

Kubernetes 101



by Jeff Geerling

The easiest introduction to
container-based infrastructure

Kubernetes 101

Learn the basics of Kubernetes and container-based infrastructure.

Jeff Geerling and Katherine Geerling

This book is for sale at <http://leanpub.com/kubernetes-101>

This version was published on 2021-07-29



Leanpub

This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2021 Jeff Geerling and Katherine Geerling

Tweet This Book!

Please help Jeff Geerling and Katherine Geerling by spreading the word about this book on [Twitter](#)!

The suggested tweet for this book is:

I just bought [Kubernetes 101](#) by Jeff Geerling on LeanPub!
Check it out at <https://leanpub.com/kubernetes-101>
[#Kube101](#)

The suggested hashtag for this book is [#kube101](#).

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

[#kube101](#)

Also By **Jeff Geerling**

Ansible for DevOps

Ansible for Kubernetes

Contents

Preface	i
Who is this book for?	ii
Typographic conventions	ii
Please help improve this book!	iv
Current Published Book Version Information .	iv
About the Author	iv
Introduction	v
Examples Repository	viii
Other resources	ix
Chapter 1 - Hello, Kubernetes!	1
Kubernetes Origins	1
Is Kubernetes Right for You?	3
Kubernetes Environments	5
Instructions for Minikube	6
Building the example Docker image	8
Chapter 2 - Containers	10
Why does Kubernetes use containers?	10
Container History: Vendor Wars	11
Docker, containerd, and runC	12
rkt and CoreOS	14
Kubernetes Container Runtime	14

CONTENTS

CRI-O	14
Modern Container Runtime options	15
How do you build a container? Docker vs Buildah	15
Instructions for ‘Hello Go’ app	16
Build the ‘Hello Go’ Docker container image	17
Push the container image to a private Docker registry	17
Chapter 3 - Deploying apps	19
Creating a Linode Cluster for cloud-based testing	19
Deploying Hello Go into Kubernetes	21
Exposing the Hello Go App	24
Scaling the Hello Go App	26
Updating the Go App	28
Rolling back the Deployment	29
Chapter 4 - Real-world apps	30
Installing Drupal on a Traditional LAMP server	30
LAMP Server Setup for drupal	31
Automating the Installation	34
Installing Drupal on Kubernetes using Bitnami’s Helm Chart	35
Install Helm	36
Install the Drupal Chart	36
Exposing a LoadBalancer in Minikube	37
Changing Chart Options	37
Cleaning Up	38
Drupal Directly in Kubernetes - Let’s Do it [Mostly	38
Deploying the Drupal Kubernetes Manifests	39
Chapter 5 - Scaling Drupal in k8s	43
Fixing the scalability issue with Drupal Pods	43
Shared Storage Options	44
Rook and Ceph	44
NFS	45

CONTENTS

Set up an NFS server	46
Reconfigure the Drupal PersistentVolumeClaim for NFS	46
Set up NFS client provisioner in K8s	47
Deploy Drupal and MySQL (MariaDB)	48
Save a File and observe it	49
Scale Drupal up... and down!	49
Use Horizontal Pod Autoscaling (HPA)	51
Set up metrics-server	51
Configuring HPA for Drupal	52
Testing HPA for Drupal	53
Scaling Databases	54
Chapter 6 - DNS, TLS, Cron, Logging	56
Setting things up from Episode 5	56
DNS and Ingress setup for Drupal	57
Set up an NGINX Ingress Controller	60
Set up Ingress for Drupal	61
External DNS Integration	61
Set up TLS with cert-manager and Let's Encrypt	62
Keeping Drupal Happy with a CronJob	65
Monitoring Drupal's Logs	68
Using an External SaaS Log Aggregator	68
Running your own ELK Stack	69
Relying on a Service Mesh	69
Using your cloud provider's solution	70
Chapter 7 - Hello, Operator!	71
What are Operators?	71
The Concept	72
The Execution	73
Why not use an Operator?	75
Popular Kubernetes Operators	76

CONTENTS

Build your own Operator	76
Building an Operator with Operator SDK . . .	78
Any language, including Python or Rust! . . .	83
Conclusion	83
Chapter 8 - Kube, Meet Pi	84
Heavy Metal Kubernetes	84
Start with Training Wheels	85
The Raspberry Pi makes for Compact Clusters	86
The Raspberry Pi sips energy, and keeps its cool	88
The Raspberry Pi teaches lessons about scalability	89
ARM is not all sunshine and roses	91
Installing a Kubernetes Distribution	92
kubeadm	93
Setting up the Raspberry Pi Dramble	94
Going Further	95
Other Guides	96
Chapter 9 - Secrets and Configuration	97
Chapter 10 - Monitoring Kubernetes	98
Two Clusters to Monitor	99
Cluster Visibility with Lens	99
Install Lens	100
Inspect your clusters with Lens	100
Explore Pod Logs	101
Log into Nodes and Pods	101
Visit web services in a browser	103
Manage resources	103
Prometheus and Grafana	104
Install Prometheus and Grafana using Helm .	104
Access Grafana	105
Grafana Dashboards	108

CONTENTS

Maintaining Grafana	108
Conclusion	108
Afterword	110

Preface

In 2020, the entire global community experienced a shock in the form of the COVID-19 pandemic.

Entire industries had to pivot overnight or risk ruin. Tech, fortunately for those of us in the industry, was suddenly in high demand.

Since I had a good deal of working knowledge about Kubernetes, and since I also finished a popular video series on Ansible earlier in the year, I decided to live-stream a series ‘Kubernetes 101’ to try to train people new to cloud-native infrastructure on this popular clustering software.

After many thousands of developers have viewed the video series, I decided to translate the content of the video series into this introductory-level book, so even more people could dive into Kubernetes and learn a new skill.

Whether you’re new to infrastructure, or you’re a veteran, Kubernetes can be complex and daunting. This book follows the progress of the video series, and starts with the basics, teaching each new concept with concrete, real-world examples.

I hope you enjoy the book as much as I enjoyed writing it!

— Jeff Geerling, 2021

Who is this book for?

If you are familiar with Linux-based infrastructure, and the basics of cloud computing (virtual machines, installing software, and deploying applications), then this book is for you.

You do not need to have any knowledge about containers, clustering, or networking; this book will teach you all the basics as we progress through containers, basic clustering, application deployment, and more.

Typographic conventions

Kubernetes uses a simple syntax (YAML) and simple command-line tools (using common POSIX conventions). Code samples and commands will be highlighted throughout the book either inline (for example: `kubectl [command]`), or in a code block (with or without line numbers) like:

```
1 ---  
2 # This is the beginning of a YAML file.
```

Some lines of YAML and other code examples require more than 70 characters per line, resulting in the code wrapping to a new line. Wrapping code is indicated by a `\` at the end of the line of code. For example:

```
1 # The line of code wraps due to the extremely long \  
2 URL .  
3 wget http://www.example.com/really/really/really/lo\  
4 ng/path/in/the/url/causes/the/line/to/wrap
```

When using the code, don't copy the `\` character, and make sure you don't use a newline between the first line with the trailing `\` and the next line.

Links to pertinent resources and websites are added inline, like the following link to [Kubernetes](#), and they can be viewed directly by clicking on them in eBook formats, or by following the URL in the footnotes.

Sometimes, asides are added to highlight further information about a specific topic:



Informational asides will provide extra information.



Warning asides will warn about common pitfalls and how to avoid them.



Tip asides will give tips for deepening your understanding or optimizing your use of Kubernetes.

When displaying commands run in a terminal session, if the commands are run under your normal/non-root user account, the commands will be prefixed by the dollar sign (`$`). If the commands are run as the root user, they will be prefixed with the pound sign (`#`).

Please help improve this book!

New revisions of this book are published on a regular basis (see current book publication stats below). If you think a particular section needs improvement or find something missing, please post an issue in the [Kubernetes 101 issue queue](#) (on GitHub).

All known issues with Kubernetes 101 will be aggregated on the book's online [Errata](#) page.

Current Published Book Version Information

- Current book version: 0.3
- Current Kubernetes version as of last publication: 1.21
- Current Date as of last publication: June 12, 2021

About the Author

Jeff Geerling is a developer who has worked in programming and reliability engineering, building hundreds of apps and services in various cloud and on-premise environments. He also manages many services offered by Midwestern Mac, LLC and has been using Kubernetes since 2017.

Introduction

To understand the need for Kubernetes, we need to take a time machine back to the days where we would have have one or two giant servers running all our applications.



This server was so important we named it!

The server you see in this photo is one that I helped manage at a radio station in St. Louis. We had a name for the server—we had names for all of our servers back then—because the server was very, very important.

Once this server was set up, we kept it running forever, as long as people still needed to access the applications that server hosted.

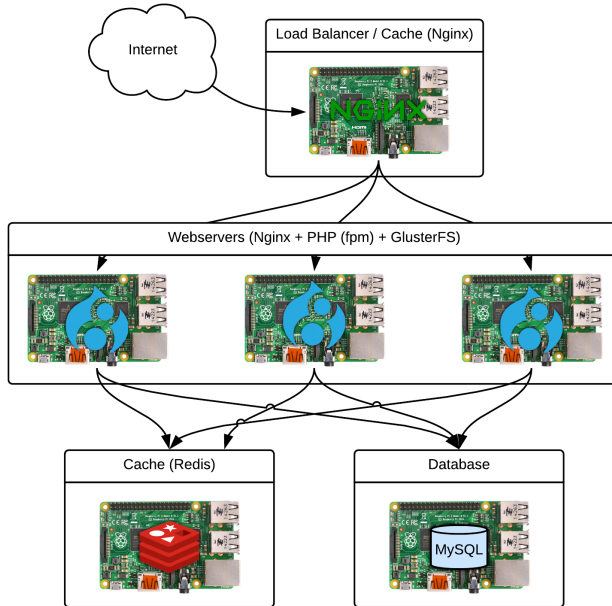
Clearly, that system had problems:

- If a power supply or hard drive failed, we would scramble to get it replaced and maybe have to restore from a slow tape backup.
- Critical changes had to be done with special care, and during odd hours early in the morning (with sleep-deprived engineers running the show) to try to avoid interruptions to business workflows.

This infrastructure was tedious, but it was the best we could do at the time.

As time went on, applications themselves focused more on redundancy and availability, and server architecture was modified for better stability and better separation of duties, with multiple redundant servers taking the place of a single server.

I have a good example of this type of architecture in the early version of my ‘Raspberry Pi Dramble’ cluster highlighted later in this book:



Original Raspberry Pi cluster architecture

Many web applications have a similar server architecture:

- A proxy or load-balancing server at the front.
- Backend web servers handling the application logic.
- Storage servers for files.
- Database servers for relational data storage.

Complexity increased with this type of clustering. Virtualized ‘cloud’ computing became extremely popular, as internal teams had trouble building out servers at the rate applications needed them.

Even with virtualization, it was daunting managing tens, hundreds, and now *thousands* of different applications.

Tools like Ansible, Chef, Puppet and management tools from AWS and other cloud providers automate and abstract things away to make them more manageable, and for some organizations, this level of automation is sufficient.

But Kubernetes takes this idea of automated cluster management to the logical end.

Instead of managing servers and the relationships between them externally and infrequently, it abstracts *everything about the system* into code, fulfilling the dream of having true ‘Infrastructure as Code’.

Kubernetes sees a pool of servers, and application (‘workload’) definitions. It’s scheduler assigns applications to your servers in the most efficient manner, and wires them together, even integrating them with external databases, load balancers, and 3rd party services.

It even handles things like auto-scaling resources, and DNS and certificate management for web applications!

There’s a good reason Kubernetes has taken hold of the industry like wildfire—though it is complex, it solves a huge number of real business needs. And it has become much easier to deploy and manage, whether you run one cluster or many.

Examples Repository

There are many code examples (manifests, configuration, etc.) throughout this book. Most of the examples are in the [Kubernetes 101 GitHub repository](#), so you can browse the code

in its final state while you're reading the book. Some of the line numbering may not match the book *exactly* (especially if you're reading an older version of the book!), but I will try my best to keep everything synchronized over time.

Other resources

We'll explore all aspects of using Kubernetes in this book, but there's no substitute for the wealth of documentation and community interaction that make these tools great. Check out the links below to find out more about the tools and their communities:

- [Kubernetes Documentation](#) - Covers Kubernetes usage patterns in depth.
- [Kubernetes Glossary](#) - If there's ever a term in this book you don't seem to fully understand, check the glossary.
- [Kubernetes SIGs and Working Groups](#) - These groups are where major changes and new features are discussed—consider joining one of these groups if the topic is of importance to you, or just follow along with the groups you're interested in.
- [Kubernetes on GitHub](#) - The official Kubernetes code repository, where the magic happens.
- [Kubernetes on Slack](#) - Chat with other Kubernetes users in the official Slack.
- [Kubernetes Blog](#)

The official documentation is continually updated and is very thorough. This book is meant as a supplement to, not a replacement for, the official documentation!

Chapter 1 - Hello, Kubernetes!

In the introduction, we learned where Kubernetes came from—the need for better management of an ever-increasing array of applications running on an ever-more-diverse and expansive set of servers in clusters, whether on-premise or in a cloud.

The basic unit that we'll deal with early in our Kubernetes journey is the Pod.

Pods are deployed to individual servers by Kubernetes, and they can contain one or more containers.

We'll get to all what all that means later, but Kubernetes distributes workloads on your servers using Pods, and can move Pods wherever there is capacity.

Kubernetes Origins

In the early 2000s, Google introduced a cluster management service they called Borg. It ran hundreds of thousands of jobs and services on their internal clusters, helming a “collective” of tens of thousands of Google's servers. Some of the engineers decided to turn Borg into a more accessible open source tool, capable of running outside Google's infrastructure, and built a new version of Borg using the up-and-coming Go programming language.

The engineers originally wanted to call the new software “Project Seven” after another Borg character, Seven of Nine, but they didn’t want to have the new software encumbered by lawyers suing over the rights to a Star Trek character’s name, so they decided to change it slightly, but still keep the spirit of that progression of naming. So they called it “Kubernetes”.



The Kubernetes logo

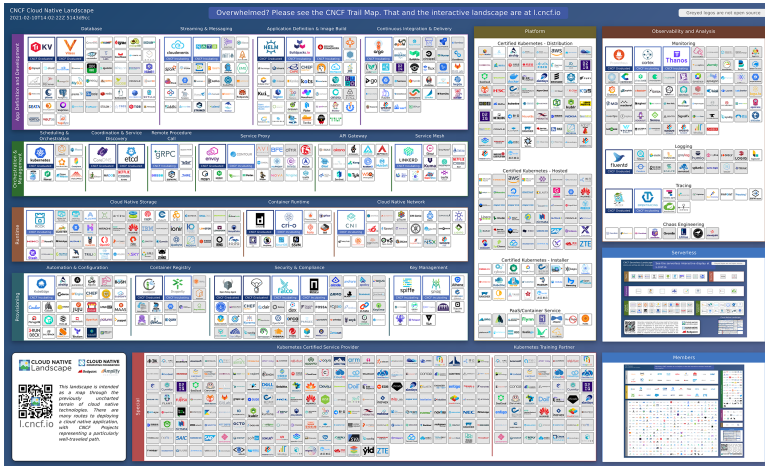
The Kubernetes logo has seven spokes around a central hub, hearkening back to “Project Seven”. And the helm is a callback to the nautical terminology prevalent in Star Trek lore. In the Kubernetes ecosystem, you’ll find a lot of nautical terminology for the same reason.

For instance, “Helm” was the first big package manager for Kubernetes, and is still very popular today.

Something else you might see a lot is the term ‘k8s’, which is a *numeronym*. These terms, like “a11y” for “accessibility”, work out to the first and last letter of a word or phrase, with the number of letters between in the middle. Often, ‘k8s’ and ‘Kubernetes’ are used interchangeably.

A year after Kubernetes’ debut, Google and the Linux Founda-

tion created the Cloud Native Computing Foundation (CNCF). The CNCF was created to foster development and collaboration in the realm of cloud native open source projects.



Kubernetes was the first CNCF project, but the overall ‘landscape’ (pictured above) has grown substantially as more cloud-native tools and vendors have joined the fray.

But don’t be worried if that image is overwhelming; to be productive in Kubernetes, you don’t need to know about more than a few of the projects in the CNCF landscape!

Is Kubernetes Right for You?

Kubernetes is a powerful tool, but it does have a learning curve, and requires more baseline infrastructure to run. There are times when it’s exactly the right tool for the job, and there are times when it is best to ignore it, especially for simpler applications.

My own website, jeffgeerling.com, is good example of a project that *doesn't* need Kubernetes; it gets a decent amount of traffic, but it runs great on a single VPS running Nginx, PHP, and MariaDB.

My website doesn't need 'five nines' of uptime, but even so, I can restore it from a backup within less than an hour, because I manage the server's configuration using Ansible. That setup has worked for nearly a decade, and it will probably work just fine for a decade more, on a single server.

On the other hand, I also operate two 'Software as a Service' products, both of which run applications spanning dozens of servers, with a highly dynamic environment.

Traditionally, I would use many automation scripts to pass data between the servers, move applications as needed, and try to keep track of server outages as well as I could.

Kubernetes is an excellent fit for these systems, because it takes care of migrating my apps between servers, handling fleet-wide upgrades, and making sure all these various applications stay running their best, with no outside intervention.

I'm already paying for dozens of larger servers, so the overhead Kubernetes requires for a high-availability control plane, and the overall operational complexity, is much less in proportion to the needs of the application.

My personal website's server cost would likely quadruple, with no tangible benefit, if I moved to Kubernetes.

My SaaS products' infrastructure costs would increase by barely a few percentage points, but the time and server scheduling efficiency savings would lead to a much greater savings over the long term.

On top of that, if you choose to manage your own Kubernetes infrastructure (without using a managed provider), you take on additional maintenance overhead keeping everything up to date and running efficiently.

Kubernetes Environments

Okay, so we know a bit about Kubernetes' history, and how to determine whether it's the right fit for our project. Let's start *using* Kubernetes!

But wait, the first problem you run into is there are actually *many* different 'flavors' of Kubernetes you can use when you want to try it out.

One easy way to get started with the full-fledged Kubernetes distribution is to set up a cluster using [Kubeadm](#). This is the official tool from Kubernetes, designed to bootstrap Kubernetes clusters on bare metal or cloud instances. It doesn't include any extra frills, it just gets Kubernetes' control plane running and attaches worker nodes to it.

But the full Kubernetes install is a bit heavyweight, requiring a couple gigs of RAM just to run Kubernetes, and that's *before* you add any of your applications running on top of it.

So there are more lightweight distributions of Kubernetes like Rancher's [K3s](#), which even runs well on ARM single-board computers like the Raspberry Pi!



We'll cover running Kubernetes on Raspberry Pis in chapter 8!

Kubernetes-in-Docker, otherwise known as [kind](#), is particularly useful for Kubernetes development, and I use it for

almost all the continuous integration (CI) jobs I run. It builds Kubernetes clusters using one or more Docker containers, and is quick to build and tear down, but it lacks some conveniences that make it difficult to use for general Kubernetes work, and is not meant for production environments.

For most people, [Minikube](#) is the most user-friendly way to develop and test things in Kubernetes locally. It's easy to install and includes many addons to make Kubernetes development efficient on your computer.

There are a lot of other distributions I haven't mentioned which may interest you, like OpenShift, which requires even more resources but is supported by RedHat, MicroK8s, another lightweight distribution by Canonical, or even k0s, yet *another* lightweight distribution, distributed as a single binary.

Now that you know some of the common Kubernetes distribution, it's time to build our first Kubernetes cluster—with Minikube!

Instructions for Minikube

[Minikube](#) and [kubect1](#) are all we need to build our first local Kubernetes cluster:

1. Install Minikube: `brew install minikube` (on a Mac with [Homebrew](#))
2. Install kubect1: `brew install kubect1`
3. Start a Minikube cluster: `minikube start`

Then you can check on the cluster's state, to make sure all the nodes—in this case, just one master node—are running and ready:

```
$ kubectl get nodes
NAME          STATUS    ROLES    AGE   VERSION
minikube      Ready     master   91s   v1.19.4
```

Now that we have a running cluster, it's time to deploy a lightweight application to it, just to make sure it's working:

```
$ kubectl create deployment hello-k8s --image=geerl\
ingguy/kube101:intro
deployment.apps/hello-k8s created
```

It seems like it's deploying correctly, so the next step is to make it so we can access the deployed application from outside the cluster. By default, Kubernetes sets up an internal network, but does not expose any of your applications to the outside world. Here's how to 'expose' the deployment to the outside using a Kubernetes service:

```
$ kubectl expose deployment hello-k8s --type=NodePo\
rt --port=80
service/hello-k8s exposed
```

We'll get into what NodePort means later, but for now, we should be able to access the deployment from our computer. Minikube has a handy command that will open up the service in a web browser directly:

```
$ minikube service hello-k8s
<should launch your web browser>
```

But you could also find the IP address for the cluster using `minikube ip`, then pair that with the high-numbered port that is returned when you run `kubectl get services hello-k8s`.

When you're finished using the cluster, run `minikube halt` to stop it, or `minikube delete` to delete the cluster.

Building the example Docker image

There is a `Dockerfile` in this directory, which is used by GitHub Actions to build the [geerlingguy/kube101:intro image on Docker Hub](#).

That image is used in this chapter to demonstrate a simple Kubernetes deployment.

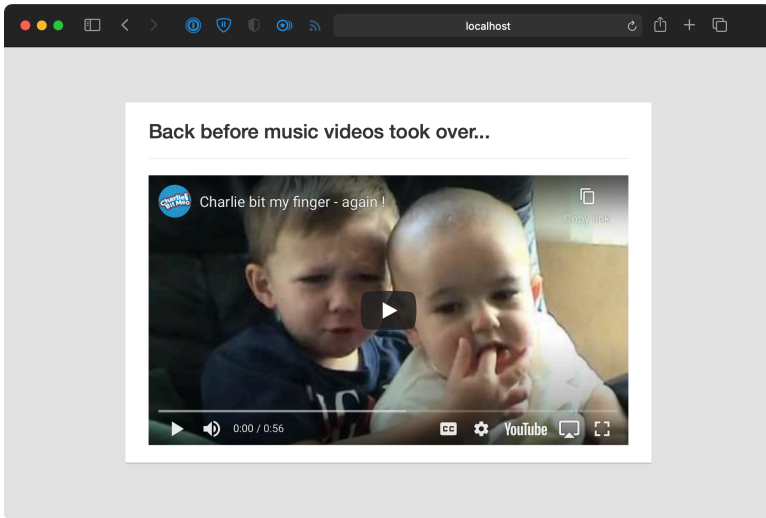
If you want to build the image on your own, locally, you can run:

```
$ docker build -t geerlingguy/kube101:intro
```

And to run the image on its own, run:

```
$ docker run -d -p 80:80 geerlingguy/kube101:intro
```

Once it's running, access the demo page at `http://localhost`, and you should see one of the most popular YouTube videos from the days before music videos and vlogging took over the platform:



YouTube used to be a simpler place

Chapter 2 - Containers

Containers are the building block of any Kubernetes cluster. In this chapter, we're going to learn what containers are, how to build them, and how to manage container images in a registry.

We'll even build a custom Go application and show how it can be deployed in a container image.

Why does Kubernetes use containers?

Containers solve many of the pain points inherent to deploying applications to servers.

First and foremost, for developers, containers help solve the 'works on my machine' problem. I know when I started in programming, the first week or two on a project were typically spent setting up local tools and maybe even a build system just so I could start developing an app on my workstation.

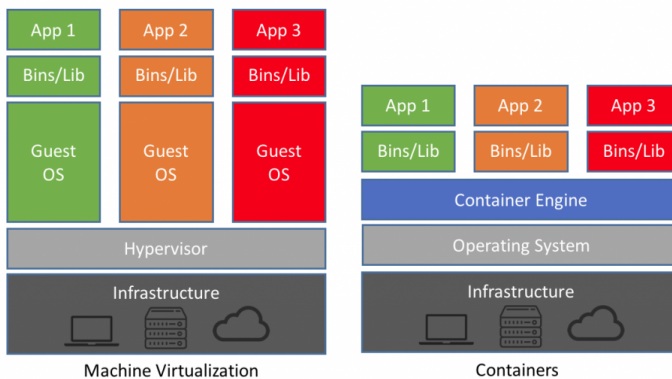
If well-documented, this process was not prone to failure, but it was still burdensome. But in reality, so often projects relied on many tools and build steps that were fragile and prone to breaking across different systems, so even keeping an app running locally was painful.

Containers solve that problem by making application environments:

- Portable
- Isolated
- Consistent
- Lightweight

And they do that by throwing away certain things like stateful data and full resource virtualization and isolation like a Virtual Machine has.

Containers do work with stateful applications, but do so by defining clear boundaries, like what directories or devices are mounted inside the container for persistent data.



Containers are *not* full virtual machines, and even though you may be able to simulate a VM-like environment inside a Kubernetes cluster to support a legacy application, that topic will not be covered in this book.

Container History: Vendor Wars

‘Containers’ as we know them today have actually been a capability of BSD and Linux for years prior to the more

standardized container management tool ‘Docker’.

You could use LXC and cgroups or jails to maintain some level of resource isolation and lightweight virtualization.

But there wasn’t a common standard for how to build these environments or share them between developers and operations teams.

Docker, containerd, and runC

Released in 2013, Docker standardized container definitions in a ‘Dockerfile’, and built Docker Hub, a central registry for container images.

Docker was quickly adopted by developers who loved the lightweight portable environments, but it took longer to gain acceptance in production environments, because the early monolithic versions were sometimes less than rock solid.

Also, since adoption was initially driven by the development side, early usage of containers often led to ‘shipping everything’ in a container image for ease of use:



Docker evolved over the years to be more efficient and stable, developers (for the most part) started building more secure and efficient containers, and the major components of the monolith (running containers, building containers, the CLI, etc.) were all broken up into more manageable projects.

Modern Docker uses `containerd` as a 'container runtime', managing all aspects of container images and management. `containerd` in turn launches containers using an 'OCI-compliant' container runtime (we'll talk about what that means soon), most often `runC`.

Both of these tools are part of 'Docker Engine', which is also installable on workstations as part of 'Docker Desktop', which adds on an easy-to-use container management UI.

rkt and CoreOS

Around the time Docker was seeing more production usage, CoreOS built `rkt`, which was a more ‘cloud-native’ competitor to Docker’s own engine. `rkt` also added concepts like ‘Pods’, which allowed multiple containers to run with a shared context.

CoreOS was acquired by Red Hat in 2018, though, and `rkt` and many of the other CoreOS tools built up around the container ecosystem were deprecated and eventually end-of-lifed.

Kubernetes Container Runtime

And this brings us to Kubernetes’ ‘Container Runtime Interface’. In the midst of the vendor wars over different container engines like Docker and `rkt`, the Kubernetes community and CNCF decided to work on standardizing the interface between clustering software like Kubernetes and running containers.

One of the fruits of the ‘Open Container Initiative’ (OCI) was the ‘[Container Runtime Interface](#)’, which defined how container runtimes interact with Kubernetes.

CRI-O

Some of the concepts behind `rkt` were inherited by Red Hat’s follow-on cloud-native runtime, CRI-O.

CRI-O’s early claim to fame was being lightweight compared to the Docker runtime, and built entirely inside the ‘cloud-native’ ecosystem, tailored to Kubernetes workloads.

Modern Container Runtime options

Today, Kubernetes clusters typically use either CRI-O or containerd as their container runtime.

What does that mean for you? Well, if you're not building a custom Kubernetes cluster, not much. As we'll find in a minute, the building and management of container images *outside* of a running cluster uses different tools entirely!

But it is helpful to at least understand the heritage and goals of different container tools that are at the heart of Kubernetes' architecture.

For even more depth on this topic, please check out Evan Baker's excellent overview, [A Comprehensive Container Runtime Comparison](#), on the Capital One website.

How do you build a container? Docker vs Buildah

Example: Webserver container from Chapter 1

```
$ docker images
```

```
$ docker build -t geerlingguy/kube-101:intro
```

```
$ docker images
```

```
$ docker run --rm -p 80:80 geerlingguy/kube101:intro
```

Instructions for 'Hello Go' app

There is a very simple Go-based web app that responds to HTTP requests on port 8180 in `cmd/hello/hello.go`.

After [installing Go](#), you can run the app directly with the command:

```
$ go run cmd/hello/hello.go
```

Or you can build the Go command hello binary using:

```
$ go build cmd/hello/hello.go
```

And then run it and monitor requests (access `localhost:8180/some-path-here` in a browser):

```
$ ./hello
2028/10/24 17:30:36 Starting webserver on :8180
2028/10/24 17:30:59 Received request for path: /som\
e-path-here
```

After you're finished, you can remove the binary with `rm hello`.

Build the 'Hello Go' Docker container image

Next up, we want to set up a container build environment that can build the Go application and then also run it (but without all the Go language cruft) in a trimmed down container image.

There is a `Dockerfile` in this directory containing a multi-stage Docker build layout which first builds the Go app using the official golang Docker image, then builds the final container based on Alpine Linux (using the official alpine Docker image).

To build the container, run:

```
$ docker build -t geerlingguy/kube101-go .
```

Once the container is built, you can see it in your list of docker images, and you can run it with the command:

```
$ docker run --rm -p 8180:8180 geerlingguy/kube101-go
```

Push the container image to a private Docker registry

When you're satisfied the container image works correctly, go ahead and push it up to a Docker registry.

For my example, I'm pushing it to a private Docker Hub repository named `geerlingguy/kube101-go`:

```
$ docker push geerlingguy/kube101-go
```

Note: Pushing to a registry typically requires authentication. Please read the documentation for a guide on how to make sure you are authenticated to your Docker Hub (or other provider) account.

Also, it's likely you won't be able to push to my namespace, so you might want to try using your own namespace instead of geerlingguy ;-)

Chapter 3 - Deploying apps

In this chapter, we're going to deploy, expose, scale, update, and roll back an app in Kubernetes. We'll use the Hello Go app and container image from Chapter 2 to learn how to manage apps into Kubernetes.

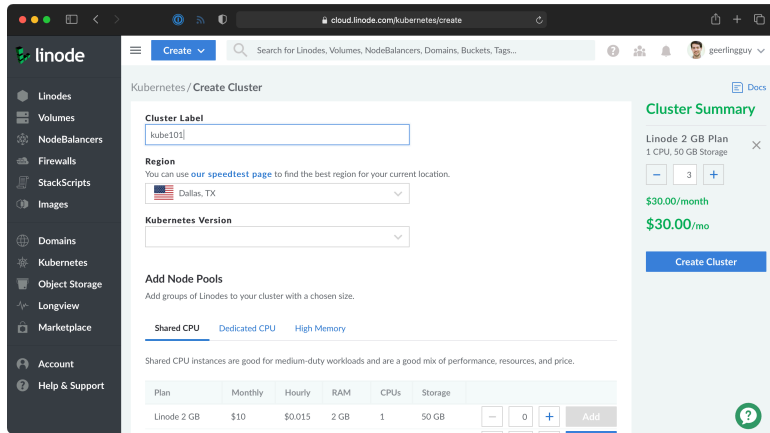
First, we'll go over how to create a cluster in the cloud for testing, in case you can't install Minikube locally.

Creating a Linode Cluster for cloud-based testing

If you're not able to install Minikube, you can use free credit from Linode to build small Kubernetes clusters for development and testing.

To do that, visit: <https://linode.com/geerling>.

Sign up for a new account, then in the Linode Cloud control panel, go to the Kubernetes section. Create a new cluster, add a few nodes to the default node pool, and click "Create".



Creating a Kubernetes cluster on Linode

The process takes about two minutes, and at the end, you can download the ‘Kubeconfig’ file to your computer, so you can use it with `kubectl` to administer the cluster.

On my computer, I copied the file into my `.kube` directory:

```
$ mv ~/Downloads/kube101-kubeconfig.yaml ~/.kube/config-kube101
```

Then I made sure `kubectl` would use this cluster config:

```
$ export KUBECONFIG=~/.kube/config-kube101
```

Now if I run any `kubectl` commands, they will work on the new Linode cluster!

```
$ kubectl get nodes
```

NAME	STATUS	ROLES	AGE\
VERSION			
lke14312-17562-5fc5708862ca	Ready	<none>	1h \
v1.18.8			
lke14312-17562-5fc570886f7d	Ready	<none>	1h \
v1.18.8			
lke14312-17562-5fc57088781f	Ready	<none>	1h \
v1.18.8			

Deploying Hello Go into Kubernetes



The example you're about to run will fail. Don't worry, that failure is an important part of this chapter!

After creating a local cluster with `minikube start`, or a cloud-based cluster, it's time to deploy 'Hello Go' into the cluster!

You can do that with:

```
$ kubectl create deployment hello-go --image=geerli\ngguy/kube101-go:1.0.0
```

Then watch the deployment status with `watch kubectl get deployment hello-go`.

Hmm... the rollout seems to be failing, as it is not showing that the Deployment is reaching a 'ready' state:

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
hello-go	0/1	1	0	51s

Let's check on the Pods for this Deployment and see what might be happening:

```
$ kubectl get pod -l app=hello-go
```

You'll likely see a status of `ErrImagePull` or `ImagePullBackOff`. Let's dig in deeper and use the `kubectl describe` command to get the details:

```
$ kubectl describe pod -l app=hello-go
Name:          hello-go-5944979865-qfxfc
Namespace:     default
...
Warning Failed      2s (x3 over 43s)  kubelet  \
Failed to pull image "geerlingguy/kube101-go:1.0.0": rpc error: code = Unknown desc = Error response from daemon: pull access denied for geerlingguy/kube101-go, repository does not exist or may require 'docker login': denied: requested access to the resource is denied
```

Ah, so Kubernetes can't pull that image, since it's in a private Docker registry. We'll have to tell Kubernetes how to authenticate to Docker Hub, since that's where the image lives. And we can do that using a special kind of Kubernetes Secret called a Docker Registry secret:

```
$ kubectl create secret docker-registry regcred \
  --docker-username=geerlingguy \
  --docker-password=[TOKEN GOES HERE] \
  --docker-email=geerlingguy@mac.com
```

For Docker Hub, you would put in your username, password (an Authentication Token, which can be generated in your Account Settings in the ‘Security’ section), and email address. For other registries, you would also need to add a `--docker-server` URL.

Once you create the secret, you need to modify the `hello-go` deployment to make sure it knows to use that secret.

So edit the deployment with:

```
$ kubectl edit deployment hello-go
```

And add a new `imagePullSecrets:` section under `spec.template.spec` like so:

```
spec:
  ...
  template:
    ...
    spec:
      imagePullSecrets:
        - name: regcred
      containers:
        - image: geerlingguy/kube101-go:1.0.0
```

Then check on the progress of the rollout with `watch kubectl get deployment hello-go`. Assuming you’re *me*, it would now

work, because you would have a valid access token that grants you access to the private geerlingguy/kube101-go image.

But since you're *not* me, and *hopefully* you don't have access to my Docker Hub account, you will need to switch to a publicly available image for the rest of this example, so run:

```
$ kubectl set image deployment/hello-go kube101-go=\
geerlingguy/kube101:hello-go
```

And confirm the image has been pulled:

```
$ kubectl describe pod -l app=hello-go
...
Events:
  Type      Reason      Age   From      Message
  ----      -
  Normal    Pulling     17s   kubelet   Pulling image "geerlingguy/kube101:hello-go"
  Normal    Pulled      11s   kubelet   Successfully pulled image "geerlingguy/kube101:hello-go"
in 5.649981045s
  Normal    Created     11s   kubelet   Created container kube101-go
  Normal    Started     11s   kubelet   Started container kube101-go
```

Exposing the Hello Go App

The next step to having a usable app is to expose it, to make it visible and usable outside the cluster.

```
$ kubectl expose deployment hello-go --port=80 --target-port=8180 --type=NodePort
```

The `kubectl expose` command sets up a Service, which allows Kubernetes to route requests to one or more Pods matching a set of conditions.

In this case, we are setting up a Service which will route requests to all Pods running in the `hello-go` deployment, and it will receive requests on port 80 and direct them to the container's port 8180—which is the port we chose to use for our Hello Go application.

Finally, we set the type to `NodePort`, and this makes it so the Hello Go App Service will be accessible on a given port on every node in the cluster.

In Minikube's case, the service would be available on the Minikube IP address, which you can get with `minikube ip`.

On a Linode Kubernetes cluster, it would be available on any of the cluster nodes' IP addresses.

You can get the port using:

```
$ kubectl get service hello-go
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	
hello-go	NodePort	10.97.34.201	<none>	
	80:32617/TCP			3m

You can access the service at any server's IP address, using the port listed in the `hello-go` service, e.g. `http://102.52.28.5:32617`.



Hello Go, successfully deployed



Note: If using a Mac running Apple Silicon and running your cluster via Minikube, the easiest way to access a service like this one is to run `minikube service hello-go` (add `--url` to not automatically launch your web browser). This sets up a tunnel into the Minikube environment.

This may change in future releases of Minikube, but as of 2021 this is the only way to reliably access services running in Minikube on modern Macs.

Scaling the Hello Go App

Next let's scale the App. Since it's stateless, meaning there's no persistent data or external database it needs to interact with, it should be easy to scale up Hello Go by increasing the number of 'replicas' in the deployment.

Edit the deployment with:


```
$ kubectl edit deployment hello-go
```

Modify the `replicas` line, and set it to 3. Save the modifications.

Now check the deployment and verify it rolled out two additional Pods:

```
$ kubectl get deployment hello-go
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
hello-go	3/3	3	3	29m

To watch traffic to the pods, you can tail all the logs in *all* the pods concurrently with this handy command:

```
$ kubectl logs -f -l app=hello-go --prefix=true
```

This will let you see which pod and container are serving which request. Head over to a web browser (or two!) and load the IP address and NodePort for the Hello Go service. Refresh the page a few times, and observe the logged requests.

You might note that the requests seem ‘sticky’; one browser keeps hitting the same pod, while a different browser hits a different pod every time. Kubernetes’ Service layer does have some customizability in this regard (how it distributes requests to different Pods), but if you need more customization, you might want to use an Ingress Controller. We’ll talk about those later.

Press Ctrl-C to stop watching the logs.

Updating the Go App

Now it's time to update the app. Marketing wanted a huge change—a change from “Hello” to “Hi” since that sounds more welcoming for the company's users. The developers just checked in a change and pushed up a new image version: `hello-go-v2`.

To deploy that image version to the Kubernetes cluster, you can modify the image directly:

```
$ kubectl set image deployment/hello-go kube101-go=\
geerlingguy/kube101:hello-go-v2
```

And you can watch how the pods are replaced using:

```
$ watch kubectl get pods -l app=hello-go
```

And confirm the version of the container image used with `kubectl describe`, either inspecting the Deployment or the Pods.



Version 2 is so much more welcoming!

Rolling back the Deployment

Let's say Marketing just realized that the word "Hi" means "I will destroy you" in an as-yet-unknown alien language, and they want you to quickly revert to the previous version of the App.

The fastest way to do that is to use `kubectl rollout undo`, and we can run this command to undo the latest version of the `hello-go` Deployment:

```
$ kubectl rollout undo deployment hello-go
```

After doing that, check on the version of the container image of the deployment with `kubectl describe deployment hello-go`, and verify it's been reverted.

Whew! What a day. If you're using a test cluster, be sure to delete it with `minikube delete`, or by deleting the cluster in the Linode Kubernetes control panel.

Chapter 4 - Real-world apps

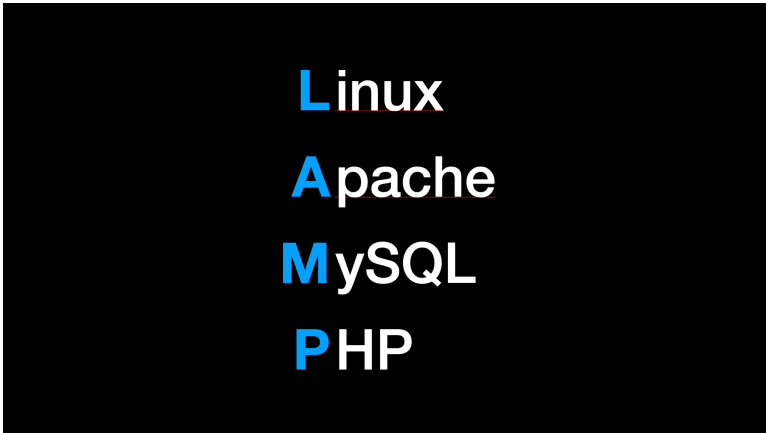


This chapter is still being translated from my rough notes and episode content from the Kubernetes 101 video series. Until this notice is removed, be aware there may be some grammar errors and gaps in certain portions.

We're going to deploy our first real-world application—a Drupal website—into Kubernetes. We'll begin by showing how Drupal is traditionally deployed on a LAMP server, then explore how that legacy architecture translates into containers and Kubernetes using Helm and custom manifests.

Installing Drupal on a Traditional LAMP server

Drupal is one of hundreds of popular PHP applications that was originally designed for the still-popular “LAMP” stack:



Over the years, the stack has evolved, and there are often other technologies added on top, but the basic components are a webserver, a database, and PHP, running on top of Linux.

LAMP Server Setup for drupal



The examples and code used in this chapter can be found in the [traditional-lamp-setup](#) directory in this book's example code repository.

This guide assumes you're running an Ubuntu 20.04 VM. The example linked in the tip above includes a Vagrantfile that you could use to build a local VM inside VirtualBox using Vagrant—but this example should work on any similar Ubuntu server or VM.

Log into the VM (using `vagrant ssh` if you're following along with the book's example), then run the following commands:

```
# Update apt caches.
$ sudo apt update

# Install MySQL (or in this case, MariaDB) and unzip
# packages.
$ sudo apt install -y mariadb-server mariadb-client\
unzip

# Run through the database installation process.
$ sudo mysql_secure_installation

# Create a Drupal database while logged into the My\
SQL cli.
$ sudo mysql -u root

CREATE DATABASE drupal;
GRANT ALL ON drupal.* TO 'drupal'@'localhost' IDENT\
IFIED BY 'mypassword';
FLUSH PRIVILEGES;
\q

# Install PHP.
$ sudo apt install -y php php-{cli,fpm,json,common,\
mysql,zip,gd,intl,mbstring,curl,xml,pear,tidy,soap,\
bcmath,xmldrpc}

# Install Apache with mod_php.
$ sudo apt install -y apache2 libapache2-mod-php

# Configure a couple important PHP settings.
$ sudo nano /etc/php/7.4/apache2/php.ini
```

Find the following lines and change them to these settings and

save the file:

```
memory_limit = 512M
date.timezone = America/Chicago
```

And back to running commands:

```
# Configure a Drupal virtual host for Apache.
$ sudo nano /etc/apache2/sites-available/drupal.conf
```

```
1 <VirtualHost *:80>
2     ServerName example.com
3     ServerAlias www.example.com
4     ServerAdmin webmaster@example.com
5     DocumentRoot /var/www/html/drupal/web
6
7     CustomLog ${APACHE_LOG_DIR}/access.log combined
8     ErrorLog ${APACHE_LOG_DIR}/error.log
9
10    <Directory /var/www/html/drupal/web>
11        Options Indexes FollowSymLinks
12        AllowOverride All
13        Require all granted
14        RewriteEngine on
15        RewriteBase /
16        RewriteCond %{REQUEST_FILENAME} !-f
17        RewriteCond %{REQUEST_FILENAME} !-d
18        RewriteRule ^(.*)$ index.php?q=$1 [L,QS\
19 A]
20    </Directory>
21 </VirtualHost>
```

(We'll finish setting up Apache later, after we have Drupal installed.)

```
# Install Composer (https://getcomposer.org/download\
d/)
$ php -r "copy('https://getcomposer.org/installer',\
'composer-setup.php');"
$ php composer-setup.php
$ sudo mv composer.phar /usr/local/bin/composer

# Prepare the /var/www directory.
$ sudo chown -R www-data:www-data /var/www

# Create a Drupal project codebase with Composer as\
the Apache user.
$ sudo su -l www-data -s /bin/bash
$ composer create-project drupal/recommended-projec\
t /var/www/html/drupal
$ exit

# Finish configuring Apache and restart it to pick \
up the new site.
$ sudo a2enmod rewrite
$ sudo a2ensite drupal.conf
$ sudo a2dissite 000-default.conf
$ sudo systemctl restart apache2
```

Visit the site in your browser: <http://192.168.80.80/>

Install Drupal (DB name: drupal, DB user: drupal, DB pass-
word: mypassword), and celebrate!

Automating the Installation

I should note that I do not set up servers using the technique mentioned in the README above. Instead, I use tools like

Ansible to automate every aspect of server provisioning and setup.

But as with traditional architecture, you should not dive into the deep end of Kubernetes without first understanding the basics. You shouldn't blindly run a 1,000 line Ansible playbook without understanding how file permissions, app configurations, and database connections work.

And you can't just skip over all that stuff when you're starting out in Kubernetes either!

Installing Drupal on Kubernetes using Bitnami's Helm Chart

On that theme, there is a very popular packaging system for Kubernetes called Helm. Helm is a convenient and fairly common way to build templates for applications and deploy and update them in clusters.

But one common pitfall I see is developers new to Kubernetes picking up very good—but very complex—Helm charts, customizing them a little bit, and running their applications this way.

I do want to show you how to deploy Drupal using Helm (which I'll do shortly), but I am intentionally not going to go deep into Helm beyond the basics, because I would rather teach how the underlying components work together before I recommend using complex templates and magic tools to manage resources in Kubernetes!

Install Helm

Install Helm; you can do this using `brew install helm` on a Mac, or follow the [Helm install instructions](#) otherwise.

Install the Drupal Chart

This guide assumes you have a Kubernetes cluster running somewhere (e.g. by running `minikube start`):

```
# Add Bitnami's Helm Chart repository.
$ helm repo add bitnami https://charts.bitnami.com/\
bitnami

# Install Drupal in your default namespace with the\
  release name 'mysite'.
$ helm install mysite bitnami/drupal
```

Now, at this point, the Helm Chart will output a message saying you can monitor the Service it sets up for Drupal until a 'LoadBalancer' IP address is assigned, using the command:

```
$ kubectl get svc --namespace default -w mysite-drupal
```

That's great, but out of the box, Minikube, by virtue of its architecture, doesn't have a Load Balancing layer like Amazon ELBs or Linode's NodeBalancers built into it. Therefore you'll be waiting forever!

Exposing a LoadBalancer in Minikube

So we'll use a little Minikube trick to enable a LoadBalancer, at least temporarily. Open a separate Terminal window and run the command:

```
$ minikube tunnel
```

This command will take a second to start, then ask for your account password to elevate privileges and create a new LoadBalancer on your machine for services like Drupal to use.

And if you run that command and watch the original terminal window, you'll notice it assigns an External IP once the LoadBalancer is running. Nice!

Now you can grab that IP address, paste it in a browser window, and access your Drupal site!

Changing Chart Options

I won't go through every option available with Helm in this chapter, but I will mention that well-maintained Charts like this Drupal chart have configurable options that allow you to control almost any aspect of what is deployed, including things like the Service type.

This Chart's documentation includes a [large selection of configurable Parameters](#) which you can specify in a Helm values file.

So, if you wanted, you could change the release from using a LoadBalancer to a NodePort by setting `service.type` to NodePort.

Cleaning Up

You can remove the Drupal site with:

```
$ helm uninstall mysite
```

R

ight!])Drupal Directly in Kubernetes - Let's Do it [Mostly] Right!

Now let's work on deploying Drupal into Kubernetes *the hard way*.

Some of the magic the Helm chart covered up:

1. Choosing or building and managing container images for Apache, MySQL, and PHP.
2. Generating a Drupal codebase.
3. Connecting all these services together correctly.

We are going to run through how to do everything step by step, and hopefully have a running Drupal installation by the end of this chapter!

I'm going to switch things up and deploy to a real Kubernetes cluster, in this case a cluster running on Linode (since they are still offering a [free \\$100 credit using this link](#)).

Deploying the Drupal Kubernetes Manifests

Please see the [README inside the k8s-manifests](#) for further instructions.

This directory contains two deployment manifests, one for MariaDB, and one for Drupal (which builds a Drupal Deployment running Drupal on top of Apache + PHP).

To apply them to a Kubernetes cluster (e.g. with Minikube: `minikube start`), run the commands:

```
# Create a namespace for the Drupal site.
$ kubectl create namespace drupal

# Create the MySQL (MariaDB) Deployment.
$ kubectl apply -f mariadb.yml

# Create the Drupal (Apache + PHP) Deployment.
$ kubectl apply -f drupal.yml
```

You can then observe the status of the deployments:

```
$ kubectl get deployments -n drupal -w
```

Then press Ctrl+C to stop watching the deployment rollout.

Working in Namespaces

If you want to do a lot of work in a particular namespace (e.g. `drupal`), you can set the current namespace context using:

```
$ kubectl config set-context --current --namespace=\ndrupal
```

And you can confirm which namespace context you're in with:

```
$ kubectl config view | grep namespace:
```

When you're done performing actions (e.g. `kubectl get pods`) only within that namespace, run:

```
$ kubectl config set-context --current --namespace=\n""
```

Accessing the Drupal site

After the Drupal deployment is complete, you can see access it via:

```
# In Minikube, this will open the URL directly.  
$ minikube service -n drupal drupal
```

```
# In other clusters, get the service to get the NodePort.  
$ kubectl get service -n drupal drupal
```

That will give you the NodePort on which the service is exposed, but what about the IP address? Well, for many commands in Kubernetes, they give a brief summary by default, but you can get a lot more information adding `-o wide`:

```
$ kubectl get nodes -o wide
```

Now you can take the IP address of any node, and pair that with the NodePort that maps to TCP port 80 on Drupal's service, and access the site. You should be taken to the Drupal installer UI.

And you can access the Drupal container's Apache logs with:

```
$ kubectl logs -f -n drupal -l app=drupal
```

Install the Drupal site using the UI, and create an Article (under Content > Add content > Article) with an image inside the Body text.

Scaling the Drupal deployment

Go ahead and edit the Drupal deployment:

```
$ kubectl edit deployment -n drupal drupal
```

Set replicas to 3, and save the changes.

Watch the Pods as they are deployed in the scale-up event:

```
$ kubectl get pods -n drupal -w
```

But oh no! It seems like these pods are stuck on 'Init'. Investigate further, with:

```
$ kubectl describe pod -n drupal -l app=drupal
```

You might notice a `Multi-Attach` error. It seems that our fancy Drupal deployment—which is similar to every other Drupal, Wordpress, and similar LAMP-based deployment I see in Kubernetes tutorials, has a major problem: *you can't scale it up!*

There goes one of the major advantages we *thought* we'd get with Kubernetes... or does it? In the next chapter we'll dig deeper into scaling Drupal, accessing Drupal with a Domain using Ingress, SSL, and other real-world app concerns.

Cleaning up

When you're finished testing, one of the best advantages of using Kubernetes namespaces is the easy ability to clean up everything in the namespace.

All you have to do is delete the namespace, and Kubernetes will clean up everything inside:

```
$ kubectl delete namespace drupal
```

After a minute or two, all traces of Drupal should be gone. Sometimes there may be remnants, however, like persistent volumes that are 'retained' for safekeeping (if you're using Linode, for example). Go ahead and delete those with:

```
$ kubectl get pv | grep Released | awk '$1 {print$1\
}' | while read vol; do kubectl delete pv/${vol}; d\
one
```

Confirm the volumes have been deleted in the cloud provider's UI as well; for Linode, at least, deleting the PV via Kubernetes does not trigger a delete of the underlying block storage device!

Chapter 5 - Scaling Drupal in k8s



This chapter is still being translated from my rough notes and episode content from the Kubernetes 101 video series. Until this notice is removed, be aware there may be some grammar errors and gaps in certain portions.

We're going to solve the problem we ran into last chapter: figuring out how to scale Drupal with a shared file system. We'll also explore Horizontal Pod Autoscaling and high-availability database options.

Fixing the scalability issue with Drupal Pods

Last chapter, we tried increasing the replica count beyond 1 Pod for the Drupal Deployment, but got a “Multi-Attach error”, because the default block storage that was connected to the Drupal pod doesn't have the ability to be attached to multiple Pods at the same time.

But if you want to scale up Drupal, Drupal has to be able to read and write files to a shared, persistent filesystem!

The problem is you can only use the ‘ReadWriteOnce’ storage mode for normal cloud storage, like Amazon EBS volumes, or Linode Volumes.

So to make ‘ReadWriteMany’ volumes available, we have to set up a different storage provisioning system for your cluster.

Shared Storage Options

In many ways, the simplest and most reliable option is to use a shared filesystem service from your cloud provider, for example, Amazon EFS.

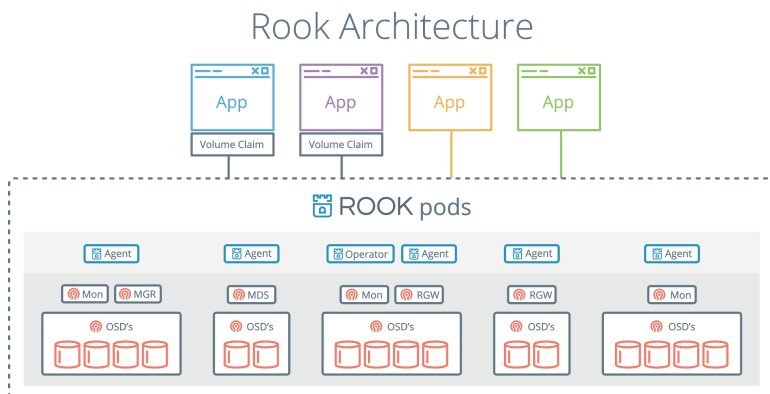
Running a filesystem cluster, whether it’s running with Ceph, Gluster, NFS, or some other storage technology, can be very difficult. The configuration is complex, there are many moving parts, backups can be tricky, and data can easily be lost if you don’t know what you’re doing!

Cloud systems like Amazon EFS make it so you click a button and have shared storage available, so I usually recommend it if you can use it in your own cloud environment.

But what if you’re running a bare metal Kubernetes cluster on your own servers? Or if you’re on a cloud environment that doesn’t have something like EFS—for example, with Linode currently?

Rook and Ceph

Initially, I was going to set up this chapter’s demo using [Rook](#) to manage an in-cluster CephFS clustered filesystem.



But after spending almost an entire workday trying to find a way to easily deploy a Rook/CephFS cluster into Kubernetes—either in Minikube, Linode Kubernetes Engine, or even Amazon’s EKS, I decided the complexity was not worth it.

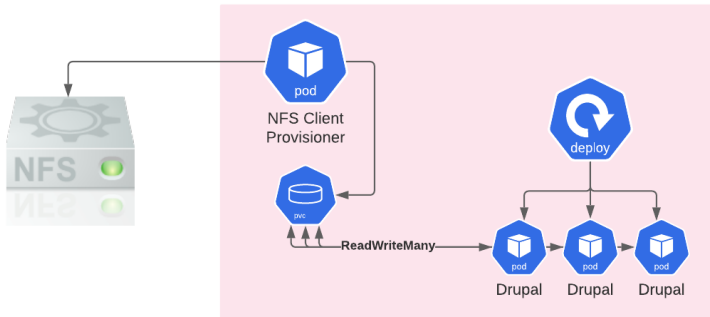
However, I am a strong believer in Rook and CephFS, and they make a potent combo for cloud-provider-agnostic flexible storage options. If you have the time and inclination, it is worth learning how they work, getting them running, and seeing if Rook might be the storage provisioner you should use for your cluster—especially if you’re building on bare metal!

Anyways, to keep things a little simpler, while still achieving the goal of having a shared storage provisioner running in the cluster for scalability, I switched gears.

NFS

I settled on NFS for simplicity’s sake, and in some circumstances, it’s actually not a bad option to just run one simple NFS server in production. That’s how I’ve had my Raspberry Pi cluster set up for years, and it’s easy to maintain, pretty

performant, and easy to restore from backups if the server dies.



Set up an NFS server

Setting up NFS is not too complicated, especially if you just need to be able to set up one shared directory, shared to one private network.

There is an Ansible playbook with its own README guiding you through the process in the [nfs-server directory](#).

Reconfigure the Drupal PersistentVolumeClaim for NFS

Assuming you have an NFS server running, and a Kubernetes cluster running which can connect to the NFS server, you will need to modify Drupal's PersistentVolumeClaim to be able to use the NFS storage.

But before you do *that*, you'll need to configure the Kubernetes cluster *itself* to be able to connect a StorageClass to the NFS server.

Set up NFS client provisioner in K8s

When you're searching around for how to configure NFS in Kubernetes so your Pods can use NFS-based volumes, you might run into a little confusion; originally, there were two provisioners:

- `nfs-server-provisioner`
- `nfs-client-provisioner`

The NFS Server provisioner would run in-cluster NFS instances to allow pods to connect to them. And that's great and handy, but many people would rather manage NFS servers separate from their clusters, and that's where the NFS Client provisioner comes in—it assumes you have an NFS server or storage cluster running elsewhere, and provisions directories inside an NFS share for PVCs in your cluster.

To add to the confusion, though, in 2020 both provisioners were moved and slightly renamed, the server provisioner moving to [nfs-ganesha-server-and-external-provisioner](#), and the client provisioner to [nfs-subdir-external-provisioner](#).

To keep things simple, I'm just going to use a pre-existing Helm Chart to deploy the client provisioner:

```
1 helm repo add ckotzbauer https://ckotzbauer.github.io/helm-charts
2
3 helm install --set nfs.server=192.168.166.68 --set \
4 nfs.path=/home/nfs ckotzbauer/nfs-client-provisioner \
5 r --version 1.0.2 --generate-name
```

For the `nfs.server`, use the Private IPv4 address for your NFS server. For the `nfs.path`, set it to the path of an NFS share on that server.

You can check if the provisioner is running with `kubectl get pods`; make sure an NFS provisioner pod is running.

Deploy Drupal and MySQL (MariaDB)

Now modify the Drupal PVC definition in the `drupal.yml` manifest to look like the following:

```
1 ---
2 kind: PersistentVolumeClaim
3 apiVersion: v1
4 metadata:
5   name: drupal-files-pvc
6   namespace: drupal
7 spec:
8   accessModes:
9     - ReadWriteMany # Was ReadWriteOnce before!
10  resources:
11    requests:
12      storage: 1Gi
13    storageClassName: nfs-client # This is new!
```

Since we're now using `nfs-client` for the storage class, we can set `ReadWriteMany` for the access mode, meaning our

Drupal deployment should be able to scale beyond just one Pod. Yay!

You can deploy Drupal and MariaDB now:

```
1 kubectl create namespace drupal
2 kubectl apply -f mariadb.yml -f drupal.yml
```

And monitor the deployment with `kubectl get deployments -n drupal -w`.

Once Drupal is ready, get the NodePort with `kubectl get service -n drupal` and an IP address of one of the servers with `kubectl get nodes -o wide`. Then access the site and install it.

Save a File and observe it

If you want to manually verify the NFS share is working, you can log into your NFS server and monitor the NFS folder, then log into Drupal and create an Article, uploading an image to the Body field of that article.

After you save the new Article, you should a new image inside the shared Drupal files directory the NFS client provisioner created in the NFS share. Fancy!

Scale Drupal up... and down!

One quick way to test how many requests you can serve on a Drupal site is to use the 'ApacheBench' tool, and we can benchmark how fast Drupal can serve its cached home page with the single replica we have:

```
1 ab -n 500 -c 10 http://45.79.40.239:32119/
```

(Substitute one of your node's IP addresses and the Drupal service NodePort in this command.)

On my test cluster, this process reported the site serving around 80 requests per second. Not bad, but we could get more out of our cluster by scaling up Drupal!

So edit the Drupal deployment, setting replicas: 5:

```
1 kubectl edit deployment -n drupal
```

Then wait for the Deployment to report being up to date with five running instances:

```
1 watch kubectl get deployment -n drupal drupal
```

Because our persistent NFS storage allows multiple Pods to mount the same volume, the Drupal deployment now successfully scales up to five instances! Hit Ctrl-C to stop watching the deployment, and run ApacheBench again:

```
1 ab -n 500 -c 10 http://45.79.40.239:32119/
```

As the caches warm up, the requests per second should go up a bit, since the load can be more evenly distributed to multiple Drupal instances.

If you set the deployment back to replicas: 1, the requests per second will likely go right back down to the original value.

Use Horizontal Pod Autoscaling (HPA)

One of the often-touted features of Kubernetes is automatic scaling. What people probably don't tell you is that, as with all things in life, nice things like autoscaling are not free. You usually have to configure the cluster and your applications to make autoscaling actually work.

Set up metrics-server

Many cluster environments do not include an essential component the [Horizontal Pod Autoscaler \(HPA\)](#) relies on, namely the metrics-server component.

metrics-server monitors resource usage in the cluster for pods and nodes and aggregates the data in Kubernetes' API.

To install it manually, run:

```

1 helm repo add bitnami https://charts.bitnami.com/bitnami
2
3 helm install -f values/metrics-server.yml metrics-server bitnami/metrics-server
4
5 And to make sure it's actually working, run kubectl top node and you should see something like the following:
6
7
8
9 $ kubectl top node
10 NAME                                CPU(cores)   CPU%   MEM\
11 EMORY(bytes)  MEMORY%
12 lke15196-18570-5fd9222ff4ff        112m         5%     9\
13 56Mi                24%
```

14	lke15196-18570-5fd9222ffd28	92m	4%	8\
15	86Mi	23%		
16	lke15196-18570-5fd92230054d	75m	3%	8\
17	39Mi	21%		

Note: See the article [How Can I Deploy the Kubernetes-Metrics Server on LKE?](#) for more details on metrics-server on Linode in particular.

Configuring HPA for Drupal

You can create an autoscaling configuration using `kubect1` just like other resources:

```
1 kubect1 autoscale -n drupal deployment drupal --min\
2 =1 --max=8 --cpu-percent=50
```

As with the Drupal deployment itself, it's best long-term to save the HPA configuration in a YAML manifest, so you aren't manually running commands to configure things in your cluster. But for now, it's easy enough to demonstrate how an HPA works using `kubect1`.

Now you can monitor the status of the HPA you just created with:

```
1 watch kubect1 get hpa -n drupal
```

But if you just sit and watch it, it's not likely anything exciting will happen, because Drupal is running pretty much idle.

You can also monitor, in a separate window, the current CPU and Memory usage of the Drupal pods, with:

```
1 watch kubectl top pods -n drupal -l app=drupal
```

Testing HPA for Drupal

To quickly create a lot of load on a Drupal site, there's no better way than hammering the Module admin page as an authenticated user.

Log into the Drupal site, and view the session cookie that is set by the site. Copy out the cookie name (starts with SESS) and the value (a random string), then paste them into the following `ab` command, which requires Apache Bench to be installed on your computer. Put in the correct IP address and NodePort for your Drupal deployment, and run the command:

```
1 ab -n 1000 -c 3 -C "session_cookie_name=key" http://\  
2 /45.79.40.239:32119/admin/modules
```

It may take a couple minutes for the Kubernetes' HPA to catch up, because it tries to not be too aggressive, but it should eventually kick off eight pods that will help handle the insane amount of load you just put on your cluster.

After the requests are complete, Kubernetes will wait for a preset cooldown period of time before it starts removing Pods, and this is to prevent *thrashing*, which could happen if the scaling were done too quickly.

That's why there's always going to be some lag between traffic and autoscaling. Autoscaling is NOT going to magically solve your scalability problems, and we've also only configured it, at this point, for the application layer—not the database, or any other services Drupal might rely on!

NOTE: The cooldown delay period is only configurable on the cluster level, and is not usually configurable in managed Kubernetes environments. There are alternative pod autoscalers with more configurability, but their setup is outside the scope of this lesson.

Scaling Databases

The last topic of this chapter is scaling databases.

And I have to share my advice that I've learned through the course of building a number of production clusters, some running less than ten applications, others running thousands of applications.

Unless you love dealing with massive complexity and scaling difficulty, I wouldn't recommend trying to configure *traditional* HA database configurations inside a Kubernetes cluster.

What most people do—and I've done often—is rely on a cloud provider's database. For example:

- Amazon Aurora with Amazon EKS
- Google Cloud SQL for Google GKS
- DigitalOcean Managed MySQL for Kubernetes

The complex database scalability concerns are taken care of by the cloud provider... but you *do* end up paying for it!

An alternative is to run single-Pod databases, without high availability, for each application that needs one. Honestly, for most of my own applications, I run a single-server database

with good backups, and it's reliable enough for many use cases.

Running things inside Kubernetes in a similar fashion is just as reliable, but could actually be more easily scalable through Kubernetes own tools, depending on the servers you set up.

But running databases inside Kubernetes—even if you use something like [Vitess](#), Bitnami's [MariaDB Cluster Helm Chart](#), or Presslabs' [MySQL Cluster Operator](#)—is challenging.

The more complex the setup, and the higher the requirements, the more you have to start considering things like:

- Using dedicated nodes for database Pod scheduling
- Ensuring database Pods have the correct taints and tolerations to not end up on the same node
- Configuring specialized storage classes for higher-performance data storage, or using on-instance high-speed storage
- Configuring robust database backups and snapshotting

Unless your needs are lightweight, you might realize the relatively high cost of cloud-managed databases is well worth it when you switch to cloud-native infrastructure!

NOTE: Another option, if you are able to adapt your applications, is to use a more 'cloud-native' database solution, like [CockroachDB](#).

Chapter 6 - DNS, TLS, Cron, Logging



This chapter is still being translated from my rough notes and episode content from the Kubernetes 101 video series. Until this notice is removed, be aware there may be some grammar errors and gaps in certain portions.

We're going to take our scalable Drupal web application and expose it to the Internet using a user-friendly domain name and HTTPS using Ingress, and then configure cron and logging using Kubernetes so it's easy to keep Drupal running it's best!

Setting things up from Episode 5

Everything in this chapter builds on the scalable Drupal configuration we set up in Chapter 5. You will need it running to be able to set up everything in this chapter.

1. Deploy the `nfs-server` from Episode 5.
2. Create a Kubernetes cluster on Linode with at least 3 worker nodes.
3. Deploy `nfs-client-provisioner` into the Kubernetes cluster:

```
1 helm repo add ckotzbauer https://ckotzbauer.github.io/helm-charts
2
3 helm install --set nfs.server=192.168.148.123 --set\
4   nfs.path=/home/nfs ckotzbauer/nfs-client-provision\
5   er --version 1.0.2 --generate-name
```

1. Deploy metrics-server into the Kubernetes cluster:

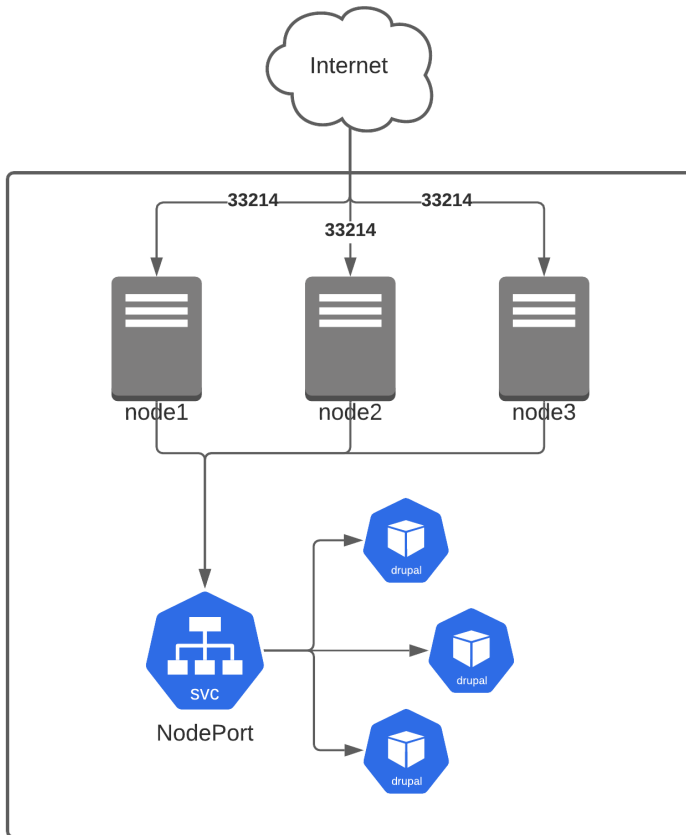
```
1 helm repo add bitnami https://charts.bitnami.com/bitnami
2
3 helm install -f values/metrics-server.yml metrics-server bitnami/metrics-server
```

1. Deploy Drupal and MariaDB into the cluster:

```
1 cd k8s-manifests
2 kubectl create namespace drupal
3 kubectl apply -f mariadb.yml -f drupal.yml
```

DNS and Ingress setup for Drupal

Up to this point, we've been accessing Drupal using a Node-Port, with the IP address of one of the servers and the assigned port, like `http://69.164.207.70:33214/`.

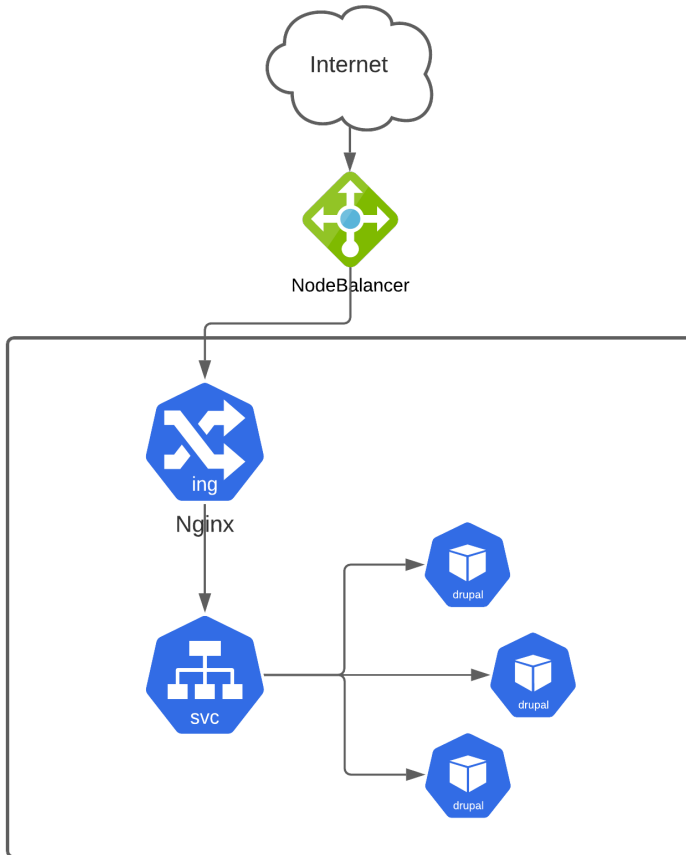


But entering IP addresses and ports isn't very useful for end users.

It would be much better if we could use a friendly DNS name, like `ep6.kube101.jeffgeerling.com`.

There are a few different ways to do this, and the easiest would be to switch Drupal's Service to a LoadBalancer, which would provision a cloud load balancer—on Linode, a NodeBal-

ancer, or an ELB on AWS.



This is easy, but if you plan on deploying many sites and publicly-accessible HTTP apps to your cluster, it can get expensive fast, because each Load balancer will incur an hourly or monthly fee.

So the preferred way to control DNS tying into backend

applications is to use what Kubernetes calls ‘Ingress’.

Instead of a Load Balancer per application, you set up one Load Balancer for an Ingress Controller, which runs software like Nginx, HAProxy, Traefik, or Envoy, and that Ingress Controller proxies web requests to backend applications like Drupal.

Set up an NGINX Ingress Controller

In this case, mostly because it’s the webserver and proxy I’m most familiar with, I’m going to set up Nginx, and there’s a simple Helm chart that sets up Nginx as a cluster Ingress Controller:

```
1 helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx
2
3 helm install ingress-nginx ingress-nginx/ingress-nginx
4
```

Notice I mentioned “a” cluster Ingress Controller, and not “the” cluster Ingress Controller? You can have multiple Ingress Controllers in your cluster, if you’d like!

Get the IP address of the nginx-ingress-controller using the command:

```
1 kubectl get svc ingress-nginx-controller
```

Now go to your DNS provider, and add an A record pointing a domain or subdomain at the “EXTERNAL-IP” address of the Ingress Controller, for example:

```
1 ep6.kube101.jeffgeerling.com --> A 45.79.240.122
```

At this point, if you visit the URL, you'll get a 404 Not Found from Nginx (assuming everything was updated correctly and the DNS change has propagated!).

Set up Ingress for Drupal

To connect the domain name to the Drupal backend, we need to add an Ingress record that tells Nginx to route requests for `ep6.kube101.jeffgeerling.com` to the drupal service on port 80.

And the `k8s-manifests/drupal-ingress.yml` file does just that!

So go ahead and take a look at it, then deploy it into the cluster with:

```
1 kubectl apply -f drupal-ingress.yml
```

After a few seconds, you should be able to access the site (e.g. `http://ep6.kube101.jeffgeerling.com/`), and you should get Drupal.

Much better now!

And since the Ingress is pointed at a Kubernetes Service in front of three or more Drupal Pods, it will make sure all the requests to Drupal are load-balanced for better scalability.

External DNS Integration

There are a few things relating to Ingress I won't cover in this 101 book, and one of them that may be pertinent, depending

on your DNS provider, is integration between Kubernetes and External DNS providers for easy DNS configuration for Services.

The [External DNS](#) integration integrates Kubernetes with a large number of external DNS providers, including AWS Route 53, Google Cloud DNS, Linode DNS, CloudFlare, and more. This makes it very easy to manage domain integration into your Kubernetes apps, but it assumes you have your DNS records configured in one of the supported services.

In the case of my example today, I used Name.com, which does not have an API that integrates with Kubernetes, so I wasn't able to demonstrate External DNS.

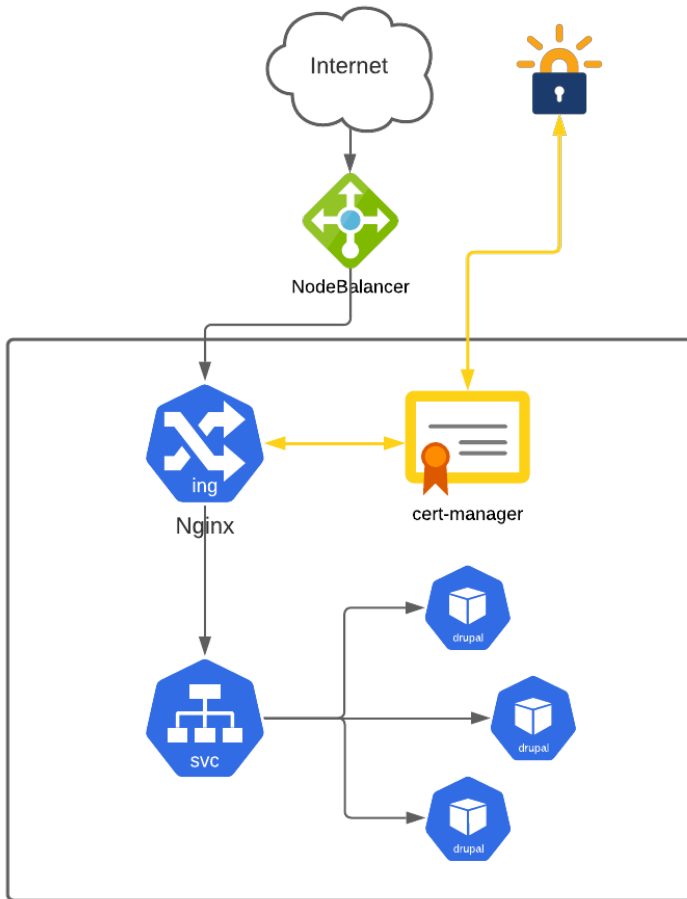
Instead, I manually grabbed the IP address of the NodeBalancer Linode provisioned for me, and added an A record in Name.com's frontend interface.

At a certain scale, managing DNS manually is very inefficient!

Set up TLS with cert-manager and Let's Encrypt

Speaking of things hard to manage manually, most web applications should be set up so they are secured using TLS encryption, meaning you access them over encrypted HTTPS and not via plaintext HTTP.

And the easiest way—and the way I have always set up my own Kubernetes clusters—to configure certificates automatically for your Ingress resources is to use [cert-manager](#).



Most of the time I use it with Let's Encrypt to get valid public certificates, but you can also use cert-manager with many other setups, including private CAs for internal applications, depending on your needs.

Installing cert-manager is easy enough.

First, create a namespace:

```
1 kubectl create namespace cert-manager
```

Then install the CRDs (Custom Resource Definitions) for cert-manager resources:

```
1 kubectl apply -f https://github.com/jetstack/cert-m\
2 anager/releases/download/v1.1.0/cert-manager.crd\
3 s.yml
```

Then add the jetstack Helm repository and install cert-manager from Jetstack's chart, into the cert-manager namespace:

```
1 helm repo add jetstack https://charts.jetstack.io
2 helm install cert-manager jetstack/cert-manager -n \
3 cert-manager --version v1.1.0
```

Check that it's running:

```
1 kubectl get pods -n cert-manager
```

Once it's running, you'll need to set up a ClusterIssuer inside the cluster that will tie cert-manager to a CA service—in our case, LetsEncrypt. Look at the configuration of the ClusterIssuer inside the [k8s-manifests/cluster-issuer.yml](#) file. There are many other options you could choose, including using Let's Encrypt's staging environment for a non-production cluster.

Create the letsencrypt-prod ClusterIssuer:

```
1 kubectl apply -f cluster-issuer.yml
```

Then deploy an updated Drupal Ingress manifest, which is the same as the ingress resource added earlier, but with TLS settings using the letsencrypt-prod ClusterIssuer we just created:

```
1 kubectl apply -f drupal-ingress-tls.yml
```

It will take cert-manager a minute or two to perform its HTTP01 challenge in the background. You can monitor progress by viewing the cert-manager pod logs:

```
1 kubectl logs -f -n cert-manager -l app=cert-manager
```

Now you should be able to refresh the Drupal site in your browser and see a secure connection, with a valid LetsEncrypt certificate! (In my case, issued by DST Root CA X3 -> R3).

Keeping Drupal Happy with a CronJob

Drupal is configured out of the box to use a special module, called “Automated Cron”, that runs its internal maintenance jobs on a regular cycle.

However, there is a downside to the way this works—it relies on external web requests to operate.

Drupal waits until an a certain amount of time passes—by default, 3 hours—and it waits for a web request to be made,

and then after it serves that request, it runs a cron job through that same request cycle.

One problem is that you don't have control over external web requests, so if you want to run Drupal's cron more frequently, say every 5 minutes, you can't rely on having steady traffic at all hours of the day.

Another problem is this ties cron runs to a web process that is tuned for external access performance, and might not even have the resources to allow Drupal to complete all its maintenance tasks efficiently.

Kubernetes has a solution in the form of a CronJob resource. Kubernetes Jobs are one-off Pod definitions which run a Pod to completion, keeping the log output for later inspection.

Kubernetes CronJobs are an efficient way of managing repetitive Jobs, like running Drupal's cron on whatever schedule you want.

I wrote an entire blog post on this topic in 2018: [Running Drupal Cron Jobs in Kubernetes](#), but I'll dive into how to do everything on our example Drupal site in this chapter.

Make sure the URL to the Drupal site's cron URL is correct (you can find this under Configuration > System > Cron) inside `k8s-manifests/drupal-cronjob.yml`, then deploy the CronJob into the cluster:

```
1 kubectl apply -f drupal-cronjob.yml
```

Then verify the CronJob exists in the Drupal namespace with:


```
1 kubectl get cronjob -n drupal
```

After you see a Job run successfully (you can monitor cron-triggered Jobs with `kubectl get job -n drupal`), check in Drupal's admin UI to make sure Cron runs are being detected.

There are a few things to be aware of when using Kubernetes CronJobs:

1. The Successful and Failed Job History Limits allow you to control how many Jobs are kept before Kubernetes prunes old Jobs, for both successful and failed runs. It may be useful to increase these limits, especially if you need to debug CronJob problems.
2. For most Jobs, it is a good idea to disable concurrency using the 'Forbid' concurrency policy. But there are use cases where you might want to change that to 'Allow' or 'Replace'.
3. When you start hitting frequent Job failures, you could run into some weird issues... at least I have in the past. Sometimes I've had to re-create a CronJob to get it to start running its schedule again after too many Job failures in a row.
4. Beware of having too many Jobs on your cluster. If you run hundreds of sites and have a CronJob for each, running every minute, this can overload your Kubernetes control plane! See my [50,000 Kubernetes Jobs for 50K subscribers](#) video for more on that!

Once you can verify the Kubernetes CronJob is running Drupal's cron successfully, it would be a good idea to disable the now-redundant Automated Cron module in Drupal!

Monitoring Drupal's Logs

Now, I had originally planned on showing a nice, simple logging setup for Drupal that I could demo in a few minutes in this chapter.

But logging is not rainbows and sunshine. It wasn't easy to do centralized logging with multiple servers *before* Kubernetes was a thing, and it's not easy with multiple containers inside Kubernetes either.

I have to punt on this topic because dealing with logs—even with one simple site—is not a quick and easy problem to solve.

Especially if you want to handle logs in a scalable and secure manner!

That said, here are a few potential solutions that I have either used in the past or think would be good for many use cases:

Using an External SaaS Log Aggregator

If you have the budget, this is going to be the easiest option. Services like [Sumo Logic](#), [DataDog](#), and [Elastic](#) provide cloud log aggregation, monitoring, and search dashboards.

They are relatively easy to set up, they have pre-made ingest plugins available for most applications (e.g. [DataDog Logs HTTP for Drupal](#)) and for Kubernetes itself, and they can usually scale to thousands of applications without a hassle.

But they do cost a *lot* (usually). That's the number one downside, but for many companies, the ease of integration and flexibility make it worthwhile.

Running your own ELK Stack

If you want to save on the costs of a hosted service, running your own ELK stack is a perfectly reasonable option. Especially for smaller clusters (where you aren't dealing in many gigabytes of logs per day), it is not impossible to manage an ELK stack with a small team.

However, maintenance is not free, and the Elastic stack (Elasticsearch, Logstash, and Kibana) require an investment in time to set up the stack, and maintain it.

And it's not lightweight, either—in my experience I've had to allocate a lot of resources to get an Elasticsearch cluster to run well beyond one or two medium-traffic sites dumping all their log data, in addition to sometimes-noisy Kubernetes services.

Relying on a Service Mesh

So-called 'Service Meshes' like Istio or Google's Anthos sometimes have their own logging integrations built-in.

I don't typically recommend a Service Mesh layer on top of a Kubernetes cluster, because I kind of see it like spraying a firehose of 'all the things' on top of a cluster, and in reality, you don't necessarily need 'all the things' that a Service Mesh provides (they sure add to your cluster operation costs though!).

It's often easier to inject specific sidecar containers—that is, containers that run alongside your application containers in the same Pod—to do specific purposes like extract log files or stream specific data to or from your application.

Using your cloud provider's solution

Many cloud providers have integrated logging in their platform. You're already paying for it (most likely), so why not use it?

Some of the solutions are not as robust as the alternatives, but the primary purpose for central logging is to be able to audit your applications and identify problems, and the basics are usually covered well.

This is one argument in favor of relying on hosting partners who are invested in the software you are using—they can offer deeper integration and insights into the apps you host with them!

Chapter 7 - Hello, Operator!



This chapter is still being translated from my rough notes and episode content from the Kubernetes 101 video series. Until this notice is removed, be aware there may be some grammar errors and gaps in certain portions.

In this chapter, we'll explore the concept of a Kubernetes Operator, and how they allow you to manage applications running in Kubernetes clusters more efficiently. We'll also talk about how you can build one on your own!

What are Operators?

As we've seen in the past few chapters, real-world applications like Drupal require a good deal of effort to deploy and maintain in a Kubernetes cluster.

There are many application concerns we haven't covered in depth, like running updates on the database schema, or performing routine operations outside of cron, like queue management or data backups.

Sometimes you can plug other tools into your Kubernetes cluster to solve some of these puzzles, but *what if there was a way to tell Kubernetes how to manage everything for you?*

Well there is, and that mechanism is an **Operator**.

We're going to throw around a lot of Kubernetes jargon in the next few paragraphs! Don't worry if the connections between all these things is crystal clear yet. A few years into my own Kubernetes journey, I'm constantly referencing documentation to make sure I'm using the terms correctly!

The Concept

Much like a human operator, a Kubernetes Operator is tasked with the management of a given application (or a whole bunch of instances of that application) inside a cluster.



The operator knows how to deploy the app, how to update it, and how to fix it if there are issues.

Digging down a couple layers into how Kubernetes works, Kubernetes uses **controllers** running in the cluster to watch the state of the cluster (watched resources), and make changes until the cluster is in the state it desires.

An analogy from the Kubernetes documentation makes a lot of sense here: a controller is like a thermostat. You set a temperature on the thermostat, then thermostat works to

inform the heating system (the Kubernetes cluster) gets to the final state where that temperature is reached.

Operators are a standard way of building controllers for **Custom Resources**, extensions to the Kubernetes API beyond the standard resources like Pods, Deployments, and Services.

In the context of this series, we would probably consider building a **Custom Resource Definition** (CRD) named “HelloGo”, or “Drupal”, and then we could build an Operator to Custom Resources that conform to that CRD.

The Execution

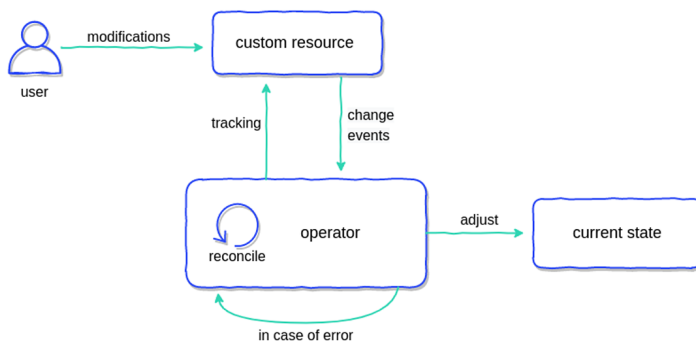
So taking Drupal as an example, an Operator would have a Custom Resource Definition.

This CRD defines the specification for individual Custom Resources (CRs) of Drupal, and at a minimum, you’d probably want to include data fields like:

- `databaseEngine`
- `filePvcType`

And anything else you might want to customize or tweak in an instance of Drupal running inside your Kubernetes cluster.

Then you could create one or more Custom Resources of `kind: Drupal`, and then the Operator would be called upon to create them if they don’t exist, or manage them when any part of the instance changes.



As an example, if you set `databaseEngine: aws_aurora`, you could write some logic in your Operator that makes sure a database exists for the Drupal application in Amazon Aurora, then connects the Drupal site to that database.

Alternatively, if you set `databaseEngine: local_mariadb`, you could have the Operator ensure a MariaDB database exists inside the Kubernetes cluster, and then connect the Drupal site to *that* database.

What if you set up your Drupal site with `aws_aurora`, then switched it to `local_mariadb`? Well, if you manage the site via an Operator, you could even build logic into that Operator to manage full database migrations!

The real power of managing applications like Drupal via Operator—especially if you manage more than one instance—is the ability to maintain the application deployment and maintenance logic in the Operator, and control instances with simple declarative YAML, just like other Kubernetes primitives.

For example, compare the following hypothetical Drupal Custom Resource to the longer examples used in chapters 4,

5, and 6:

```
1 apiVersion: "drupal.example.com/v1"
2 kind: Drupal
3 metadata:
4   name: my-drupal-site
5 spec:
6   databaseEngine: local_mariadb
7   version: 1.2.3
8   filePvcType: efs
```

A lot simpler to manage! Now imagine you're building a platform that's running 300 different Drupal sites. Would you rather build the automation into one Operator that can manage 300 instances, or have to build automation that runs outside of Kubernetes to do the same thing?

Why not use an Operator?

There are some valid reasons for sticking with primitives, especially if you don't have more complex needs, like running many instances of an application in one or more clusters.

In many cases, you might only need to run one instance of an application or microservice, and it's not something that would benefit from an extra layer of automation.

Especially early on, when something is in development, it is easier to iterate using Kubernetes resources directly, rather than to build and maintain an Operator, which then manages those resources for you.

And in some cases, Operators are just one extra layer of automation that you might not want to maintain.

Popular Kubernetes Operators

Even if you don't build your own operator, though, there's a good chance you'll end up using one or more in your Kubernetes clusters.

Almost every cluster I've ever built needed Prometheus for monitoring, and the standard way to install Prometheus and Alertmanager in a Kubernetes cluster is to use the [Prometheus Operator](#).

There are many other operators available, like the [Argo CD Operator](https://argocd-operator.readthedocs.io/en/latest/) for deploying one of the most popular Continuous Deployment tools in Kubernetes, or the [OpenEBS Operator](#).

Operators aren't really centrally visible, like Docker images on [Docker Hub](#), or Helm charts on [Artifact Hub](#).

But there is one central location that's aggregating a large number of operators, and that's [OperatorHub.io](#).

Currently, OperatorHub lists almost 200 operators, but I know there are hundreds more that aren't listed there. Usually I find them through a direct Google or GitHub search.

Some are better maintained than others, though. The only way to get a good grasp on whether an existing operator is right for you is to install it in a test cluster.

Build your own Operator

So what if there isn't a good operator for the software you're running in your cluster? Or what if your application needs a custom operator?

Early on, building an Operator required fairly deep knowledge of the Go programming language and Kubernetes APIs, but thanks to a lot of work by the community, there are now a variety of ways you can build operators—even if you don’t know Go at all!

The two main ways I’ve seen used for building operators are [Kubebuilder](#) and the [Operator SDK](#).

The main difference between them used to be that Kubebuilder only helped build Go-based operators, requiring knowledge of Go, whereas Operator SDK worked with Go, Ansible, or Helm-based operators.

But in the past year, the Operator SDK upstreamed some of its customizations into Kubebuilder, so Go-based operators built by Kubebuilder and Operator SDK are practically identical.

Operator SDK is still the only easy way to build non-Go operators with Ansible or Helm, though.

There are some other projects that help with building operators, or operator-like tooling, like [KUDO](#), but I won’t cover them in this chapter.

There are some tradeoffs if you’re not using Go to build your operator with Operator SDK, though. The Operator SDK shows a graph of ‘[operator capability levels](#)’, and this shows different types of operators are better at different levels of automation.

Helm-based operators are great for basic app install and upgrades.

Ansible-based operators can also perform more app management, including metrics integrations, config management, and external integrations.

Go-based operators can do everything, with the highest amount of granularity and available performance tuning.

But Go-based operators are the most difficult to set up and maintain if you don't already know the Go programming language. And Ansible might be more complex than what you require too, if you just want to be able to install and upgrade many instances of your application via Helm.

Building an Operator with Operator SDK

Since I'm most familiar with Ansible, though, I'm going to demonstrate building a custom operator in Ansible, using the guide from the Operator SDK.

Installing Operator SDK

First, you need to make sure Operator SDK is installed on your system. As with everything else on my Mac, I use homebrew to install it:

```
1 brew install operator-sdk
```

You can also download the release binary from GitHub if you want.

Building an Ansible Operator

The first step to building an operator is to create a project directory, and initialize an Operator SDK project inside:

```

1 mkdir memcached-operator
2 cd memcached-operator
3 operator-sdk init --plugins=ansible --domain=exampl\
4 e.com

```

Next you need to create an API for Kubernetes with a role for the API to run in the cluster:

```

1 operator-sdk create api --group cache --version v1 \
2 --kind Memcached --generate-role

```

Next you need to create an Ansible task to actually manage a Memcached instance whenever a new Memcached Custom Resource (CR) is added.

So open `roles/memcached/tasks/main.yml`—this is the task file that is run when the Ansible Operator identifies a new or changed resource. Add the following inside:

```

1 ---
2 - name: Manage a memcached deployment with {{ size }}
3   replicas.
4   community.kubernetes.k8s:
5     definition:
6       kind: Deployment
7       apiVersion: apps/v1
8       metadata:
9         name: '{{ ansible_operator_meta.name }}-mem\
10  cached'
11         namespace: '{{ ansible_operator_meta.namesp\
12  ace }}'
13       spec:
14         replicas: "{{ size }}"

```

```

15         selector:
16             matchLabels:
17                 app: memcached
18         template:
19             metadata:
20                 labels:
21                     app: memcached
22             spec:
23                 containers:
24                     - name: memcached
25                       command:
26                         - memcached
27                         - -m=64
28                         - -o
29                         - modern
30                         - -v
31                       image: "docker.io/memcached:1.4.36-al\
32 pine"
33                 ports:
34                     - containerPort: 11211

```

Next, modify the spec inside `config/samples/cache_v1_-memcached.yaml` so it provides a custom size value for the Ansible operator—and delete the existing `foo: bar` entry:

```

1 spec:
2   size: 3

```

To be complete, you should also set an Ansible role default for the size variable in case the user didn't set one, but I'm skipping that step here.

The operator runs inside a container image, so before you can start using the operator, you have to build the docker image

and push it somewhere your Kubernetes cluster can pull it from:

```
1 make docker-build docker-push IMG=ttl.sh/example-me\  
2 mcached:1h
```

[ttl.sh](#) allows you to push ephemeral container images to a public repository for testing purposes. Note that the image will be automatically removed after an hour. For real-world usage, you should push the operator image to a persistent registry!

Currently there's [a bug](#) in the Operator SDK that prevents the generated Makefile from working on macOS Big Sur. The fix is to comment out the `SHELL` line in the Makefile before building.

Running an Ansible Operator

First, make sure you have a Kubernetes cluster running somewhere to which you have access as a `cluster-admin`. The easiest thing for testing is to use Kind or Minikube. In my case, I'll start up a Minikube cluster:

```
1 minikube start
```

Then, install the Operator's CRD into the cluster:

```
1 make install
```

And finally, deploy the Operator into the cluster, so it can start watching for custom resources:

```
1 make deploy IMG=ttl.sh/example-memcached:1h
```

After it's deployed, you can check on the new operator pod running in the cluster, which should be in the `memcached-operator-system` namespace:

```
1 kubectl get pods -n memcached-operator-system
```

Once it's Running, you can start deploying instances of your application and the Operator will start managing them!

Create a CR, and let the Operator Operate!

Now deploy the Memcached Custom Resource sample modified earlier:

```
1 kubectl apply -f config/samples/cache_v1_memcached.\
2 yaml
```

And watch the Operator perform a 'reconciliation', making sure the CR is in the proper state:

```
1 kubectl logs deployment.apps/memcached-operator-con\
2 troller-manager -n memcached-operator-system -c man\
3 ager
```

After you're done, you can clean everything up by deleting the CR, then un-deploying the operator and CRDs:


```
1 kubectl delete -f config/samples/cache_v1_memcached\  
2 .yaml  
3 make undeploy
```

That example was fairly simple, but the idea is you can add Ansible tasks to create whatever resources are necessary, manage connections between them, run necessary installation or database update tasks, and keep things running correctly whenever a change occurs in the cluster.

Any language, including Python or Rust!

Since Operators are basically Kubernetes Controllers that interact with the Kubernetes API directly, you can write them in any language.

Besides Go, Ansible, and Helm, there are also guides and frameworks to assist with building a [controller in Rust](#), or [an operator in Python using Kopf](#).

Conclusion

There are hundreds of Kubernetes Operators that you can look at for inspiration. I even maintain a [Drupal Operator](#) that could be used to manage multiple Drupal instances in a cluster!

Operators are not the solution for every application, but they do help in many ways, especially if you have a lot of application lifecycle management to automate in a cluster, or if you run many instances of an application.

Chapter 8 - Kube, Meet Pi



This chapter is still being translated from my rough notes and episode content from the Kubernetes 101 video series. Until this notice is removed, be aware there may be some grammar errors and gaps in certain portions.

Now we're going to get Kubernetes running on a bare metal cluster of Raspberry Pis! Two of my favorite things, together in one fun chapter!

Heavy Metal Kubernetes

For the most part, the production Kubernetes clusters I've worked on were hosted in the cloud, where cloud vendors operate the servers, storage, load balancers, and in most cases, even the Kubernetes control plane itself, that schedules Pods onto Nodes and manages the cluster.

This is very convenient for many applications, but there are situations where Kubernetes is a good fit for your architecture, but you need more control over the cluster, more performance, or more security (e.g. an 'air-gapped' cluster).

And in these cases, you'll want to install Kubernetes on 'bare metal'.



Bare metal is a fancy way of saying ‘a particular physical server dedicated to one task.’ And in our case, that task is operating as part of a Kubernetes cluster.

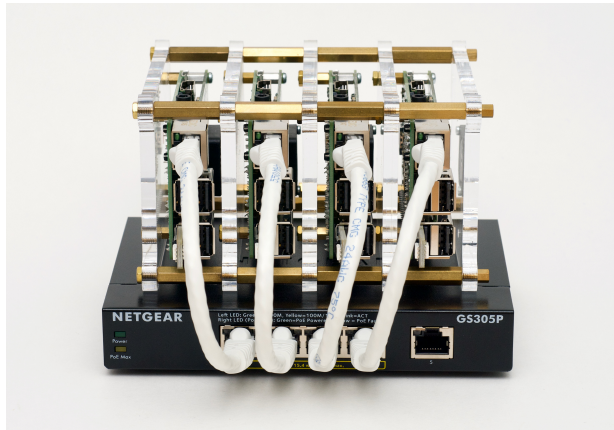
There are some things that can be more difficult to manage in a bare-metal environment, including:

- Cluster networking
- High-availability for the Kubernetes control plane
- Storage backends
- External load balancing and network ingress
- Certificate management (especially on non-public networks)

Start with Training Wheels

And in my experience, the best way to learn how to do something well is to start with the basics: in this case, find the simplest, most inexpensive way to build a local bare metal Kubernetes cluster.

My weapon of choice? The humble Raspberry Pi!



Now, immediately I know you're thinking a Raspberry Pi is woefully underpowered with its mobile CPU, its maximum 8 GB of RAM, and its embarrassingly slow microSD storage.

But what the Pi lacks in niceties it makes up for in size, energy consumption, and teaching ability.

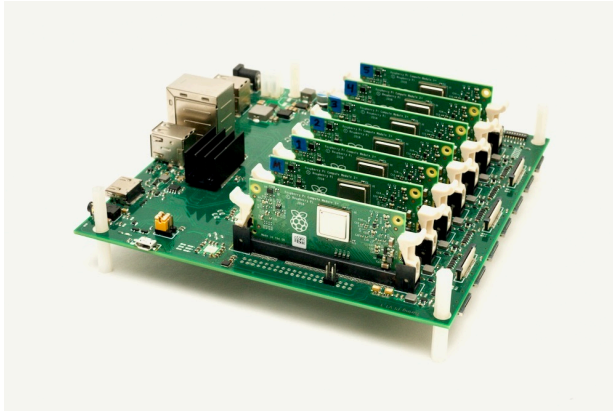
The Raspberry Pi makes for Compact Clusters

Skeptics of Pi-based clusters suggest networking together a pile of old laptops or cheap corporate thin-client desktops to get more performance, more memory, and faster storage for the same price.



That's definitely possible, but once you have three or more nodes, you start to yearn for the efficiency of a single board computer, especially combined with Power over Ethernet. Not having to take up a couple square feet of real estate on your desk or in a cabinet gives you the freedom to grab your entire cluster with one hand.

Not being tied to four separate power adapters plugged into four separate electrical outlets—plus an extra one for a network switch—means you can quickly move your cluster around as-needed. That's especially true of the latest trend in SBC clustering: cluster boards like the Turing Pi!



In my experience, the simpler and smaller an educational tool is, the more likely it comes out of my closet for some quick testing or learning.

And this is to say nothing of the heat and noise that comes out of a typical PC with a traditional X86 processor!

The Raspberry Pi sips energy, and keeps its cool

And that leads me to another advantage of learning on a Pi. Unless you're running heavyweight apps full-tilt 24x7, the Raspberry Pi is pretty easy to keep cool with either heat sinks or a gentle slow fan blowing over the cluster.

Indeed, most of the cluster builds I've seen end up using either passive cooling with an open case, or they have one large-ish fan blowing over all the Pis quietly.

And the entire cluster—including a PoE switch—can operate on less than 50W of power.

At idle, it sips less than 10W of power.

When you're running a Kubernetes cluster, you're not going to take advantage of sleep or hibernation, so running on old laptops where those modes save a ton of power won't help you.



While raw power usage is lower on a cluster of Pis, performance per watt is likely going to be higher with even a few-generations-old Intel or AMD CPU. But unless you are running intense CPU-hungry applications, or recompiling software on the cluster constantly, that's not necessarily going to result in better power efficiency (over a longer term).

The Raspberry Pi teaches lessons about scalability

Many developers today are running the latest processors, with dozens of gigabytes of RAM and the fastest NVMe storage. And often you get a budget to buy beefy cloud instances with similar specs and multi-gigabit networking.

But there's an old saying that reminds me of the importance of starting small:

“Wax on. Wax off.”



Just like [Mr. Miyagi in The Karate Kid](#), I like to teach the lesson that you build a foundation for advanced usage of complex tools like Kubernetes by starting small, focusing on a first step, perfecting your knowledge of that step, then moving on.

So start learning little things, with great constraints. In Kubernetes, you quickly learn lessons that will pay back in spades later:

- Always add resource limits
- Monitor CPU and memory usage
- Optimize your applications so they don't end up thrashing—especially when you end up in situations where your storage, networking, or compute resources start going haywire.

And you might be skeptical: “I have a budget of \$50,000/month for multiple 32 vCPU nodes with 64 GB of RAM and fast SSD-backed storage!” I hear you say.

But if you can build an efficient cluster using nothing but small Raspberry Pis, I guarantee your end users will appreciate the incredible speed and scalability you can get when you throw in a faster processor, faster networking, and faster

storage, plus the resilience of configuration that deals with outages and slowdowns with aplomb.

ARM is not all sunshine and roses

That's not to say everything is peachy when building on the Raspberry Pi. It uses a 64-bit ARM processor, and the default Pi OS is actually still *32-bit*, meaning you're starting out a bit crippled, at least early in the 2020s, where Intel and AMD processors still dominate the server landscape.

Through my four years building ARM-based Kubernetes clusters, I have grown to hate—but immediately recognize—the dreaded:

```
Exec format error.
```

This error would show up when the cluster would pull a container image, try running it, and realize the image doesn't support the ARM architecture of the Pi.

Luckily there are four forces that have come together to make this situation much more rare lately:

- Raspberry Pi OS is now available as a 64-bit beta, and Canonical publishes a supported 64-bit build for Raspberry Pi as well.
- Amazon's less-expensive Graviton processors have encouraged more developers to build ARM-compatible images.
- Apple's M1 processor has proven the ARM architecture has great performance and energy efficiency potential.

- Tools like Docker's Buildx make building multi-arch images (that work on ARM64, AMD64, and even PowerPC or other more exotic architectures) much easier.

So this problem has become much less prevalent, though there are still many popular and widely-used images that stubbornly refuse to support multiple architectures (*cough MySQL cough*).

When you run into these instances, you can either find an alternative image or build one on your own; I've done both of those things in support of my Raspberry Pi Dramble cluster.

Installing a Kubernetes Distribution

So now that we've chosen to build a cluster on Raspberry Pis, the next question is *what flavor of Kubernetes?*

In case you weren't aware, there are actually a number of different 'distributions' that maintain compatibility with the Kubernetes API and are managed in a similar way, including:

- Kubernetes (K8s)
- [MicroK8s](#) (maintained by Canonical, the makers of Ubuntu)
- [K3s](#) (maintained by Rancher)
- [K0s](#) (maintained by Marantis)
- [OpenShift](#) (maintained by Red Hat)

And these are not all, heck *Docker* even ships their own Kubernetes-in-a-box with every download of Docker Desktop.

But not all of these distributions are suitable for running on the Raspberry Pi. For example, OpenShift requires at least 16 GB of RAM on master nodes, and from my own experience it really wants 32 or more GB per controller.

Other distributions are focused on the ‘edge’ use case and more minimalist requirements. For example, I ran K3s on a Turing Pi cluster earlier this year, and those nodes only had 1 GB of RAM and 100 megabit networking.

Kubernetes itself is not lightweight, but it runs surprisingly well if you have at least 2 GB of RAM on your Raspberry Pis. It’s a little slower than MicroK8s and K3s, but it is nice to learn how to run the ‘full’ Kubernetes on a homelab built with Pis, so that’s what we’re going to do here.

kubeadm

Now, we want to install Kubernetes—but how do we do it?

There are actually a number of ways to do that too, including using pre-built automation like [Kubespray](#) or [Kops](#), two officially-sanctioned installers.

But the most direct way to install Kubernetes is using [kubeadm](#), which is described as:

a tool built to provide best-practice “fast paths” for creating Kubernetes clusters. It performs the actions necessary to get a minimum viable, secure cluster up and running in a user friendly way.

There’s also the option of building a Kubernetes cluster *the hard way*, but that is about a thousand times deeper than we can go in a 101-level series!

Anyways, I chose to build some light Ansible automation around `kubeadm` to set a Kubernetes controller on one Raspberry Pi, then configure the other three Raspberry Pis in my cluster as nodes that are joined to the controller.

Setting up the Raspberry Pi Dramble

And if you're building a bare metal cluster, you'll need to put everything together yourself. Instead of going into a cloud dashboard and clicking a plus button a few times, then 'Go', you have to get your hands dirty and plug things together!

So first, let's take a short pause and listen to the [ASMR version of the Raspberry Pi Dramble cluster assembly](#).

Okay, now that that's out of the way, here's a [detailed build video](#) where I show how exactly I put together the Raspberry Pi Dramble, my four-node Kubernetes cluster that's set up with `kubeadm` through an Ansible playbook, with NFS for shared storage, and Traefik for ingress across all four nodes.

If you want to build a cluster just like this, I've documented everything in excruciating detail on the website [pidramble.com](#), and the great thing is it's easy to add as many nodes as you want!

I even have some [Sourcekit PiTray minis](#) on hand now, so I could add in nodes using the Compute Module in the same footprint as my normal Pi 4 model B's!

A lot of people ask me why I chose to build a cluster with four nodes. Well, back when I had the cluster running on Raspberry Pi 2's, I actually had a 6-node cluster. But back

then I had a cheap 8-port gigabit switch. With my new cluster, I used power over ethernet, which means the switch is a lot more expensive! My little four port PoE switch cost more than \$50, so I decided to trim the cluster size to 4 nodes instead of doubling my budget for the switch and Pi PoE HATs.

Going Further

There are a few things I'm not covering in this chapter that you'll probably want to look into more deeply.

First of all, the setup I use in my Drabble cluster uses an Ingress controller running on every node to sort out requests to the Drupal site.

If you're going to run more applications and you want true load balancing at a higher level, like what you'd get with a cloud service, you should look into [MetalLB](#).

There's a good article on setting it up on Raspberry Pi on Opensource.com: [Install a Kubernetes load balancer on your Raspberry Pi homelab with MetalLB](#).

Second, the current configuration doesn't include any monitoring system, even though I set up pretty thorough monitoring—including Pi CPU temperature monitoring—using Prometheus and Grafana. To see how I did that, check out my [Raspberry Pi Cluster Episode 4](#), starting around the 5 minute mark.

Finally, I can't wait for the [Turing Pi 2](#) to be released. In my [review of the original Turing Pi](#), I noted that it is much easier to set up and manage, hardware-wise, than a cluster of Pi 4 model B computers—but that the Compute Module 3+ is a far cry from the Pi 4 generation.

The Turing Pi 2 will support the usually-twice-as-fast Compute Module 4, which means you could have a single board with four nodes and one Ethernet connection for the entire setup, saving even more cabling and power hassle.

Other Guides

It seems like building a Raspberry Pi Kubernetes cluster is a hot topic, since you'll find hundreds of guides to do it with any Kubernetes distribution if you search it on the web.

The Raspberry Pi Dramble's heritage dates back to 2014, when it was originally a discrete cluster of separate nodes running different applications, but it has evolved as cloud computing itself has evolved, and it'll be interesting to see how I can make it better as the clustering world moves on to newer and better things.

Chapter 9 - Secrets and Configuration



This chapter is still being translated from my rough notes and episode content from the Kubernetes 101 video series. Until this notice is removed, be aware there may be some grammar errors and gaps in certain portions.

Please stand by as this chapter is translated from my notes. Sorry for the absence of any content so far!

Chapter 10 - Monitoring Kubernetes

In the entire Kubernetes 101 series, I've avoided cluster visualization tools and abstractions, mostly because I find learning the basics, like the command line interface and basic architecture, is better to do *before* you start throwing in abstraction layers.

Once you've learned the basic architecture, the visualizations make more sense.

It's useful to have abstracted visualizations of the cluster and resource consumption, because we, the operators of these clusters, are humans, and our brains often need visual cues to help us identify anomalies or monitor the overall state of our applications.

In this chapter, I'll cover three tools I've found essential to monitoring production Kubernetes clusters.

Lens is billed as an 'IDE' for Kubernetes development, but I think of it as a handy dashboard, one that's easier to use than the traditional web-based dashboards Kubernetes has included.

Prometheus is a metrics monitoring tool, and is the *de facto* standard for monitoring system metrics in Kubernetes clusters.

Grafana was originally forked from Kibana 3, but is developed with a focus on real-time metrics monitoring dash-

boards. Most of the fancy-looking monitoring dashboards you've seen in Kubernetes screenshots are built with Grafana. The CNCF even [uses Grafana to plot out project activity](#).

Lens is often used by individual Kubernetes developers, and can even be used by other members of a team to inspect deployments and explore application logs, following the permissions they have in the cluster.

Prometheus and Grafana are often used together to gather cluster and custom application metrics, and display them in user-friendly dashboards.

Two Clusters to Monitor

To help with the examples in this chapter, I recommend creating two Kubernetes clusters:

1. A local Minikube cluster following the example from chapter 1, with a kubeconfig file in `~/.kube/config`.
2. A Linode Kubernetes Engine cluster running Drupal, following the example from chapters 4, 5 and 6, with a kubeconfig file in `~/.kube/linode.yaml`.

Cluster Visibility with Lens

If you work with multiple clusters, managing context in the command line with `kubectl` can get annoying.

Luckily, the open source [Lens](#) app, available for Mac, Windows, or Linux, makes it easier to browse multiple clusters and dig into any resource you have the permission to manage.

Lens was originally created by a team at Kontena, but after acquiring Kontena in early 2020, Mirantis also acquired ownership of the Lens app later in 2020.

The app's source is still maintained under the MIT license, though, so I wouldn't worry about the license being swapped like some other vendors have done recently.

Install Lens

The easiest way to get Lens is to download a binary from the Lens website, at <https://k8slens.dev>.

On my Mac, I prefer to install things via Homebrew if possible, and Lens is available as a cask, so I can install it with:

```
$ brew install --cask lens
```

Lens is also available as a Snap for Linux.

Inspect your clusters with Lens

Once installed, you can open Lens and start adding clusters.

Click the '+' icon, and browse to a kubeconfig file. I added two clusters:

1. Local Minikube
 - kubeconfig: ~/.kube/config
 - context: minikube
2. Linode cluster
 - kubeconfig: ~/.kube/linode.yaml
 - context: lke18701-ctx (will vary)

After adding both clusters, I was able to see node information and browse all the resources in the cluster.

Explore Pod Logs

One of my favorite features of Lens is the ability to jump into any Pod's logs quickly through the UI.

If you go to the Workloads > Pods section, and select any Pod, you can click on the 'Logs' icon, and choose any of the containers in that Pod, to see it's logs.

While viewing the logs, you can switch containers, show timestamps (for containers that don't log timestamps in their own output), and even save the log file locally for deeper inspection or archive.

Log into Nodes and Pods

Another handy feature is the ability to log into any Kubernetes Node or Pod to which you have access.



WAIT WHAT!? You may ask... how can Lens allow users to log into a Node?

Well, if you didn't know this already, prepare for your mind to be blown: if you give someone admin privileges in a typical Kubernetes cluster, that user will be able to use Linux kernel namespace features to manage the *nodes themselves* as root.

This is a *feature*, not a bug, though there are more attempts lately to run Kubernetes in a more locked-down fashion.

But the mechanism Lens uses is fairly simple: when you request shell access to a Node, it runs a Pod on that node with privileged access, and runs the command `nsenter -t 1 -m -u -i -n sleep 14000`, thereby allowing that Pod to access resources on the node as root.

Read up on [Pod Security Policies](#) to learn about one way to mitigate attack vectors through privileged Pods.

If you weren't strict about granting admin-level access in your clusters before, you should probably think about being more strict! Anybody with that level of access effectively has root-level access to all nodes in your cluster.

You can visit any Node or Pod, and click the 'Node shell' or 'Pod shell' icon to be dropped into a terminal session inside that Node or Pod.

One major caveat: if the container you're logging into is running a minimal base image like Alpine or Scratch, it might not have a full-fledged interactive shell environment (like

bash), so the ability to debug anything inside the container may be limited.

Visit web services in a browser

Another handy feature is the ability for Lens to automatically create proxy connections between your cluster and your local computer, so you can open web services in your browser.

Just find a service exposed in your cluster under Network > Services, then click on the exposed TCP port. After a few seconds, it should open up in your browser.

This is equivalent to running `kubectl port-forward`, but Lens does all the magic for you.

Manage resources

You can edit any resource to which you have access by clicking the 'Edit' icon. You can scale Deployments, you can create new resources with the '+' icon... and you can even open up a Terminal session on your local computer with `kubectl` configured and ready to go by clicking +, then selecting 'Terminal session'.

You might notice, however, that Lens shows a blank section on the main Cluster overview, with the warning:

Metrics are not available due to missing or invalid Prometheus configuration.

Wouldn't it be nice to have those dashboards filled in, too?

While features like Horizontal Pod Autoscaling and `kubectl top` require metrics-server to be running in your cluster, Lens

relies on *Prometheus* for metrics data, so now's a good time to install it.

Prometheus and Grafana

Prometheus was created at SoundCloud in 2012, when they wanted to both simplify and scale their metrics monitoring beyond what they could do with StatsD and Graphite.

The project's governance was transferred to the CNCF in 2016, and Prometheus became the second 'CNCF graduated' project (after Kubernetes itself) in 2018.

Grafana is maintained by Grafana Labs, and has hundreds of integrations that make it the most flexible open source dashboard visualization and alerting tool. It doesn't only work with Prometheus, but most people looking for an open source monitoring stack for Kubernetes use it along with Prometheus.

Install Prometheus and Grafana using Helm

For the past few years, an effort was made to build a 'first class' out of the box experience for Kubernetes users integrating Prometheus and Grafana, and the fruit of that labor is the `kube-prometheus-stack` Helm chart, based on the [kube-prometheus](#) project.

This Helm chart installs the following in your cluster:

- Prometheus

- kube-state-metrics (gathers metrics from cluster resources)
- Prometheus Node Exporter (gathers metrics from Kubernetes nodes)
- Grafana
- Grafana dashboards and Prometheus rules for Kubernetes monitoring

To install the Helm chart, first add the Prometheus Community Helm repo and run `helm repo update`:

```
$ helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
$ helm repo update
```

Then create a monitoring namespace and install the stack inside:

```
$ kubectl create namespace monitoring
$ helm install prometheus --namespace monitoring prometheus-community/kube-prometheus-stack
```

Watch the progress in Lens, or via `kubectl`:

```
$ kubectl get deployments -n monitoring -w
```

Access Grafana

Once deployed, you can access Grafana using the default admin account and the default password `prom-operator`.

The Grafana password is stored in the `prometheus-grafana` secret, which you can view with the following command:

```
$ kubectl get secret -n monitoring prometheus-grafana\
na -o jsonpath="{.data.admin-password}" | base64 --\
decode ; echo
```

You can change the Grafana admin password via the parameters passed to the Helm chart when you install it. You should override this password if you're running Grafana in production!

To access Grafana in your browser, run:

```
$ kubectl port-forward -n monitoring service/promet\
heus-grafana 3000:80
```

Then open your browser and visit <http://localhost:3000/> and log in with the password you found from the earlier command.

But wait! If our cluster already has an Ingress Controller and cert-manager (as it did at the end of chapter 6), it's even easier to access Grafana. You can add an Ingress resource for Grafana just like we did for Drupal, and Nginx and cert-manager will work in tandem to give you a nice, friendly URL and TLS certificate for HTTPS access.

Create an ingress manifest named `grafana-ingress-tls.yml`, with the following contents:


```
1  ---
2  apiVersion: networking.k8s.io/v1
3  kind: Ingress
4  metadata:
5    name: grafana
6    namespace: monitoring
7    annotations:
8      kubernetes.io/ingress.class: nginx
9      cert-manager.io/cluster-issuer: letsencrypt-prod
10 spec:
11   tls:
12   - hosts:
13     - grafana.kube101.jeffgeerling.com
14     secretName: grafana-tls
15   rules:
16   - host: grafana.kube101.jeffgeerling.com
17     http:
18       paths:
19       - pathType: ImplementationSpecific
20         backend:
21           service:
22             name: prometheus-grafana
23             port:
24               number: 80
```

Apply it to the cluster with the command:

```
$ kubectl apply -f grafana-ingress-tls.yml
```

After a couple minutes, a cert should be acquired, and you can access Grafana at a friendly, secure URL—in my example, <https://grafana.kube101.jeffgeerling.com/>.

Grafana Dashboards

If you go to Dashboards > Manage, you'll see a list of the default dashboards that ship with the kube-prometheus-stack, including a number of dashboards for Compute Resources, Networking, and even Nodes.

Click on one of the dashboards (e.g. 'Nodes') to view the dashboard, and you should see live data going back to the time when Prometheus was installed.

The 'Cluster' dashboard is helpful for a quick overview of which namespaces are consuming the most resources.

Maintaining Grafana

Grafana has a full fledged access control system, a plugin system, custom dashboards, and more.

In that sense, it's a stateful application, and like Drupal, care must be taken if you want to persist your configuration and data like user accounts.

It's beyond the scope of this chapter, but if you are running Grafana in a production environment, you should install the monitoring stack with persistence enabled (the Grafana chart uses a `persistence.enabled` parameter that is set to `false` by default), and make sure you take backups or snapshots of the persistent volume Grafana uses.

Conclusion

With Prometheus installed, you should also be able to see cluster metrics in Lens now, though you may need to restart the app before metrics start coming in.

Now that you have more insights into your cluster's resource usage, you can make decisions about how and when to scale more easily, and also consider adding alerts either via AlertManager or Grafana which can be delivered through email, chat apps, or even services like PagerDuty.

Getting the right alerts set up for your own applications is often a trial-and-error process. In my experience, applications run differently on every cloud provider, and some elevated metrics that are concerning on one provider might never be a problem on another provider.

Don't get too bogged down in using a tool like Lens, though. Especially early on in your Kubernetes journey, a tool that abstracts away the basic commands to manage cluster resources can be detrimental to your actual understanding of Kubernetes!

Afterword

You should be well on your way towards streamlined infrastructure management. Many developers and sysadmins have been helped by this book, and many have even gone further and contributed *back* to the book, in the form of corrections, suggestions, and fruitful discussion!

Thanks to you for purchasing and reading this book, and a special thanks to all those who have given direct feedback in the form of corrections, PRs, or suggestions for improvement:

TODO: List of contributors goes here!