will allow you fewer configuration choices, but

will be *a lot* easier than rolling your own solution. Let's call this the *fully managed* option.

Starting Small with Your Local Machine

Sometimes the easiest way to learn something is to install it locally and start poking at it. Installing a full bare metal Kubernetes solution is not trivial, but you *can* start smaller by running all the components on your local machine.

Linux

If you're running Linux locally—or in a VM you can easily access—then it's pretty easy to get started.

1. Install Docker and make sure it's in your path. If you already have Docker installed, then make sure it's at least version 1.3 by running the *docker --version* command.
2. Install etcd, and make sure it's in your path.
3. Make sure go is installed and also in your path. Check to make sure your version is also at least 1.3 by running *go version*.

Once you've completed these steps you should follow along with this getting started guide. It will tell you everything you need to know to get up and running.

Windows/Mac

If you're on Windows or a Mac, on the other hand, the process is a little (but not much) more complicated. There are a few different ways to do it, but the one I'm going to recommend is to use a tool called Vagrant.

Vagrant is an application that automatically sets up and manages self-contained runtime environments. It was created so that different software developers could be certain that each of them was running an identical configuration on their local machines.

The basic idea is that you install a copy of Vagrant and tell it that you want to create a Kubernetes environment. It will run some scripts and set everything up for you. You can try this yourself by following along with the handy setup guide here.

Bare Metal

After you've experimented a little and have gotten the feel for installing and configuring Kubernetes on your local machine, you might get the itch to deploy a more realistic configuration on some spare servers you have lying around. (Who among us *doesn't* have a few servers sitting in a closet someplace?)

This setup—a fully bare metal setup—is definitely the most difficult path you can choose, but it does have the advantage of keeping absolutely everything under your control.

The first question you should ask yourself is do you *prefer* one Linux distribution over another? Some people are really familiar with Fedora or RHEL, while others are more in the Ubuntu or Debian camps. You don't *need* to have a preference—but some people do.

Here are my recommendations for soup-to-nuts getting-started guides for some of the more popular distributions:

1. **Fedora, RHEL**—There are many such tutorials, but I think the easiest one is here. If you're looking for something that goes into some of the grittier details, then this might be more to your liking.
2. **Ubuntu**—Another popular choice. I prefer this guide, but a quick Google search shows many others.
3. **CentOS**—I've used this guide and found it to be very helpful.
4. **Other**—Just because I don't list a guide for your preferred distribution doesn't mean one doesn't exist or that the task is undoable. I found a really good getting-started guide that will apply to pretty much *any* bare metal installation here.

Virtual Metal (IaaS on a Public Cloud)

So maybe you *don't* have a bunch of spare servers lying around in a closet like I do—or maybe you just don't want to have to worry about cabling, power, cooling, etc. In that case, it's a pretty straightforward exercise to build your own Kubernetes cluster from scratch using VMs you spin up on one of the major public clouds.

NOTE

This is a different process than installing on bare metal because your choice of network layout and configuration is governed by your choice of provider. Whichever bare metal guides you may have read in the previous section will only be *mostly* helpful in a public cloud.

Here are some quick resources to get you started.

1. **AWS**—The easiest way is to use this guide, though it also points you to some other resources if you're looking for a little more configuration control.
2. **Azure**—Are you a fan of Microsoft Azure? Then this is the guide for you.
3. **Google Cloud Platform (GCP)**—I'll bet it won't surprise you to find out that far and away the most documented way to run Kubernetes in the virtual metal configuration is for GCP. I found hundreds of pages of tips and setup scripts and guides, but the easiest one to start with is this guide.
4. **Rackspace**—A reliable installation guide for Rackspace has been a bit of a moving target. The most recent guide is here, but things seem to change enough every few months such that it is not always perfectly reliable. You can see a discussion on this topic here. If you're an experienced Linux administrator then you can probably work around the rough edges reasonably easily. If not, you might want to check back later.

Other Configurations

The previous two sections are **by no means** an exhaustive list of configuration options or getting-started guides. If you're interested in other possible configurations, then I recommend two things:

1. Start with this list. It's continuously maintained at the main Kubernetes Github site and contains lots of really useful pointers.
2. Search Google. Really. Things are changing a lot in the Kubernetes space. New guides and scripts are being published nearly every day. A simple Google search every now and again will keep you up to date. If you're like me and you absolutely want to

know as soon as something new pops up, then I recommend you set up a Google alert. You can start here.

Fully Managed

By far, your easiest path into the world of clusters and global scaling will be to use a fully managed service provided by one of the large public cloud providers (AWS, Google, and Microsoft). Strictly speaking, however, only one of them is actually Kubernetes.

Let me explain.

Amazon recently announced a brand new managed offering named Elastic Container Service (ECS). It's designed to manage Docker containers and shares many of the same organizing principles as Kubernetes. It does not, however, appear to actually *use* Kubernetes under the hood. AWS doesn't say what the underlying technology is, but there are enough configuration and deployment differences that it appears they have rolled their own solution. (If you know differently, please feel free to email me and I'll update this text accordingly.)

In April of 2015, Microsoft announced Service Fabric for their Azure cloud offering. This new service lets you build microservices using containers and is apparently the same technology that has been powering their underlying cloud offerings for the past five years. Mark Russinovich (Azure's CTO) gave a helpful overview session of the new service at their annual //Build conference. He was pretty clear that the underlying technology in the new service was not Kubernetes—though Microsoft has contributed knowledge to the project GitHub site on how to configure Kubernetes on Azure VMs.

As far as I know, the only fully managed Kubernetes service on the market among the large public cloud providers is Google Container Engine (GKE). So if your goal is to use the things I've discussed in this paper to build a web-scale service, then GKE is pretty much your only fully managed offering. Additionally, since Kubernetes is an open source project with full source code living on GitHub, you can really dig into the mechanics of how GKE operates by studying the code directly.

A Word about Multi-Cloud Deployments

What if you could create a service that seamlessly spanned your bare metal and several public cloud infrastructures? I think we can agree that would be pretty handy. It certainly would make it hard for your service to go offline under any circumstances short of a large meteor strike or nuclear war.

Unfortunately, that's still a little bit of a fairy tale in the clustering world. People are thinking hard about the problem, and a few are even taking some tentative steps to create the frameworks necessary to achieve it.

One such effort is being led by my colleague Quinton Hoole, and it's called Kubernetes Cluster Federation, though it's also cheekily sometimes called *Ubernetes*. He keeps his current thinking and product design docs on the main Kubernetes GitHub site here, and it's a pretty interesting read—though it's still early days.

Getting Started with Some Examples

The main Kubernetes GitHub page keeps a running list of example deployments you can try. Two of the more popular ones are the WordPress and Guestbook examples.

The WordPress example will walk you through how to set up the popular WordPress publishing platform with a MySQL backend whose data will survive the loss of a container or a system reboot. It assumes you are deploying on GKE, though you can pretty easily adapt the example to run on bare/virtual metal.

The Guestbook example is a little more complicated. It takes you step-by-step through configuring a simple guestbook web application (written in Go) that stores its data in a Redis backend. Although this example has more moving parts, it does have the advantage of being easily followed on a bare/virtual metal setup. It has no dependencies on GKE and serves as an easy introduction to replication.

Where to Go for More

There are a number of good places you can go on the Web to continue your learning about Kubernetes.

- The main Kubernetes homepage is here and has all the official documentation.
- The project GitHub page is here and contains all the source code plus a wealth of other configuration and design documentation.
- If you've decided that you want to use the GKE-managed offering, then you'll want to head over here.
- When I have thorny questions about a cluster I'm building, I often head to Stack Overflow and grab all the Kubernetes discussion here.
- You can also learn a lot by reading bug reports at the official Kubernetes issues tracker.
- Finally, if you want to contribute to the Kubernetes project, you will want to start here.

These are exciting days for cloud computing. Some of the key technologies that we will all be using to build and deploy our future applications and services are being created and tested right around us. For those of us old enough to remember it, this feels a lot like the early days of personal computing or perhaps those first few key years of the World Wide Web. This *is* where the world is going, and those of our peers that are patient enough to tolerate the inevitable fits and starts will be in the best position to benefit.

Good luck, and thanks for reading.

About the Author

Dave Rensin, Director of Global Cloud Support and Services at Google, also served as Senior Vice President of Products at Novitas Group, and Principal Solutions Architect at Amazon Web Services. As a technology entrepreneur, he co-founded and sold several businesses, including one for more than $1 billion. Dave is the principal inventor on 15 granted U.S. patents.

Acknowledgments

Everytime I finish a book I *solemnly swear* on a stack of bibles that I'll never do it again. Writing is hard.

I know. This isn't Hemingway, but a blank page is a blank page, and it will torture you equally whether you're writing a poem, a polemic, or a program.

Helping you through all your self-imposed (and mostly ridiculous) angst is an editor—equal parts psychiatrist, tactician, and task master.

I'd like to thank Brian Anderson for both convincing me to do this and for being a fine editor. He cajoled when he had to, reassured when he needed to, and provided constant and solid advice on both clarity and composition.

My employer—Google—encourages us to write and to generally contribute knowledge to the world. I've worked at other places where that was not true, and I really appreciate the difference that makes.

In addition, I'd like to thank my colleagues Henry Robertson and Daz Wilkins for providing valuable advice on this text as I was writing it.

I'd very much like to hear your opinions about this work—good or bad—so please feel free to contribute them liberally via O'Reilly or to me directly at rensin@google.com.

Things are changing a lot in our industry and sometimes it's hard to know how to make the right decision. I hope this text helps—at least a little.