

LFS458

Kubernetes

Administration

Version 1.30.1



Version 1.30.1

© Copyright The Linux Foundation 2024 All rights reserved.

© Copyright The Linux Foundation 2024 All rights reserved.

The training materials provided or developed by The Linux Foundation in connection with the training services are protected by copyright and other intellectual property rights.

Open source code incorporated herein may have other copyright holders and is used pursuant to the applicable open source license.

The training materials are provided for individual use by participants in the form in which they are provided. They may not be copied, modified, distributed to non-participants or used to provide training to others without the prior written consent of The Linux Foundation.

No part of this publication may be reproduced, photocopied, stored on a retrieval system, or transmitted without express prior written consent.

Published by:

the Linux Foundation

<https://www.linuxfoundation.org>

No representations or warranties are made with respect to the contents or use of this material, and any express or implied warranties of merchantability or fitness for any particular purpose or specifically disclaimed.

Although third-party application software packages may be referenced herein, this is for demonstration purposes only and shall not constitute an endorsement of any of these software applications.

Linux is a registered trademark of Linus Torvalds. Other trademarks within this course material are the property of their respective owners.

If there are any questions about proper and fair use of the material herein, please go to <https://trainingsupport.linuxfoundation.org>.

Nondisclosure of Confidential Information

“Confidential Information” shall not include any of the following, even if marked confidential or proprietary: (a) information that relates to the code base of any open source or open standards project (collectively, “Open Project”), including any existing or future contribution thereto; (b) information generally relating or pertaining to the formation or operation of any Open Project; or (c) information relating to general business matters involving any Open Project.

This course does not include confidential information, nor should any confidential information be divulged in class.

Contents

1	Introduction	1
1.1	The Linux Foundation	2
1.2	The Linux Foundation Training	4
1.3	The Linux Foundation Certifications	8
1.4	The Linux Foundation Digital Badges	11
1.5	Laboratory Exercises, Solutions and Resources	12
1.6	Things Change in Linux and Open Source Projects	14
1.7	E-Learning Course: LFS258	15
1.8	Platform Details	16
2	Basics of Kubernetes	17
2.1	Define Kubernetes	18
2.2	Cluster Structure	19
2.3	Adoption	20
2.4	Project Governance and CNCF	28
2.5	Labs	30
3	Installation and Configuration	31
3.1	Getting Started With Kubernetes	32
3.2	Minikube	35
3.3	kubeadm	36
3.4	More Installation Tools	39
3.5	Labs	43
4	Kubernetes Architecture	65
4.1	Kubernetes Architecture	66
4.2	Networking	81
4.3	Other Cluster Systems	87
4.4	Labs	88
5	APIs and Access	105
5.1	API Access	106
5.2	Annotations	111
5.3	Working with A Simple Pod	112
5.4	kubectl and API	113
5.5	Swagger and OpenAPI	120
5.6	Labs	122
6	API Objects	127
6.1	API Objects	128
6.2	The v1 Group	129
6.3	API Resources	131
6.4	RBAC APIs	136
6.5	Labs	137
7	Managing State With Deployments	147

7.1	Deployment Overview	148
7.2	Deployments and Replica Sets	149
7.3	DaemonSets	158
7.4	Labels	159
7.5	Labs	161
8	Helm	175
8.1	Overview	176
8.2	Helm	177
8.3	Using Helm	180
8.4	Labs	182
9	Volumes and Data	187
9.1	Volumes Overview	188
9.2	Volumes	189
9.3	Persistent Volumes	193
9.4	Rook	197
9.5	Passing Data To Pods	198
9.6	ConfigMaps	201
9.7	Labs	203
10	Services	225
10.1	Overview	226
10.2	Accessing Services	228
10.3	DNS	234
10.4	Labs	236
11	Ingress	249
11.1	Overview	250
11.2	Ingress Controller	251
11.3	Ingress Rules	256
11.4	Service Mesh	258
11.5	Labs	259
12	Scheduling	269
12.1	Overview	270
12.2	Scheduler Settings	271
12.3	Pod Specification	276
12.4	Affinity Rules	278
12.5	Taints and Tolerations	283
12.6	Labs	286
13	Logging and Troubleshooting	295
13.1	Overview	296
13.2	Troubleshooting Flow	297
13.3	Basic Start Sequence	299
13.4	Monitoring	300
13.5	Plugins	301
13.6	Logging	305
13.7	Troubleshooting Resources	306
13.8	Labs	307
14	Custom Resource Definition	315
14.1	Overview	316
14.2	Custom Resource Definitions	317
14.3	Aggregated APIs	321
14.4	Labs	322
15	Security	327

15.1	Overview	328
15.2	Accessing the API	330
15.3	Authentication and Authorization	331
15.4	Admission Controller	335
15.5	Network Policies	337
15.6	Labs	341
16	High Availability	351
16.1	Overview	352
16.2	Stacked Database	353
16.3	External Database	354
16.4	Labs	355
17	Closing and Evaluation Survey	363
17.1	Evaluation Survey	363
Appendices		367
A	Domain Review	367
A.1	CKA Exam	368
A.2	Exam Domain Review	369

List of Figures

2.1	K8s Official Logo	19
2.2	Official Project Logos	21
2.3	The Kubernetes Lineage (by Chip Childers, Cloud Foundry Foundation, from LinkedIn Slideshare)	22
2.4	Kubernetes Architecture (image v2)	23
2.5	Kubernetes Users	26
2.6	CNCF	28
3.1	External Access via Browser	62
4.1	Architectural Overview	66
4.2	K8s Architectural Review	79
4.3	Pod Network	81
4.4	Container/Services Networking	82
4.5	Services	83
4.6	Mesos Architecture	87
5.1	Swagger Screenshot	120
9.1	K8s Pod Volumes	189
10.1	Built in services	230
10.2	Service Traffic	231
10.3	Example of Cluster Networking	232
11.1	Ingress Controller for inbound connections	251
11.2	Istio Service Mesh	258
11.3	Main Linkerd Page	260
11.4	Now shows meshed	261
11.5	Five meshed pods	262
11.6	Ingress Traffic	266
11.7	Linkerd Top Metrics	268
13.1	External Access via Browser	312
13.2	External Access via Browser	313
13.3	External Access via Browser	313
15.1	Accessing the API from kubernetes.io	330
16.1	Initial HAProxy Status	357
16.2	Multiple HAProxy Status	360
16.3	HAProxy Down Status	362
17.1	Course Survey	363

Chapter 1

Introduction



1.1	The Linux Foundation	2
1.2	The Linux Foundation Training	4
1.3	The Linux Foundation Certifications	8
1.4	The Linux Foundation Digital Badges	11
1.5	Laboratory Exercises, Solutions and Resources	12
1.6	Things Change in Linux and Open Source Projects	14
1.7	E-Learning Course: LFS258	15
1.8	Platform Details	16

1.1 The Linux Foundation

What is The Linux Foundation?

- A non-profit consortium, dedicated to fostering the growth of:
 - **Linux**
 - Many other **Open Source Software (OSS)** projects and communities
- Supports the creation of sustainable **OSS** ecosystems by providing:
 - Financial and intellectual resources and services
 - Training
 - Events
- Originally founded to protect, support and improve **Linux** development and sponsors the work of **Linux** creator Linus Torvalds
- Supported by leading technology companies and developers in a neutral collaborative environment
- See <https://linuxfoundation.org>.

The Linux Foundation provides a neutral, trusted hub for developers to code, manage, and scale open technology projects. Founded in 2000, **The Linux Foundation** is supported by more than 1,000 members and is the world's leading home for collaboration on open source software, open standards, open data and open hardware. **The Linux Foundation**'s methodology focuses on leveraging best practices and addressing the needs of contributors, users and solution providers to create sustainable models for open collaboration.

The Linux Foundation hosts **Linux**, the world's largest and most pervasive open source software project in history. It is also home to **Linux** creator Linus Torvalds and lead maintainer Greg Kroah-Hartman. The success of **Linux** has catalyzed growth in the open source community, demonstrating the commercial efficacy of open source and inspiring countless new projects across all industries and levels of the technology stack.

As a result, **The Linux Foundation** today hosts far more than **Linux**; it is the umbrella for many critical open source projects that power corporations today, spanning virtually all industry sectors. Some of the technologies we focus on include big data and analytics, networking, embedded systems and IoT, web tools, cloud computing, edge computing, automotive, security, blockchain, and many more.

The Linux Foundation Events

This is only a very partial list of **The Linux Foundation** events. Some are held in multiple locations yearly, such as North America, Europe and Asia.

- Open Source Summit
- Embedded Linux Conference North
- Open Networking & Edge Summit
- KubeCon + CloudNativeCon
- Automotive Linux Summit
- KVM Forum
- Linux Storage Filesystem and Memory Management Summit
- Linux Security Summit
- Linux Kernel Maintainer Summit
- The Linux Foundation Member Summit
- Open Compliance Summit
- And many more.

Over 85,000 open source technologists and leaders worldwide gather at **The Linux Foundation** events annually to share ideas, learn and collaborate. **The Linux Foundation** events are the meeting place of choice for open source maintainers, developers, architects, infrastructure managers, and sysadmins and technologists leading open source program offices, and other critical leadership functions.

These events are the best place to gain visibility within the open source community quickly and advance open source development work by forming connections with the people evaluating and creating the next generation of technology. They provide a forum to share and gain knowledge, help organizations identify software trends early to inform future technology investments, connect employers with talent, and showcase technologies and services to influential open source professionals, media, and analysts around the globe.

1.2 The Linux Foundation Training

Training Venues

The Linux Foundation offers several types of training:

- Physical Classroom (often On-Site)
- Online Virtual Classroom
- Individual Self-Paced E-learning over the Internet
- Events-Based

The Linux Foundation's training is for the community, by the community, and features instructors and content straight from the leaders of the developer community.

Attendees receive training that is operating system and/or **Linux** distribution-flexible, technically advanced and created with the actual leaders of the development community themselves. **The Linux Foundation** courses give attendees the broad, foundational knowledge and networking needed to thrive in their careers today. With either online or in person training, **The Linux Foundation** classes can keep you or your developers ahead of the curve on the essentials of open source administration and development.

Training Offerings

Our current course offerings include:

- Linux Programming & Development Training
- Enterprise IT & Linux System Administration Courses
- Open Source Compliance Courses

For more information see <https://training.linuxfoundation.org>

The Linux Foundation also offers a wide range of free **MOOCs** (**M**assively **O**pen **O**nline **C**ourses) offered through **edX** at <https://edx.org>. These cover basic as well as rather advanced topics associated with open source.

To find them at the **edX** website, search on "The Linux Foundation"

Copyright

- The contents of this course and all its related materials, including hand-outs, are © Copyright The Linux Foundation 2024 All rights reserved.



Do not copy or distribute

No part of this publication may be stored in a retrieval system, transmitted or reproduced in any way, including, but not limited to, photocopy, photograph, magnetic, electronic or other record, without the prior written permission of The Linux Foundation.

This training, including all material provided herein, is supplied without any guarantees from **The Linux Foundation**. **The Linux Foundation** assumes no liability for damages or legal action arising from the use or misuse of contents or details contained herein.

If you believe **The Linux Foundation** materials are being used, copied, or otherwise improperly distributed please go to <https://trainingsupport.linuxfoundation.org>.

About Confidential Information



Nondisclosure of Confidential Information

“Confidential Information” shall not include any of the following, even if marked confidential or proprietary: (a) information that relates to the code base of any open source or open standards project (collectively, “Open Project”), including any existing or future contribution thereto; (b) information generally relating or pertaining to the formation or operation of any Open Project; or (c) information relating to general business matters involving any Open Project.

This course does not include confidential information, nor should any confidential information be divulged in class.

1.3 The Linux Foundation Certifications

The Linux Foundation Certification Exams

- **The Linux Foundation** offers comprehensive **Certification Programs**.
- Full details about this program can be found at <https://training.linuxfoundation.org/certification>. This information includes a thorough description of the **Domains** and **Competencies** covered by each exam.
- Besides the **LFCS** (Linux Foundation Certified Sysadmin) exam, **The Linux Foundation** provides certification exams for many other open source projects. The list is constantly expanding so please see <https://training.linuxfoundation.org/certification> for the current list.
- For additional information, including course descriptions, technical requirements and other logistics, see <https://training.linuxfoundation.org>.

Certification/Training Firewall

- **The Linux Foundation** has two separate training divisions:
 - Certification
 - Course Delivery
- These are separated by a **firewall**:
 - Enables third party organizations to develop and deliver **LF** certification preparation classes
 - Prevents using **secret sauce** in **LF** courses
 - Prevents **teaching the test** in **LF** courses
- Instructors (including today) are guided entirely by publicly available information

The curriculum development and maintenance division of **The Linux Foundation** training department has no direct role in developing, administering, or grading certification exams.

Enforcing this self-imposed **firewall** ensures that independent organizations and companies can develop third party training material, geared to helping test takers pass their certification exams.

Furthermore, it ensures that there are no secret “tips” (or secrets in general) that one needs to be familiar with to succeed.

It also permits **The Linux Foundation** to develop a very robust set of courses that do far more than **teach the test**, but rather equip attendees with a broad knowledge of many areas they may be required to master to have a successful career in **Linux** system administration.

Preparation Resources

- Before doing anything else, download:

<https://training.linuxfoundation.org/download-free-certification-prep-guide>

- Sections on:

- Domains and Competencies
 - Free Training Resources
 - Paid Training Resources
 - Taking the Exam

- Also get the **Candidate Handbook** at:

https://training.linuxfoundation.org/go/candidate_handbook

- Also read exam-specific information at:

<https://training.linuxfoundation.org/certification/lfcs>

We will discuss some (but not all!) of the issues in the documents quoted above, all of which can also be easily be found through links at **The Linux Foundation** web page at <https://training.linuxfoundation.org/certification>.

Topics covered include:

- What material is covered in the exam
- Candidate Requirements, including identification, authentication, eligibility, accessibility, confidentiality requirements.
- Exam registration and fees, refund policy, etc.
- **Linux** distribution choices
- Checking your hardware and software environment for suitability for the exam
- How to start and complete the exam
- Exam Interface and format
- Exam results, scoring and re-scoring requests, and retake policy
- Certificate issuance, verification, expiration, renewal etc.
- Accessing tech support

1.4 The Linux Foundation Digital Badges

The Linux Foundation Digital Badges

- Digital Badges communicate abilities and credentials
- Can be used in email signatures, digital resumes, social media sites (**LinkedIn**, **Facebook**, **Twitter**, etc)
- Contain verified metadata describing qualifications and process that earned them.
- Available to students who successfully complete **The Linux Foundation** courses and certifications
- Details at <https://training.linuxfoundation.org/badges/>

The Linux Foundation is committed to providing you with the tools necessary to achieve your professional goals. We understand communicating your abilities and credentials can be challenging. For this reason, we have partnered with **Credly** to provide you with a digital version of your credentials through its **Acclaim** platform. These badges can be used in email signatures or digital resumes, as well as on social media sites such as **LinkedIn**, **Facebook**, and **Twitter**. This digital image contains verified metadata that describes your qualifications and the process required to earn them.

- Badges are shareable via any digital platform: social media, embedded in your résumé, email signature, or the web.
- All badges can be verified by anyone simply by clicking on the badge.
- Badges will be issued to everyone who passes one of our certification exams as well as those who purchase training courses directly from The Linux Foundation or an authorized training partner.
- Badges will also be issued to those who contributed on our exam or course development teams

How it Works

1. You will receive an email notifying you to claim your badge at our partner **Credly's Acclaim** platform website.
2. Click the link in that email.
3. Create an account on the **Acclaim** platform site and confirm your email.
4. Claim your badge.
5. Start sharing.

1.5 Laboratory Exercises, Solutions and Resources

Labs

- Hands-on exercises provided at the end of each session.
- Can be done on either virtual machines or bare-metal
 - Unless otherwise specified
- Some exercises marked as optional
- Solutions available at the end of each exercise.

Obtaining Course Solutions and Resources

- If this course has such material, lab exercise solutions, suggestions and other resources may be downloaded from: <https://cm.lf.training/LFS458>
- If you do not have a browser available you can obtain with:

```
$ wget --user=LFtraining --password=<ask instructor> \
  https://cm.lf.training/LFS458/LFS458_V1.30.1.SOLUTIONS.tar.xz
```

and similarly for the RESOURCES file.

```
$ wget --user=LFtraining --password=<ask instructor> \
  https://cm.lf.training/LFS458/LFS458_V1.30.1.RESOURCES.tar
```
- Any errata, updated solutions, etc. will also be posted on that site
- These files may be unpacked with:

```
$ tar xvf LFS458_V1.30.1.SOLUTIONS.tar.xz
$ tar xvf LFS458_V1.30.1.RESOURCES.tar
```

You will see subdirectories in the both the **SOLUTIONS** and **RESOURCES** directories such as s_01, s_10, s_13 for each section that has files you either need or may find of interest. We may refer to these files in future sections.

Depending on the course, there may not be a **RESOURCES** file, which is intended for binary files such as archives and videos.

Binary files, such as source tarballs for various labs, will be resourced with instructions as needed.

If the course has demonstration videos included, these are mostly intended for supplementary study outside of lecture time. However, the instruction might elect to show some of them to get a hopefully clean demonstration, or give an example of practices on alternative **Linux** distributions.

1.6 Things Change in Linux and Open Source Projects

Open Source Software Changes Often

- **The Linux Foundation** courses are constantly updated to synchronize with upstream versions of projects such as **Kubernetes** or the **Linux** kernel.
- There will be unavoidable breakage from time to time and deprecated features will disappear.
- We try to minimize problems, but often we have to respond to upstream changes we cannot control.
- The alternative is to have stale material which is lacking important features.
- Working with these shifting sources and targets is part of the *fun* of working with open source software!

The **Linux Foundation** courses are constantly updated, some of them with every new version release of projects they depend on, such as the **Linux** kernel or **Kubernetes**.

For example, no matter how hard we have worked to stay current, **Linux** is constantly evolving, both at the technical level (including kernel features) and at the distribution and interface level.

So please keep in mind we have tried to be as up to date as possible at the time this class was released, but there will be changes and new features we have not discussed. It is unavoidable.

As a result of this churn, no matter how hard we try there are inevitably features in the material that might suffer from one of the following:

- They will stop working on a **Linux** distribution we support.
- A change in **API** will break a laboratory exercise with a particular kernel or upstream library or product.
- Deprecated features we have been supporting will finally disappear.

There are students who just expect everything in the class to always work all the time. While we strive to achieve this, we know:

- It is an unrealistic expectation.
- Figuring out the problems and how to solve this is an active part of the class, part of the “adventure” of working with cutting edge open source projects. Moving targets are always harder to hit.

If we tried to avoid all such problems, we would have material that goes stale quickly and does not keep up with changes. We rely on our instructors and students to notice any such problems and inform courseware designers so they can incorporate the appropriate remedies in the next release.

This is the true open source way of doing things.

1.7 E-Learning Course: LFS258

Companion E-Learning Course: LFS258

- The Linux Foundation offers a self-paced, e-learning closely related course:

LFS258: Kubernetes Fundamentals

- This course covers much of the same material and shares many of the laboratory exercises as this course.
- Access to materials online is available for one year and you can take as much time as you need to cover all the content and exercises.
- Note that this instructor-led version cannot give adequate time to all subjects either in breadth or depth due to time constraints, and some topics are marked as optional.
- You may have received a subscription to **LFS258** as part of your enrollment in this course. Otherwise you can contact separately through <https://training.linuxfoundation.org/>

1.8 Platform Details

Platform Details

Any modern operating system can be used:

- **Linux**
- **MacOS**
- **Windows**
- **Unix**

Required Software:

- Modern Web Browser
- Terminal Emulation Program (**ssh** or **PuTTY**)

Chapter 2

Basics of Kubernetes



2.1	Define Kubernetes	18
2.2	Cluster Structure	19
2.3	Adoption	20
2.4	Project Governance and CNCF	28
2.5	Labs	30

2.1 Define Kubernetes

What Is Kubernetes

- Orchestration
 - <http://kubernetes.io>
- “Open-source software for automating deployment, scaling, and management of containerized applications”***
- Easy to run, potentially complex to integrate
 - Built with lessons from **Google**
 - Open and extensible
 - From Greek κυβερνητης, pilot or Helmsman

Running a container on a laptop is relatively simple. But connecting containers across multiple hosts, scaling them, deploying applications without downtime, and service discovery among several aspects, can be difficult.

Kubernetes addresses those challenges from the start with a set of primitives and a powerful open and extensible API. The ability to add new objects and controllers allows easy customization for various production needs.

A key aspect of Kubernetes is that it builds on 15 years of experience at **Google** in a project called **borg**.

Google's infrastructure started reaching high scale before virtual machines became pervasive in the datacenter, and containers provided a fine-grained solution for packing clusters efficiently. Efficiency in using clusters and managing distributed applications has been at the core of **Google** challenges.

The Kubernetes Name

In Greek, κυβερνητης means the Helmsman, or pilot of the ship. Keeping with the maritime theme of containers, Kubernetes is the pilot of a ship of containers.

Due to difficulty in pronouncing the name many will use a nickname, **K8s**, as Kubernetes has eight letters between K and S. The nickname is said like **Kate's**.

2.2 Cluster Structure

Components of Kubernetes

- Microservices
- Decoupling
- Built to be transient
- API call driven
- YAML to JSON, written in **Go**

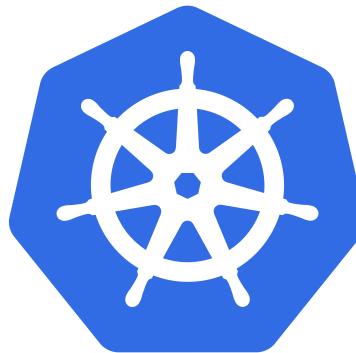


Figure 2.1: K8s Official Logo

Deploying containers and using Kubernetes may require a change in development and system administration approach to deploying applications. In a traditional environment, an application (such as a web server) would be a monolithic application placed on a dedicated server. As the web traffic increases the application would be tuned and perhaps moved to bigger and bigger hardware. After a couple of years, lot of customization may have been done in order to meet the current web traffic needs.

Instead of using a large server, Kubernetes approaches the same issue by deploying a large number of small servers, or **microservices**. The server and client sides of the application are written to expect there to be many possible agents available to respond to a request. It is also important that clients expect the server processes to die and be replaced, leading to a transient server deployment. Instead of a large **Apache** web server with many **httpd** daemons responding to page requests, there would be many **nginx** servers, each responding.

The transient nature of smaller services also allows for decoupling. Each aspect of the traditional application is replaced with a dedicated, but transient, microservice or agent. To join these agents, or their replacements, together we use **services**. A service ties traffic from one agent to another (for example a front end web server to a back end database) and handles new IP or other information should either one die and be replaced.

Communication is entirely API call driven, which allows for flexibility. Cluster configuration information is stored in **JSON** format inside of **etcd**, but is most often written in **YAML** by the community. Kubernetes agents convert the YAML to JSON prior to persistence to the database.

Go

Kubernetes is written in **Go Language**, a portable language which is like a hybridization between C++. Python, and Java. Some claim it incorporates the best (while some claim the worst) parts of each.

2.3 Adoption

Challenges

- Easy to build container images
- Simple to share images via registries
- Powerful user tools to manage containers
- Network considerations
- Flexible storage

Containers provide a great way to package, ship, and run applications; that is the **Docker** motto.

The developer experience has been boosted tremendously thanks to containers. Containers (and **Docker** specifically) have empowered developers with ease of building container images, simplicity of sharing images via **Docker** registries, and providing a powerful user experience to manage containers.

However, managing containers at scale and designing a distributed application based on microservices' principles may be challenging.

A smart first step is deciding on a continuous integration / continuous delivery (CI/CD) pipeline to build, test, and verify container images. Tools such as <https://www.spinnaker.io>, <https://www.jenkins.io>, and <https://helm.sh> can be helpful to use, among other possible tools. This will help with the challenge of a dynamic environment.

Then you need a cluster of machines acting as your base infrastructure on which to run your containers. You also need a system to launch your containers, watch over them when things fail and replace as required. Rolling updates and easy rollbacks of containers is an important feature, and eventually tear down the resource when no longer needed.

All of these actions require flexible, scalable, and easy to use network and storage. As containers are launched on any worker node, the network must join the resource to other containers, while still keeping the traffic secure from others. We also need a storage structure which provides and keeps or recycles storage in a seamless manner.

While Kubernetes answers these concerns one of the biggest challenges to adoption is the applications themselves, running inside the container. They need to be written, or re-written, to be truly transient. A good question to ponder: If you were to deploy **Chaos Monkey**, which could terminate any containers at any time, would your customers notice?

Other Solutions

- Docker Swarm
- Apache Mesos
- Nomad
- Rancher



Figure 2.2: Official Project Logos

Built on open source and easily extensible, **Kubernetes** (which we will be covering in this course) is definitely a solution to manage containerized applications.

There are other solutions as well, including:

- **Docker Swarm** is the **Docker** Inc. solution. It has been re-architected recently and is based on SwarmKit. It is embedded with the **Docker** Engine. <https://github.com/docker/swarmkit>
- **Apache Mesos** is a data center scheduler, which can run containers through the use of frameworks. **Marathon** is the framework that lets you orchestrate containers. <https://mesosphere.github.io/marathon/>
- **Nomad** from HashiCorp, the makers of Vagrant and Consul, is another solution for managing containerized applications. Nomad schedules tasks defined in Jobs. It has a **Docker** driver which lets you define a running container as a task. <https://www.nomadproject.io/>
- **Rancher** is a container orchestrator agnostic system, which provides a single pane of glass interface to managing applications. It supports **Mesos**, **Swarm**, **Kubernetes**, as well as its native system, **Cattle**. As of 2.0 **Cattle** has been dropped in favor of Kubernetes. <http://rancher.com>

Borg Heritage

- Orchestration system to manage all **Google** applications at scale
- Described publicly in 2015

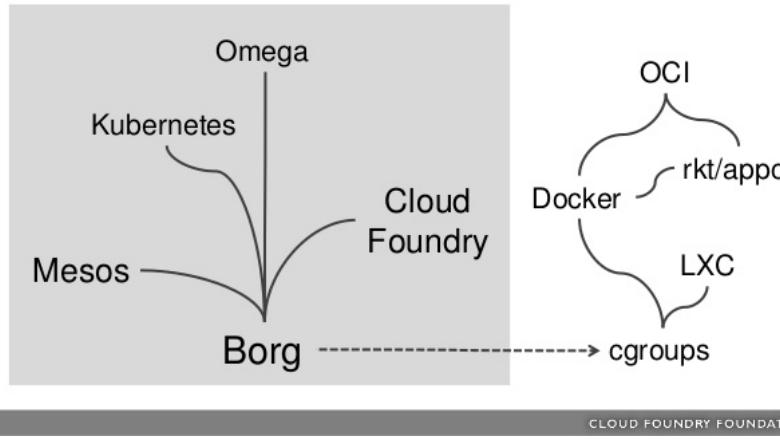


Figure 2.3: The Kubernetes Lineage (by Chip Childers, Cloud Foundry Foundation, from LinkedIn Slideshare)

What primarily distinguishes **Kubernetes** from other systems is its heritage. **Kubernetes** is inspired by **Borg**, the internal system used by **Google** to manage its applications (e.g **Gmail**, **Apps**, **GCE**).

With **Google** pouring the valuable lessons they learned from writing and operating Borg for over 15 years into **Kubernetes**, this makes **Kubernetes** a safe choice when having to decide on what system to use to manage containers. While a powerful tool it was secret and internal, part of the current growth in Kubernetes is making it easier to work with and handle workloads not found in a **Google** datacenter.

To learn more about the ideas behind Kubernetes, you can read the ***Large-scale cluster management at Google with Borg*** paper. <https://research.google.com/pubs/pub43438.html>

Borg has inspired current data center systems, as well as the underlying technologies used in container runtime today. **Google** contributed **cgroups** to the Linux kernel in 2007; it limits the resources used by collection of processes. Both **cgroups** and **Linux namespaces** are at the heart of containers today, including **Docker**.

Mesos was inspired by discussions with **Google** when **Borg** was still a secret. Indeed, **Mesos** builds a multi-level scheduler, which aims to better use a data center cluster.

Cloud Foundry Foundation embraces the 12 factor application principles. These principles provide great guidance to build web applications that can scale easily, can be deployed in the Cloud, and whose build is automated. Borg and Kubernetes address these principles as well.

Graphic retrieved from goo.gl/vwaAG4

Kubernetes Architecture

- Control Planes
 - API server, Scheduler, Controllers, and etcd storage system
- Worker nodes

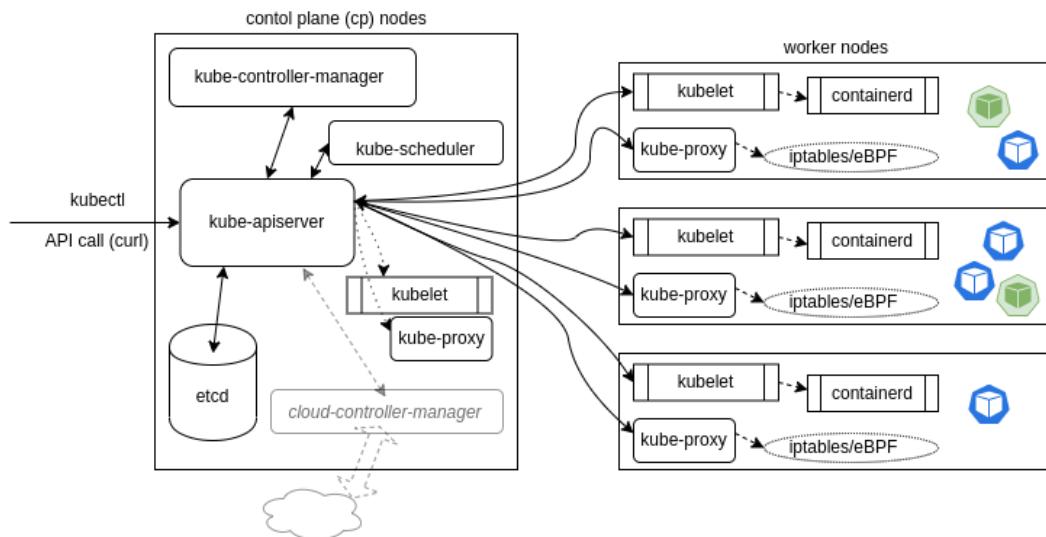


Figure 2.4: Kubernetes Architecture (Image v2)

To quickly demystify Kubernetes, let's have a look at a graphic, which shows a high-level architecture diagram of the system components. Not all components are shown. Every node running a container would have **kubelet** and **kube-proxy** for example.

In its simplest form, Kubernetes is made of **control plane nodes** (aka **cp** nodes) and worker nodes, once called minions. We will see in a follow-on chapter how you can actually run everything on a single node for testing purposes. The cp runs an API server, a scheduler, various controllers and a storage system to keep the state of the cluster, container settings, and networking configuration.

Kubernetes exposes an API via the API server. You can communicate with the API using a local client called **kubectl** or you can write your own client and use **curl** commands. The **kube-scheduler** is forwarded the pod spec for running containers coming to the API and finds a suitable node to run those containers. Each node in the cluster runs two processes, a **kubelet**, which is often a **systemd** process, not a container, and a **kube-proxy**. The **kubelet** receives requests to run the containers, manages any resources necessary, and works with the container engine to manage them on the local node. The local container engine could be **Docker**, **cri-o**, **containerd** or some other.

The **kube-proxy** creates and manages networking rules to expose the container on the network to other containers or the outside world.

Using an API based communication scheme allows for non-Linux worker nodes and containers. Support for **Windows Server 2019** was graduated to Stable with the 1.14 release. Only **Linux** nodes can be cp of the cluster at this time.

Terminology

- Pod
- Namespace
- Operator/Controller/Watch-loop
 - ReplicaSet
 - Deployment
 - DaemonSet
 - Jobs
 - Service
- Label
- Taint
- Toleration
- Annotation

We have learned that Kubernetes is an orchestration system to deploy and manage containers. Containers are not managed individually; instead they are part of a larger object called a Pod. A Pod consists of one or more containers which share an IP address, access to storage and namespace. Typically one container in a Pod runs an application while other containers support the primary application.

Kubernetes uses namespaces to keep objects distinct from each other, for resource control and multi-tenant considerations. Some objects are cluster scoped, others are scoped to one namespace at a time. As the namespace is a segregation of resources pods would need to leverage services to communicate.

Orchestration is managed through a series of watch-loops, also called controllers or operators. Each interrogates the kube-apiserver for a particular object state, then modifying the object until the declared state matches the current state. These controllers are compiled into the kube-controller-manager, but others can be added using custom resource definitions. The default and feature-filled operator for containers is a Deployment. A Deployment does not directly work with pods. Instead it manages ReplicaSets. The ReplicaSet is an operator which will create or terminate pods according to a podspec. The podspec is sent to kubelet which then interacts with the container engine, to download and make available required resources then spawn or terminate containers until the status matches the spec.

The service operator requests existing IP addresses and information from the endpoint operator, and will manage the network connectivity based on labels. A service is used to communicate between pods, namespaces, and outside the cluster. There are also Jobs and CronJobs to handle single or recurring tasks, among other default operators.

To easily manage thousands of Pods across hundreds of nodes could be difficult. To make management easier we can use labels, arbitrary strings which become part of the object metadata. These can then be used when checking or changing the state of objects without having to know individual names or UIDs. Nodes can have taints to discourage Pod assignments, unless the Pod has a toleration in its metadata.

There is also space in metadata for annotations which remain with the object but is not used as a selector. This information could be used by third-party agents, or other tools.

Innovation

- Given to open source June 2014
- Thousands of contributors
- More than 100K commits
- Tens of thousands on **Slack**
- Currently four month minor release cycle
- Patch releases every ten days or so
- Constant change

Since its inception, Kubernetes has seen a terrific pace of innovation and adoption. The community of developers, users, testers, and advocates is continuously growing every day. The software is also moving at an extremely fast pace, which is even putting GitHub to the test.

Understanding and expecting constant change is often a challenge for those used to single-vendor, monolithic applications. While there are major releases every four months, you can expect minor releases every ten days. Constant testing of new releases and changes is essential to properly maintain a Kubernetes cluster.

User Community

Kubernetes Users



Figure 2.5: Kubernetes Users

Kubernetes is being adopted at a very rapid pace. To learn more, you should check out the case studies presented on the Kubernetes website. **Ebay**, **box**, **Pearson** and **Wikimedia** have all shared their stories.

Pokemon Go, the fastest growing mobile game, also runs on **Google Container Engine (GKE)**, the Kubernetes service from **Google Cloud Platform (GCP)**.

Tools

- Several tools to choose from
 - Minikube
 - kubeadm
 - kubectl
 - Dashboard
 - kops
 - helm
 - kompose

There are several tools to work with Kubernetes. As the project has grown new tools are made and old ones deprecated. **Minikube** is a very simple tool meant to run inside of **VirtualBox**. If you have limited resources and do not want much hassle it is the easiest way to get up and running. We mention it for those who are not interested in a typical production environment, but want to use the tool.

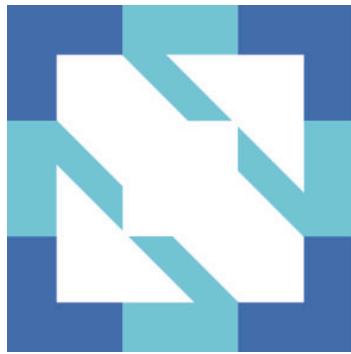
Our labs will focus on use of **kubeadm** and **kubectl**, which are very powerful and complex tools.

In a later chapter we will work with **helm**, an easy tool for using Kubernetes, to search for and install software using charts. There are also helpful commands like **Kompose** to translate **Docker Compose** files into Kubernetes objects, should that be desired.

Expect these tools to change often.

2.4 Project Governance and CNCF

Cloud Native Computing Foundation



CLOUD NATIVE
COMPUTING
FOUNDATION

Figure 2.6: CNCF

Kubernetes is open source software using an **Apache** license. **Google** donated Kubernetes to a newly formed collaborative project within the **Linux Foundation** in July 2015, when Kubernetes reached the v1.0 release. This project was named the **Cloud Native Computing Foundation (CNCF)**.

CNCF is not just about Kubernetes; it serves as the governing body for open source software that solves specific issues faced by Cloud native applications (i.e applications that are written specifically for a Cloud environment).

CNCF has many corporate members that collaborate, such as **Cisco**, the **Cloud Foundry Foundation**, **AT&T**, **Box**, **Goldman Sachs**, and many others.



Please Note

Since **CNCF** now owns the Kubernetes copyright, contributors to the source need to sign a contributor license agreement (CLA) with **CNCF**, just like any contributor to an Apache-licensed project signs a CLA with the Apache Software Foundation.

Resource Recommendations

- The Borg Paper
- John Wilkes speech
- Local community hangout
- Slack channel
- Stack Overflow community

Various URLs for more information:

<https://research.google.com/pubs/pub43438.html>

<https://www.gcppodcast.com/post/episode-46-borg-and-k8s-with-john-wilkes/>

<https://github.com/kubernetes/community>

<http://slack.kubernetes.io/>

<http://stackoverflow.com/search?q=kubernetes>

2.5 Labs

Exercise 2.1: View Online Resources

Visit kubernetes.io

With such a fast changing project, it is important to keep track of updates. The main place to find documentation of the current version is <https://kubernetes.io/>.

1. Open a browser and visit the <https://kubernetes.io/> website.
2. In the upper right hand corner, use the drop down to view the versions available. It will say something like v1.30.
3. Select the top level link for Documentation. The links on the left of the page can be helpful in navigation.
4. As time permits navigate around other sub-pages such as SETUP, CONCEPTS, and TASKS to become familiar with the layout.

Track Kubernetes Issues

There are hundreds, perhaps thousands, working on Kubernetes every day. With that many people working in parallel there are good resources to see if others are experiencing a similar outage. Both the source code as well as feature and issue tracking are currently on github.com.

1. To view the main page use your browser to visit [https://github.com/kubernetes/kubernetes/](https://github.com/kubernetes/kubernetes)
2. Click on various sub-directories and view the basic information available.
3. Update your URL to point to <https://github.com/kubernetes/kubernetes/issues>. You should see a series of issues, feature requests, and support communication.
4. In the search box you probably see some existing text like `is:issue is:open:` which allows you to filter on the kind of information you would like to see. Append the search string to read: `is:issue is:open label:kind/bug:` then press enter.
5. You should now see bugs in descending date order. Across the top of the issues a menu area allows you to view entries by author, labels, projects, milestones, and assignee as well. Take a moment to view the various other selection criteria.
6. Some times you may want to exclude a kind of output. Update the URL again, but precede the label with a minus sign, like: `is:issue is:open -label:kind/bug`. Now you see everything except bug reports.

Chapter 3

Installation and Configuration



3.1	Getting Started With Kubernetes	32
3.2	Minikube	35
3.3	kubeadm	36
3.4	More Installation Tools	39
3.5	Labs	43

3.1 Getting Started With Kubernetes

Installation Tools

- **kubeadm** community tool
- Google Kubernetes Engine (**GKE**)
- Amazon Elastic Kubernetes Service (**EKS**)
- Digital Ocean Kubernetes
- **Minikube**
- Kubernetes Operations (**kops**)
- MicroK8s
- May need **kubectl** or proprietary command

This chapter is about Kubernetes installation and configuration. We are going to review a few installation mechanisms that you can use to create your own Kubernetes cluster.

To get started without having to dive right away into installing and configuring a cluster, there are a few choices.

One way is to use **Google Container Engine (GKE)**, a Cloud service from the **Google Cloud Platform**, that lets you request a Kubernetes cluster with the latest stable version. **Amazon** has a service **Elastic Kubernetes Service (EKS)**, which allows more control of the cp nodes. Another easy way to get started is to use **Minikube**. It is a single binary which deploys into **Oracle VirtualBox** software, which can run in several operating systems. While Minikube is local and single node, it will give you a learning, testing, and development platform. A newer tool is from Canonical, **MicroK8s**, aimed at easy installation. Aimed at appliance-like installations it currently only runs on Ubuntu 16.04 and later. More can be found here <https://microk8s.io/docs/>.

To be able to use the Kubernetes cluster, you will need to have installed the Kubernetes command line, called **kubectl**, or a wrapper command such as **gcloud**. This runs locally on your machine and targets the API server endpoint. It allows you to create, manage, and delete all Kubernetes resources (e.g. Pods, Deployments, Services). It is a powerful CLI that we will use throughout the rest of this course. So, you should become familiar with it.

We will use **kubeadm**, the community-suggested tool from the Kubernetes project, that makes installing Kubernetes easy and avoids vendor-specific installers. Getting a cluster running involves basically two commands: **kubeadm init**, that you run on a cp node, and then, **kubeadm join**, that you run on your worker or redundant cp nodes, and your cluster bootstraps itself. The flexibility of these tools allows Kubernetes to be deployed in a number of places. The lab exercise uses this method, as does the high availability chapter.

Other installation mechanisms, such as **kubespray** or **kops**, are used to create a Kubernetes cluster on AWS nodes. As Kubernetes is so popular there are several other tools you may use to configure a cluster. Some may become popular, others may become defunct within months.

Installing kubectl

- Install or compile **kubectl**
- Main binary for working with objects
- Available for common distributions via dedicated repos
- Configuration file: `$HOME/.kube/config`
 - endpoints
 - SSL keys
 - contexts

To configure and manage your cluster, you will probably use the **kubectl** command. You can use **RESTful** calls or the **Go** language as well.

Enterprise **Linux** distributors have the various Kubernetes utilities and other files available in their repositories. For example, on **RHEL/CentOS**, you would find **kubectl** in the **kubernetes-client** package. On **OpenShift** they use a command very similar to **kubectl** called **oc**.

You can (if needed) download the code from github.com/kubernetes/kubernetes/tree/master/pkg/kubectl and go through the usual steps to compile and install.

This command line will use `$HOME/.kube/config` as a configuration file. This contains all the Kubernetes endpoints that you might use. If you examine it, you will see cluster definitions (i.e. IP endpoints), credentials, and contexts.

A context is a combination of a cluster and user credentials. You can pass these parameters on the command line or switch the shell between contexts with a command as in:

```
$ kubectl config use-context foobar
```

This is handy when going from a local environment to a cluster in the Cloud, or from one cluster to another, such as from development to production.

Using Google Kubernetes Engine (GKE)

- Create account on **GKE**
- Add method of payment
- Install and use **gcloud**
 - Vendor-specific command to manage GKE
- More details:
<https://console.cloud.google.com/getting-started>

Google takes every Kubernetes release through rigorous testing and makes it available via its **GKE** service. To be able to use GKE, you will need an account on **Google Cloud**, a method of payment for the services you will use, and the **gcloud** command line client.

There is an extensive documentation to get it installed. Pick your favorite method of installation and set it up.

See <https://cloud.google.com/sdk/downloads#linux>.

You will then be able to follow the GKE quickstart guide and you will be ready to create your first Kubernetes cluster:

```
$ gcloud container clusters create linuxfoundation  
$ gcloud container clusters list  
$ kubectl get nodes
```

By installing **gcloud**, you will have automatically installed **kubectl**. In the commands above, we created the cluster, listed it, and then, listed the nodes of the cluster with **kubectl**.

Once you are done, **do not forget to delete your cluster** otherwise you will keep on getting charged for it:

```
$ gcloud container clusters delete linuxfoundation
```

3.2 Minikube

Using Minikube

- Open source project within GitHub Kubernetes
- Download from **Google**
- Assumes **VirtualBox** already installed
- Useful for developers
- Uses **Go** binary **localkube**
- Also uses **Docker**

While you can download a release from **GitHub** and compile following listed directions, it may be easier to download a pre-compiled binary. Make sure to verify and get the latest version.

```
$ curl -Lo minikube https://storage.googleapis.com/minikube/releases/latest/minikube-darwin-amd64  
$ chmod +x minikube  
$ sudo mv minikube /usr/local/bin
```

With Minikube now installed, starting Kubernetes on your local machine is very easy:

```
$ minikube start  
$ kubectl get nodes
```

This will start a **VirtualBox** virtual machine that will contain a single node Kubernetes deployment and the **Docker** engine. Internally, **minikube** runs a single Go binary called **localkube**. This binary runs all the components of Kubernetes together. This makes Minikube simpler than a full Kubernetes deployment. In addition, the Minikube VM also runs **Docker**, in order to be able to run containers.

3.3 kubeadm

Install With kubeadm

- Available since Kubernetes 1.4.0, added HA in 1.15.0
- Works with current versions of **Ubuntu** and **CentOS**
- Main steps
 - Run `kubeadm init` on the control plane node
 - Create a network for IP-per-Pod criteria
 - Run `kubeadm join` on workers or secondary cp nodes

Once you become familiar with Kubernetes using Minikube, you may want to start building a real cluster. Currently, the most straightforward method is to use **kubeadm**, which appeared in Kubernetes 1.4.0, and can be used to bootstrap a cluster quickly. As the community has focused on **kubeadm** it has moved from beta to stable and added high availability with v1.15.0.

Documentation on how to do this is available on the Kubernetes website: <https://kubernetes.io/docs/setup/independent/create-cluster-kubeadm/>.

Package repositories are available for current versions of **Ubuntu** and **CentOS**, among others. We will work with **Ubuntu** in our lab exercises.

To join other nodes to the cluster you will need at least one token and a SHA256 hash. This information is returned by the command, **kubeadm init**. Once the cp has initialized you would apply a network plugin. You can also create the network with **kubectl**, by using a resource manifest of the network plugin to be used. Each plugin may have a different method of installation.

For example, to use the **Weave** network, you would do the following:

```
$ kubectl create -f https://git.io/weave-kube
```

Once all the steps are completed, workers and other cp nodes joined, you will have a functioning multi-node Kubernetes cluster and will be able to use **kubectl** to interact with it.

kubeadm upgrade

- Allows validation prior to and eventual upgrade
- Upgrade to alpha, beta, release candidate, or stable release
- Sub-commands for various functions
 - plan
 - apply
 - diff
 - node

If you build your cluster with **kubeadm**, you also have the option to upgrade the cluster using the **kubeadm upgrade** command. While most choose to remain with a version for as long as possible, and will often skip several releases, this does offer a useful path to regular upgrades for security reasons.

plan - This will check the installed version against the newest found in the repository, and verify the cluster can be upgraded.

apply - Upgrades the first control plane node of the cluster to the specified version.

diff - Similar to an **apply --dry-run** this command will show the differences applied during an upgrade.

node - This allows for updating the local kubelet configuration on worker nodes, or the control planes of other cp nodes if there is more than one. Also will accept a **phase** command to step through the upgrade process.

General upgrade process:

- Update the software
- Check the software version
- Drain the control plane
- View the planned upgrade
- Apply the upgrade
- Uncordon the control plane to allow pods to be scheduled

Detailed steps can be found here: <https://kubernetes.io/docs/tasks/administer-cluster/kubeadm/kubeadm-upgrade/>

Install A Pod Network

- Only one pod network per cluster
- Several to choose from
 - **Calico**
 - **Canal**
 - **Flannel**
 - **Kube-router**
 - **Cilium**
- Can become complicated to manage
- Several add-ons available

Prior to initializing the Kubernetes cluster, the network must be considered and IP conflicts avoided. There are several Pod networking choices, in varying levels of development and feature set.

Calico (projectcalico.org)

A flat Layer 3 network which communicates without IP encapsulation used in production with software such as Kubernetes, **OpenShift**, **Docker**, **Mesos** and **OpenStack**. Viewed as a simple and flexible it scales well for large environments. Another network option **Canal**, also part of this project, allows for integration with **Flannel**. Allows for implementation of network policies.

Flannel (github.com/coreos/flannel)

A Layer 3 IPv4 network between nodes of a cluster. Developed by **CoreOS** it has a long history with Kubernetes. Focused on traffic between hosts, not how containers configure local networking, it can use one of several backend mechanisms such as VXLAN. A **flanneld** agent on each node allocates subnet leases for the host. While it can be configured after deployment it is much easier prior to any Pods being added.

Kube-router (github.com/cloudnativelabs/kube-router)

Feature-filled single binary which claims to "do it all". The project is alpha stage but promises to offer a distributed load balancer, firewall, and router purpose built for Kubernetes.

Cilium (<https://cilium.io/>) A newer but incredibly powerful network plugin which is used by major cloud providers. Via the use of eBPF and other features this network plugin has become so powerful it is considered a service mesh, which we will discuss later in the course.

Many of the projects will mention **Container Network Interface (CNI)** which is a **CNCF** project. Several container runtimes currently use CNI. As a standard to handle deployment management and cleanup of network resources, it will become more popular.

3.4 More Installation Tools

More Installation Tools

- **kubespray**
- **kops**
- **kube-aws**
- **kind**

Since Kubernetes is, after all, like any other application installed on a server (whether physical or virtual), all of the configuration management systems (e.g. **Chef**, **Puppet**, **Ansible**, **Terraform**) can be used. Various recipes are available on the Internet.

Here are just a few examples of installation tools that you can use:

- **kubespray**

Now in the Kubernetes incubator. It is an advanced **Ansible** playbook which allows you to setup a Kubernetes cluster on various operating systems and use different network providers. Was once known as **kargo**.

- **kops**

Lets you create a Kubernetes cluster on **AWS** via a single command line. Also in beta for **GKE** and alpha for **VMWare**.

- **kube-aws**

A command line tool that makes use of the **AWS Cloud Formation** to provision a Kubernetes cluster on **AWS**.

- **kind**

One of a few methods to run Kubernetes locally. Currently written to work with **Docker**.



Please Note

The best way to learn how to install Kubernetes using step-by-step manual commands is to examine Kelsey Hightower's walk through: <https://github.com/kelseyhightower/kubernetes-the-hard-way>.

Installation Considerations

- Which provider should I use?
 - Public or private cloud?
- Which operating system should I use?
- Which networking solution should I use?
 - Do I need an overlay?
- Where should I run my **etcd** cluster?
- Should I configure Highly-Available head nodes?

To begin the installation process, you should start experimenting with a single-node deployment. This single-node will run all the Kubernetes components (e.g. API server, controller, scheduler, **kubelet**, and **kube-proxy**). You can do this with Minikube, for example.

Once you want to deploy on a cluster of servers (physical or virtual), you will have many choices to make, just like with any other distributed system.

To learn more about how to choose the best options, you can see **Picking the Right Solution** at <http://kubernetes.io/docs/getting-started-guides/>.

With **systemd** now the dominant **init** system on **Linux**, your Kubernetes components will end up being run as **systemd** unit files in most cases. Or, they will be run via a **kubelet** running on the head node (i.e. **kubadbm**).

The lab exercises were written using **Google Compute Engine (GCE)** nodes. Each node with 2 vCPUs and 7.5G of memory, running Ubuntu 20.04. Smaller nodes should work, but you should expect slow response. Other operating system images are also possible, but there may slight difference in some command output. Use of **GCE** requires setting up an account and will incur expense if using nodes of the size suggested. You can view Getting Started pages here: <https://cloud.google.com/compute/docs/quickstart-linux>

Amazon Web Service (AWS) is another provider of cloud-based nodes, and requires an account and will incur expense for nodes of the suggested size. You can find videos and getting-started information here: <https://aws.amazon.com/getting-started/tutorials/launch-a-virtual-machine>

Virtual machines such as **KVM**, **VirtualBox**, or **VMWare** can also be used for the lab systems. Putting the VMs on a private network can make troubleshooting easier.

Finally using bare-metal nodes, with access to the internet, will also work for the lab exercises.

Main Deployment Configurations

- Single-node
- Single head node, multiple workers
- Multiple head nodes with HA, multiple workers
- HA etcd, HA head nodes, multiple workers
- Federation also provides higher availability

At a high level, you have four main deployment configurations. Which of the four you will use will depend on how advanced you are in your Kubernetes journey, but also on what your goals are.

With a single-node deployment all the components run on the same server. This is great for testing, learning, and developing around Kubernetes.

Adding more workers, a single head node and multiple workers typically will consist of a single node **etcd** instance running on the head node with the API, the scheduler, and the controller-manager.

Multiple head nodes in an HA configuration and multiple workers add more durability to the cluster. The API server will be fronted by a load balancer, the scheduler and the controller-manager will elect a leader (which is configured via flags). The **etcd** setup can still be single node.

The most advanced and resilient setup would be an HA **etcd** cluster, with HA head nodes and multiple workers. Also **etcd** would run as a true cluster, which would provide HA and would run on nodes separate from the Kubernetes head nodes.

The use of Kubernetes Federation also offers higher availability. Multiple clusters are joined together with a common control plane allowing movement of resources from one cluster to another administratively or after failure. While Federation has had some issues there is hope v2 will be a stronger product. <https://github.com/kubernetes-sigs/Kubefed>

Compiling from Source

- Configure **Golang** environment
- Clone source code
- May need to install other compiler and libraries

The list of binary releases is available on GitHub: <https://github.com/kubernetes/kubernetes/releases>. Together with **gcloud**, **minikube**, and **kubeadm**, these cover several scenarios to get started with Kubernetes.

Kubernetes can also be compiled from source relatively quickly. You can clone the repository from GitHub, and then use the [Makefile](#) to build the binaries. You can build them natively on your platform if you have a **Golang** environment properly setup, or via containers or virtual machines.

To build natively with **Golang**, first install **Golang**. Download and directions can be found here: <https://golang.org/doc/install>. Once **Golang** is working, you can clone the kubernetes repository, around 500 MB in size. Change into the directory and use **make**:

```
$ cd $GOPATH  
$ git clone https://github.com/kubernetes/kubernetes  
$ cd kubernetes  
$ make
```

There may be other software and settings you need in order for the **make** to work properly. Review the output until it completes properly.

The [_output/bin](#) directory will contain the newly built binaries.

You may find some materials via <https://gitlab.com/>, but most resources are on <https://github.com/> at the moment.

3.5 Labs

Exercise 3.1: Install Kubernetes

Overview

There are several Kubernetes installation tools provided by various vendors. In this lab we will learn to use **kubeadm**. As a community-supported independent tool, it is planned to become the primary manner to build a Kubernetes cluster.



Platforms: Digital Ocean, GCP, AWS, VirtualBox, etc

The labs were written using **Ubuntu 20.04** instances running on **Google Cloud Platform (GCP)**. They have been written to be vendor-agnostic so could run on AWS, local hardware, or inside of virtualization to give you the most flexibility and options. Each platform will have different access methods and considerations. As of v1.21.0 the minimum (as in barely works) size for **VirtualBox** is 3vCPU/4G memory/5G minimal OS for cp and 1vCPU/2G memory/5G minimal OS for worker node. Most other providers work with 2CPU/7.5G.

If using your own equipment you will have to disable swap on every node, and ensure there is only one network interface. Multiple interfaces are supported but require extra configuration. There may be other requirements which will be shown as warnings or errors when using the **kubeadm** command. While most commands are run as a regular user, there are some which require root privilege. If you are accessing the nodes remotely, such as with **GCP** or **AWS**, you will need to use an SSH client such as a local terminal or **PuTTY** if not using **Linux** or a Mac. You can download **PuTTY** from www.putty.org. You would also require a **.pem** or **.ppk** file to access the nodes. Each cloud provider will have a process to download or create this file. If attending in-person instructor led training the file will be made available during class.



Very Important

Please disable any firewalls while learning Kubernetes. While there is a list of required ports for communication between components, the list may not be as complete as necessary. If using **GCP** you can add a rule to the project which allows all traffic to all ports. Should you be using **VirtualBox** be aware that inter-VM networking will need to be set to promiscuous mode.

In the following exercise we will install Kubernetes on a single node then grow the cluster, adding more compute resources. Both nodes used are the same size, providing 2 vCPUs and 7.5G of memory. Smaller nodes could be used, but would run slower, and may have strange errors.



YAML files and White Space

Various exercises will use YAML files, which are included in the text. You are encouraged to write some of the files as time permits, as the syntax of YAML has white space indentation requirements that are important to learn. An important note, **do not** use tabs in your YAML files, **white space only. Indentation matters.**

If using a PDF the use of copy and paste often does not paste the single quote correctly. It pastes as a back-quote instead. You will need to modify it by hand. The files mentioned in labs have also been made available as a compressed **tar** file. You can view the resources by navigating to this URL:

<https://cm.lf.training/LFS458>

To login use user: LFtraining and a password of: Penguin2014

Once you find the name and link of the current file, which will change as the course updates, use **wget** to download the file into your node from the command line then expand it like this:

```
$ wget https://cm.lf.training/LFS458/LFS458_V1.30.1_SOLUTIONS.tar.xz \
--user=LFtraining --password=Penguin2014
```

```
$ tar -xvf LFS458_V1.30.1_SOLUTIONS.tar.xz
```

(Note: depending on your PDF viewer, if you are cutting and pasting the above instructions, the underscores may disappear and be replaced by spaces, so you may have to edit the command line by hand!)

Install Kubernetes

Log into your control plane (cp) and worker nodes. If attending in-person instructor led training the node IP addresses will be provided by the instructor. You will need to use a `.pem` or `.ppk` key for access, depending on if you are using `ssh` from a terminal or **PuTTY**. The instructor will provide this to you.

1. Open a terminal session on your first node. For example, connect via **PuTTY** or **SSH** session to the first **GCP** node. The user name may be different than the one shown, `student`. Create a non-root user if one is not present. The IP used in the example will be different than the one you will use. You may need to adjust the access mode of your pem or ppk key. The example shows how a Mac or Linux system would change mode. Windows may have a similar process.

```
[student@laptop ~]$ chmod 400 LFS458.pem
[student@laptop ~]$ ssh -i LFS458.pem student@35.226.100.87
```

```
The authenticity of host '54.214.214.156 (35.226.100.87)' can't be established.
ECDSA key fingerprint is SHA256:IPvznbkx93/Wc+ACwXrCcDDgvBwmvEXC9vmYhk2Wo1E.
ECDSA key fingerprint is MD5:d8:c9:4b:b0:b0:82:d3:95:08:08:4a:74:1b:f6:e1:9f.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '35.226.100.87' (ECDSA) to the list of known hosts.
<output_omitted>
```

2. Use the `wget` command above to download and extract the course tarball to your node. Again copy and paste won't always paste the underscore characters.
3. You are encouraged to type out commands, if using a PDF or eLearning, instead of copy and paste. By typing the commands you have a better chance to remember both the command and the concept. There are a few exceptions, such as when a long hash or output is much easier to copy over, and does not offer a learning opportunity.
4. Become root and update and upgrade the system. You may be asked a few questions. If so, allow restarts and keep the local version currently installed. Which would be a yes then a 2.

```
student@cp:~$ sudo -i
```

```
root@cp:~# apt-get update && apt-get upgrade -y
```

```
<output_omitted>

You can choose this option to avoid being prompted; instead,
all necessary restarts will be done for you automatically
so you can avoid being asked questions on each library upgrade.
```

```
Restart services during package upgrades without asking? [yes/no] yes
```

```
<output_omitted>
```

```
A new version (/tmp/fileEbke6q) of configuration file /etc/ssh/sshd_config is
available, but the version installed currently has been locally modified.
```

1. install the package maintainer's version
2. keep the local version currently installed
3. show the differences between the versions
4. show a side-by-side difference between the versions
5. show a 3-way difference between available versions
6. do a 3-way merge between available versions
7. start a new shell to examine the situation

What do you want to do about modified configuration file `sshd_config`? 2

```
<output_omitted>
```

5. Install a text editor like **nano** (an easy to use editor), **vim**, or **emacs**. Any will do, the labs use a popular option, **vim**.

```
root@cp:~# apt-get install -y vim
```

```
<output-omitted>
```

6. The main choices for a container environment are **containerd**, **cri-o**, and **Docker** on older clusters. We suggest **containerd** for class, as it is easy to deploy and commonly used by cloud providers.

Please note, install one engine only. If more than one are installed the **kubeadm** init process search pattern will use Docker at the moment. Also be aware that engines other than **containerd** may show different output on some commands.

7. There are several packages we should install to ensure we have all dependencies take care of. Please note the backslash is not necessary and can be removed if typing on a single line.

```
root@cp:~# apt install curl apt-transport-https git wget \
software-properties-common lsb-release ca-certificates -y
```

```
<output-omitted>
```

8. Disable swap if not already done. Cloud providers disable swap on their images.

```
root@cp:~# swapoff -a
```

9. Load modules to ensure they are available for following steps.

```
root@cp:~# modprobe overlay
root@cp:~# modprobe br_netfilter
```

10. Update kernel networking to allow necessary traffic. Be aware the shell will add a greater than sign (>) to indicate the command continues after a carriage return.

```
root@cp:~# cat << EOF | tee /etc/sysctl.d/kubernetes.conf
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
net.ipv4.ip_forward = 1
EOF
```

```
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
net.ipv4.ip_forward = 1
```

11. Ensure the changes are used by the current kernel as well

```
root@cp:~# sysctl --system
* Applying /etc/sysctl.d/10-console-messages.conf ...
kernel.printk = 4 4 1 7
* Applying /etc/sysctl.d/10-ipv6-privacy.conf ...
net.ipv6.conf.all.use_tempaddr = 2
net.ipv6.conf.default.use_tempaddr = 2
* Applying /etc/sysctl.d/10-kernel-hardening.conf ...
kernel.kptr_restrict = 1
<output_omitted>
```

12. Install the necessary key for the software to install

```
root@cp:~# mkdir -p /etc/apt/keyrings
root@cp:~# curl -fsSL https://download.docker.com/linux/ubuntu/gpg \
| sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg

root@cp:~# echo \
"deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg] \
https://download.docker.com/linux/ubuntu \
$(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

13. Install the containerd software.

```
root@cp:~# apt-get update && apt-get install containerd.io -y
root@cp:~# containerd config default | tee /etc/containerd/config.toml
root@cp:~# sed -e 's/SystemdCgroup = false/SystemdCgroup = true/g' -i /etc/containerd/config.toml
root@cp:~# systemctl restart containerd
```

```
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following packages were automatically installed and are no longer required:
<output_omitted>
```

14. Download the public signing key for the Kubernetes package repositories

```
root@cp:~# mkdir -p -m 755 /etc/apt/keyrings
root@cp:~# curl -fsSL https://pkgs.k8s.io/core:/stable:/v1.29/deb/Release.key \
| sudo gpg --dearmor -o /etc/apt/keyrings/kubernetes-apt-keyring.gpg
```

15. Add the appropriate Kubernetes apt repository. Please note that this repository have packages only for Kubernetes 1.29; for other Kubernetes minor versions, you need to change the Kubernetes minor version in the URL to match your desired minor version

```
root@cp:~# echo "deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg] \
https://pkgs.k8s.io/core:/stable:/v1.29/deb/ /" \
| sudo tee /etc/apt/sources.list.d/kubernetes.list
```

```
deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg]
→ https://pkgs.k8s.io/core:/stable:/v1.29/deb/ /
```

16. Update with the new repo declared, which will download updated repo information.

```
root@cp:~# apt-get update
```

```
<output-omitted>
```

17. Install the Kubernetes software. There are regular releases, the newest of which can be used by omitting the equal sign and version information on the command line. Historically new versions have lots of changes and a good chance of a bug or five. As a result we will hold the software at the recent but stable version we install. In a later lab we will update the cluster to a newer version.

```
root@cp:~# apt-get install -y kubeadm=1.29.1-1.1 kubelet=1.29.1-1.1 kubectl=1.29.1-1.1
```

```
<output-omitted>
```

```
root@cp:~# apt-mark hold kubelet kubeadm kubectl
```

```
kubelet set on hold.  
kubeadm set on hold.  
kubectl set on hold.
```

18. Find the IP address of the primary interface of the cp server. The example below would be the ens4 interface and an IP of 10.128.0.3, yours may be different. There are two ways of looking at your IP addresses.

```
root@cp:~# hostname -i
```

```
10.128.0.3
```

```
root@cp:~# ip addr show
```

```
....  
2: ens4: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1460 qdisc mq state UP group default qlen 1000  
    link/ether 42:01:0a:80:00:18 brd ff:ff:ff:ff:ff:ff  
    inet 10.128.0.3/32 brd 10.128.0.3 scope global ens4  
        valid_lft forever preferred_lft forever  
        inet6 fe80::4001:aff:fe80:18/64 scope link  
            valid_lft forever preferred_lft forever  
....
```

19. Add an local DNS alias for our cp server. Edit the `/etc/hosts` file and add the above IP address and assign a name k8scp.

```
root@cp:~# vim /etc/hosts
```

```
10.128.0.3 k8scp    #<-- Add this line  
127.0.0.1 localhost  
....
```

20. Create a configuration file for the cluster. There are many options we could include, and they differ for **containerd**, **Docker**, and **cri-o**. Use the file included in the course tarball. After our cluster is initialized we will view other default values used. Be sure to use the node alias we added to `/etc/hosts`, not the IP so the network certificates will continue to work when we deploy a load balancer in a future lab. The file is also in the course tarball.

```
root@cp:~# cp /home/student/LFS458/SOLUTIONS/s_03/kubeadm-config.yaml /root/
```

```
root@cp:~# vim kubeadm-config.yaml
```



kubeadm-config.yaml

```

1 apiVersion: kubeadm.k8s.io/v1beta3
2 kind: ClusterConfiguration
3 kubernetesVersion: 1.29.1
4 controlPlaneEndpoint: "k8scp:6443"           #<-- Use the word stable for newest version
5 networking:                                #<-- Use the alias we put in /etc/hosts not the IP
6   podSubnet: 192.168.0.0/16
7

```

21. Initialize the cp. Scan through the output. Expect the output to change as the software matures. At the end are configuration directions to run as a non-root user. The token is mentioned as well. This information can be found later with the **kubeadm token list** command. The output also directs you to create a pod network to the cluster, which will be our next step. Pass the network settings **Cilium** has in its configuration file. **Please note:** the output lists several commands which following exercise steps will complete.

```
root@cp:~# kubeadm init --config=kubeadm-config.yaml --upload-certs \
| tee kubeadm-init.out                         #<-- Save output for future review
```

```
[init] Using Kubernetes version: v1.29.1
[preflight] Running pre-flight checks
```

```
<output_omitted>
```

You can now join any number of the control-plane node running the following command on each as root:

```
kubeadm join k8scp:6443 --token vapzqi.et2p9zbkzk29wwth \
--discovery-token-ca-cert-hash
→ sha256:f62bf97d4fba6876e4c3ff645df3fcfa969c06169dee3865aab9d0bca8ec9f8cd \
--control-plane --certificate-key
→ 911d41fcada89a18210489afaa036cd8e192b1f122ebb1b79cce1818f642fab8
```

Please note that the certificate-key gives access to cluster sensitive data, keep it secret!

As a safeguard, uploaded-certs will be deleted in two hours; If necessary, you can use

"`kubeadm init phase upload-certs --upload-certs`" to reload certs afterward.

Then you can join any number of worker nodes by running the following on each as root:

```
kubeadm join k8scp:6443 --token vapzqi.et2p9zbkzk29wwth \
--discovery-token-ca-cert-hash
→ sha256:f62bf97d4fba6876e4c3ff645df3fcfa969c06169dee3865aab9d0bca8ec9f8cd
```

22. As suggested in the directions at the end of the previous output we will allow a non-root user admin level access to the cluster. Take a quick look at the configuration file once it has been copied and the permissions fixed.

```
root@cp:~# exit
```

```
logout
```

```
student@cp:~$ mkdir -p $HOME/.kube
```

```
student@cp:~$ sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
```

```
student@cp:~$ sudo chown $(id -u):$(id -g) $HOME/.kube/config
student@cp:~$ less .kube/config
```

```
apiVersion: v1
clusters:
- cluster:
<output_omitted>
```

23. Deciding which pod network to use for Container Networking Interface (**CNI**) should take into account the expected demands on the cluster. There can be only one pod network per cluster, although the **CNI-Genie** project is trying to change this.

The network must allow container-to-container, pod-to-pod, pod-to-service, and external-to-service communications. We will use **Cilium** as a network plugin which will allow us to use Network Policies later in the course. Currently **Cilium** does not deploy using CNI by default.

Cilium is generally installed using "cilium install" or using "helm install" commands. We have generated the cilium-cni.yaml file using the below commands for your convenience. **Note:** You don't need to execute the commands in this box, they are just for reference.

```
$ helm repo add cilium https://helm.cilium.io/
$ helm repo update
$ helm template cilium cilium/cilium --version 1.14.1 \
--namespace kube-system > cilium.yaml
```

```
student@cp:~$ find $HOME -name cilium-cni.yaml
student@cp:~$ kubectl apply -f /home/student/LFS458/SOLUTIONS/s_03/cilium-cni.yaml
serviceaccount/cilium created
serviceaccount/cilium-operator created
secret/cilium-ca created
configmap/cilium-config created
<output_omitted>
```

24. While many objects have short names, a **kubectl** command can be a lot to type. We will enable **bash** auto-completion. Begin by adding the settings to the current shell. Then update the **\$HOME/.bashrc** file to make it persistent. Ensure the **bash-completion** package is installed. If it was not installed, log out then back in for the shell completion to work.

```
student@cp:~$ sudo apt-get install bash-completion -y
<exit and log back in>
student@cp:~$ source <(kubectl completion bash)
student@cp:~$ echo "source <(kubectl completion bash)" >> $HOME/.bashrc
```

25. Test by describing the node again. Type the first three letters of the sub-command then type the **Tab** key. Auto-completion assumes the default namespace. Pass the namespace first to use auto-completion with a different namespace. By pressing **Tab** multiple times you will see a list of possible values. Continue typing until a unique name is used. First look at the current node (your node name may not start with cp), then look at pods in the **kube-system** namespace. If you see an error instead such as `-bash: _get_comp_words_by_ref: command not found` revisit the previous step, install the software, log out and back in.

```
student@cp:~$ kubectl des<Tab> n<Tab><Tab> cp<Tab>
student@cp:~$ kubectl -n kube-s<Tab> g<Tab> po<Tab>
```

26. Explore the **kubectl help** command. The output has been omitted from commands. Take a moment to review help topics.

```
student@cp:~$ kubectl help
student@cp:~$ kubectl help create
```

27. View other values we could have included in the `kubeadm-config.yaml` file when creating the cluster.

```
student@cp:~$ sudo kubeadm config print init-defaults
```

```
apiVersion: kubeadm.k8s.io/v1beta3
bootstrapTokens:
- groups:
  - system:bootstrappers:kubeadm:default-node-token
    token: abcdef.0123456789abcdef
    ttl: 24h0m0s
    usages:
    - signing
    - authentication
  kind: InitConfiguration
<output_omitted>
```

Exercise 3.2: Grow the Cluster

Open another terminal and connect into your second node. Install **containerd** and Kubernetes software. These are the many, but not all, of the steps we did on the cp node.

This book will use the **worker** prompt for the node being added to help keep track of the proper node for each command. Note that the prompt indicates both the user and system upon which run the command. It can be helpful to change the colors and fonts of your terminal session to keep track of the correct node.

1. Using the same process as before connect to a second node. If attending an instructor-led class session, use the same .pem key and a new IP provided by the instructor to access the new node. Giving a different title or color to the new terminal window is probably a good idea to keep track of the two systems. The prompts can look very similar.

2. `student@worker:~$ sudo -i`

3. `root@worker:~# apt-get update && apt-get upgrade -y`

<If asked allow services to restart and keep the local version of software>

4. Install the containerd engine, starting with dependent software.

```
root@worker:~# apt install curl apt-transport-https vim git wget \
software-properties-common lsb-release ca-certificates -y

root@worker:~# swapoff -a

root@worker:~# modprobe overlay

root@worker:~# modprobe br_netfilter
```

```

root@worker:~# cat << EOF | tee /etc/sysctl.d/kubernetes.conf
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
net.ipv4.ip_forward = 1
EOF

root@worker:~# sysctl --system

root@worker:~# mkdir -p /etc/apt/keyrings
root@worker:~# curl -fsSL https://download.docker.com/linux/ubuntu/gpg \
| sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg

root@worker:~# echo \
"deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg] \
https://download.docker.com/linux/ubuntu \
$(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

root@worker:~# apt-get update && apt-get install containerd.io -y
root@worker:~# containerd config default | tee /etc/containerd/config.toml
root@worker:~# sed -e 's/SystemdCgroup = false/SystemdCgroup = true/g' -i /etc/containerd/config.toml
root@worker:~# systemctl restart containerd

```

5. Add Kubernetes repo

```

root@worker:~# curl -fsSL https://pkgs.k8s.io/core:/stable:/v1.29/deb/Release.key \
| sudo gpg --dearmor -o /etc/apt/keyrings/kubernetes-apt-keyring.gpg

```

6. Get the GPG key for the software

```

root@worker:~# echo "deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg] \
https://pkgs.k8s.io/core:/stable:/v1.29/deb/ /" \
| sudo tee /etc/apt/sources.list.d/kubernetes.list

```

7. Update repos then install the Kubernetes software. Be sure to match the version on the cp.

```
root@worker:~# apt-get update
```

8. root@worker:~# apt-get install -y kubeadm=1.29.1-1.1 kubelet=1.29.1-1.1 kubectl=1.29.1-1.1

9. Ensure the version remains if the system is updated.

```
root@worker:~# apt-mark hold kubeadm kubelet kubectl
```

10. Find the IP address of your **cp** server. The interface name will be different depending on where the node is running. Currently inside of **GCE** the primary interface for this node type is **ens4**. Your interfaces names may be different. From the output we know our cp node IP is 10.128.0.3.

```
student@cp:~$ hostname -i
```

10.128.0.3

```
student@cp:~$ ip addr show ens4 | grep inet
```

```
inet 10.128.0.3/32 brd 10.128.0.3 scope global ens4
inet6 fe80::4001:aff:fe8e:2/64 scope link
```

11. At this point we could copy and paste the **join** command from the cp node. That command only works for 2 hours, so we will build our own **join** should we want to add nodes in the future. Find the token on the cp node. The token lasts 2 hours by default. If it has been longer, and no token is present you can generate a new one with the **sudo kubeadm token create** command, seen in the following command.

```
student@cp:~$ sudo kubeadm token create --print-join-command
```

```
kubeadm join k8scp:6443 --token kcu55w.7js085i0e2dsn05y \
--discovery-token-ca-cert-hash
↳ sha256:0fb62b3c47bfd3af3c15d21f2ab6082fad1f913b244d5980816f8147ce9936ef
```

12. On the **worker node** add a local DNS alias for the cp server. Edit the `/etc/hosts` file and add the cp IP address and assign the name k8scp. The entry should be exactly the same as the edit on the cp.

```
root@worker:~# vim /etc/hosts
```

```
10.128.0.3 k8scp    #<-- Add this line
127.0.0.1 localhost
....
```

13. Use the token and hash, in this case as sha256:long-hash to join the cluster from the **second/worker** node. Use the **private** IP address of the cp server and port 6443. The output of the **kubeadm init** on the cp also has an example to use, should it still be available.

```
root@worker:~# kubeadm join \
k8scp:6443 --token wcal99.hxv9v0gtnz42g6dr \
--discovery-token-ca-cert-hash \
sha256:0fb62b3c47bfd3af3c15d21f2ab6082fad1f913b244d5980816f8147ce9936ef
```

```
[preflight] Running pre-flight checks
```

```
[preflight] Reading configuration from the cluster...
[preflight] FYI: You can look at this config file with 'kubectl -n kube-system get cm
↳ kubeadm-config -oyaml'
[kubelet-start] Writing kubelet configuration to file "/var/lib/kubelet/config.yaml"
[kubelet-start] Writing kubelet environment file with flags to file
↳ "/var/lib/kubelet/kubeadm-flags.env"
[kubelet-start] Activating the kubelet service
[kubelet-start] Waiting for the kubelet to perform the TLS Bootstrap...
```

This node has joined the cluster:

- * Certificate signing request was sent to apiserver and a response was received.
- * The Kubelet was informed of the new secure connection details.

Run 'kubectl get nodes' on the control-plane to see this node join the cluster.

14. Try to run the **kubectl** command on the secondary system. It should fail. You do not have the cluster or authentication keys in your local `.kube/config` file.

```
root@worker:~# exit
```

```
student@worker:~$ kubectl get nodes
```

```
The connection to the server localhost:8080 was refused - did you specify the right host or port?
```

```
student@worker:~$ ls -l .kube
```

```
ls: cannot access '.kube': No such file or directory
```

Exercise 3.3: Finish Cluster Setup

- View the available nodes of the cluster. It can take a minute or two for the status to change from NotReady to Ready. The NAME field can be used to look at the details. Your node name may be different, use YOUR control-plane name in future commands, if different than the book.

```
student@cp:~$ kubectl get node
```

NAME	STATUS	ROLES	AGE	VERSION
cp	Ready	control-plane	28m	v1.29.1
worker	Ready	<none>	50s	v1.29.1

- Look at the details of the node. Work line by line to view the resources and their current status. Notice the status of Taints. The cp won't allow non-infrastructure pods by default for security and resource contention reasons. Take a moment to read each line of output, some appear to be an error until you notice the status shows False.

```
student@cp:~$ kubectl describe node cp
```

Name:	cp
Roles:	control-plane
Labels:	beta.kubernetes.io/arch=amd64 beta.kubernetes.io/os=linux kubernetes.io/arch=amd64 kubernetes.io/hostname=cp kubernetes.io/os=linux node-role.kubernetes.io/control-plane= node.kubernetes.io/exclude-from-external-load-balancers=
Annotations:	kubeadm.alpha.kubernetes.io/cri-socket: → unix:///var/run/containerd/containerd.sock node.alpha.kubernetes.io/ttl: 0 volumes.kubernetes.io/controller-managed-attach-detach: true
CreationTimestamp:	Mon, 19 Jun 2024 15:00:23 +0000
Taints:	node-role.kubernetes.io/control-plane:NoSchedule <output_omitted>

- Allow the cp server to run non-infrastructure pods. The cp node begins tainted for security and performance reasons. We will allow usage of the node in the training environment, but this step may be skipped in a production environment. Note the **minus sign (-)** at the end, which is the syntax to remove a taint. As the second node does not have the taint you will get a not found error. There may be more than one taint. Keep checking and removing them until all are removed.

```
student@cp:~$ kubectl describe node | grep -i taint
```

Taints:	node-role.kubernetes.io/control-plane:NoSchedule
Taints:	<none>

```
student@cp:~$ kubectl taint nodes --all node-role.kubernetes.io/control-plane-
```

node/cp untainted
error: taint "node-role.kubernetes.io/control-plane" not found

```
student@cp:~$ kubectl describe node | grep -i taint
```

Taints:	<none>
Taints:	<none>

- Determine if the DNS and Cilium pods are ready for use. They should all show a status of Running. It may take a minute or two to transition from Pending.

```
student@cp:~$ kubectl get pods --all-namespaces
```

NAMESPACE	NAME	READY	STATUS	RESTARTS	AGE
kube-system	cilium-operator-788c7d7585-tnsph	1/1	Running	0	95m
kube-system	cilium-swjsj	1/1	Running	0	95m
kube-system	coredns-5d78c9869d-dwds8	1/1	Running	0	100m
kube-system	coredns-5d78c9869d-t24p5	1/1	Running	0	100m

<output_omitted>

5. Only if you notice the coredns- pods are stuck in ContainerCreating status you may have to delete them, causing new ones to be generated. Delete both pods and check to see they show a Running state. Your pod names will be different.

```
student@cp:~$ kubectl get pods --all-namespaces
```

NAMESPACE	NAME	READY	STATUS	RESTARTS	AGE
kube-system	cilium-swjsj	2/2	Running	0	12m
kube-system	coredns-576cbf47c7-rn6v4	0/1	ContainerCreating	0	3s
kube-system	coredns-576cbf47c7-vq5dz	0/1	ContainerCreating	0	94m

<output_omitted>

```
student@cp:~$ kubectl -n kube-system delete \
pod coredns-576cbf47c7-vq5dz coredns-576cbf47c7-rn6v4
```

```
pod "coredns-576cbf47c7-vq5dz" deleted
pod "coredns-576cbf47c7-rn6v4" deleted
```

6. When it finished you should see more interfaces will be created . It may take up to a minute to be created. You will notice interfaces such as cilium interfaces when you deploy pods, as shown in the output below.

```
student@cp:~$ ip a
```

```
<output_omitted>
3: cilium_net@cilium_host: <BROADCAST,MULTICAST,NOARP,UP,LOWER_UP> mtu 1460 qdisc noqueue state UP
    ↓  group default qlen 1000
        link/ether be:19:22:da:62:ac brd ff:ff:ff:ff:ff:ff
        inet6 fe80::bc19:22ff:fed:a62ac/64 scope link
            valid_lft forever preferred_lft forever
5: cilium_vxlan: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1460 qdisc noqueue state UNKNOWN group
    ↓  default qlen 1000
        link/ether ca:22:e7:23:42:89 brd ff:ff:ff:ff:ff:ff
        inet6 fe80::c822:e7ff:fe23:4289/64 scope link
            valid_lft forever preferred_lft forever
<output_omitted>
```

7. Containerd may still be using an out of date notation for the runtime-endpoint. You may see errors about an undeclared resource type such as unix://. We will update the **crictl** configuration. There are many possible configuration options. We will set one, and view the configuration file that is created. We will also set this configuration on worker node as well for our convenience.

```
student@cp:~$ sudo crictl config --set \
runtime-endpoint=unix:///run/containerd/containerd.sock \
--set image-endpoint=unix:///run/containerd/containerd.sock
```

```
student@worker:~$ sudo crictl config --set \
```

```
runtime-endpoint=unix:///run/containerd/containerd.sock \
--set image-endpoint=unix:///run/containerd/containerd.sock

student@cp:~$ sudo cat /etc/crictl.yaml
```

```
runtime-endpoint: "unix:///run/containerd/containerd.sock"
image-endpoint: "unix:///run/containerd/containerd.sock"
timeout: 0
debug: false
pull-image-on-create: false
disable-pull-on-run: false
```

📝 Exercise 3.4: Deploy A Simple Application

We will test to see if we can deploy a simple application, in this case the **nginx** web server.

1. Create a new deployment, which is a Kubernetes object, which will deploy an application in a container. Verify it is running and the desired number of containers matches the available.

```
student@cp:~$ kubectl create deployment nginx --image=nginx
```

```
deployment.apps/nginx created
```

```
student@cp:~$ kubectl get deployments
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
nginx	1/1	1	1	8s

2. View the details of the deployment. Remember auto-completion will work for sub-commands and resources as well.

```
student@cp:~$ kubectl describe deployment nginx
```

```
Name: nginx
Namespace: default
CreationTimestamp: Wed, 23 Jun 2024 22:38:32 +0000
Labels: app=nginx
Annotations: deployment.kubernetes.io/revision: 1
Selector: app=nginx
Replicas: 1 desired | 1 updated | 1 total | 1 ava....
StrategyType: RollingUpdate
MinReadySeconds: 0
RollingUpdateStrategy: 25% max unavailable, 25% max surge
<output_omitted>
```

3. View the basic steps the cluster took in order to pull and deploy the new application. You should see several lines of output. The first column shows the age of each message, note that due to JSON lack of order the time LAST SEEN time does not print out chronologically. Eventually older messages will be removed.

```
student@cp:~$ kubectl get events
```

```
<output_omitted>
```

4. You can also view the output in **yaml** format, which could be used to create this deployment again or new deployments. Get the information but change the output to yaml. Note that halfway down there is status information of the current deployment.

```
student@cp:~$ kubectl get deployment nginx -o yaml
```

YAML

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   annotations:
5     deployment.kubernetes.io/revision: "1"
6   creationTimestamp: 2024-06-24T18:21:25Z
7 <output_omitted>
8
```

- Run the command again and redirect the output to a file. Then edit the file. Remove the creationTimestamp, resourceVersion, and uid lines. Also remove all the lines including and after status:, which should be somewhere around line 120, if others have already been removed.

```
student@cp:~$ kubectl get deployment nginx -o yaml > first.yaml
```

```
student@cp:~$ vim first.yaml
```

<Remove the lines mentioned above>

- Delete the existing deployment.

```
student@cp:~$ kubectl delete deployment nginx
```

deployment.apps "nginx" deleted

- Create the deployment again this time using the file.

```
student@cp:~$ kubectl create -f first.yaml
```

deployment.apps/nginx created

- Look at the yaml output of this iteration and compare it against the first. The creation time stamp, resource version and unique ID we had deleted are in the new file. These are generated for each resource we create, so we may need to delete them from yaml files to avoid conflicts or false information. You may notice some time stamp differences as well. The status should not be hard-coded either.

```
student@cp:~$ kubectl get deployment nginx -o yaml > second.yaml
```

```
student@cp:~$ diff first.yaml second.yaml
```

<output_omitted>

- Now that we have worked with the raw output we will explore two other ways of generating useful YAML or JSON. Use the --dry-run option and verify no object was created. Only the prior nginx deployment should be found. The output lacks the unique information we removed before, but does have the same essential values.

```
student@cp:~$ kubectl create deployment two --image=nginx --dry-run=client -o yaml
```

**Y
A
M
L**

```

1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   creationTimestamp: null
5   labels:
6     app: two
7     name: two
8 spec:
9 <output_omitted>
10

```

student@cp:~\$ kubectl get deployment

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
nginx	1/1	1	1	7m

- Existing objects can be viewed in a ready to use YAML output. Take a look at the existing **nginx** deployment.

student@cp:~\$ kubectl get deployments nginx -o yaml

**Y
A
M
L**

```

1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   annotations:
5     deployment.kubernetes.io/revision: "1"
6   creationTimestamp: null
7   generation: 1
8   labels:
9     run: nginx
10 <output_omitted>
11

```

- The output can also be viewed in JSON output.

student@cp:~\$ kubectl get deployment nginx -o json

JSON

```

1 {
2   "apiVersion": "apps/v1",
3   "kind": "Deployment",
4   "metadata": {
5     "annotations": {
6       "deployment.kubernetes.io/revision": "1"
7     },
8   <output_omitted>
9

```

- The newly deployed **nginx** container is a light weight web server. We will need to create a service to view the default welcome page. Begin by looking at the help output. Note that there are several examples given, about halfway through the output.

student@cp:~\$ kubectl expose -h

<output_omitted>

13. Now try to gain access to the web server. As we have not declared a port to use you will receive an error.

```
student@cp:~$ kubectl expose deployment/nginx
error: couldn't find port via --port flag or introspection
See 'kubectl expose -h' for help and examples.
```

14. To change an object configuration one can use subcommands apply, edit or patch for non-disruptive updates. The apply command does a three-way diff of previous, current, and supplied input to determine modifications to make. Fields not mentioned are unaffected. The edit function performs a get, opens an editor, then an apply. You can update API objects in place with JSON patch and merge patch or strategic merge patch functionality.

If the configuration has resource fields which cannot be updated once initialized then a disruptive update could be done using the replace --force option. This deletes first then re-creates a resource.

Edit the file. Find the container name, somewhere around line 31 and add the port information as shown below.

```
student@cp:~$ vim first.yaml
```



```
YAML
first.yaml

1  ....
2  spec:
3    containers:
4      - image: nginx
5        imagePullPolicy: Always
6        name: nginx
7        ports:          # Add these
8          - containerPort: 80      # three
9            protocol: TCP        # lines
10           resources: {}
```

15. Due to how the object was created we will need to use replace to terminate and create a new deployment.

```
student@cp:~$ kubectl replace -f first.yaml --force
deployment.apps/nginx replaced
```

16. View the Pod and Deployment. Note the AGE shows the Pod was re-created.

```
student@cp:~$ kubectl get deploy,pod
NAME                      READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/nginx     1/1     1           1           2m4s
NAME                      READY   STATUS      RESTARTS   AGE
pod/nginx-7db75b8b78-qjffm 1/1     Running    0          8s
```

17. Try to expose the resource again. This time it should work.

```
student@cp:~$ kubectl expose deployment/nginx
service/nginx exposed
```

18. Verify the service configuration. First look at the service, then the endpoint information. Note the ClusterIP is not the current endpoint. Cilium provides the ClusterIP. The Endpoint is provided by kubelet and kube-proxy. Take note of the current endpoint IP. In the example below it is 192.168.1.5:80. We will use this information in a few steps.

```
student@cp:~$ kubectl get svc nginx
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
nginx	ClusterIP	10.100.61.122	<none>	80/TCP	3m

```
student@cp:~$ kubectl get ep nginx
```

NAME	ENDPOINTS	AGE
nginx	192.168.1.5:80	26s

19. Determine which node the container is running on. Log into that node and use **tcpdump**, which you may need to install using **apt-get install**, to view traffic on the tun0, as in tunnel zero, interface. The second node in this example. You may also see traffic on an interface which starts with **cili** and some string. Leave that command running while you run **curl** in the following step. You should see several messages go back and forth, including a HTTP HTTP/1.1 200 OK: and a ack response to the same sequence.

```
student@cp:~$ kubectl describe pod nginx-7cbc4b4d9c-d27xw \
    | grep Node:
```

```
Node: worker/10.128.0.5
```

```
student@worker:~$ sudo tcpdump -i cilium_vxlan
```

```
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on cilium_vxlan, link-type EN10MB (Ethernet), capture size 262144 bytes
<output_omitted>
```

20. Test access to the Cluster IP, port 80. You should see the generic nginx installed and working page. The output should be the same when you look at the ENDPOINTS IP address. If the **curl** command times out the pod may be running on the other node. Run the same command on that node and it should work.

```
student@cp:~$ curl 10.100.61.122:80
```

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
<output_omitted>
```

```
student@cp:~$ curl 192.168.1.5:80
```

21. Now scale up the deployment from one to three web servers.

```
student@cp:~$ kubectl get deployment nginx
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
nginx	1/1	1	1	12m

```
student@cp:~$ kubectl scale deployment nginx --replicas=3
```

```
deployment.apps/nginx scaled
```

```
student@cp:~$ kubectl get deployment nginx
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
nginx	3/3	3	3	12m

22. View the current endpoints. There now should be three. If the UP-TO-DATE above said three, but AVAILABLE said two wait a few seconds and try again, it could be slow to fully deploy.

```
student@cp:~$ kubectl get ep nginx
```

NAME	ENDPOINTS	AGE
nginx	192.168.0.3:80,192.168.1.5:80,192.168.1.6:80	7m40s

23. Find the oldest pod of the **nginx** deployment and delete it. The Tab key can be helpful for the long names. Use the AGE field to determine which was running the longest. You may notice activity in the other terminal where **tcpdump** is running, when you delete the pod. The pods with 192.168.0 addresses are probably on the cp and the 192.168.1 addresses are probably on the worker

```
student@cp:~$ kubectl get pod -o wide
```

NAME	READY	STATUS	RESTARTS	AGE	IP
nginx-1423793266-7f1qw	1/1	Running	0	14m	192.168.1.5
nginx-1423793266-8w2nk	1/1	Running	0	86s	192.168.1.6
nginx-1423793266-fbt4b	1/1	Running	0	86s	192.168.0.3

```
student@cp:~$ kubectl delete pod nginx-1423793266-7f1qw
```

```
pod "nginx-1423793266-7f1qw" deleted
```

24. Wait a minute or two then view the pods again. One should be newer than the others. In the following example nine seconds instead of four minutes. If your **tcpdump** was using the veth interface of that container it will error out. Also note we are using a short name for the object.

```
student@cp:~$ kubectl get po
```

NAME	READY	STATUS	RESTARTS	AGE
nginx-1423793266-13p69	1/1	Running	0	9s
nginx-1423793266-8w2nk	1/1	Running	0	4m1s
nginx-1423793266-fbt4b	1/1	Running	0	4m1s

25. View the endpoints again. The original endpoint IP is no longer in use. You can delete any of the pods and the service will forward traffic to the existing backend pods.

```
student@cp:~$ kubectl get ep nginx
```

NAME	ENDPOINTS	AGE
nginx	192.168.0.3:80,192.168.1.6:80,192.168.1.7:80	12m

26. Test access to the web server again, using the ClusterIP address, then any of the endpoint IP addresses. Even though the endpoints have changed you still have access to the web server. This access is only from within the cluster. When done use **ctrl-c** to stop the **tcpdump** command.

```
student@cp:~$ curl 10.100.61.122:80
```

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body
<output_omitted>
```

Exercise 3.5: Access from Outside the Cluster

You can access a Service from outside the cluster using a DNS add-on or environment variables. We will use environment variables to gain access to a Pod.

1. Begin by getting a list of the pods.

```
student@cp:~$ kubectl get po
```

NAME	READY	STATUS	RESTARTS	AGE
nginx-1423793266-13p69	1/1	Running	0	4m10s
nginx-1423793266-8w2nk	1/1	Running	0	8m2s
nginx-1423793266-fbt4b	1/1	Running	0	8m2s

2. Choose one of the pods and use the exec command to run **printenv** inside the pod. The following example uses the first pod listed above.

```
student@cp:~$ kubectl exec nginx-1423793266-13p69 \
-- printenv |grep KUBERNETES
```

```
KUBERNETES_SERVICE_PORT=443
KUBERNETES_SERVICE_HOST=10.96.0.1
KUBERNETES_SERVICE_PORT_HTTPS=443
KUBERNETES_PORT=tcp://10.96.0.1:443
<output_omitted>
```

3. Find and then delete the existing service for **nginx**.

```
student@cp:~$ kubectl get svc
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP	4h
nginx	ClusterIP	10.100.61.122	<none>	80/TCP	17m

4. Delete the service.

```
student@cp:~$ kubectl delete svc nginx
```

```
service "nginx" deleted
```

5. Create the service again, but this time pass the LoadBalancer type. Check to see the status and note the external ports mentioned. The output will show the External-IP as pending. Unless a provider responds with a load balancer it will continue to show as pending.

```
student@cp:~$ kubectl expose deployment nginx --type=LoadBalancer
```

```
service/nginx exposed
```

```
student@cp:~$ kubectl get svc
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP	4h
nginx	LoadBalancer	10.104.249.102	<pending>	80:32753/TCP	6s

6. Open a browser on your local system, not the lab exercise node, and use the public IP of your node and node port 32753, shown in the output above. If running the labs on remote nodes like AWS or GCE use the public IP you used with PuTTY or SSH to gain access. You may be able to find the IP address using curl.

```
student@cp:~$ curl ifconfig.io
```

```
54.214.214.156
```



Figure 3.1: External Access via Browser

7. Scale the deployment to zero replicas. Then test the web page again. Once all pods have finished terminating accessing the web page should fail.

```
student@cp:~$ kubectl scale deployment nginx --replicas=0
```

```
deployment.apps/nginx scaled
```

```
student@cp:~$ kubectl get po
```

```
No resources found in default namespace.
```

8. Scale the deployment up to two replicas. The web page should work again.

```
student@cp:~$ kubectl scale deployment nginx --replicas=2
```

```
deployment.apps/nginx scaled
```

```
student@cp:~$ kubectl get po
```

NAME	READY	STATUS	RESTARTS	AGE
nginx-1423793266-7x181	1/1	Running	0	6s
nginx-1423793266-s6vcz	1/1	Running	0	6s

9. Delete the deployment to recover system resources. Note that deleting a deployment does not delete the endpoints or services.

```
student@cp:~$ kubectl delete deployments nginx
```

```
deployment.apps "nginx" deleted
```

```
student@cp:~$ kubectl delete svc nginx
```

```
service "nginx" deleted
```


Chapter 4

Kubernetes Architecture



4.1	Kubernetes Architecture	66
4.2	Networking	81
4.3	Other Cluster Systems	87
4.4	Labs	88

4.1 Kubernetes Architecture

Main Components

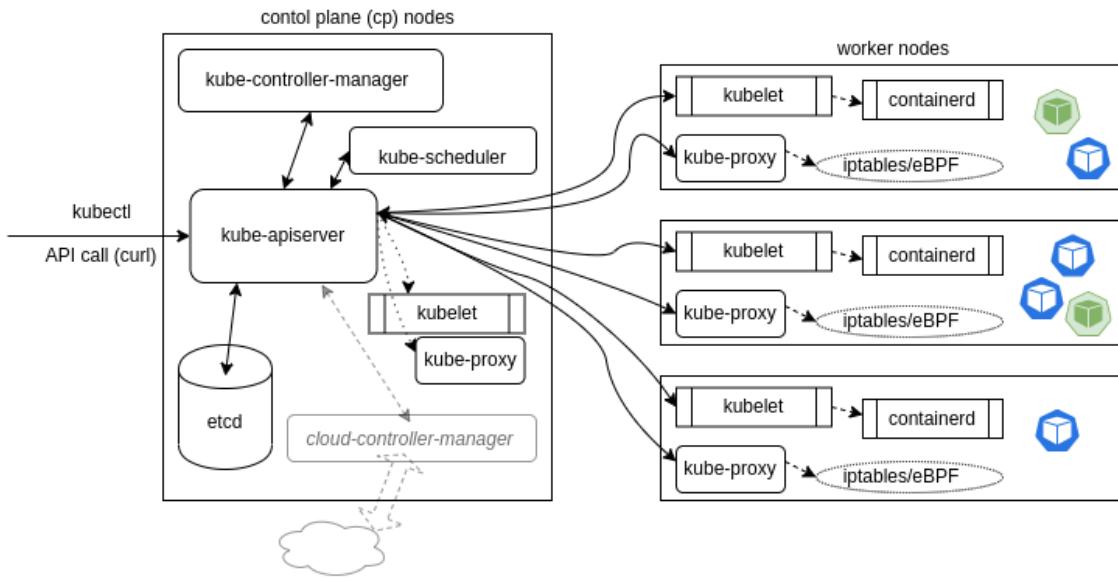


Figure 4.1: Architectural Overview

Main Components:

- Control Plane(s) and worker node(s)
- Operators
- Services
- Pods of Containers
- Namespaces and quotas
- Network and policies
- Storage

As mentioned in the previous chapter, a Kubernetes cluster is made of one or more cp node and a set of worker nodes. The cluster is all driven via API calls to operators. A network plugin helps handle both interior as well as exterior traffic. We will take a closer look at these components on following pages

Most of the processes are executed inside of a container. There are some differences depending on vendor and tool used to build the cluster.

When upgrading a cluster be aware that each of these components are developed to work together by multiple teams. Care should be taken ensure a proper match of versions. The **kubeadm upgrade plan** command is useful to discover this information.

Control Plane Node

- **kube-apiserver**
- **kube-scheduler**
- **etcd**
- **kube-controller-manager**
- **cloud-controller-manager**
- **CoreDNS**
- Network plugin
- Add-ons
 - Dashboard - Web UI
 - Cluster-level resource monitoring
 - cluster-level logging

The Kubernetes cp runs various server and manager processes for the cluster. As the software has matured, new components have been created to handle dedicated needs, such as the **cloud-controller-manager**; it handles tasks once handled by the **kube-controller-manager** to interact with other tools such as **Rancher** or **DigitalOcean** for third-party cluster management and reporting.

There are several add-ons which have become essential to a typical production cluster, such as DNS services. Others are third-party solutions where Kubernetes has not yet developed a local component such as cluster-level logging and monitoring.

As a concept the various pods responsible for ensuring the current state of the cluster matches the desired state are called the **control plane**.

When building a cluster using **kubeadm** the **kubelet** process is managed by **systemd**. Once running it will start every pod found in [/etc/kubernetes/manifests/](#).

kube-apiserver

- Front-end of cluster's shared state
- Control Plane for the cluster
- All components work through it
- Validates and configures data for API objects
- Services REST operations
- Only component to connect to **etcd** database

The **kube-apiserver** is central to the operation of the Kubernetes cluster.

All calls, both internal and external traffic are handled via this agent. All actions are accepted and validated by this agent and it is the only connection to the **etcd** database. As a result it acts as a cp process for the entire cluster, and acts as a front-end of the cluster's shared state.

Starting as an beta feature in v1.18 **Konnectivity service** provides the ability to separate user-initiated traffic from server-initiated traffic. Until these features are developed most network plugins commingle the traffic, which has performance, capacity, and security ramifications.

kube-scheduler

- Uses algorithm to determine Pod placement
- Checks quota restrictions
- Custom scheduling policies possible
- Affinity rules to place pods on specific nodes
- Taints can be used to repel pods
- Pod bindings can force particular scheduling

The **kube-scheduler** uses an algorithm to determine which node will host a Pod of containers. The scheduler will try to view available resources (such as volumes) to bind, and then try and retry to deploy the Pod based on availability and success.

There are several ways you can affect the algorithm, or a custom scheduler could be used instead. You can also bind a Pod to a particular node, though the Pod may remain in a pending state due to other settings.

One of the first settings referenced is if the Pod can be deployed within the current quota restrictions. If so, then the taints, tolerations, and labels of the Pods are used along with metadata of the nodes to determine the proper placement.

The details of the scheduler can be found at: <https://raw.githubusercontent.com/kubernetes/kubernetes/master/pkg/scheduler/scheduler.go>

etcd Database

- Multiversion persistent b+tree key-value store
- Append only, regular compaction
- Shed oldest version of superseded data
- Works with **curl** and other HTTP libraries
- Provides reliable watch queries
- Distributed consensus protocol for leadership

The state of the cluster, networking and other persistent information is kept in an **etcd** database, or more accurately a b+tree key-value store. Rather than find and change an entry, values are always appended to the end. Previous copies of the data are then marked for future removal by a compaction process.

Simultaneous requests to update a value all travel via the **kube-apiserver**, which then passes along the request to **etcd** in a series. The first request would update the database. The second request would no longer have the same version number, in which case the **kube-apiserver** would reply with an error 409 to the requester. There is no logic past that response on the server side, meaning the client needs to expect this and act upon the denial to update.

There is a Leader database along with possible Followers, or non-voting Learners who are in the process of joining the cluster. They communicate with each other on an ongoing basis to determine which will be the Leader and determine another in the event of failure. While very fast and potentially durable, there have been some hiccups with new tools such as **kubeadm** and features like whole cluster upgrades.

While most objects of Kubernetes are designed to be decoupled, transient microservice which can be terminated without much concern **etcd** is the exception. As it is the persistent state of the entire cluster it must be protected and secured. Before upgrades or maintenance you should plan on backing up **etcd**. The **etcdctl** command allows for `snapshot save` and `snapshot restore`.

Other Agents

- **kube-controller-manager**
 - Daemon which embeds core control-loops
 - Watches state of cluster
 - Works to make current state match desired state
- **cloud-controller-manager (ccm)**
 - Interacts with outside cloud managers
 - Allows features to be developed outside of core release cycle
 - Each **kubelet** must use --cloud-provider-external
 - Handles tasks once part of kube-controller-manager
 - This is an optional agent which takes a few steps to enable
- Network plugins
- **CoreDNS**

The **kube-controller-manager** is a core control loop daemon which interacts with the **kube-apiserver** to determine the state of the cluster. If the state does not match, the manager will contact the necessary controller to match the desired state. There are several controllers in use, such as endpoints, namespace, and replication. The full list has expanded as Kubernetes has matured.

Remaining in **beta** since v1.11 the **cloud-controller-manager** interacts with agents outside of the cloud. It handles tasks once handled by **kube-controller-manager**. This allows faster changes without altering the core kubernetes control process. Each **kubelet** must use the --cloud-provider-external settings passed to the binary. You can also develop your own CCM, which can be deployed as a daemonset as an in-tree deployment or as a free-standing out-of-tree installation. More can be found here: kubernetes.io/docs/tasks/administer-cluster/running-cloud-controller

Depending on which network plugin has been chosen there may be various pods to control network traffic. To handle DNS queries, Kubernetes service discovery, and other functions the **CoreDNS** server has replaced kube-dns. Using chains of plugins, one of many provided or custom written, the server is easily extensible.

Worker Nodes

- kubelet
- kube-proxy
- Docker engine, cri-o, containerd, etc.
- Network plugin pods
- Fluentd
- Prometheus

All nodes run the **kubelet** and **kube-proxy**, as well as the container engine such as **containerd** or **cri-o**, among several options. Other management daemons are deployed to watch these agents or provide services not yet included with Kubernetes.

The **kubelet** interacts with the underlying container engine also installed on all the nodes and makes sure that the containers that need to run are actually running. The **kube-proxy** is in charge of managing the network connectivity to the containers. It does so through the use of **iptables** entries. It also has **userspace** mode in which it monitors **Services** and **Endpoints** using a random port to proxy traffic via **ipvs**. A network plugin pod, such as **cilium-xxxxx**, may be found depending on the plugin in use.

Each node could run a different engine. It is likely that Kubernetes will support additional container runtime engines.

Kubernetes does not have cluster-wide logging yet. Instead, another **CNCF** project is used, called **Fluentd** (<http://www.fluentd.org>). When implemented it provides a unified logging layer for the cluster which filters, buffers and routes messages.

Cluster-wide metrics is another area with limited functionality the **metrics-server** SIG provides basic node and pod CPU and memory utilization. For more metrics many use the **Prometheus** project.

kubelet

- systemd process on each node
- Uses PodSpec
- Mounts volumes to Pod
- Downloads secrets
- Passes request to local container engine
- Reports status of Pods and node to cluster

The **kubelet** systemd process is the heavy-lifter for changes and configuration on worker nodes. It accepts the API calls for Pod specifications. It will work to configure the local node until the specification has been met.

Should a Pod require access to storage, secrets or ConfigMaps the **kubelet** will ensure access or creation. It also sends back status to the **kube-apiserver** for eventual persistence.

Kubelet calls other components such as the Topology Manager, which uses hints from other components to configure topology aware resource NUMA assignments such as for CPU and hardware accelerators. As an alpha feature it is not enabled by default.

Operators

- Watch based control loop monitoring delta
- Informer / SharedInformer
- Workqueue
- Shipped operators
 - Deployment operator
 - replicaSet operator
 - Service operator
 - endpoints operator
 - namespace operator
 - serviceaccounts operator

An important concept for orchestration is the use of operators, otherwise known as controllers or watch-loops. Various operators ship with Kubernetes, and you can create your own as well. A simplified view of a operator is an agent, or `Informer` and a downstream store. Using a `DeltaFIFO` queue, the source and downstream are compared. A loop process received an `obj` or `object`, which is an array of deltas from the `FIFO` queue. As long as the delta is not of the type `Deleted`, the logic of the operator is used to create or modify some object until it matches the spec.

The `Informer` which uses the API server as a source, requests the state of an object via API call. The data is cached to minimize API server transactions. A similar agent is the `SharedInformer`; objects are often used by multiple other objects. It creates a shared cache of the state for multiple requests.

A `Workqueue` uses a key to hand out tasks to various workers. The standard `Go` work queues of rate limiting, delayed, and time queue are typically used.

The `endpoints`, `namespace`, and `serviceaccounts` controllers each manage the eponymous resources for Pods. Deployments manage `replicaSets` which manage Pods running the same `podSpec`, or replicas.

Service Operator

- Connect Pods together
- Expose Pods to Internet
- Decouple settings
- Define Pod access policy
- Operator monitoring a different operator

With every object and agent decoupled we need a flexible and scalable agent which connects resources together and will reconnect should something die and a replacement is spawned. A service is an operator which listens to endpoint operator to provide a persistent IP for Pods. Pods have ephemeral IP addresses chosen from a pool.

Then the service operator sends messages via the **kube-apiserver** which forwards settings to **kube-proxy** on every node as well as the network plugin such as **cilium-agent**

A service also handles access policies for inbound requests, useful for resource control as well as for security.

Pods

- One or more containers
- Smallest unit to work with
- Only one, shared IP address per Pod

The whole point of Kubernetes is to orchestrate the life cycle of a container. We do not interact with particular containers. Instead the smallest unit we can work with is a Pod. Some would say a pod of whales or peas-in-a-pod. Due to shared resources, the design of a Pod typically follows a one process per container architecture.

Containers in a Pod are started in parallel. As a result, there is no way to determine which container becomes available first inside a pod. The use of `InitContainers` can order startup, to some extent. To support a single process running in a container, you may need logging, a proxy, or special adapter. These tasks are often handled by other containers in the same pod.

There is only one IP address per Pod, for almost every network plugin. If there is more than one container in a pod, they must share the IP. To communicate with each other they can either use IPC, the loopback interface, or a shared filesystem.

While Pods are often deployed with one application container in each, a common reason to have multiple containers in a Pod is for logging. You may find the term `sidecar` for a container dedicated to performing a helper task, like handling logs and responding to requests as the primary application container may have this ability. The term `sidecar`, like `ambassador` and `adapter`, does not have a special setting but refers to the concept of what secondary pods are included to do.

Containers

- Not worked with directly
- Usage limits passed to container engine

```
resources:  
  limits:  
    cpu: "1"  
    memory: "4Gi"  
  requests:  
    cpu: "0.5"  
    memory: "500Mi"
```

- ResourceQuota
- PriorityClass

While Kubernetes orchestration does not allow direct manipulation on a container level, we can manage the resources containers are allowed to consume.

In the `resources` section of the PodSpec you can pass parameters which will be passed to the container runtime on the scheduled node.

Another way to manage resource usage of the containers is by creating a `ResourceQuota` object which allows hard and soft limits to be set in a namespace. The quotas allow management of more resources than just CPU and memory and allows limiting several objects.

`scopeSelector` field in the quota spec is used to run a pod at a specific priority if it has the appropriate `priorityClassName` in its pod spec.

Init Containers

- Block app containers until precondition met
- Can contain code or utilities not in an app
- Independent security from app container

```
spec:  
  containers:  
    - name: main-app  
      image: databaseD  
  initContainers:  
    - name: wait-database  
      image: busybox  
      command: ['sh', '-c', 'until ls /db/dir ; do sleep 5; done; ']
```

Not all containers are the same. Standard containers are sent to the container engine at the same time, and may start in any order. LivenessProbes, ReadinessProbes, and StatefulSets can be used to determine order but can add complexity. Another option can be an **Init Container** which must complete before app containers will be started. Should the init container fail it will be restarted until completion, without the app container running.

The init container can have a different view of the storage and security settings which allow utilities and commands to be used which the application would not be allowed to use.

The code above will run the init container until the **ls** command succeeds, then will database container.

Component Review

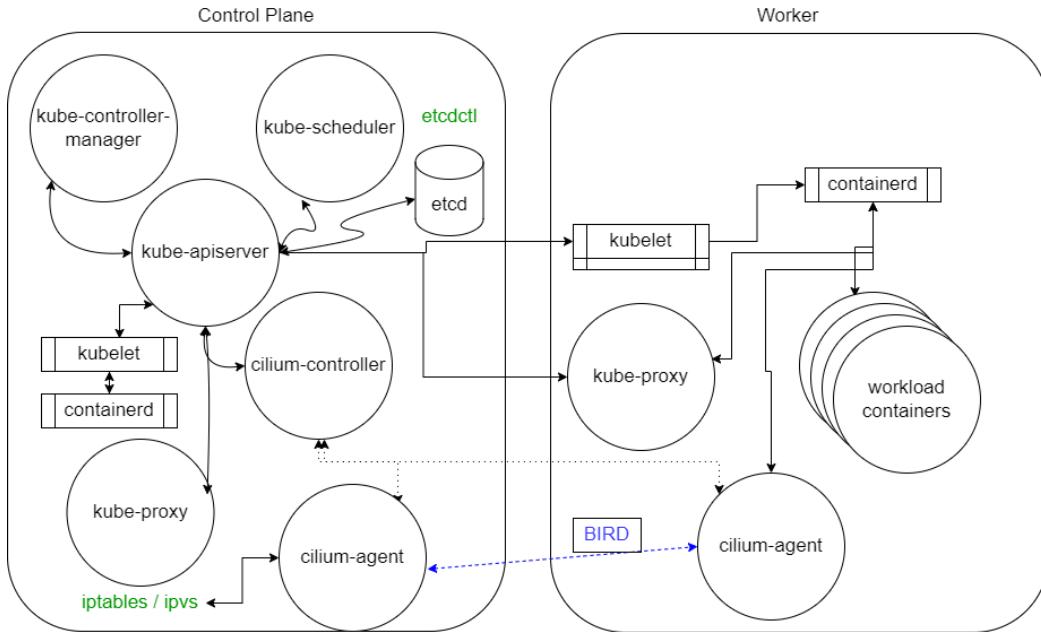


Figure 4.2: K8s Architectural Review

Now that we have seen some of the components, lets take another look with some of the connections shown. **Not** all connections are shown in this diagram. Note that all of the components are communicating with **kube-apiserver**. Only **kube-apiserver** communicates with the **etcd** database.

We also see some commands, which we may need to install separately, to work with various components. There is a **etcdctl** command to interrogate the database and **cilium** to view more of how the network is configured.

Node

- Created outside of cluster
- **NodeStatus**
- **NodeLease**
- View resource usage with **kubectl describe node**

A node is an API object created outside of the cluster representing an instance. While a cp must be **Linux**, worker nodes can also be **Microsoft Windows Server 2019**. Once the node has the necessary software installed it is ingested into the API server.

At the moment you can create a cp node with **kubeadm init** and worker nodes by passing **join**. In the near future secondary cp nodes and/or **etcd** nodes may be joined.

If the **kube-apiserver** cannot communicate with the **kubelet** on a node for five minutes, the default **NodeLease**, it will schedule the node for deletion, and the **NodeStatus** will change from **ready**. The pods will be evicted once connection is reestablished. They are no longer forcibly removed and rescheduled by the cluster.

Each node object exists in the **kube-node-lease** namespace. To remove a node from the cluster first use **kubectl delete node <node-name>** to remove it from the API server. This will cause pods to be evacuated. Then use **kubeadm reset** to remove cluster specific information. You may also need to remove **iptables** information, depending on if you plan on re-using the node.

To view CPU, memory, and other resource usage, requests and limits use the **kubectl describe node** command. The output will show capacity and pods allowed as well as details on current pods and resource utilization.

4.2 Networking

Single IP per Pod

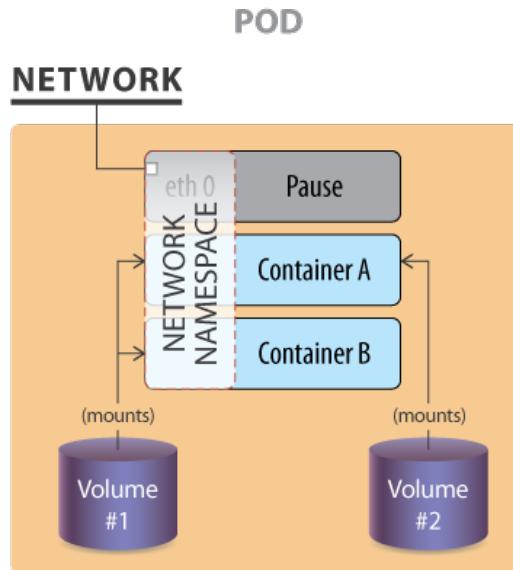


Figure 4.3: Pod Network

As well as a single IP address, a Pod represents a group of co-located containers with some associated data volumes. All containers in a pod share the same network **namespace**.

The graphic shows a pod with two containers, A and B, and two data volumes, 1 and 2. Containers A and B share the network namespace of a third container, known as the **pause container**. The pause container is used to get an IP address, then all the containers in the pod will use its network namespace. Volumes 1 and 2 are shown for completeness.

To communicate with each other containers within pods can use the loopback interface, write to files on a common filesystem or via inter-process communication (IPC). There is now a network plugin from **HPE Labs** which allows multiple IP addresses per pod, but this feature has not grown past this new plugin.

Container to Outside Path

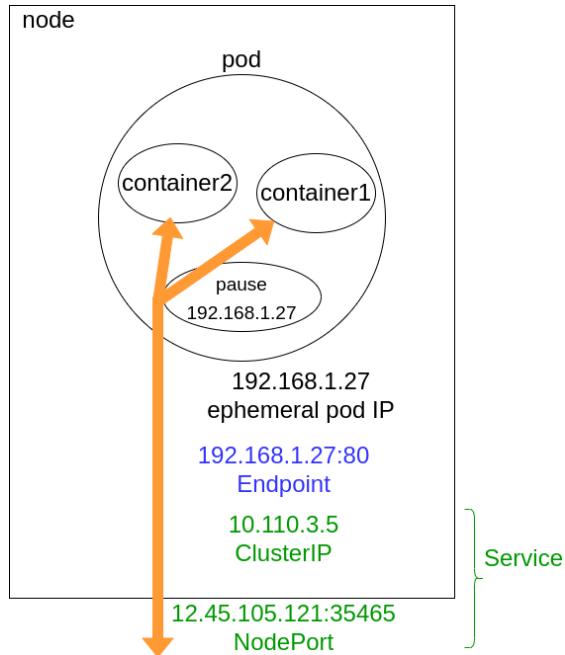


Figure 4.4: Container/Services Networking

This graphic shows a node with a single, dual-container pod. A NodePort service connects the Pod to the outside network. Even though there are two containers they share the same namespace and the same IP address, which would be configured by **kubelet** working with **kube-proxy**. The IP address is assigned before the containers are started and will be inserted into the containers. The container will have an interface like `eth0@tun10`. This IP is set for the life of the pod.

The endpoint is created at the same time as the service. Note that it uses the pod IP address, but also includes a port. The service connects network traffic from a node high-number port to the endpoint using iptables with ipvs on the way. The **kube-controller-manager** handles the watch loops to monitor the need for **endpoints** and **services**, as well as any updates or deletions.

Services

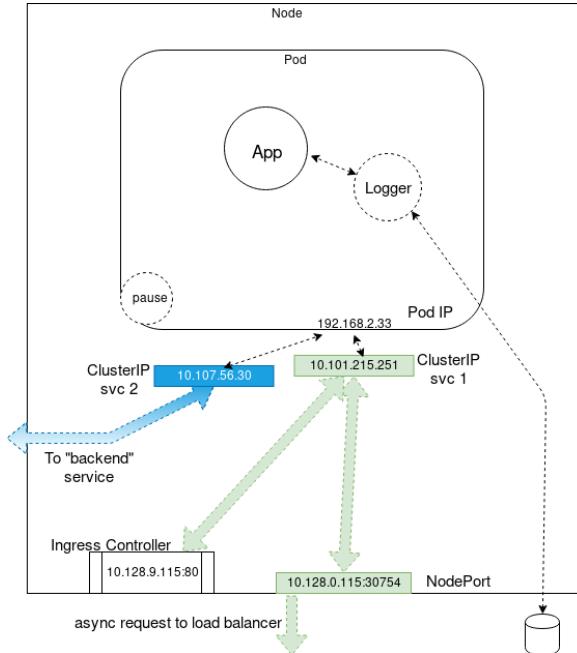


Figure 4.5: Services

We can use a service to connect one pod to another, or to outside of the cluster. This graphic shows a pod with a primary container, App, with an optional sidecar Logger. Also seen is the pause container, which is used by the cluster to reserve the IP address in the namespace prior to starting the other pods. This container is not seen from within Kubernetes, but can be seen using `docker` and `cubectl`.

This graphic shows a ClusterIP which is used to connect inside the cluster, not the IP of the cluster. As the graphic shows this can be used to connect to a NodePort for outside the cluster, an IngressController or proxy, or another "backend" pod or pods.

Networking Setup

- Main networking challenges:
 - Coupled container-to-container communications
 - Pod-to-pod communications
 - External-to-pod communications (solved by the services concept, to be discussed later)
- Admin configuration required
- IP assigned to pod, not container

Getting all the previous components running is a common task for system administrators who are accustomed to configuration management. But to get a fully functional Kubernetes cluster, the network will need to be setup properly as well.

A detailed explanation about the Kubernetes networking model can be seen at: <https://kubernetes.io/docs/concepts/cluster-administration/networking/>.

If you have experience deploying virtual machines (VMs) based on IaaS solutions, this will sound familiar. The only caveat is that in Kubernetes, the lowest compute unit is not a **container**, but what we call a **pod**.

A pod is a group of co-located containers that share the same IP address. From a networking perspective, a pod can be seen as a virtual machine or a physical host. The network needs to assign IP addresses to pods, and needs to provide traffic routes between all pods on any nodes. There are some network plugins which can allocate more than one IP to a pod, but most do not.

The three main networking challenges to solve in a container orchestration system are:

- Coupled container-to-container communications which is solved by the pod concept.
- Pod-to-pod communications.
- External-to-pod communications which is solved by the services concept.

Kubernetes expects the network configuration to enable pod-to-pod communications to be available; it will not do it for you.

Tim Hockin, one of the lead Kubernetes developers, has created a very useful slide deck to understand Kubernetes networking. You can check it out at: <https://speakerdeck.com/thockin/illustrated-guide-to-kubernetes-networking>.

CNI Network Configuration File

```
{  
    "cniVersion": "0.2.0",  
    "name": "mynet",  
    "type": "bridge",  
    "bridge": "cni0",  
    "isGateway": true,  
    "ipMasq": true,  
    "ipam": {  
        "type": "host-local",  
        "subnet": "10.22.0.0/16",  
        "routes": [  
            { "dst": "0.0.0.0/0" }  
        ]  
    }  
}
```

To provide container networking, Kubernetes is standardizing on the **Container Network Interface (CNI)** specification. Since v1.6.0, **kubeadm** (the Kubernetes cluster bootstrapping tool) the goal has been to use CNI as the default network interface mechanism. Various plugins can use CNI, but you may need to recompile to do so.

CNI is an emerging specification with associated libraries to write plugins that configure container networking, and remove allocated resources when the container is deleted. Its aim is to provide a common interface between the various networking solutions and container runtimes. As the CNI spec is language agnostic, there are many plugins from Amazon ECS to SR-IOV to Cloud Foundry and many more.

This configuration defines a standard Linux bridge named `cni0`, which will give out IP addresses in the subnet `10.22.0.0/16`. The `bridge` plugin will configure the network interfaces in the correct namespaces to define the container network properly.

The main [README](#) of the CNI GitHub repository (<https://github.com/containernetworking/cni>) has more information.

Pod-to-Pod Communication

- All pods can communicate with each other across nodes.
- All nodes can communicate with all pods.
- No Network Address Translation (NAT).

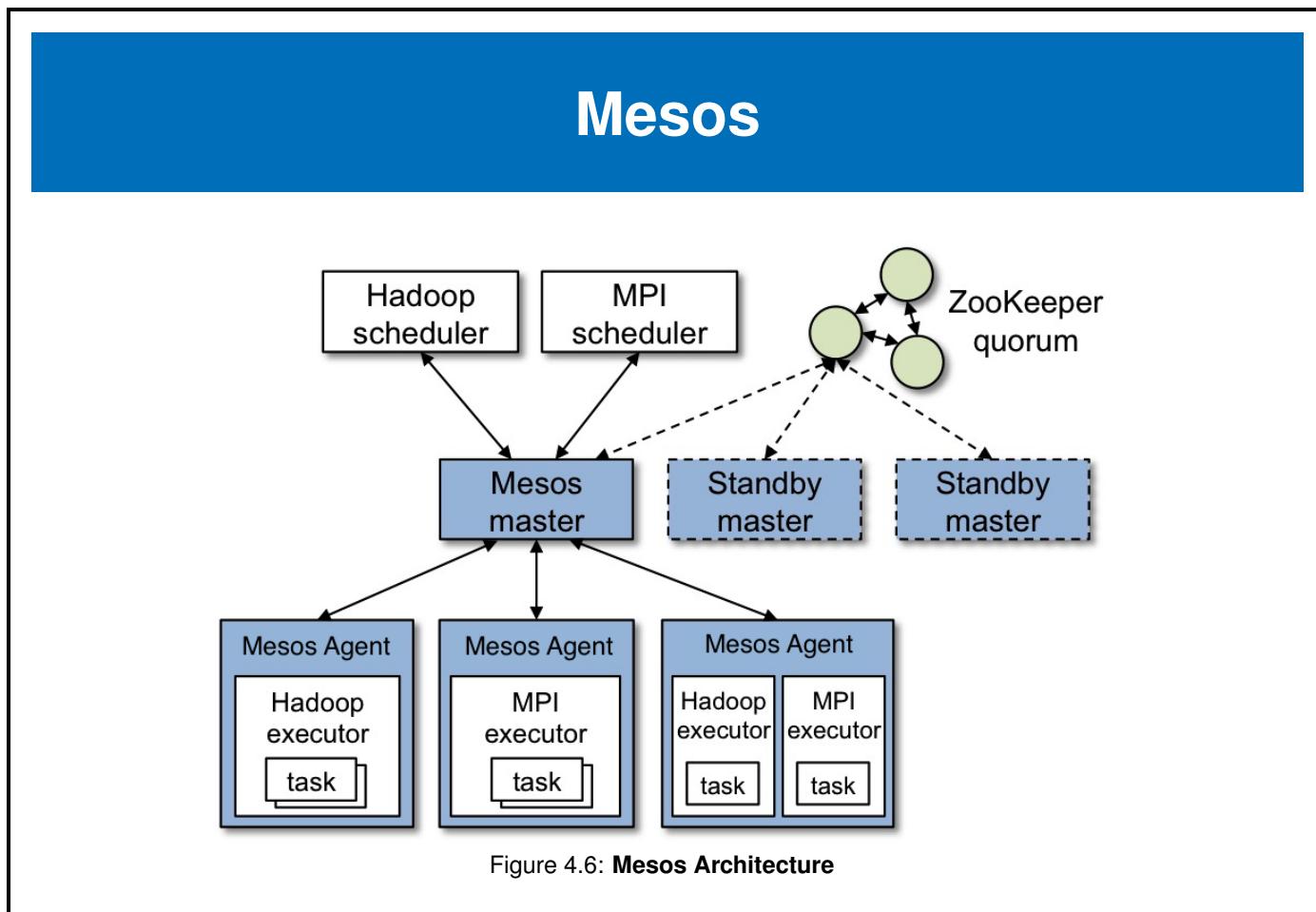
While a CNI plugin can be used to configure the network of a pod and provide a single IP per pod, CNI does not help you with pod-to-pod communications across nodes.

Basically, all IPs involved (nodes and pods) are routable without NAT. This can be achieved at the physical network infrastructure if you have access to it (e.g. GKE). Or this can be achieved with a software defined overlay with solutions like:

- Cilium
<https://docs.cilium.io/en/stable/>
- Flannel
<https://coreos.com/flannel/docs/latest/>
- Calico
<https://www.projectcalico.org/>
- Romana
<http://romana.io/>

See this documentation page or the list of networking add-ons for a more complete list.

4.3 Other Cluster Systems



At a high level, there is nothing different between Kubernetes and other clustering system.

A central manager exposes an API, a scheduler places the workloads on a set of nodes, and the state of the cluster is stored in a persistent layer.

For example, you could compare Kubernetes with **Mesos**, and you would see the similarities. In Kubernetes, however, the persistence layer is implemented with **etcd**, instead of **Zookeeper** for Mesos.

You should also consider systems like **OpenStack** and **CloudStack**. Think about what runs on their head node, and what runs on their worker nodes. How do they keep state? How do they handle networking? If you are familiar with those systems, Kubernetes will not seem that different.

What really sets Kubernetes apart is its features oriented towards fault-tolerance, self-discovery, and scaling, coupled with a mindset that is purely API-driven.

4.4 Labs

Exercise 4.1: Basic Node Maintenance

In this section we will backup the **etcd** database then update the version of Kubernetes used on control plane nodes and worker nodes.

Backup The etcd Database

While the upgrade process has become stable, it remains a good idea to backup the cluster state prior to upgrading. There are many tools available in the market to backup and manage etcd, each with a distinct backup and restore process. We will use the included snapshot command, but be aware the exact steps to restore will depend on the tools used, the version of the cluster, and the nature of the disaster being recovered from.

- Find the data directory of the **etcd** daemon. All of the settings for the pod can be found in the manifest.

```
student@cp:~$ sudo grep data-dir /etc/kubernetes/manifests/etcd.yaml
- --data-dir=/var/lib/etcd
```

- Log into the **etcd** container and look at the options **etcdctl** provides. Use tab to complete the container name, which has the node name appended to it.

```
student@cp:~$ kubectl -n kube-system exec -it etcd-<Tab> -- sh
```



On Container

- View the arguments and options to the **etcdctl** command. Take a moment to view the options and arguments available. As the Bourne shell does not have many features it may be easier to copy/paste the majority of the command and arguments after typing them out the first time.

```
# etcdctl -h
NAME:
    etcdctl - A simple command line client for etcd3.

USAGE:
    etcdctl [flags]
    <output_omitted>
```

- In order to use TLS, find the three files that need to be passed with the **etcdctl** command. Change into the directory and view available files. Newer versions of **etcd** image have been minimized. As a result you may no longer have the **find** command, or really most commands. One must remember the URL **/etc/kubernetes/pki/etcd**. As the **ls** command is also missing we can view the files using **echo** instead.

```
# cd /etc/kubernetes/pki/etcd
# echo *
ca.crt ca.key healthcheck-client.crt healthcheck-client.key
peer.crt peer.key server.crt server.key
```

- Typing out each of these keys, especially in a locked-down shell can be avoided by using an environmental parameter. Log out of the shell and pass the various paths to the necessary files.



```
# exit
```

3. Check the health of the database using the loopback IP and port 2379. You will need to pass then peer cert and key as well as the Certificate Authority as environmental variables. The command is commented, you do not need to type out the comments or the backslashes.

```
student@cp:~$ kubectl -n kube-system exec -it etcd-cp -- sh \
-c "ETCDCTL_API=3 \
ETCDCTL_CACERT=/etc/kubernetes/pki/etcd/ca.crt \
ETCDCTL_CERT=/etc/kubernetes/pki/etcd/server.crt \
ETCDCTL_KEY=/etc/kubernetes/pki/etcd/server.key \
etcdctl endpoint health"      #The command to test the endpoint
```

```
https://127.0.0.1:2379 is healthy: successfully committed proposal: took = 11.942936ms
```

4. Determine how many databases are part of the cluster. Three and five are common in a production environment to provide 50%+1 for quorum. In our current exercise environment we will only see one database. Remember you can use up-arrow to return to the previous command and edit the command without having to type the whole command again.

```
student@cp:~$ kubectl -n kube-system exec -it etcd-cp -- sh \
-c "ETCDCTL_API=3 \
ETCDCTL_CACERT=/etc/kubernetes/pki/etcd/ca.crt \
ETCDCTL_CERT=/etc/kubernetes/pki/etcd/server.crt \
ETCDCTL_KEY=/etc/kubernetes/pki/etcd/server.key \
etcdctl --endpoints=https://127.0.0.1:2379 member list"
```

```
fb50b7ddbf4930ba, started, cp, https://10.128.0.35:2380,
https://10.128.0.35:2379, false
```

5. You can also view the status of the cluster in a table format, among others passed with the **-w** option. Again, up-arrow allows you to edit just the last part of the long string easily.

```
student@cp:~$ kubectl -n kube-system exec -it etcd-cp -- sh \
-c "ETCDCTL_API=3 \
ETCDCTL_CACERT=/etc/kubernetes/pki/etcd/ca.crt \
ETCDCTL_CERT=/etc/kubernetes/pki/etcd/server.crt \
ETCDCTL_KEY=/etc/kubernetes/pki/etcd/server.key \
etcdctl --endpoints=https://127.0.0.1:2379 member list -w table"
```

ID	STATUS	NAME	PEER ADDRS	CLIENT ADDRS	IS LEARNER
802d78549985d5a8	started	cp	https://10.128.0.15:2380	https://10.128.0.15:2379	false

6. Now that we know how many etcd databases are in the cluster, and their health, we can back it up. Use the snapshot argument to save the snapshot into the container data directory/`/var/lib/etcd/`

```
student@cp:~$ kubectl -n kube-system exec -it etcd-cp -- sh \
-c "ETCDCTL_API=3 \
ETCDCTL_CACERT=/etc/kubernetes/pki/etcd/ca.crt \
ETCDCTL_CERT=/etc/kubernetes/pki/etcd/server.crt \
ETCDCTL_KEY=/etc/kubernetes/pki/etcd/server.key \
```

```
etcdctl --endpoints=https://127.0.0.1:2379 snapshot save /var/lib/etcd/snapshot.db "
```

```
{"level":"info","ts":1598380941.6584022,"caller":"snapshot/v3_snapshot.go:110","msg":"created temporary db file","path":"/var/lib/etcd/snapshot.db.part"} {"level":"warn","ts":"2023-02-25T18:42:21.671Z","caller":"clientv3/retry_interceptor.go:116","msg":"retry stream intercept"} {"level":"info","ts":1598380941.6736135,"caller":"snapshot/v3_snapshot.go:121","msg":"fetching snapshot","endpoint":"https://127.0.0.1:2379"} {"level":"info","ts":1598380941.7519674,"caller":"snapshot/v3_snapshot.go:134","msg":"fetched snapshot","endpoint":"https://127.0.0.1:2379","took":0.093466104} {"level":"info","ts":1598380941.7521122,"caller":"snapshot/v3_snapshot.go:143","msg":"saved","path":"/var/lib/etcd/snapshot.db"} Snapshot saved at /var/lib/etcd/snapshot.db
```

- Verify the snapshot exists from the node perspective, the file date should have been moments earlier.

```
student@cp:~$ sudo ls -l /var/lib/etcd/
```

```
total 3888
drwx----- 4 root root 4096 Aug 25 11:22 member
-rw----- 1 root root 3973152 Aug 25 18:42 snapshot.db
```

- Backup the snapshot as well as other information used to create the cluster both locally as well as another system in case the node becomes unavailable. Remember to create snapshots on a regular basis, perhaps using a cronjob to ensure a timely restore. When using the snapshot restore it's important the database not be in use. An HA cluster would remove and replace the control plane node, and not need a restore.

```
student@cp:~$ mkdir $HOME/backup
student@cp:~$ sudo cp /var/lib/etcd/snapshot.db $HOME/backup/snapshot.db-$(date +%-m-%d-%y)
student@cp:~$ sudo cp /root/kubeadm-config.yaml $HOME/backup/
student@cp:~$ sudo cp -r /etc/kubernetes/pki/etcd $HOME/backup/
```

- Any mistakes during restore may render the cluster unusable. Instead of issues, and having to rebuild the cluster, please attempt a database restore after the final lab exercise of the course. More on the restore process can be found here: <https://kubernetes.io/docs/tasks/administer-cluster/configure-upgrade-etcd/#restoring-an-etcd-cluster>

Upgrade the Cluster

- Begin by updating the package metadata for APT.

```
student@cp:~$ sudo apt update
```

```
Hit:1 http://us-central1.gce.archive.ubuntu.com/ubuntu bionic InRelease
Get:2 http://us-central1.gce.archive.ubuntu.com/ubuntu bionic-updates InRelease [88.7 kB]
Get:3 http://us-central1.gce.archive.ubuntu.com/ubuntu bionic-backports InRelease [74.6 kB]
Get:5 http://security.ubuntu.com/ubuntu bionic-security InRelease [88.7 kB]
<output_omitted>
```



Introducing Kubernetes Community-Owned Package Repositories

Note: The legacy package repositories (`apt.kubernetes.io` and `yum.kubernetes.io`) have been deprecated and frozen starting from September 13, 2023. Using the new package repositories hosted at `pkgs.k8s.io` is strongly recommended and required in order to install Kubernetes versions released after September 13, 2023. The deprecated legacy repositories, and their contents, might be removed at any time in the future and without a further notice period. The new package repositories provide downloads for Kubernetes versions starting with v1.24.0.



For further information please refer the URL <https://kubernetes.io/blog/2023/08/15/pkgs-k8s-io-introduction/>

- Replace the apt repository definition so that apt points to the new repository instead of the Google-hosted repository. Make sure to replace the Kubernetes minor version in the command below with the minor version that you are going to upgrade.

```
student@cp:~$ sudo sed -i 's/29/30/g' /etc/apt/sources.list.d/kubernetes.list
```

```
student@cp:~$ sudo apt-get update
```

- View the available packages.

```
student@cp:~$ sudo apt-cache madison kubeadm
```

```
kubeadm | 1.30.2-1.1 | https://pkgs.k8s.io/core:/stable:/v1.30/deb Packages
kubeadm | 1.30.1-1.1 | https://pkgs.k8s.io/core:/stable:/v1.30/deb Packages
kubeadm | 1.30.0-1.1 | https://pkgs.k8s.io/core:/stable:/v1.30/deb Packages
<output_omitted>
```

- Remove the hold on **kubeadm** and update the package. Remember to update to the next major release's update 1.

```
student@cp:~$ sudo apt-mark unhold kubeadm
```

```
Canceled hold on kubeadm.
```

```
student@cp:~$ sudo apt-get install -y kubeadm=1.30.1-1.1
```

```
Reading package lists... Done
Building dependency tree
Reading state information... Done
<output_omitted>
```

- Hold the package again to prevent updates along with other software.

```
student@cp:~$ sudo apt-mark hold kubeadm
```

```
kubeadm set on hold.
```

- Verify the version of **Kubeadm**. It should indicate the new version you just installed.

```
student@cp:~$ sudo kubeadm version
```

```
kubeadm version: &version.Info{Major:"1", Minor:"30", GitVersion:"v1.30.1",
→ GitCommit:"6911225c3f747e1cd9d109c305436d08b668f086", GitTreeState:"clean",
→ BuildDate:"2024-05-14T10:49:05Z", GoVersion:"go1.22.2", Compiler:"gc", Platform:"linux/amd64"}
```

- To prepare the cp node for update we first need to evict as many pods as possible. The nature of daemonsets is to have them on every node, and some such as Cilium must remain. Change the node name to your node's name, and add the option to ignore the daemonsets.

```
student@cp:~$ kubectl drain cp --ignore-daemonsets
```

```

node/cp cordoned
Warning: ignoring DaemonSet-managed Pods: kube-system/cilium-5tv9d, kube-system/kube-proxy-8x9c5
evicting pod kube-system/coredns-5d78c9869d-z5ngb
evicting pod kube-system/cilium-operator-788c7d7585-wnb5b
evicting pod kube-system/coredns-5d78c9869d-4h2bs
pod/cilium-operator-788c7d7585-wnb5b evicted
pod/coredns-5d78c9869d-z5ngb evicted
pod/coredns-5d78c9869d-4h2bs evicted
node/cp drained

```

8. Use the **upgrade plan** argument to check the existing cluster and then update the software. You may notice that there are versions available later than the .1 update in use. If you initialized the cluster in a previous lab using 1.25.1, use upgrade to 1.26.1. If you initialized using 1.26.1, upgrade 1.27.1, etc. Read through the output and get a feel for what would be changed in an upgrade.

```
student@cp:~$ sudo kubeadm upgrade plan
```

```

[preflight] Running pre-flight checks.
[upgrade/config] Reading configuration from the cluster...
[upgrade/config] FYI: You can look at this config file with 'kubectl -n kube-system get cm
→ kubeadm-config -o yaml'
[upgrade] Running cluster health checks
[upgrade] Fetching available versions to upgrade to
[upgrade/versions] Cluster version: 1.29.1
[upgrade/versions] kubeadm version: v1.30.1
[upgrade/versions] Target version: v1.30.2
[upgrade/versions] Latest version in the v1.29 series: v1.29.6

<output_omitted>

```

9. We are now ready to actually upgrade the software. There will be a lot of output. Be aware the command will ask if you want to proceed with the upgrade, answer **y** for yes. Take a moment and look for any errors or suggestions, such as upgrading the version of etcd, or some other package. Again, Use the next minor release, update 1 from the version used previous lab which initialized the cluster. The process will take several minutes to complete.

```
student@cp:~$ sudo kubeadm upgrade apply v1.30.1
```

```

[upgrade/config] Making sure the configuration is correct:
[upgrade/config] Reading configuration from the cluster...
[upgrade/config] FYI: You can look at this config file with 'kubectl -n kube-system get cm
→ kubeadm-config -o yaml'
[preflight] Running pre-flight checks.
[upgrade] Running cluster health checks
[upgrade/version] You have chosen to change the cluster version to "v1.30.1"
[upgrade/versions] Cluster version: v1.29.1
[upgrade/versions] kubeadm version: v1.30.1
[upgrade] Are you sure you want to proceed? [y/N]: y
[upgrade/prepull] Pulling images required for setting up a Kubernetes cluster
[upgrade/prepull] This might take a minute or two, depending on the speed of your internet
→ connection
[upgrade/prepull] You can also perform this action in beforehand using 'kubeadm config images
→ pull'
[upgrade/apply] Upgrading your Static Pod-hosted control plane to version "v1.30.1" (timeout:
→ 5m0s)...
<output_omitted>

```

10. Check the status of the nodes. The cp should show scheduling disabled. Also as we have not updated all the software and restarted the daemons it will show the previous version.

```
student@cp:~$ kubectl get node
```

NAME	STATUS	ROLES	AGE	VERSION
cp	Ready, SchedulingDisabled	control-plane	109m	v1.29.1
worker	Ready	<none>	61m	v1.29.1

11. Release the hold on **kubelet** and **kubectl**.

```
student@cp:~$ sudo apt-mark unhold kubelet kubectl
```

```
Canceled hold on kubelet.  
Canceled hold on kubectl.
```

12. Upgrade both packages to the same version as **kubeadm**.

```
student@cp:~$ sudo apt-get install -y kubelet=1.30.1-1.1 kubectl=1.30.1-1.1
```

```
Reading package lists... Done  
Building dependency tree  
Reading state information... Done  
<output omitted>
```

13. Again add the hold so other updates don't update the Kubernetes software.

```
student@cp:~$ sudo apt-mark hold kubelet kubectl
```

```
kubelet set on hold.  
kubectl set on hold.
```

14. Restart the daemons.

```
student@cp:~$ sudo systemctl daemon-reload
```

```
student@cp:~$ sudo systemctl restart kubelet
```

15. Verify the cp node has been updated to the new version. Then update other cp nodes, if you should have them, using the same process except **sudo kubeadm upgrade node** instead of **sudo kubeadm upgrade apply**.

```
student@cp:~$ kubectl get node
```

NAME	STATUS	ROLES	AGE	VERSION
cp	Ready, SchedulingDisabled	control-plane	113m	v1.30.1
worker	Ready	<none>	65m	v1.29.1

16. Now make the cp available for the scheduler, again change the name to match the cluster node name on your control plane.

```
student@cp:~$ kubectl uncordon cp
```

```
node/cp uncordoned
```

17. Verify the cp now shows a Ready status.

```
student@cp:~$ kubectl get node
```

NAME	STATUS	ROLES	AGE	VERSION
cp	Ready	control-plane	114m	v1.30.1
worker	Ready	<none>	66m	v1.29.1

18. Now update the worker node(s) of the cluster. **Open a second terminal session to the worker.** Note that you will need to run a couple commands on the cp as well, having two sessions open may be helpful. Begin by allowing the software to update on the worker.

```
student@worker:~$ sudo apt-mark unhold kubeadm
```

```
Canceled hold on kubeadm.
```



Introducing Kubernetes Community-Owned Package Repositories

Note: The legacy package repositories (`apt.kubernetes.io` and `yum.kubernetes.io`) have been deprecated and frozen starting from September 13, 2023. Using the new package repositories hosted at `pkgs.k8s.io` is strongly recommended and required in order to install Kubernetes versions released after September 13, 2023. The deprecated legacy repositories, and their contents, might be removed at any time in the future and without a further notice period. The new package repositories provide downloads for Kubernetes versions starting with v1.24.0. <https://kubernetes.io/blog/2023/08/15/pkgs-k8s-io-introduction/>

19. Replace the apt repository definition so that apt points to the new repository instead of the Google-hosted repository. Make sure to replace the Kubernetes minor version in the command below with the minor version that you are going to upgrade.

```
student@worker:~$ sudo sed -i 's/29/30/g' /etc/apt/sources.list.d/kubernetes.list
```

```
student@worker:~$ sudo apt-get update
```

20. View the available packages.

```
student@worker:~$ sudo apt-cache madison kubeadm
```

```
kubeadm | 1.30.2-1.1 | https://pkgs.k8s.io/core:/stable:/v1.30/deb Packages
kubeadm | 1.30.1-1.1 | https://pkgs.k8s.io/core:/stable:/v1.30/deb Packages
kubeadm | 1.30.0-1.1 | https://pkgs.k8s.io/core:/stable:/v1.30/deb Packages
<output omitted>
```

21. Update the **`kubeadm`** package to the same version as the cp node.

```
student@worker:~$ sudo apt-mark unhold kubeadm
student@worker:~$ sudo apt-get update && sudo apt-get install -y kubeadm=1.30.1-1.1
```

```
<output omitted>
Setting up kubeadm (1.30.1-1.1) ...
```

22. Hold the package again.

```
student@worker:~$ sudo apt-mark hold kubeadm
```

```
kubeadm set on hold.
```

23. Back on the **cp terminal session** drain the worker node, but allow the daemonsets to remain.

```
student@cp:~$ kubectl drain worker --ignore-daemonsets
```

```

node/worker cordoned
Warning: ignoring DaemonSet-managed Pods: kube-system/cilium-gzdk6, kube-system/kube-proxy-lpsmq
evicting pod kube-system/cilium-operator-788c7d7585-hc9wf
evicting pod kube-system/coredns-5d78c9869d-h4p7v
evicting pod kube-system/coredns-5d78c9869d-d4nv8
pod/cilium-operator-788c7d7585-hc9wf evicted
pod/coredns-5d78c9869d-h4p7v evicted
pod/coredns-5d78c9869d-d4nv8 evicted
node/worker drained

```

24. Return to the **worker** node and download the updated node configuration.

```
student@worker:~$ sudo kubeadm upgrade node
```

```

[upgrade] Reading configuration from the cluster...
[upgrade] FYI: You can look at this config file with 'kubectl -n kube-system get cm
kubeadm-config -o yaml'
[preflight] Running pre-flight checks
[preflight] Skipping prepull. Not a control plane node.
[upgrade] Skipping phase. Not a control plane node.
[kubelet-start] Writing kubelet configuration to file "/var/lib/kubelet/config.yaml"
[upgrade] The configuration for this node was successfully updated!
[upgrade] Now you should go ahead and upgrade the kubelet package using your package manager.

```

25. Remove the hold on the software then update to the same version as set on the cp.

```
student@worker:~$ sudo apt-mark unhold kubelet kubectl
```

```

Canceled hold on kubelet.
Canceled hold on kubectl.

```

```
student@worker:~$ sudo apt-get install -y kubelet=1.30.1-1.1 kubectl=1.30.1-1.1
```

```

Reading package lists... Done
Building dependency tree
<output_omitted>
Setting up kubectl (1.30.1-1.1) ...
Setting up kubelet (1.30.1-1.1) ...

```

26. Ensure the packages don't get updated when along with regular updates.

```
student@worker:~$ sudo apt-mark hold kubelet kubectl
```

```

kubelet set on hold.
kubectl set on hold.

```

27. Restart daemon processes for the software to take effect.

```
student@worker:~$ sudo systemctl daemon-reload
```

```
student@worker:~$ sudo systemctl restart kubelet
```

28. Return to the cp node. View the status of the nodes. Notice the worker status.

```
student@cp:~$ kubectl get node
```

NAME	STATUS	ROLES	AGE	VERSION
cp	Ready	control-plane	118m	v1.30.1
worker	Ready, SchedulingDisabled	<none>	70m	v1.30.1

29. Allow pods to be deployed to the worker node. Remember to use YOUR worker name. TAB can be helpful to enter the name if command line completion is enabled.

```
student@cp:~$ kubectl uncordon worker
```

```
node/worker uncordoned
```

30. Verify the nodes both show a Ready status and the same upgraded version.

```
student@cp:~$ kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
cp	Ready	control-plane	119m	v1.30.1
worker	Ready	<none>	71m	v1.30.1

Exercise 4.2: Working with CPU and Memory Constraints

Overview

We will continue working with our cluster, which we built in the previous lab. We will work with resource limits, more with namespaces and then a complex deployment which you can explore to further understand the architecture and relationships.

Use **SSH** or **PuTTY** to connect to the nodes you installed in the previous exercise. We will deploy an application called **stress** inside a container, and then use resource limits to constrain the resources the application has access to use.

1. Use a container called **stress**, in a deployment which we will name **hog**, to generate load. Verify you have the container running.

```
student@cp:~$ kubectl create deployment hog --image vish/stress
```

```
deployment.apps/hog created
```

```
student@cp:~$ kubectl get deployments
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
hog	1/1	1	1	13s

2. Use the **describe** argument to view details, then view the output in YAML format. Note there are no settings limiting resource usage. Instead, there are empty curly brackets.

```
student@cp:~$ kubectl describe deployment hog
```

Name:	hog
Namespace:	default
CreationTimestamp:	Thu, 23 Aug 2023 10:15:53 +0000
Labels:	app=hog
Annotations:	deployment.kubernetes.io/revision: 1
<output omitted>	

```
student@cp:~$ kubectl get deployment hog -o yaml

apiVersion: apps/v1
kind: Deployment
Metadata:
<output_omitted>

template:
  metadata:
    creationTimestamp: null
    labels:
      app: hog
  spec:
    containers:
    - image: vish/stress
      imagePullPolicy: Always
      name: stress
      resources: {}
      terminationMessagePath: /dev/termination-log
<output_omitted>
```

3. We will use the YAML output to create our own configuration file.

```
student@cp:~$ kubectl get deployment hog -o yaml > hog.yaml
```

4. Probably good to remove the status output, creationTimestamp and other settings. We will also add in memory limits found below.

```
student@cp:~$ vim hog.yaml
```



hog.yaml

```
1 ....
2     imagePullPolicy: Always
3     name: hog
4     resources:           # Edit to remove {}
5       limits:            # Add these 4 lines
6         memory: "4Gi"
7       requests:
8         memory: "2500Mi"
9     terminationMessagePath: /dev/termination-log
10    terminationMessagePolicy: File
11 ....
12
```

5. Replace the deployment using the newly edited file.

```
student@cp:~$ kubectl replace -f hog.yaml
```

```
deployment.apps/hog replaced
```

6. Verify the change has been made. The deployment should now show resource limits.

```
student@cp:~$ kubectl get deployment hog -o yaml
```

```
....  
resources:  
  limits:  
    memory: 4Gi  
  requests:  
    memory: 2500Mi  
  terminationMessagePath: /dev/termination-log  
....
```

7. View the stdio of the hog container. Note how much memory has been allocated.

```
student@cp:~$ kubectl get po
```

NAME	READY	STATUS	RESTARTS	AGE
hog-64cbfcc7cf-lwq66	1/1	Running	0	2m

```
student@cp:~$ kubectl logs hog-64cbfcc7cf-lwq66
```

```
I1102 16:16:42.638972      1 main.go:26] Allocating "0" memory, in  
"4Ki" chunks, with a 1ms sleep between allocations  
I1102 16:16:42.639064      1 main.go:29] Allocated "0" memory
```

8. Open a second and third terminal to access both cp and second nodes. Run **top** to view resource usage. You should not see unusual resource usage at this point. The **containerd** and **top** processes should be using about the same amount of resources. The **stress** command should not be using enough resources to show up. Use the **kubectl get events** to see if any pods are evicted when too many resources are in use.
9. Edit the hog configuration file and add arguments for **stress** to consume CPU and memory. The **args:** entry should be indented the same number of spaces as **resources:**.

```
student@cp:~$ vim hog.yaml
```

YAML hog.yaml

```
1 ....  
2     resources:  
3       limits:  
4         cpu: "1"  
5         memory: "4Gi"  
6       requests:  
7         cpu: "0.5"  
8         memory: "500Mi"  
9       args:  
10      - --cpus  
11      - "2"  
12      - --mem-total  
13      - "950Mi"  
14      - --mem-alloc-size  
15      - "100Mi"  
16      - --mem-alloc-sleep  
17      - "1s"  
18 ....  
19
```

10. Delete and recreate the deployment. You should see increased CPU usage almost immediately and memory allocation happen in 100M chunks, allocated to the **stress** program via the running **top** command. Check both nodes as the

container could be deployed to either. Be aware that nodes with a small amount of memory or CPU may encounter issues. Symptoms include CPU node infrastructure pods failing. Adjust the amount of resources used to allow standard pods to run without error.

```
student@cp:~$ kubectl delete deployment hog
```

```
deployment.apps "hog" deleted
```

```
student@cp:~$ kubectl create -f hog.yaml
```

```
deployment.apps/hog created
```



Only if top does not show high usage

Should the resources not show increased use, there may have been an issue inside of the container. Kubernetes may show it as running, but the actual workload has failed. Or the container may have failed; for example if you were missing a parameter the container may panic.

```
student@cp:~$ kubectl get pod
```

NAME	READY	STATUS	RESTARTS	AGE
hog-1985182137-5bz2w	0/1	Error	1	5s

```
student@cp:~$ kubectl logs hog-1985182137-5bz2w
```

```
panic: cannot parse '150mi': unable to parse quantity's suffix

goroutine 1 [running]:
panic(0x5ff9a0, 0xc820014cb0)
    /usr/local/go/src/runtime/panic.go:481 +0x3e6
k8s.io/kubernetes/pkg/api/resource.MustParse(0x7ffe460c0e69, 0x5, 0x0, 0x0, 0x0, 0x0, 0x0,
    ↳ 0x0, 0x0)
    /usr/local/google/home/vishnuk/go/src/k8s.io/kubernetes/pkg/api/resource/quantity.go:134
    ↳ +0x287
main.main()
    /usr/local/google/home/vishnuk/go/src/github.com/vishh/stress/main.go:24 +0x43
```

Here is an example of an improper parameter. The container is running, but not allocating memory. It should show the usage requested from the YAML file.

```
student@cp:~$ kubectl get po
```

NAME	READY	STATUS	RESTARTS	AGE
hog-1603763060-x3vnn	1/1	Running	0	8s

```
student@cp:~$ kubectl logs hog-1603763060-x3vnn
```

```
I0927 21:09:23.514921      1 main.go:26] Allocating "0" memory, in "4ki" chunks, with a 1ms
    ↳ sleep \
          between allocations
I0927 21:09:23.514984      1 main.go:39] Spawning a thread to consume CPU
I0927 21:09:23.514991      1 main.go:39] Spawning a thread to consume CPU
I0927 21:09:23.514997      1 main.go:29] Allocated "0" memory
```

Exercise 4.3: Resource Limits for a Namespace

The previous steps set limits for that particular deployment. You can also set limits on an entire namespace. We will create a new namespace and configure another hog deployment to run within. When set hog should not be able to use the previous amount of resources.

1. Begin by creating a new namespace called low-usage-limit and verify it exists.

```
student@cp:~$ kubectl create namespace low-usage-limit
```

```
namespace/low-usage-limit created
```

```
student@cp:~$ kubectl get namespace
```

NAME	STATUS	AGE
default	Active	1h
kube-node-lease	Active	1h
kube-public	Active	1h
kube-system	Active	1h
low-usage-limit	Active	42s

2. Create a YAML file which limits CPU and memory usage. The kind to use is LimitRange. Remember the file may be found in the example tarball.

```
student@cp:~$ cp /home/student/LFS458/SOLUTIONS/s_04/low-resource-range.yaml .
```

```
student@cp:~$ vim low-resource-range.yaml
```



low-resource-range.yaml

```

1 apiVersion: v1
2 kind: LimitRange
3 metadata:
4   name: low-resource-range
5 spec:
6   limits:
7     - default:
8       cpu: 1
9       memory: 500Mi
10    defaultRequest:
11      cpu: 0.5
12      memory: 100Mi
13    type: Container

```

3. Create the LimitRange object and assign it to the newly created namespace low-usage-limit. You can use --namespace or -n to declare the namespace.

```
student@cp:~$ kubectl create -f low-resource-range.yaml -n low-usage-limit
```

```
limitrange/low-resource-range created
```

4. Verify it works. Remember that every command needs a namespace and context to work. Defaults are used if not provided.

```
student@cp:~$ kubectl get LimitRange
```

No resources found in default namespace.

```
student@cp:~$ kubectl get LimitRange --all-namespaces
```

NAMESPACE	NAME	CREATED AT
low-usage-limit	low-resource-range	2024-06-23T10:23:57Z

5. Create a new deployment in the namespace.

```
student@cp:~$ kubectl -n low-usage-limit \
    create deployment limited-hog --image vish/stress
```

deployment.apps/limited-hog created

6. List the current deployments. Note hog continues to run in the default namespace. If you chose to use the **Cilium** network policy you may see a couple more than what is listed below.

```
student@cp:~$ kubectl get deployments --all-namespaces
```

NAMESPACE	NAME	READY	UP-TO-DATE	AVAILABLE	AGE
default	hog	1/1	1	1	7m57s
kube-system	cilium-operator	1/1	1	1	2d10h
kube-system	coredns	2/2	2	2	2d10h
low-usage-limit	limited-hog	1/1	1	1	9s

7. View all pods within the namespace. Remember you can use the **tab** key to complete the namespace. You may want to type the namespace first so that tab-completion is appropriate to that namespace instead of the default namespace.

```
student@cp:~$ kubectl -n low-usage-limit get pods
```

NAME	READY	STATUS	RESTARTS	AGE
limited-hog-2556092078-wnpnv	1/1	Running	0	2m11s

8. Look at the details of the pod. You will note it has the settings inherited from the entire namespace. The use of shell completion should work if you declare the namespace first.

```
student@cp:~$ kubectl -n low-usage-limit \
    get pod limited-hog-2556092078-wnpnv -o yaml
```

```
<output_omitted>
spec:
  containers:
  - image: vish/stress
    imagePullPolicy: Always
    name: stress
    resources:
      limits:
        cpu: "1"
        memory: 500Mi
      requests:
        cpu: 500m
        memory: 100Mi
      terminationMessagePath: /dev/termination-log
<output_omitted>
```

9. Copy and edit the config file for the original `hog` file. Add the `namespace:` line so that a new deployment would be in the `low-usage-limit` namespace. Delete the `selfLink` line, if it exists.

```
student@cp:~$ cp hog.yaml hog2.yaml
```

```
student@cp:~$ vim hog2.yaml
```

YAML

hog2.yaml

```
1 ....
2   labels:
3     app: hog
4   name: hog
5   namespace: low-usage-limit      #<<--- Add this line, delete following
6   selfLink: /apis/apps/v1/namespaces/default/deployments/hog
7 spec:
8 ....
9
```

10. Open up extra terminal sessions so you can have `top` running in each. When the new deployment is created it will probably be scheduled on the node not yet under any stress.

Create the deployment.

```
student@cp:~$ kubectl create -f hog2.yaml
```

```
deployment.apps/hog created
```

11. View the deployments. Note there are two with the same name, `hog` but in different namespaces. You may also find the `cilium` deployment has no pods, nor has any requested. Our small cluster does not need to add **Cilium** pods via this autoscaler.

```
student@cp:~$ kubectl get deployments --all-namespaces
```

NAMESPACE	NAME	READY	UP-TO-DATE	AVAILABLE	AGE
default	hog	1/1	1	1	24m
kube-system	cilium-operator	1/1	0	0	4h
kube-system	coredns	2/2	2	2	4h
low-usage-limit	hog	1/1	1	1	26s
low-usage-limit	limited-hog	1/1	1	1	5m11s

12. Look at the `top` output running in other terminals. You should find that both `hog` deployments are using about the same amount of resources, once the memory is fully allocated. Per-deployment settings override the global namespace settings. You should see something like the following lines one from each node, which indicates use of one processor and about 12 percent of your memory, were you on a system with 8G total.

```
25128 root      20  0  958532 954672   3180 R 100.0 11.7   0:52.27 stress
24875 root      20  0  958532 954800   3180 R 100.3 11.7  41:04.97 stress
```

13. Delete the `hog` deployments to recover system resources.

```
student@cp:~$ kubectl -n low-usage-limit delete deployment hog limited-hog
```

```
deployment.apps "hog" deleted
deployment.apps "limited-hog" deleted
```

```
student@cp:~$ kubectl delete deployment hog
```

```
deployment.apps "hog" deleted
```


Chapter 5

APIs and Access



5.1	API Access	106
5.2	Annotations	111
5.3	Working with A Simple Pod	112
5.4	kubectl and API	113
5.5	Swagger and OpenAPI	120
5.6	Labs	122

5.1 API Access

API Access

- API driven architecture
- API groups
- RESTful style
- Standard HTTP verbs
- Deprecation process now being honored

Kubernetes has a powerful **REST**-based API. The entire architecture is API driven. Knowing where to find resource endpoints and understanding how the API changes between versions can be important to ongoing administrative tasks, as there is much ongoing change and growth. Starting with v1.16 deprecated objects are no longer honored by the API server.

As we learned in the Architecture chapter, the main agent for communication between cluster agents and from outside the cluster is the `kube-apiserver`. A `curl` query to the agent will expose the current API groups. Groups may have multiple versions which evolve independently of other groups, and follow a domain-name format with several names reserved, such as single-word domains, the empty group and any name ending in `.k8s.io`.

RESTful

- Responds to typical HTTP verbs (GET, POST, DELETE ...)
- Allows easy interaction with other ecosystems
- Scripting and interaction with deployment tools
- User impersonation headers

kubectl makes API calls on your behalf. You can also make calls externally, using **curl** or other program. With the appropriate certs and keys, you can make requests or pass **json** files to make configuration changes.

```
$ curl --cert userbob.pem --key userBob-key.pem \  
--cacert /path/to/ca.pem \  
https://k8sServer:6443/api/v1/pods
```

The ability to impersonate other users or groups, subject to RBAC configuration, allows a manual override authentication. This can be helpful for debugging authorization policies of other users.

Checking Access

- Several ways to authenticate
- auth can-i subcommand to query authorization
- Accepts can-i and reconcile arguments
- More in Security chapter to follow.

While there is more detail on security in a later chapter, it is helpful to check the current authorizations, both as an admin, as well as another user. The following shows what user bob could do in the default namespace and the developer namespace:

```
$ kubectl auth can-i create deployments
```

```
yes
```

```
$ kubectl auth can-i create deployments --as bob
```

```
no
```

```
$ kubectl auth can-i create deployments --as bob --namespace developer
```

```
yes
```

There are currently three APIs which can be applied to set who and what can be queried.

- `SelfSubjectAccessReview`
Access review for any user, helpful for delegating to others.
- `LocalSubjectAccessReview`
Review is restricted to a specific namespace.
- `SelfSubjectRulesReview`
A review which shows allowed actions for a user within a particular namespace.

The use of `reconcile` allows a check of authorization necessary to create an object from a file. No output indicates the creation would be allowed.

Optimistic Concurrency

- Currently leverage JSON
- `resourceVersion`
- Clients must handle 409 CONFLICT Errors

The default serialization for API calls must be JSON. There is an effort to use **Google's** protobuf serialization, but this remains experimental. While we may work with files in YAML format, they are converted to and from JSON.

Kubernetes uses the `resourceVersion` value to determine API updates and implement optimistic concurrency. In other words, an object is not locked from the time it has been read until the object is written.

Instead, upon an updated call to an object, the `resourceVersion` is checked and a 409 CONFLICT is returned should the number have changed. The `resourceVersion` is currently backed via the `modifiedIndex` parameter in the `etcd` database, and is unique to the namespace, kind and server. Operations which do not change an object such as WATCH or GET do not update this value.

5.2 Annotations

Using Annotations

- Distinct from Labels
- Non-identifying metadata
- Key/value maps
- Metadata otherwise held in exterior databases
- Useful for third-party automation

Labels are used to work with objects or collections of objects; annotations are not.

Instead annotations allow for metadata to be included with an object that may be helpful outside of Kubernetes object interaction. Similar to labels, they are key to value maps. They also are able to hold more information, and more human readable information than labels.

Having this kind of metadata can be used to track information such as a timestamp, pointers to related objects from other ecosystems, or even an email from the developer responsible for that object's creation.

The annotation data could otherwise be held in an exterior database, but that would limit the flexibility of the data. The more this metadata is included, the easier to integrate management and deployment tools or shared client libraries.

For example, to annotate only Pods within a namespace, then overwrite the annotation and finally delete it:

```
$ kubectl annotate pods --all description='Production Pods' -n prod  
$ kubectl annotate --overwrite pod webpod description="Old Production Pods" -n prod  
$ kubectl -n prod annotate pod webpod description=
```

5.3 Working with A Simple Pod

Simple Pod

- Lowest compute unit of K8s
- Typically multiple containers grouped together
- Created from PodSpec
- Few required, many optional
 - apiVersion
Must match existing API group
 - kind
The type of object to create
 - metadata
At least a name
 - spec
What to create and parameters

As discussed earlier, a Pod is the lowest compute unit and individual object we can work with in Kubernetes. It can be a single container, but often it will consist of a primary application container and one or more supporting containers.

Below is an example of a simple pod manifest in YAML format. You can see the `apiVersion`, the `kind`, the `metadata`, and its `spec`, which define the container that actually runs in this pod:

YAML

```
1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: firstpod
5 spec:
6   containers:
7     - image: nginx
8       name: stan
9
```

You can use the `kubectl create` command to create this pod in Kubernetes. Once it is created, you can check its status with `kubectl get pods`. Output is omitted to save space:

```
$ kubectl create -f simple.yaml
$ kubectl get pods
$ kubectl get pod firstpod -o yaml
$ kubectl get pod firstpod -o json
```

5.4 kubectl and API

Manage API Resources with kubectl

- API exposed via RESTful interface
- Use **curl** to access and test
- Use verbose mode
- Leverages HTTP verbs

Kubernetes exposes resources via RESTful API calls, which allows all resources to be managed via HTTP, JSON or even XML. the typical protocol being HTTP. The state of the resources can be changed using standard HTTP verbs (e.g. GET, POST, PATCH, DELETE, etc.).

kubectl has a verbose mode argument which shows details from where the command gets and updates information. Other output includes **curl** commands you could use to obtain the same result. While the verbosity accepts levels from zero to any number, there is currently no verbosity value greater than ten. You can check this out for **kubectl get**. The output below has been formatted for clarity:

```
$ kubectl --v=10 get pods firstpod
.....
I1215 17:46:47.860958 29909 round_trippers.go:417]
curl -k -v -XGET -H "Accept: application/json"
-H "User-Agent: kubectl/v1.8.5 (linux/amd64) kubernetes/cce11c6"
https://10.128.0.3:6443/api/v1/namespaces/default/pods/firstpod
....
```

Access From Outside The Cluster

- Can use **curl** from outside cluster
- Must use SSL/TLS for secure access
- Information found in `~/.kube/config`
- View server information via **kubectl config view**

The primary tool used from the command line will be **kubectl**, which calls **curl** on your behalf. You can also use the **curl** command from outside the cluster to view or make changes.

The basic server information, with redacted TLS certificate information can be found in the output of

```
$ kubectl config view
```

If you view verbose output from a previous page, you will note that the first line references a config file where this information is pulled from, `~/.kube/config`.

```
I1215 17:35:46.725407 27695 loader.go:357] Config loaded from file /home/student/.kube/config
```

Without the certificate authority, key and certificate from this file, only insecure **curl** commands can be used, which will not expose much due to security settings. We will use curl to access our cluster using TLS in an upcoming lab.

~/.kube/config

```

apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: LS0tLS1CRUdF.....
    server: https://10.128.0.3:6443
    name: kubernetes
contexts:
- context:
    cluster: kubernetes
    user: kubernetes-admin
    name: kubernetes-admin@kubernetes
current-context: kubernetes-admin@kubernetes
kind: Config
preferences: {}
users:
- name: kubernetes-admin
  user:
    client-certificate-data: LS0tLS1CRUdJTib.....
    client-key-data: LS0tLS1CRUdJTi....

```

The output above shows 19 lines of output with each of the keys being heavily truncated. While the keys may look similar close examination shows them to be distinct.

- **apiVersion**
As with other objects, this instructs the `kube-apiserver` where to assign the data.
- **clusters**
This contains the name of the cluster as well as where to send the API calls. The `certificate-authority-data` is passed to authenticate the `curl` request.
- **contexts**
A setting which allows easy access to multiple clusters, possibly as various users, from one config file. It can be used to set `namespace`, `user`, and `cluster`.
- **current-context**
Shows which cluster and user `kubectl` would use. These settings can also be passed on a per-command basis.
- **kind**
Every object within Kubernetes must have this setting, in this case a declaration of object type `Config`.
- **preferences**
Currently not used optional settings for the `kubectl` command, such as colorizing output.
- **users**
A nickname associated with client credentials which can be client key and certificate, username and password, and a token. Token and username/password are mutually exclusive. These can be configured via the `kubectl config set-credentials` command.

Namespaces

- Linux kernel feature
 - Segregates system resources
 - Core functionality of containers
- API Object
 - Four namespaces to begin with:
 - * default
 - * kube-node-lease
 - * kube-public
 - * kube-system
 - - - **all-namespaces**

The term **namespace** is used to both reference the Kernel feature as well as the segregation of API objects by Kubernetes. Both are means to keep resources distinct.

Every API call includes a namespace, using default if not otherwise declared: <https://10.128.0.3:6443/api/v1/namespaces/default/pods>.

Namespaces are intended to isolate multiple groups and the resources they have access to work with via quotas. Eventually, access control policies will work on namespace boundaries as well. One could use Labels to group resources for administrative reasons.

There are four namespaces when a cluster is first created.

- default
This is where all resources are assumed unless set otherwise.
- kube-node-lease The namespace where worker node lease information is kept.
- kube-public
A namespace readable by all, even those not authenticated. General information is often included in this namespaces.
- kube-system
Contains infrastructure pods.

Should you want to see all resources on a system you must pass the `--all-namespaces` option to the **kubectl** command.

Working with Namespaces

```
$ kubectl get ns  
$ kubectl create ns linuxcon  
$ kubectl describe ns linuxcon  
$ kubectl get ns/linuxcon -o yaml  
$ kubectl delete ns/linuxcon
```

The above commands show how to view, create and delete namespaces. Note that the **describe** subcommand shows several settings such as Labels, Annotations, resource quotas, and resource limits which we will discuss later in the course. Once a namespace has been created you can reference via YAML when creating resource:

```
$ cat redis.yaml
```

YAML redis.yaml

```
1 apiVersion: V1  
2 kind: Pod  
3 metadata:  
4   name: redis  
5   namespace: linuxcon  
6 ...  
7
```

API Resources with kubectl

- All available via **kubectl**
- `kubectl [command] [type] [Name] [flag]`
- `kubectl help` for more information
- Abbreviated names

All API resources exposed are available via **kubectl**. Expect the list to change.

- all
- certificatesigningrequests (csr)
- clusterrolebindings
- clusterroles
- clusters (valid only for federation apiservers)
- componentstatuses (cs)
- configmaps (cm)
- controllerrevisions
- cronjobs
- customresourcedefinition (crd)
- daemonsets (ds)
- deployments (deploy)
- endpoints (ep)
- events (ev)
- horizontalpodautoscalers (hpa)
- ingresses (ing)
- jobs
- limitranges (limits)
- namespaces (ns)
- networkpolicies (netpol)
- nodes (no)
- persistentvolumeclaims (pvc)
- persistentvolumes (pv)
- poddisruptionbudgets (pdb)
- podpreset
- pods (po)
- podsecuritypolicies (psp)
- podtemplates
- replicaset (rs)
- replicationcontrollers (rc)
- resourcequotas (quota)
- rolebindings
- roles
- secrets
- serviceaccounts (sa)
- services (svc)
- statefulsets
- storageclasses

Additional Resource Methods

- Various Endpoints
- CLI --help
- Online documentation

In addition to basic resource management via REST, the API also provides some extremely useful endpoints for certain resources.

For example, you can access the logs of a container, exec into it, and watch changes to it with the following endpoints:

```
$ curl --cert /tmp/client.pem --key /tmp/client-key.pem \  
--cacert /tmp/ca.pem -v -XGET \  
https://10.128.0.3:6443/api/v1/namespaces/default/pods/firstpod/log
```

This would be the same as the following. If the container does not have any standard out, there would be no logs.

```
$ kubectl logs firstpod
```

Other calls you could make, following the various API groups on your cluster:

```
GET /api/v1/namespaces/{namespace}/pods/{name}/exec  
GET /api/v1/namespaces/{namespace}/pods/{name}/log  
GET /api/v1/watch/namespaces/{namespace}/pods/{name}
```

5.5 Swagger and OpenAPI

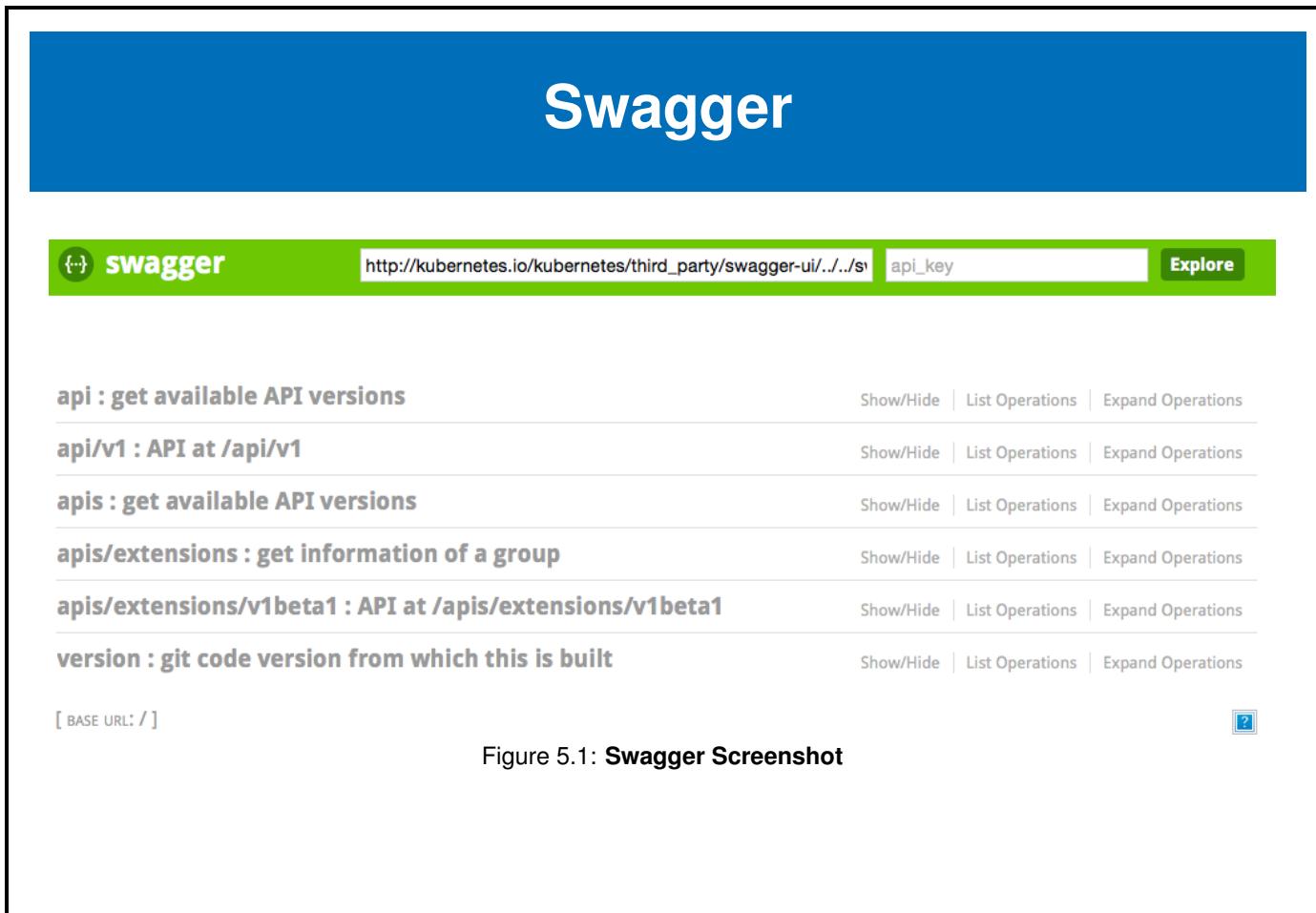


Figure 5.1: **Swagger Screenshot**

The entire Kubernetes API was built using a **Swagger** specification. This has been evolving towards the **OpenAPI** initiative. It is extremely useful, as it allows, for example, to auto-generate client code. All the stable resources definitions are available on the documentation site.

You can browse some of the API groups via a Swagger UI at <https://swagger.io/specification/>.

API Maturity

- Versioning of API levels for easier growth
- Not directly tied to software versioning
- Versions imply level of support
 - Alpha
 - Beta
 - Stable

The use of API groups and different versions allows for development to advance without changes to an existing group of APIs. This allows for easier growth and separation of work among separate teams. While there is an attempt to maintain some consistency between API and software versions they are only indirectly linked.

The use of JSON and **Google's** Protobuf serialization scheme will follow the same release guidelines.

An Alpha level release, noted with `alpha` in the names, may be buggy and is disabled by default. Features could change or disappear at any time, and backward compatibility is not guaranteed. Only use these features on a test cluster which can often be rebuilt.

The Beta levels, found with `beta` in the names, has more well tested code and is enabled by default. It also ensures that as changes move forward they will be tested for backwards compatibility between versions. It has not been adopted and tested enough to be called stable. Expect some bugs and issues.

Use of the Stable version, denoted by only an integer which may be preceded by the letter `v`, is for stable APIs. At the moment `v1` is the only stable version.

5.6 Labs

Exercise 5.1: Configuring TLS Access

Overview

Using the Kubernetes API, **kubectl** makes API calls for you. With the appropriate TLS keys you could run **curl** as well use a **golang** client. Calls to the `kube-apiserver` get or set a PodSpec, or desired state. If the request represents a new state the **Kubernetes Control Plane** will update the cluster until the current state matches the specified state. Some end states may require multiple requests. For example, to delete a ReplicaSet, you would first set the number of replicas to zero, then delete the ReplicaSet.

An API request must pass information as JSON. **kubectl** converts `.yaml` to JSON when making an API request on your behalf. The API request has many settings, but must include `apiVersion`, `kind` and `metadata`, and `spec` settings to declare what kind of container to deploy. The `spec` fields depend on the object being created.

We will begin by configuring remote access to the `kube-apiserver` then explore more of the API.

1. Begin by reviewing the **kubectl** configuration file. We will use the three certificates and the API server address.

```
student@cp:~$ less $HOME/.kube/config
```

<output_omitted>

2. We will create a variables using certificate information. You may want to double-check each parameter as you set it. Begin with setting the `client-certificate-data` key.

```
student@cp:~$ export client=$(grep client-cert $HOME/.kube/config |cut -d" " -f 6)
```

```
student@cp:~$ echo $client
```

```
LS0tLS1CRUdJTiBDRVJUSUZJQOFURS0tLS0tCk1JSUM4akNDQWRxZ0F3SUJ
BZ01JRy9wbC9rWEpNdmd3RFFZSktvWklodmN0QVFTEJRQXdGVEVUTUJFR0
ExVUUKQXhNS2EzVmLaWEp1WIhSbGN6QWVGdzB4TnpFeU1UTXh0e1EyTXpKY
UZ3MHhPREV5TVRNeE56UTJNe1JhTURReApGekFWQmdOVkJBb1REbk41YzNS
<output_omitted>
```

3. Almost the same command, but this time collect the `client-key-data` as the key variable.

```
student@cp:~$ export key=$(grep client-key-data $HOME/.kube/config |cut -d " " -f 6)
```

```
student@cp:~$ echo $key
```

<output_omitted>

4. Finally set the auth variable with the `certificate-authority-data` key.

```
student@cp:~$ export auth=$(grep certificate-authority-data $HOME/.kube/config |cut -d " " -f 6)
```

```
student@cp:~$ echo $auth
```

<output_omitted>

5. Now encode the keys for use with **curl**.

```
student@cp:~$ echo $client | base64 -d - > ./client.pem
student@cp:~$ echo $key | base64 -d - > ./client-key.pem
student@cp:~$ echo $auth | base64 -d - > ./ca.pem
```

6. Pull the API server URL from the config file. Your hostname or IP address may be different.

```
student@cp:~$ kubectl config view |grep server
```

```
server: https://k8scp:6444
```

7. Use **curl** command and the encoded keys to connect to the API server. Use your hostname, or IP, found in the previous command, which may be different than the example below.

```
student@cp:~$ curl --cert ./client.pem \
--key ./client-key.pem \
--cacert ./ca.pem \
https://k8scp:6443/api/v1/pods
```

```
{
  "kind": "PodList",
  "apiVersion": "v1",
  "metadata": {
    "selfLink": "/api/v1/pods",
    "resourceVersion": "239414"
  },
<output_omitted>
```

8. If the previous command was successful, create a JSON file to create a new pod. Remember to use **find** and search for this file in the tarball output, it can save you some typing.

```
student@cp:~$ cp /home/student/LFS458/SOLUTIONS/s_05/curlpod.json .
```

```
student@cp:~$ vim curlpod.json
```

```
{
  "kind": "Pod",
  "apiVersion": "v1",
  "metadata": {
    "name": "curlpod",
    "namespace": "default",
    "labels": {
      "name": "examplepod"
    }
  },
  "spec": {
    "containers": [
      {
        "name": "nginx",
        "image": "nginx",
        "ports": [{"containerPort": 80}]
      }
    ]
  }
}
```

9. The previous **curl** command can be used to build a XPOST API call. There will be a lot of output, including the scheduler and taints involved. Read through the output. In the last few lines the phase will probably show Pending, as it's near the beginning of the creation process.

```
student@cp:~$ curl --cert ./client.pem \
--key ./client-key.pem --cacert ./ca.pem \
https://k8scp:6443/api/v1/namespaces/default/pods \
-XPOST -H'Content-Type: application/json' \
-d@curlpod.json

{
  "kind": "Pod",
  "apiVersion": "v1",
  "metadata": {
    "name": "curlpod",
  }
<output_omitted>
```

10. Verify the new pod exists and shows a Running status.

```
student@cp:~$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
curlpod	1/1	Running	0	45s

✍ Exercise 5.2: Explore API Calls

1. One way to view what a command does on your behalf is to use **strace**. In this case, we will look for the current endpoints, or targets of our API calls. Install the tool, if not present.

```
student@cp:~$ sudo apt-get install -y strace
```

```
student@cp:~$ kubectl get endpoints
```

NAME	ENDPOINTS	AGE
kubernetes	10.128.0.3:6443	3h

2. Run this command again, preceded by **strace**. You will get a lot of output. Near the end you will note several **openat** functions to a local directory, `/home/student/.kube/cache/discovery/k8scp_6443`. If you cannot find the lines, you may want to redirect all output to a file and grep for them. This information is cached, so you may see some differences should you run the command multiple times. As well your IP address may be different.

```
student@cp:~$ strace kubectl get endpoints
```

```
execve("/usr/bin/kubectl", ["kubectl", "get", "endpoints"], [/*.....
.....
openat(AT_FDCWD, "/home/student/.kube/cache/discovery/k8scp_6443...
<output_omitted>
```

3. Change to the parent directory and explore. Your endpoint IP will be different, so replace the following with one suited to your system.

```
student@cp:~$ cd /home/student/.kube/cache/discovery/
```

```
student@cp:~/.kube/cache/discovery$ ls
```

```
k8scp_6443
```

```
student@cp:~/.kube/cache/discovery$ cd k8scp_6443/
```

4. View the contents. You will find there are directories with various configuration information for kubernetes.

```
student@cp:~/.kube/cache/discovery/k8scp_6443$ ls
```

```

admissionregistration.k8s.io  certificates.k8s.io          node.k8s.io
apiextensions.k8s.io         coordination.k8s.io        policy
apiregistration.k8s.io       cilium.io                rbac.authorization.k8s.io
apps                         discovery.k8s.io          scheduling.k8s.io
authentication.k8s.io        events.k8s.io           servergroups.json
authorization.k8s.io         extensions               storage.k8s.io
autoscaling                   flowcontrol.apiserver.k8s.io v1
batch                         networking.k8s.io

```

5. Use the find command to list out the subfiles. The prompt has been modified to look better on this page.

```
student@cp:./k8scp_6443$ find .
```

```
.
./storage.k8s.io
./storage.k8s.io/v1beta1
./storage.k8s.io/v1beta1/serverresources.json
./storage.k8s.io/v1
./storage.k8s.io/v1/serverresources.json
./rbac.authorization.k8s.io
<output_omitted>
```

6. View the objects available in version 1 of the API. For each object, or kind:, you can view the verbs or actions for that object, such as create seen in the following example. Note the prompt has been truncated for the command to fit on one line. Some are HTTP verbs, such as GET, others are product specific options, not standard HTTP verbs. The command may be **python**, depending on what version is installed.

```
student@cp:.$ python3 -m json.tool v1/serverresources.json
```

```

1  {
2      "apiVersion": "v1",
3      "groupVersion": "v1",
4      "kind": "APIResourceList",
5      "resources": [
6          {
7              "kind": "Binding",
8              "name": "bindings",
9              "namespaced": true,
10             "singularName": "",
11             "verbs": [
12                 "create"
13             ]
14         },
15     <output_omitted>
16

```

7. Some of the objects have shortNames, which makes using them on the command line much easier. Locate the shortName for endpoints.

```
student@cp:.$ python3 -m json.tool v1/serverresources.json | less
```

JSON

serverresources.json

```

1 ...
2 {
3   "kind": "Endpoints",
4   "name": "endpoints",
5   "namespaced": true,
6   "shortNames": [
7     "ep"
8   ],
9   "singularName": "",
10  "verbs": [
11    "create",
12    "delete",
13 ...
14

```

8. Use the shortName to view the endpoints. It should match the output from the previous command.

```
student@cp:~$ kubectl get ep
```

NAME	ENDPOINTS	AGE
kubernetes	10.128.0.3:6443	3h

9. We can see there are 37 objects in version 1 file.

```
student@cp:~$ python3 -m json.tool v1/serverresources.json | grep kind
```

```

"kind": "APIResourceList",
"kind": "Binding",
"kind": "ComponentStatus",
"kind": "ConfigMap",
"kind": "Endpoints",
"kind": "Event",
<output_omitted>

```

10. Looking at another file we find nine more.

```
student@cp:~$ python3 -m json.tool apps/v1/serverresources.json | grep kind
```

```

"kind": "APIResourceList",
"kind": "ControllerRevision",
"kind": "DaemonSet",
"kind": "DaemonSet",
"kind": "Deployment",
<output_omitted>

```

11. Delete the curlpod to recoup system resources.

```
student@cp:~$ kubectl delete po curlpod
```

```
pod "curlpod" deleted
```

12. Take a look around the other files in this directory as time permits.

Chapter 6

API Objects



6.1	API Objects	128
6.2	The v1 Group	129
6.3	API Resources	131
6.4	RBAC APIs	136
6.5	Labs	137

6.1 API Objects

Overview

- Ongoing growth in API objects
- Track release notes to find new objects
- In this chapter we will introduce common API objects:
 - Deployment the typical object used
 - DaemonSets, ReplicaSets now apps/v1
 - StatefulSets (once called PetSets) part of apps/v1 since v1.9
 - Jobs and CronJob now batch/v1
 - RBAC moved from v1alpha1 all the way to v1 in one release
- Explain which API group contains these new API objects.
- Find additional resources to start using the new API objects.

This chapter is about additional API resources or objects. We will learn about resources in the v1 API group, among others. Stability increases and code becomes more stable as objects move from alpha versions, to beta, then v1 indicating stability.

DaemonSets, which ensure a Pod on every node, and StatefulSets, which stick a container to a node and otherwise act like a deployment, have progressed to apps/v1 stability.

As a fast moving project keeping track of changes, and possible changes can be an important part of ongoing system administration. Release notes, as well as discussions to release notes can be found in version-dependent sub-directories at: <https://github.com/kubernetes/enhancements/>. For example, the release feature status can be found here: <https://kubernetes.io/docs/setup/release/notes/>.

Starting with v1.16 deprecated API object versions will respond with an error instead of being accepted. This is an important change from historic behavior.

6.2 The v1 Group

v1 API Group

- Pod
- Node
- Service Account
- Resource Quota
- Endpoint
- More added with each release

The v1 API is no longer a single group, but rather a collection of groups for each main object category. For example there is a v1 group, a storage.k8s.io/v1 group, and rbac.authorization.k8s.io/v1 etc... Currently there are many v1 groups.

We have touched on several objects in lab exercises. Here are details for some of them:

- **Node**

Represents a machine (physical or virtual) that is part of your Kubernetes cluster. You can get more information about nodes with the `kubectl get nodes` command. You can turn on and off the scheduling to a node with the `kubectl cordon/uncordon` commands.

- **Service Account**

Provides an identifier for processes running in a pod to access the API server and performs actions that it is authorized to do.

- **Resource Quota**

It is an extremely useful tool, allowing you to define quotas per namespace. For example, if you want to limit a specific namespace to only run a given number of pods, you can write a `resourcequota` manifest, create it with `kubectl` and the quota will be enforced.

- **Endpoint**

Generally, you do not manage endpoints. They represent the set of IPs for pods that match a particular service. They are handy when you want to check that a service actually matches some running pods. If an endpoint is empty, then it means that there are no matching pods and something is most likely wrong with your service definition.

Discovering API Groups

```
$ curl https://localhost:6443/apis \
--header "Authorization: Bearer $token" -k

{
  "kind": "APIGroupList",
  "apiVersion": "v1",
  "groups": [
    {
      "name": "apiregistration.k8s.io",
      "versions": [
        {
          "groupVersion": "apiregistration.k8s.io/v1",
          "version": "v1"
        }
      ],
      "preferredVersion": {
        "groupVersion": "apiregistration.k8s.io/v1",
        "version": "v1"
      }
    }
  ]
}
```

We can take a closer look at the output of request for current APIs. Each of the name values can be appended to the URL to see details of that group. For example you could drill down to find included objects at this URL: <https://localhost:6443/apis/apiregistration.k8s.io/v1beta1>

If you follow this URL you will find only one resource, with a name of apiservices. If it seems to be listed twice the lower output is for status. You'll note there are different verbs or actions for each. Another entry is if this object is namespaced, or restricted to only one namespace. In this case it is not.

You could **curl** each of these URLs and discover additional API objects, their characteristics and associated verbs.

6.3 API Resources

Deploying an Application

- Deployment
- ReplicaSet
- Pod

Using the **kubectl create** command we can quickly deploy an application. We have looked at the Pods created running the application, like **nginx**. Looking closer you will find that a Deployment was created which manages a ReplicaSet which then deploys the Pod. Lets take a closer look at each object.

- **Deployment** - A controller which manages the state of ReplicaSets and the pods within. The higher level control allows for more flexibility with upgrades and administration. Unless you have a good reason, use a deployment.
- **ReplicaSet** - Orchestrates individual Pod life cycle and updates. These are newer versions of Replication Controllers which differ only in selector support.
- **Pod** - As we've mentioned the lowest unit we can manage, runs the application container, possibly support containers.

DaemonSets

- Ensures every node runs a single pod
- Similar to ReplicaSet
- Often used for logging, metrics and security pods.
- Can be configured to avoid nodes

Should you want to have a logging application on every node a DaemonSet may be a good choice. The controller ensures that a single pod, of the same type, runs on every node in the cluster. When a new node is added to the cluster a Pod, same as deployed on the other nodes, is started. When the node is removed the DaemonSet makes sure the local Pod is deleted.

As usual, you get all the CRUD operations via **kubectl**:

```
$ kubectl get daemonsets  
$ kubectl get ds
```

StatefulSet

- Similar to Deployment
- Ensures unique pods
- Guarantee ordering

Pods deployed using a StatefulSet use the same Pod specification. How this is different than a Deployment is that a StatefulSet considers each Pod as unique and provides ordering to Pod deployment.

In order to track each Pod as a unique object the controller uses identity composed of stable storage, stable network identity, and an ordinal. This identity remains with the node regardless to which node the Pod is running on at any one time.

The default deployment scheme is sequential starting with 0, such as app-0, app-1, app-2 etc. A following Pod will not launch until the current Pod reaches a running and ready state. They are not deployed in parallel.

Autoscaling

- Agents which add or remove resources from the cluster.
- Horizontal Pod Autoscaling (HPA)
 - Scale based on current CPU usage, or custom metric
 - Must have **Metrics Server** or custom component running
- Vertical Pod Autoscaler (under development)
- Cluster Autoscaler (CA)
 - Add or remove nodes based on utilization
 - Makes request to cloud provider
 - Pods which cannot be evicted prevent scale-down

In the autoscaling group we find the **Horizontal Pod Autoscalers (HPA)**. This is a stable resource. HPAs automatically scale Replication Controllers, ReplicaSets, or Deployments based on a target of 80% CPU usage by default. The usage is checked by kubelet every 15 seconds and retrieved by Metrics Server API call every minute. HPA checks with Metrics Server every 15 seconds. Should a Pod be added it takes immediate action and when needs to be removed HPA waits 300 seconds before further action.

Other metrics can be used and queried via REST. The autoscaler does not collect the metrics, it only makes a request for the aggregated information and increases or decreases the number of replicas to match the configuration.

The Cluster Autoscaler (CA) adds or removes nodes to the cluster based off of inability to deploy a Pod or having nodes with low utilization for at least 10 minutes. This allows dynamic requests of resources from the cloud provider and minimizes expense for unused nodes. If you are using CA nodes should be added and removed through `cluster-autoscaler-` commands. Scale-up and down of nodes is checked every 10 seconds, but decisions are made on a node every 10 minutes. Should a scale-down fail the group will be rechecked in 3 minutes, with the failing node being eligible in five minutes. The total time to allocate a new node is largely dependent on the cloud provider.

Another project still under development is the Vertical Pod Autoscaler. This component will adjust the amount of CPU and memory requested by Pods.

Jobs

- Part of Batch API group
- Jobs run Pod until number of completions reached
 - Batch processing or one-off Pods
 - Ensure specified number of pods successfully terminate
 - Can run multiple Pods in parallel
- Cronjob to run Job on regular basis
 - Creates a Job about once per executing time
 - Some issues, job should be idempotent
 - Can run in serial or parallel
 - Same time syntax as Linux cron job

Jobs are part of the batch API group. They are used to run a set number of pods to completion. If a pod fails, it will be restarted until the number of completion is reached.

While they can be seen as a way to do batch processing in Kubernetes, they can also be used to run one-off pods. A Job specification will have a parallelism and a completion key. If omitted, they will be set to one. If they are present, the parallelism number will set the number of pods that can be running concurrently and the completion number will set how many pods need to run successfully for the Job itself to be considered done. Several Job patterns can be implemented, like a traditional work queue.

Cronjobs work in the similar manner to **Linux** jobs with the same time syntax. There are some cases where a job would not be run during a time period or could run twice, as a result the requested Pod should be idempotent.

An option spec field is `.spec.concurrencyPolicy` which determines how to handle existing jobs should the time segment expire. If set to `Allow`, the default, another concurrent job will be run. If set to `Forbid` the current job continues and the new job is skipped. A value of `Replace` cancels the current job and starts a new job in its place.

6.4 RBAC APIs

RBAC

- rbac.authorization.k8s.io
- Provide resources
 - ClusterRole
 - ClusterRoleBinding
 - RoleBinding
 - Role
- Often combined with quotas for typical production deployments

The last API resources that we will look at are in the `rbac.authorization.k8s.io` group. We actually have four resources: `ClusterRole`, `Role`, `ClusterRoleBinding`, and `RoleBinding`. They are used for **Role Based Access Control (RBAC)** to Kubernetes.

```
$ curl localhost:8080/apis/rbac.authorization.k8s.io/v1
```

```
...  
  "groupVersion": "rbac.authorization.k8s.io/v1",  
  "resources": [  
    "kind": "ClusterRoleBinding"  
    "kind": "ClusterRole"  
    "kind": "RoleBinding"  
    "kind": "Role"  
  ...
```

These resources allow us to define Roles within a cluster and associate users to these Roles. For example, we can define a Role for someone who can only read pods in a specific namespace, or a Role that can create deployments, but no services.



Please Note

More on RBAC is covered later in the Security chapter.

6.5 Labs

Exercise 6.1: RESTful API Access

Overview

We will continue to explore ways of accessing the control plane of our cluster. In the security chapter we will discuss there are several authentication methods, one of which is use of a Bearer token. We will work with one then deploy a local proxy server for application-level access to the Kubernetes API.

We will use the **curl** command to make API requests to the cluster, in an insecure manner. Once we know the IP address and port, then the token we can retrieve cluster data in a RESTful manner. By default most of the information is restricted, but changes to authentication policy could allow more access.

- First we need to know the IP and port of a node running a replica of the API server. The cp system will typically have one running. Use **kubectl config view** to get overall cluster configuration, and find the server entry. This will give us both the IP and the port.

```
student@cp:~$ kubectl config view
```

```
apiVersion: v1
clusters:
- cluster:
  certificate-authority-data: DATA+OMITTED
  server: https://k8scp:6443
  name: kubernetes
<output_omitted>
```

- The creation of token secrets by Kubernetes is no longer automatic in recent releases. It then falls on the user to design them as desired. Use the command shown below to create the token.

```
student@cp:~$ export token=$(kubectl create token default)
```

- Test to see if you can get basic API information from your cluster. We will pass it the server name and port, the token and use the **-k** option to avoid using a cert.

```
student@cp:~$ curl https://k8scp:6443/apis --header "Authorization: Bearer $token" -k
```

```
{
  "kind": "APIGroupList",
  "apiVersion": "v1",
  "groups": [
    {
      "name": "apiregistration.k8s.io",
      "versions": [
        {
          "groupVersion": "apiregistration.k8s.io/v1",
          "version": "v1"
        }
      ]
    }
  ]
}<output_omitted>
```

- Try the same command, but look at API v1. Note that the path has changed to api.

```
student@cp:~$ curl https://k8scp:6443/api/v1 --header "Authorization: Bearer $token" -k
```

```
<output_omitted>
```

5. Now try to get a list of namespaces. This should return an error. It shows our request is being seen as `system:serviceaccount:`, which does not have the RBAC authorization to list all namespaces in the cluster.

```
student@cp:~$ curl \
  https://k8scp:6443/api/v1/namespaces --header "Authorization: Bearer $token" -k

<output_omitted>
  "message": "namespaces is forbidden: User \\"system:serviceaccount:default...
<output_omitted>
```

Exercise 6.2: Using the Proxy

Another way to interact with the API is via a proxy. The proxy can be run from a node or from within a Pod through the use of a sidecar. In the following steps we will deploy a proxy listening to the loopback address. We will use `curl` to access the API server. If the `curl` request works, but does not from outside the cluster, we have narrowed down the issue to authentication and authorization instead of issues further along the API ingestion process.

1. Begin by starting the proxy. It will start in the foreground by default. There are several options you could pass. Begin by reviewing the help output.

```
student@cp:~$ kubectl proxy -h

Creates a proxy server or application-level gateway between localhost
and the Kubernetes API Server. It also allows serving static content
over specified HTTP path. All incoming data enters through one port
and gets forwarded to the remote kubernetes API Server port, except
for the path matching the static content path.

Examples:
  # To proxy all of the kubernetes api and nothing else, use:

  $ kubectl proxy --api-prefix=/
<output_omitted>
```

2. Start the proxy while setting the API prefix, and put it in the background. You may need to use `enter` to view the prompt. Take note of the process ID, 225000 in the example below, we'll use it to kill the process when we are done.

```
student@cp:~$ kubectl proxy --api-prefix=/ &

[1] 22500
Starting to serve on 127.0.0.1:8001
```

3. Now use the same `curl` command, but point toward the IP and port shown by the proxy. The output should be the same as without the proxy, but may be formatted differently.

```
student@cp:~$ curl http://127.0.0.1:8001/api/
<output_omitted>
```

4. Make an API call to retrieve the namespaces. The command did not work in the previous section due to permissions, but should work now as the proxy is making the request on your behalf.

```
student@cp:~$ curl http://127.0.0.1:8001/api/v1/namespaces

{
  "kind": "NamespaceList",
```

```

"apiVersion": "v1",
"metadata": {
    "selfLink": "/api/v1/namespaces",
    "resourceVersion": "86902"
<output_omitted>

```

- Stop the proxy service as we won't need it any more. Use the process ID from a previous step. Your process ID may be different.

```
student@cp:~$ kill 22500
```

Exercise 6.3: Working with Jobs

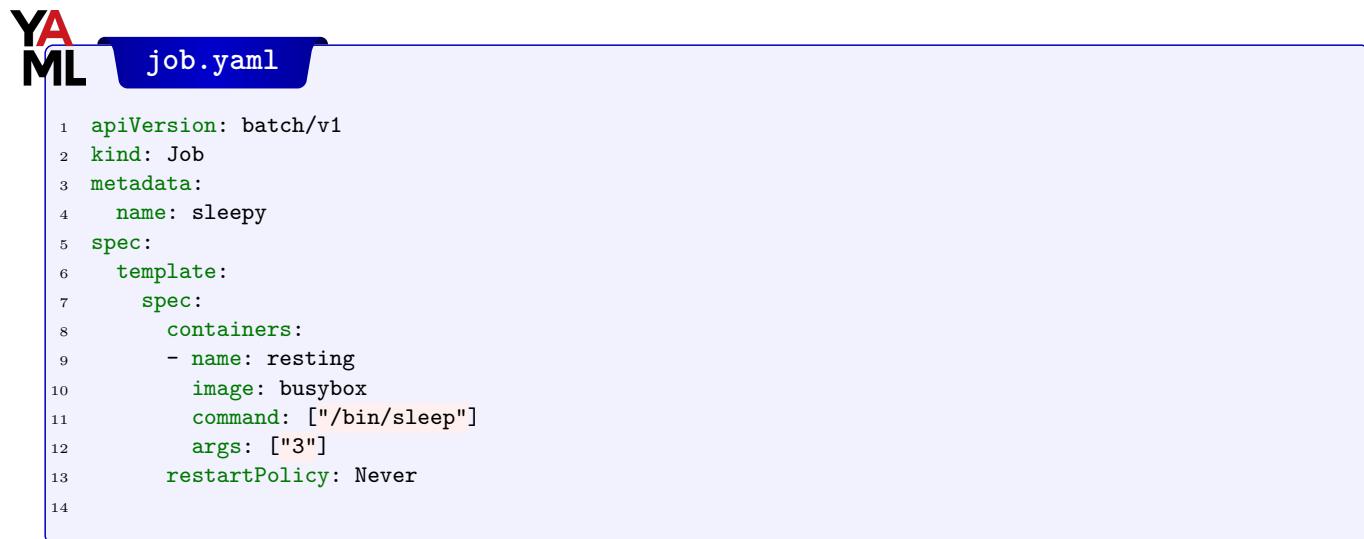
While most API objects are deployed such that they continue to be available there are some which we may want to run a particular number of times called a Job, and others on a regular basis called a CronJob

Create A Job

- Create a job which will run a container which sleeps for three seconds then stops.

```
student@cp:~$ cp /home/student/LFS458/SOLUTIONS/s_06/job.yaml .
```

```
student@cp:~$ vim job.yaml
```



```

YAML job.yaml
1 apiVersion: batch/v1
2 kind: Job
3 metadata:
4   name: sleepy
5 spec:
6   template:
7     spec:
8       containers:
9         - name: resting
10        image: busybox
11        command: ["/bin/sleep"]
12        args: ["3"]
13      restartPolicy: Never
14

```

- Create the job, then verify and view the details. The example shows checking the job three seconds in and then again after it has completed. You may see different output depending on how fast you type.

```
student@cp:~$ kubectl create -f job.yaml
```

```
job.batch/sleepy created
```

```
student@cp:~$ kubectl get job
```

NAME	COMPLETIONS	DURATION	AGE
sleepy	0/1	3s	3s

```
student@cp:~$ kubectl describe jobs.batch sleepy
```

```
Name:          sleepy
Namespace:    default
Selector:     controller-uid=24c91245-d0fb-11e8-947a-42010a800002
Labels:       controller-uid=24c91245-d0fb-11e8-947a-42010a800002
              job-name=sleepy
Annotations:  <none>
Parallelism: 1
Completions: 1
Start Time:   Thu, 23 Aug 2023 10:47:53 +0000
Completed At: Thu, 23 Aug 2023 10:48:00 +0000
Duration:     5s
Pods Statuses: 0 Running / 1 Succeeded / 0 Failed
<output_omitted>
```

student@cp:~\$ kubectl get job

NAME	COMPLETIONS	DURATION	AGE
sleepy	1/1	5s	17s

- View the configuration information of the job. There are three parameters we can use to affect how the job runs. Use **-o yaml** to see these parameters. We can see that backoffLimit, completions, and the parallelism. We'll add these parameters next.

student@cp:~\$ kubectl get jobs.batch sleepy -o yaml

```
<output_omitted>
uid: c2c3a80d-d0fc-11e8-947a-42010a800002
spec:
  backoffLimit: 6
  completions: 1
  parallelism: 1
  selector:
    matchLabels:
      <output_omitted>
```

- As the job continues to AGE in a completion state, delete the job.

student@cp:~\$ kubectl delete jobs.batch sleepy

```
job.batch "sleepy" deleted
```

- Edit the YAML and add the completions: parameter and set it to 5.

student@cp:~\$ vim job.yaml



```
1  <output_omitted>
2  metadata:
3    name: sleepy
4  spec:
5    completions: 5    #--Add this line
6    template:
7      spec:
8        containers:
```



```
9 <output_omitted>
10
```

6. Create the job again. As you view the job note that COMPLETIONS begins as zero of 5.

```
student@cp:~$ kubectl create -f job.yaml
```

```
job.batch/sleepy created
```

```
student@cp:~$ kubectl get jobs.batch
```

NAME	COMPLETIONS	DURATION	AGE
sleepy	0/5	5s	5s

7. View the pods that running. Again the output may be different depending on the speed of typing.

```
student@cp:~$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
sleepy-z5tnh	0/1	Completed	0	8s
sleepy-zd692	1/1	Running	0	3s
<output_omitted>				

8. Eventually all the jobs will have completed. Verify then delete the job.

```
student@cp:~$ kubectl get jobs
```

NAME	COMPLETIONS	DURATION	AGE
sleepy	5/5	26s	10m

```
student@cp:~$ kubectl delete jobs.batch sleepy
```

```
job.batch "sleepy" deleted
```

9. Edit the YAML again. This time add in the parallelism: parameter. Set it to 2 such that two pods at a time will be deployed.

```
student@cp:~$ vim job.yaml
```



job.yaml

```
1 <output_omitted>
2   name: sleepy
3   spec:
4     completions: 5
5     parallelism: 2 #<-- Add this line
6     template:
7       spec:
8 <output_omitted>
9
```

10. Create the job again. You should see the pods deployed two at a time until all five have completed.

```
student@cp:~$ kubectl create -f job.yaml
```

```
job.batch/sleepy created
```

```
student@cp:~$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
sleepy-8xwpc	1/1	Running	0	5s
sleepy-xjqnf	1/1	Running	0	5s
<output_omitted>				

```
student@cp:~$ kubectl get jobs
```

NAME	COMPLETIONS	DURATION	AGE
sleepy	3/5	11s	11s

11. Add a parameter which will stop the job after a certain number of seconds. Set the activeDeadlineSeconds: to 15. The job and all pods will end once it runs for 15 seconds. We will also increase the sleep argument to five, just to be sure does not expire by itself.

```
student@cp:~$ vim job.yaml
```

YAML

```
<output_omitted>
1   completions: 5
2   parallelism: 2
3   activeDeadlineSeconds: 15    #<-- Add this line
4   template:
5     spec:
6       containers:
7         - name: resting
8           image: busybox
9           command: ["/bin/sleep"]
10          args: ["5"]           #<-- Edit this line
11    <output_omitted>
12
13
```

12. Delete and recreate the job again. It should run for 15 seconds, usually 3/5, then continue to age without further completions.

```
student@cp:~$ kubectl delete jobs.batch sleepy
```

```
job.batch "sleepy" deleted
```

```
student@cp:~$ kubectl create -f job.yaml
```

```
job.batch/sleepy created
```

```
student@cp:~$ kubectl get jobs
```

NAME	COMPLETIONS	DURATION	AGE
sleepy	1/5	6s	6s

```
student@cp:~$ kubectl get jobs
```

NAME	COMPLETIONS	DURATION	AGE
sleepy	3/5	16s	16s

13. View the message: entry in the Status section of the object YAML output.

```
student@cp:~$ kubectl get job sleepy -o yaml
```

```
<output_omitted>
status:
  conditions:
  - lastProbeTime: 2023-08-23T10:48:00Z
    lastTransitionTime: 2023-08-23T10:48:00Z
    message: Job was active longer than specified deadline
    reason: DeadlineExceeded
    status: "True"
    type: Failed
  failed: 2
  startTime: 2023-08-23T10:48:00Z
  succeeded: 3
```

14. Delete the job.

```
student@cp:~$ kubectl delete jobs.batch sleepy
```

```
job.batch "sleepy" deleted
```

Create a CronJob

A CronJob creates a watch loop which will create a batch job on your behalf when the time becomes true. We Will use our existing Job file to start.

1. Copy the yaml file from the tarball.

```
student@cp:~$ cp /home/student/LFS458/SOLUTIONS/s_06/cronjob.yaml .
```

2. Verify the file to look like the annotated file shown below. Edit the lines mentioned below if needed. The three parameters we added will need to be removed. Other lines will need to be further indented if needed.

```
student@cp:~$ vim cronjob.yaml
```

Y
A
M
L

```
apiVersion: batch/v1
kind: CronJob          #<-- Update this line to CronJob
metadata:
  name: sleepy
spec:
  schedule: "*/* * * * *"      #<-- Add Linux style cronjob syntax
  jobTemplate:                #<-- New jobTemplate and spec move
    spec:
      template:              #<-- This and following lines move
        spec:                  #<-- four spaces to the right
          containers:
            - name: resting
```

YAML

```

13     image: busybox
14     command: ["/bin/sleep"]
15     args: ["5"]
16   restartPolicy: Never
17

```

3. Create the new CronJob. View the jobs. It will take two minutes for the CronJob to run and generate a new batch Job.

```
student@cp:~$ kubectl create -f cronjob.yaml
```

```
cronjob.batch/sleepy created
```

```
student@cp:~$ kubectl get cronjobs.batch
```

NAME	SCHEDULE	SUSPEND	ACTIVE	LAST SCHEDULE	AGE
sleepy	*/2 * * * *	False	0	<none>	8s

```
student@cp:~$ kubectl get jobs.batch
```

```
No resources found.
```

4. After two minutes you should see jobs start to run.

```
student@cp:~$ kubectl get cronjobs.batch
```

NAME	SCHEDULE	SUSPEND	ACTIVE	LAST SCHEDULE	AGE
sleepy	*/2 * * * *	False	0	21s	2m1s

```
student@cp:~$ kubectl get jobs.batch
```

NAME	COMPLETIONS	DURATION	AGE
sleepy-1539722040	1/1	5s	18s

```
student@cp:~$ kubectl get jobs.batch
```

NAME	COMPLETIONS	DURATION	AGE
sleepy-1539722040	1/1	5s	5m17s
sleepy-1539722160	1/1	6s	3m17s
sleepy-1539722280	1/1	6s	77s

5. Ensure that if the job continues for more than 10 seconds it is terminated. We will first edit the **sleep** command to run for 30 seconds then add the **activeDeadlineSeconds**: entry to the container.

```
student@cp:~$ vim cronjob.yaml
```

YAML

```

1 ....
2   jobTemplate:
3     spec:
4       template:
5         spec:

```

YAML

```

6      activeDeadlineSeconds: 10  #<-- Add this line
7      containers:
8          - name: resting
9      ....
10     command: ["/bin/sleep"]
11     args: ["30"]           #<-- Edit this line
12     restartPolicy: Never
13 ....
14

```

6. Delete and recreate the CronJob. It may take a couple of minutes for the batch Job to be created and terminate due to the timer.

```
student@cp:~$ kubectl delete cronjobs.batch sleepy
```

```
cronjob.batch "sleepy" deleted
```

```
student@cp:~$ kubectl create -f cronjob.yaml
```

```
cronjob.batch/sleepy created
```

```
student@cp:~$ kubectl get jobs
```

NAME	COMPLETIONS	DURATION	AGE
sleepy-1539723240	0/1	61s	61s

```
student@cp:~$ kubectl get cronjobs.batch
```

NAME	SCHEDULE	SUSPEND	ACTIVE	LAST SCHEDULE	AGE
sleepy	*/2 * * * *	False	1	72s	94s

```
student@cp:~$ kubectl get jobs
```

NAME	COMPLETIONS	DURATION	AGE
sleepy-1539723240	0/1	75s	75s

```
student@cp:~$ kubectl get jobs
```

NAME	COMPLETIONS	DURATION	AGE
sleepy-1539723240	0/1	2m19s	2m19s
sleepy-1539723360	0/1	19s	19s

```
student@cp:~$ kubectl get cronjobs.batch
```

NAME	SCHEDULE	SUSPEND	ACTIVE	LAST SCHEDULE	AGE
sleepy	*/2 * * * *	False	2	31s	2m53s

7. Clean up by deleting the CronJob.

```
student@cp:~$ kubectl delete cronjobs.batch sleepy
```

```
cronjob.batch "sleepy" deleted
```

Chapter 7

Managing State With Deployments



7.1	Deployment Overview	148
7.2	Deployments and Replica Sets	149
7.3	DaemonSets	158
7.4	Labels	159
7.5	Labs	161

7.1 Deployment Overview

Overview

- Deployments
- Application Updates
- Labels
- ReplicaSet

As with other objects a, deployment can be made from a YAML or JSON spec file. When added to the cluster, the controller will create a ReplicaSet and a Pod automatically. The containers, their settings and applications can be modified via an update, which generates a new ReplicaSet which in turn generates new Pods.

The updated objects can be staged to replace previous objects as a block or as a rolling update, which is determined as part of the deployment spec. Most updates can be configured by editing a YAML file and running **kubectl apply**. You can also use **kubectl edit** to modify the in-use configuration. Previous versions of the ReplicaSets are kept, allowing a roll-back to return to a previous configuration.

We will also talk more about **labels**. Labels are essential to administration in Kubernetes, but are not an API resource. They are user defined key-value pairs which can be attached to any resource, and are stored in the metadata. Labels are used to query or select resources in your cluster, allowing for flexible and complex management of the cluster.

As a label is arbitrary, you could select all resources used by developers, or belonging to a user, or any attached string without having to figure out what kind or how many of such resources exist.

7.2 Deployments and Replica Sets

Deployment Details

- Generate YAML of newly created objects
`kubectl get deployments,rs,pods -o yaml`
- Sometimes JSON output can make it more clear
`kubectl get deployments,rs,pods -o json`

```
apiVersion: v1
items:
- apiVersion: apps/v1
  kind: Deployment
```

In the previous slide we created a new deployment running a particular version of the **nginx** web server. Now we will look at the YAML output which also shows default values, not passed to the object when created.

- **apiVersion**

A value of **v1** indicates this object is considered to be a stable resource. In this case it is not the deployment. It is a reference to the **List** type.

- **items**

As the previous line is a **List** this declares the list of items the command is showing.

- - **apiVersion**

The dash is a YAML indication of the first item of the list, which declares the **apiVersion** of the object as **apps/v1**. This indicates the object is considered stable. Deployments are an operator used in many cases.

- **kind**

This is where the type of object to create is declared, in this case a deployment.

Deployment Configuration Metadata

```
metadata:  
  annotations:  
    deployment.kubernetes.io/revision: "1"  
  creationTimestamp: 2022-12-21T13:57:07Z  
  generation: 1  
  labels:  
    app: dev-web  
    name: dev-web  
    namespace: default  
  resourceVersion: "774003"  
  uid: d52d3a63-e656-11e7-9319-42010a800003
```

Continuing with the YAML output we see the next general block of output concerns the metadata of the deployment. This is where we would find labels, annotations, and other non-configuration information. Note that this output will not show all possible configuration. Many settings which are set to false by default are not shown, like `podAffinity` or `nodeAffinity`.

- `annotations`: These values do not configure the object, but provide further information that could be helpful to third-party applications or administrative tracking. Unlike `labels` they cannot be used to select an object with `kubectl`.
- `creationTimestamp` : When the object was originally created. Does not update if object edited.
- `generation` : How many times this object has been edited, changing the number of replicas for example.
- `labels` : Arbitrary strings used to select or exclude objects for use with `kubectl`, or other API calls. Helpful for admins to select objects outside of typical object boundaries.
- `name` : This is a **required** string, which we passed from the command line. The name must be unique to the namespace.
- `resourceVersion` : A value tied to the `etcd` database to help with concurrency of objects. Any changes to the database will cause this number to change.
- `uid` : remains a unique ID for the life of the object.

Deployment Configuration Spec

```
spec:  
  progressDeadlineSeconds: 600  
  replicas: 1  
  revisionHistoryLimit: 10  
  selector:  
    matchLabels:  
      app: dev-web  
  strategy:  
    rollingUpdate:  
      maxSurge: 25%  
      maxUnavailable: 25%  
    type: RollingUpdate
```

There are two spec declarations for the deployment. The first will modify the ReplicaSet created, while the second will pass along Pod configuration.

- **spec** : A declaration that following items will configure the object being created.
- **progressDeadlineSeconds** : Time in seconds until a progress error is reported during a change. Reasons could be quotas, image issues, or limit ranges.
- **replicas** : As the object being created is a ReplicaSet this parameter determines how many Pods should be created. If you were to use **kubectl edit** and change this value to two, a second Pod would be generated.
- **revisionHistoryLimit** : How many old ReplicaSet specifications to retain for rollback.
- **selector** : A collection of values ANDed together. All must be satisfied for the replica to match. Do not create Pods which match these selectors as the deployment controller may try to control the resource leading to issues.
- **matchLabels** : set-based requirements of the Pod selector. Often found with **matchExpressions** statement to further designate where the resource should be scheduled.
- **strategy** : A header for values having to do with updating Pods. Works with the later listed **type**. Could also be set to **Recreate**, which would delete all existing pods before new pods are created. With **RollingUpdate** you can control how many Pods are deleted at a time with the following parameters.
- **maxsurge** : Maximum number of Pods over desired number of Pods to create. Can be a percentage, default of 25%, or an absolute number. This creates a certain number of new Pods before deleting old, for continued access.
- **maxUnavailable** : A number or percentage of Pods which can be in a state other than Ready during the update process.
- **type** : Even though listed last in the section, due to level of white space indentation it is read as the type of object being configured. **rollingUpdate**

Deployment Configuration Pod Template

```

template:
  metadata:
    creationTimestamp: null
    labels:
      app: dev-web
  spec:
    containers:
      - image: nginx:1.17.7-alpine
        imagePullPolicy: IfNotPresent
        name: dev-web
        resources: {}
        terminationMessagePath: /dev/termination-log
        terminationMessagePolicy: File
    dnsPolicy: ClusterFirst
    restartPolicy: Always
    schedulerName: default-scheduler
    securityContext: {}
    terminationGracePeriodSeconds: 30
  
```

We will see some similar values as we view the configuration for the Pods to be deployed. If the meaning is basically the same we will not define it again.

- **template :**
Data being passed to the ReplicaSet to determine how to deploy an object, in this case containers.
- **containers :** Key word indicating that the following items of this indentation are for a container.
- **image :** This is the image name passed to the container engine, typically **Docker**. The engine will pull the image and create the Pod.
- **imagePullPolicy :** Policy settings passed along to container engine about when and if an image should be downloaded or used from a local cache.
- **name :** The leading stub of the Pod names. A unique string will be appended.
- **resources :** By default empty this is where you would set resource restrictions and settings such as a limit on CPU or memory for the containers.
- **terminationMessagePath :** A customizable location of where to output success or failure information of a container

- **terminationMessagePolicy :** The default value is **File** which holds the termination method. Could also be set to **FallbackToLogsOnError** which will use the last chunk of container log if the message file is empty and the container shows an error.
- **dnsPolicy :** Determine if DNS queries should go to **coredns** or, if set to Default, use the node's DNS resolution configuration.
- **restartPolicy :** Should the container be restarted if killed. Automatic restarts is part of the typical strength of Kubernetes.
- **schedulerName :** Allows for the use of a custom scheduler instead of the Kubernetes default.
- **securityContext :** Flexible setting to pass one or more security settings such as **SELinux** context, **AppArmor** values, users and UIDs for the containers to use.
- **terminationGracePeriodSeconds :** Amount of time to wait for a SIGTERM to run until a SIGKILL is used to terminate the container.

Deployment Configuration Status

```
status:  
  availableReplicas: 2  
  conditions:  
    - lastTransitionTime: 2022-12-21T13:57:07Z  
      lastUpdateTime: 2022-12-21T13:57:07Z  
      message: Deployment has minimum availability.  
      reason: MinimumReplicasAvailable  
      status: "True"  
      type: Available  
    - lastTransitionTime: "2021-07-29T06:00:24Z"  
      lastUpdateTime: "2021-07-29T06:00:33Z"  
      message: ReplicaSet "test-5f6778868d" has successfully progressed.  
      reason: NewReplicaSetAvailable  
      status: "True"  
      type: Progressing  
  observedGeneration: 2  
  readyReplicas: 2  
  replicas: 2  
  updatedReplicas: 2
```

The Status output is generated when the information is requested. The output above shows what the same deployment were to look like if the number of replicas were increased to two. The times are different than when the deployment was first generated.

- **availableReplicas :**
indicates how many were configured by the ReplicaSet. This would be compared to the later value of `readyReplicas` which would be used to determine if all replicas have been fully generated and without error.
- **observedGeneration :**
shows how often the deployment has been updated. This information can be used to understand the roll-out and roll-back situation of the deployment.

Scaling and Rolling Updates

- Deployment configuration can be dynamically updated Controllers
- Use set argument to create new Replica Set
- Also use edit to trigger update
- Update YAML file and use kubectl apply

The API server allows for the configurations settings to be updated for most values. There are some immutable values, which may be different depending on the version of Kubernetes you have deployed.

A common update is to change the number of replicas running. If this number is set to zero there would be no containers, but there would still be a ReplicaSet and Deployment. This is the backend process when a Deployment is deleted.

```
$ kubectl scale deploy/dev-web --replicas=4
```

```
deployment "dev-web" scaled
```

```
$ kubectl get deployments
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
dev-web	4/4	4	4	20s

Non-immutable values can be edited via text editor as well. For example to change the deployed version of the **nginx** web server to an older version:

```
$ kubectl edit deployment nginx
```

```
....  
  containers:  
    - image: nginx:1.8      #<<----Set to an older version  
      imagePullPolicy: IfNotPresent  
      name: dev-web  
....
```

This would trigger a rolling update of the deployment. While the deployment would show an older age, a review of the Pods would show a recent update and older version of the web server application deployed.

Deployment Rollbacks

- Previous ReplicaSets retained for rollback
- Can pause and resume
- Deployment edits replica counts decrementing old and incrementing new replicaSet

With some of previous the replicaSets of a Deployment being kept, you can also roll back to a previous revision by scaling up and down. The number of previous configurations kept is configurable, and has changed from version to version.

```
$ kubectl create deploy ghost --image=ghost
$ kubectl annotate deployment/ghost kubernetes.io/change-cause="kubectl create deploy ghost --image=ghost"
$ kubectl get deployments ghost -o yaml
```

```
deployment.kubernetes.io/revision: "1"
kubernetes.io/change-cause: kubectl create deploy ghost --image=ghost
```

Should an update fail, an improper image version for example, you can roll-back the change to a working version with `kubectl rollout undo`:

```
$ kubectl set image deployment/ghost ghost:ghost:09 --all
```

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
ghost-2141819201-tcths	0/1	ImagePullBackOff	0	1m

```
$ kubectl rollout undo deployment/ghost ; kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
ghost-3378155678-eq5i6	1/1	Running	0	7s

Deployment Rollbacks (cont.)

```
$ kubectl rollout pause deployment/ghost  
$ kubectl rollout resume deployment/ghost
```

You can roll back to a specific revision with the `--to-revision=2` option.

You can also edit a Deployment using the `kubectl edit` command.

You can also pause a Deployment, and then resume.

Please note that you can still do a rolling update on Replication Controllers with the `kubectl rolling-update` command, but this is done on the client side. Hence, if you close your client, the rolling update will stop.

7.3 DaemonSets

Using DaemonSets

- Runs on every node
- Same image on each
- Added and removed dynamically
- Use kind: DaemonSet

A newer object to work with is the DaemonSet. This controller ensures that a single pod exists on each node in the cluster. Every Pod uses the same image. Should a new node be added, the DaemonSet controller will deploy a new Pod on your behalf. Should a node be removed, the controller will delete the Pod also.

The use of a DaemonSet allows for ensuring a particular container is always running. In a large and dynamic environment, it can be helpful to have a logging or metric generation application on every node without an admin remembering to deploy that application.

There are ways of effecting `kube-scheduler` such that some nodes will not run a DaemonSet.

7.4 Labels

Labels

- Can exist in every resource metadata
- Hash label created by default
- Immutable as of API version apps/v1

Part of the metadata of an object is a label. Though they are not an API object, they are an important tool for cluster administration. They can be used to select an object based on an arbitrary string, regardless of object type.

Every resource can contain labels in its metadata. By default, creating a Deployment with `kubectl create` adds a label as we saw:

```
....  
  labels:  
    pod-template-hash: "3378155678"  
    run: ghost  
....
```

You could then view labels in new columns:

```
$ kubectl get pods -l run=ghost
```

NAME	READY	STATUS	RESTARTS	AGE
ghost-3378155678-eq5i6	1/1	Running	0	10m

```
$ kubectl get pods -L run
```

NAME	READY	STATUS	RESTARTS	AGE	RUN
ghost-3378155678-eq5i6	1/1	Running	0	10m	ghost
nginx-3771699605-4v27e	1/1	Running	1	1h	nginx

Labels (cont.)

- During creation or on the fly
- Easy to query or select

While you typically define labels in pod templates and in specifications of Deployments, you can also add labels on the fly:

```
$ kubectl label pods ghost-3378155678-eq5i6 foo=bar
```

```
$ kubectl get pods --show-labels
```

NAME	READY	STATUS	RESTARTS	AGE	LABELS
ghost-3378155678-eq5i6	1/1	Running	0	11m	foo=bar, pod-template-hash=3378155678,run=ghost

For example, if you want to force the scheduling of a pod on a specific node, you can use a `nodeSelector` in a pod definition, add specific labels to certain nodes in your cluster and use those labels in the pod.

```
....  
spec:  
  containers:  
    - image: nginx  
  nodeSelector:  
    disktype: ssd
```

7.5 Labs

Exercise 7.1: Working with ReplicaSets

Overview

Understanding and managing the state of containers is a core Kubernetes task. In this lab we will first explore the API objects used to manage groups of containers. The objects available have changed as Kubernetes has matured, so the Kubernetes version in use will determine which are available. Our first object will be a ReplicaSet, which does not include newer management features found with Deployments. A Deployment operator manages ReplicaSet operators for you. We will also work with another object and watch loop called a DaemonSet which ensures a container is running on newly added node.

Then we will update the software in a container, view the revision history, and roll-back to a previous version.

A ReplicaSet is a next-generation of a Replication Controller, which differs only in the selectors supported. The only reason to use a ReplicaSet anymore is if you have no need for updating container software or require update orchestration which won't work with the typical process.

1. View any current ReplicaSets. If you deleted resources at the end of a previous lab, you should have none reported in the default namespace.

```
student@cp:~$ kubectl get rs
```

```
No resources found in default namespace.
```

2. Create a YAML file for a simple ReplicaSet. The apiVersion setting depends on the version of Kubernetes you are using. The object is stable using the apps/v1 apiVersion. We will use an older version of **nginx** then update to a newer version later in the exercise.

```
student@cp:~$ cp /home/student/LFS458/SOLUTIONS/s_07/rs.yaml .
```

```
student@cp:~$ vim rs.yaml
```

YAML **rs.yaml**

```

1  apiVersion: apps/v1
2  kind: ReplicaSet
3  metadata:
4    name: rs-one
5  spec:
6    replicas: 2
7    selector:
8      matchLabels:
9        system: ReplicaOne
10   template:
11     metadata:
12       labels:
13         system: ReplicaOne
14     spec:
15       containers:
16         - name: nginx
17           image: nginx:1.15.1
18         ports:
19           - containerPort: 80

```

3. Create the ReplicaSet:

```
student@cp:~$ kubectl create -f rs.yaml
replicaset.apps/rs-one created
```

4. View the newly created ReplicaSet:

```
student@cp:~$ kubectl describe rs rs-one

Name:           rs-one
Namespace:      default
Selector:       system=ReplicaOne
Labels:          <none>
Annotations:    <none>
Replicas:       2 current / 2 desired
Pods Status:   2 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:        system=ReplicaOne
  Containers:
    nginx:
      Image:      nginx:1.15.1
      Port:       80/TCP
      Host Port:  0/TCP
      Environment: <none>
      Mounts:     <none>
      Volumes:    <none>
  Events:        <none>
```

5. View the Pods created with the ReplicaSet. From the yaml file created there should be two Pods. You may see a Completed busybox which will be cleared out eventually.

```
student@cp:~$ kubectl get pods

NAME        READY   STATUS    RESTARTS   AGE
rs-one-2p9x4 1/1     Running   0          5m4s
rs-one-3c6pb 1/1     Running   0          5m4s
```

6. Now we will delete the ReplicaSet, but not the Pods it controls.

```
student@cp:~$ kubectl delete rs rs-one --cascade=orphan
replicaset.apps "rs-one" deleted
```

7. View the ReplicaSet and Pods again:

```
student@cp:~$ kubectl describe rs rs-one
Error from server (NotFound): replicasets.apps "rs-one" not found
```

```
student@cp:~$ kubectl get pods

NAME        READY   STATUS    RESTARTS   AGE
rs-one-2p9x4 1/1     Running   0          7m
rs-one-3c6pb 1/1     Running   0          7m
```

8. Create the ReplicaSet again. As long as we do not change the selector field, the new ReplicaSet should take ownership. Pod software versions cannot be updated this way.

```
student@cp:~$ kubectl create -f rs.yaml
```

```
replicaset.apps/rs-one created
```

9. View the age of the ReplicaSet and then the Pods within:

```
student@cp:~$ kubectl get rs
```

NAME	DESIRED	CURRENT	READY	AGE
rs-one	2	2	2	46s

```
student@cp:~$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
rs-one-2p9x4	1/1	Running	0	8m
rs-one-3c6pb	1/1	Running	0	8m

10. We will now isolate a Pod from its ReplicaSet. Begin by editing the label of a Pod. We will change the system: parameter to be IsolatedPod.

```
student@cp:~$ kubectl edit pod rs-one-3c6pb
```

```
....  
labels:  
  system: IsolatedPod  #<-- Change from ReplicaOne  
managedFields:  
....
```

11. View the number of pods within the ReplicaSet. You should see two running.

```
student@cp:~$ kubectl get rs
```

NAME	DESIRED	CURRENT	READY	AGE
rs-one	2	2	2	4m

12. Now view the pods with the label key of system. You should note that there are three, with one being newer than others. The ReplicaSet made sure to keep two replicas, replacing the Pod which was isolated.

```
student@cp:~$ kubectl get po -L system
```

NAME	READY	STATUS	RESTARTS	AGE	SYSTEM
rs-one-3c6pb	1/1	Running	0	10m	IsolatedPod
rs-one-2p9x4	1/1	Running	0	10m	ReplicaOne
rs-one-dq5xd	1/1	Running	0	30s	ReplicaOne

13. Delete the ReplicaSet, then view any remaining Pods.

```
student@cp:~$ kubectl delete rs rs-one
```

```
replicaset.apps "rs-one" deleted
```

```
student@cp:~$ kubectl get po
```

NAME	READY	STATUS	RESTARTS	AGE
rs-one-3c6pb	1/1	Running	0	14m
rs-one-dq5xd	0/1	Terminating	0	4m

14. In the above example the Pods had not finished termination. Wait for a bit and check again. There should be no ReplicaSets, but one Pod.

```
student@cp:~$ kubectl get rs
```

```
No resources found in default namespaces.
```

```
student@cp:~$ kubectl get pod
```

NAME	READY	STATUS	RESTARTS	AGE
rs-one-3c6pb	1/1	Running	0	16m

15. Delete the remaining Pod using the label.

```
student@cp:~$ kubectl delete pod -l system=IsolatedPod
```

```
pod "rs-one-3c6pb" deleted
```

✍ Exercise 7.2: Working with Deployments

A Deployment is a watch loop object which we have been working with in the previous labs. A Deployment provides a declarative update to Pods and ReplicaSets and ensure a particular number of pods are created in general, several could be on a single node. Deployment is a high-level resource object that is used to manage the rollout and scaling of containerized applications. A Deployment describes the desired state of the application, such as the number of replicas, and the container image to use. When a Deployment is created, Kubernetes will automatically create and manage the necessary replica sets, which in turn will create and manage the necessary pods to ensure that the desired state of the application is met. Deployment also provide rolling updates, which allow for updating an application to a new version without downtime by gradually replacing the old replicas with new ones. Using Deployment in Kubernetes makes it easy to manage and scale containerized applications while ensuring high availability and reliability.

1. We begin by creating a yaml file. In this case the kind would be set to deployment. We can generate the yaml file using the imperative method

```
student@cp:~$ kubectl create deploy webserver --image nginx:1.22.1 --replicas=2 \
--dry-run=client -o yaml | tee dep.yaml
```

```
student@cp:~$ cat dep.yaml
```

YAML dep.yaml

```
1 ....
2 kind: Deployment
3 ....
4 name: webserver
5 ....
6 replicas: 2
```



```

7  ....
8      app: webserver
9  ....
10

```

2. Create and verify the newly formed Deployment. There should be two replicas of Pods created in the cluster.

```
student@cp:~$ kubectl create -f dep.yaml
```

```
deployment.apps/webserver created
```

```
student@cp:~$ kubectl get deploy
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
webserver	2/2	2	2	14s

```
student@cp:~$ kubectl get pod
```

NAME	READY	STATUS	RESTARTS	AGE
webserver-6cbc654ddc-lssbm	1/1	Running	0	42s
webserver-6cbc654ddc-xpmtl	1/1	Running	0	42s

3. Verify the image running inside the Pods. We will use this information in the next section.

```
student@cp:~$ kubectl describe pod webserver-6cbc654ddc-lssbm | grep Image:
```

```
Image:          nginx:1.22.1
```

Exercise 7.3: Rolling Updates and Rollbacks using Deployment

One of the advantages of micro-services is the ability to replace and upgrade a container while continuing to respond to client requests. We will use the recreate setting that upgrades a container when the predecessor is deleted, then the use the RollingUpdate feature as well, which begins a rolling update immediately.



nginx versions

The **nginx** software updates on a distinct timeline from Kubernetes. If the lab shows an older version please use the current default, and then a newer version. Versions can be verified on the repositories on the registry

1. Begin by viewing the current strategy setting for the Deployment created in the previous section.

```
student@cp:~$ kubectl get deploy webserver -o yaml | grep -A 4 strategy
```

```

strategy:
rollingUpdate:
  maxSurge: 25%
  maxUnavailable: 25%
type: RollingUpdate

```

2. Edit the object to use the Recreate update strategy. This would allow the manual termination of some of the pods, resulting in an updated image when they are recreated.

```
student@cp:~$ kubectl edit deploy webserver

.....
strategy:
rollingUpdate:          # <-- remove this line
  maxSurge: 25%         # <-- remove this line
  maxUnavailable: 25%   # <-- remove this line
type: Recreate           # <-- Edit this line
:q....
```

3. Update the Deployment to use a newer version of the **nginx** server. This time use the **set** command instead of **edit**. Set the version to be **1.23.1-alpine**.

```
student@cp:~$ kubectl set image deploy webserver nginx=nginx:1.23.1-alpine
deployment.apps/webserver image updated
```

4. Verify that the **Image:** parameter for the Pod checked in the previous section is unchanged.

```
student@cp:~$ kubectl get pod
```

NAME	READY	STATUS	RESTARTS	AGE
webserver-6cf9cd5c74-qjph4	1/1	Running	0	35s
webserver-6cf9cd5c74-zc6x9	1/1	Running	0	35s

```
student@cp:~$ kubectl describe po webserver-6cf9cd5c74-qjph4 |grep Image:
```

```
Image:      nginx:1.23.1-alpine
```

5. View the history of changes for the Deployment. You should see two revisions listed. As we did not add the the **change-cause** annotation we didn't see why the object updated.

```
student@cp:~$ kubectl rollout history deploy webserver
```

```
deployment.apps/webserver
REVISION  CHANGE-CAUSE
1          <none>
2          <none>
```

6. View the settings for the various versions of the Deployment. The **Image:** line should be the only difference between the two outputs.

```
student@cp:~$ kubectl rollout history deploy webserver --revision=1
```

```
deployment.apps/webserver with revision #1
Pod Template:
  Labels:      app=webserver
               pod-template-hash=6cbc654ddc
  Containers:
    nginx:
      Image:      nginx:1.22.1
```

```
Port:      <none>
Host Port: <none>
Environment:      <none>
Mounts:      <none>
Volumes:     <none>
```

```
student@cp:~$ kubectl rollout history deploy webserver --revision=2
```

```
....  
Image:      nginx:1.23.1-alpine  
....
```

7. Use `kubectl rollout undo` to change the Deployment back to previous version.

```
student@cp:~$ kubectl rollout undo deploy webserver
```

```
deployment.apps/webserver rolled back
```

```
student@cp:~$ kubectl get pod
```

NAME	READY	STATUS	RESTARTS	AGE
webserver-6cbc654ddc-7wb5q	1/1	Running	0	37s
webserver-6cbc654ddc-svbtj	1/1	Running	0	37s

```
student@cp:~$ kubectl describe pod webserver-6cbc654ddc-7wb5q |grep Image:
```

```
Image:      nginx:1.22.1
```

8. Clean up the system by removing the Deployment.

```
student@cp:~$ kubectl delete deploy webserver
```

```
deployment.apps "webserver" deleted
```

Exercise 7.4: Working with DaemonSets

A DaemonSet is a watch loop object like a Deployment which we have been working with in the rest of the labs. The DaemonSet ensures that when a node is added to a cluster, a pod will be created on that node. A Deployment would only ensure a particular number of pods are created in general, several could be on a single node. Using a DaemonSet can be helpful to ensure applications are on each node, helpful for things like metrics and logging especially in large clusters where hardware may be swapped out often. Should a node be removed from a cluster the DaemonSet would ensure the Pods are garbage collected before removal. Starting with Kubernetes v1.12 the scheduler handles DaemonSet deployment which means we can now configure certain nodes to not have a particular DaemonSet pods.

This extra step of automation can be useful for using with products like **ceph** where storage is often added or removed, but perhaps among a subset of hardware. They allow for complex deployments when used with declared resources like memory, CPU or volumes.

- We begin by creating a yaml file. In this case the kind would be set to DaemonSet. For ease of use we will copy the previously created `rs.yaml` file and make a couple edits. Remove the Replicas: 2 line.

```
student@cp:~$ cp rs.yaml ds.yaml
```

```
student@cp:~$ vim ds.yaml
```

YAML

ds.yaml

```

1 ....
2 kind: DaemonSet
3 ....
4   name: ds-one
5 ....
6   replicas: 2 #<<<----Remove this line
7 ....
8     system: DaemonSetOne #<<-- Edit both references
9 ....
10

```

2. Create and verify the newly formed DaemonSet. There should be one Pod per node in the cluster.

```
student@cp:~$ kubectl create -f ds.yaml
```

```
daemonset.apps/ds-one created
```

```
student@cp:~$ kubectl get ds
```

NAME	DESIRED	CURRENT	READY	UP-TO-DATE	AVAILABLE	NODE-SELECTOR	AGE
ds-one	2	2	2	2	2	<none>	1m

```
student@cp:~$ kubectl get pod
```

NAME	READY	STATUS	RESTARTS	AGE
ds-one-b1dcv	1/1	Running	0	2m
ds-one-z31r4	1/1	Running	0	2m

3. Verify the image running inside the Pods. We will use this information in the next section.

```
student@cp:~$ kubectl describe pod ds-one-b1dcv | grep Image:
```

```
Image: nginx:1.15.1
```

Exercise 7.5: Rolling Updates and Rollbacks using DaemonSet

One of the advantages of micro-services is the ability to replace and upgrade a container while continuing to respond to client requests. We will use the `OnDelete` setting that upgrades a container when the predecessor is deleted, then the use the `RollingUpdate` feature as well, which begins a rolling update immediately.



nginx versions

The `nginx` software updates on a distinct timeline from Kubernetes. If the lab shows an older version please use the current default, and then a newer version. Versions can be seen with this command: `sudo docker image ls nginx`

1. Begin by viewing the current `updateStrategy` setting for the DaemonSet created in the previous section.

```
student@cp:~$ kubectl get ds ds-one -o yaml | grep -A 4 Strategy
```

```
updateStrategy:
  rollingUpdate:
    maxSurge: 0
    maxUnavailable: 1
  type: RollingUpdate
```

2. Edit the object to use the `OnDelete` update strategy. This would allow the manual termination of some of the pods, resulting in an updated image when they are recreated.

```
student@cp:~$ kubectl edit ds ds-one
```

```
....  
updateStrategy:  
  rollingUpdate:  
    maxUnavailable: 1  
  type: OnDelete          #<-- Edit to be this line  
status:  
....
```

3. Update the DaemonSet to use a newer version of the `nginx` server. This time use the `set` command instead of `edit`. Set the version to be `1.16.1-alpine`.

```
student@cp:~$ kubectl set image ds ds-one nginx=nginx:1.16.1-alpine
```

```
daemonset.apps/ds-one image updated
```

4. Verify that the `Image:` parameter for the Pod checked in the previous section is unchanged.

```
student@cp:~$ kubectl describe po ds-one-b1dcv |grep Image:
```

```
Image:           nginx:1.15.1
```

5. Delete the Pod. Wait until the replacement Pod is running and check the version.

```
student@cp:~$ kubectl delete po ds-one-b1dcv
```

```
pod "ds-one-b1dcv" deleted
```

```
student@cp:~$ kubectl get pod
```

NAME	READY	STATUS	RESTARTS	AGE
ds-one-xc86w	1/1	Running	0	19s
ds-one-z31r4	1/1	Running	0	4m8s

```
student@cp:~$ kubectl describe pod ds-one-xc86w |grep Image:
```

```
Image:           nginx:1.16.1-alpine
```

6. View the image running on the older Pod. It should still show version 1.15.1.

```
student@cp:~$ kubectl describe pod ds-one-z31r4 |grep Image:
```

```
Image:           nginx:1.15.1
```

7. View the history of changes for the DaemonSet. You should see two revisions listed. As we did not add the change-cause annotation we didn't see why the object updated.

```
student@cp:~$ kubectl rollout history ds ds-one
```

```
daemonsets "ds-one"
REVISION      CHANGE-CAUSE
1            <none>
2            <none>
```

8. View the settings for the various versions of the DaemonSet. The Image: line should be the only difference between the two outputs.

```
student@cp:~$ kubectl rollout history ds ds-one --revision=1
```

```
daemonsets "ds-one" with revision #1
Pod Template:
  Labels:      system=DaemonSetOne
  Containers:
    nginx:
      Image:      nginx:1.15.1
      Port:       80/TCP
      Environment: <none>
      Mounts:     <none>
      Volumes:   <none>
```

```
student@cp:~$ kubectl rollout history ds ds-one --revision=2
```

```
.....
  Image:      nginx:1.16.1-alpine
.....
```

9. Use kubectl rollout undo to change the DaemonSet back to an earlier version. As we are still using the OnDelete strategy there should be no change to the Pods.

```
student@cp:~$ kubectl rollout undo ds ds-one --to-revision=1
```

```
daemonset.apps/ds-one rolled back
```

```
student@cp:~$ kubectl describe pod ds-one-xc86w |grep Image:
```

```
Image:      nginx:1.16.1-alpine
```

10. Delete the Pod, wait for the replacement to spawn then check the image version again.

```
student@cp:~$ kubectl delete pod ds-one-xc86w
```

```
pod "ds-one-xc86w" deleted
```

```
student@cp:~$ kubectl get pod
```

NAME	READY	STATUS	RESTARTS	AGE
ds-one-qc72k	1/1	Running	0	10s

```
ds-one-xc86w      0/1      Terminating   0      12m
ds-one-z31r4      1/1      Running     0      28m
```

student@cp:~\$ kubectl describe po ds-one-qc72k |grep Image:

Image:	nginx:1.15.1
--------	--------------

- View the details of the DaemonSet. The Image should be v1.15.1 in the output.

student@cp:~\$ kubectl describe ds |grep Image:

Image:	nginx:1.15.1
--------	--------------

- View the current configuration for the DaemonSet in YAML output. Look for the updateStrategy: the type:

student@cp:~\$ kubectl get ds ds-one -o yaml

```
apiVersion: apps/v1
kind: DaemonSet
.....
terminationGracePeriodSeconds: 30
updateStrategy:
  type: OnDelete
status:
  currentNumberScheduled: 2
....
```

- Create a new DaemonSet, this time setting the update policy to RollingUpdate. Begin by generating a new config file.

student@cp:~\$ kubectl get ds ds-one -o yaml > ds2.yaml

- Edit the file. Change the name, around line 69 and the update strategy around line 100, back to the default RollingUpdate.

student@cp:~\$ vim ds2.yaml

```
....
  name: ds-two
...
  type: RollingUpdate
```

- Create the new DaemonSet and verify the nginx version in the new pods.

student@cp:~\$ kubectl create -f ds2.yaml

daemonset.apps/ds-two created

student@cp:~\$ kubectl get pod

NAME	READY	STATUS	RESTARTS	AGE
ds-one-qc72k	1/1	Running	0	28m
ds-one-z31r4	1/1	Running	0	57m
ds-two-10khc	1/1	Running	0	5m
ds-two-kzp9g	1/1	Running	0	5m

student@cp:~\$ kubectl describe po ds-two-10khc |grep Image:

Image:	nginx:1.15.1
--------	--------------

16. Edit the configuration file and set the image to a newer version such as 1.16.1-alpine.

```
student@cp:~$ kubectl edit ds ds-two
```

```
....  
    - image: nginx:1.16.1-alpine  
....
```

17. View the age of the DaemonSets. It should be around ten minutes old, depending on how fast you type.

```
student@cp:~$ kubectl get ds ds-two
```

NAME	DESIRED	CURRENT	READY	UP-TO-DATE	AVAILABLE	NODE-SELECTOR	AGE
ds-two	2	2	2	2	2	<none>	10m

18. Now view the age of the Pods. Two should be much younger than the DaemonSet. They are also a few seconds apart due to the nature of the rolling update where one then the other pod was terminated and recreated.

```
student@cp:~$ kubectl get pod
```

NAME	READY	STATUS	RESTARTS	AGE
ds-one-qc72k	1/1	Running	0	36m
ds-one-z31r4	1/1	Running	0	1h
ds-two-2p8vz	1/1	Running	0	34s
ds-two-8lx7k	1/1	Running	0	32s

19. Verify the Pods are using the new version of the software.

```
student@cp:~$ kubectl describe po ds-two-8lx7k |grep Image:
```

Image:	nginx:1.16.1-alpine
--------	---------------------

20. View the rollout status and the history of the DaemonSets.

```
student@cp:~$ kubectl rollout status ds ds-two
```

daemon set "ds-two" successfully rolled out

```
student@cp:~$ kubectl rollout history ds ds-two
```

REVISION	CHANGE-CAUSE
1	<none>
2	<none>

21. View the changes in the update they should look the same as the previous history, but did not require the Pods to be deleted for the update to take place.

```
student@cp:~$ kubectl rollout history ds ds-two --revision=2
```

```
...  
Image:      nginx:1.16.1-alpine
```

22. Clean up the system by removing the DaemonSets.

```
student@cp:~$ kubectl delete ds ds-one ds-two
```

```
daemonset.apps "ds-one" deleted  
daemonset.apps "ds-two" deleted
```


Chapter 8

Helm



8.1	Overview	176
8.2	Helm	177
8.3	Using Helm	180
8.4	Labs	182

8.1 Overview

Deploying Complex Applications

- Package your Kubernetes application via **chart** template
- **Helm** client to request install of **chart**
- **Helm** creates cluster resources according to **chart**
- Version 3 was major update.

We have used Kubernetes tools to deploy simple containers and services. Also necessary was to have a canonical location for software. **Helm** is similar to a package manager like **yum** or **apt**, with a **chart** being similar to a package, in that it has the binaries as well as installation and removal scripts.

A typical containerized application will have several manifests. Manifests for deployments, services, and configMaps. You will probably also create some secrets, Ingress, and other objects. Each of these will need a manifest.

With **Helm**, you can package all those manifests and make them available as a single tarball. You can put the tarball in a repository, search that repository, discover an application, and then, with a single command, deploy and start the entire application, one or more times.

The tarballs can be collected in a repository for sharing. You can connect to multiple repositories of applications, including those provided by vendors.

You will also be able to upgrade, or roll-back, an application easily from the command line.

8.2 Helm

Helm v3

- Many changes compared to version 2
- No more Tiller pod
- 3-way strategic merge patches
- Name or generated name now required on install

With the near complete overhaul of Helm the processes and commands have changed quite a bit. Expect to spend some time updating and integrating these changes if you are currently using the outdated Helm v2.

One of the most noticeable changes is the removal of the Tiller pod. This was an ongoing security issues as the pod needed elevated permissions to deploy charts. The functionality is in the command alone, and no longer requires initialization to use.

In version 2 an update to a chart and deployment used a 2-way strategic merge for patching. This compared the previous manifest to the intended manifest, but not the possible edits done outside of helm commands. The third way now checked is the live state of objects.

Among other changes software installation no longer generates a name automatically. One must be provided, or the --generated-name option must be passed.

Chart Contents

```
|-- Chart.yaml
|-- README.md
|-- templates
|   |-- NOTES.txt
|   |-- _helpers.tpl
|   |-- configmap.yaml
|   |-- deployment.yaml
|   |-- pvc.yaml
|   |-- secrets.yaml
|   |-- svc.yaml
|-- values.yaml
```

A chart is an archive set of Kubernetes resource manifests that make up a distributed application. You can check out the GitHub repository where the Kubernetes community is curating charts. Others exist and can be easily created, for example by a vendor providing software. Similar to the use of independent **YUM** repositories.

`Chart.yaml` contains some metadata about the chart, like its name, version, keywords, and so on, in this case for **MariaDB**. `values.yaml` contains keys and values that are used to generate the release in your Cluster. These values are replaced in the resource manifests using the **Go** templating syntax. And finally, the `templates` directory contains the resource manifests that make up this MariaDB application.

More about creating charts can found at:
<https://helm.sh/docs/topics/charts/>.

Templates

```
apiVersion: v1
kind: Secret
metadata:
  name: {{ template "fullname" . }}
  labels:
    app: {{ template "fullname" . }}
    chart: "{{ .Chart.Name }}-{{ .Chart.Version }}"
    release: "{{ .Release.Name }}"
    heritage: "{{ .Release.Service }}"
type: Opaque
data:
  mariadb-root-password: {{ default "" .Values.mariadbRootPassword | b64enc \
                           | quote }}
  mariadb-password: {{ default "" .Values.mariadbPassword | b64enc | quote }}
```

The template are resource manifests which use the **Go** templating syntax. Variables defined in the values file, for example, get injected in the template when a release is created. In the **MariadDB** example provided, the database passwords are stored in a Kubernetes secret, and the database configuration is stored in a Kubernetes ConfigMap.

We can see that a set of labels are defined in the secret metadata using the chart name, Release name, etc. The actual values of the passwords are read from [values.yaml](#).

8.3 Using Helm

Chart Repositories and Hub

- Search for charts
- Add a new repository
- Simple HTTP servers with index file and tarball of charts
- **helm repo**
- ArtifactHub to replace Docker hub

Repositories are currently simple HTTP servers that contain an index file and a tarball of all the Charts present. Prior to adding a repository you can only search Artifact Hub <https://artifacthub.io/>, using **helm search hub**.

```
$ helm search hub redis
```

You can interact with a repository using the **helm repo** commands.

```
$ helm repo add bitnami https://charts.bitnami.com/bitnami
$ helm repo list
```

NAME	URL
bitnami	https://charts.bitnami.com/bitnami

Once you have a repository available, you can search for Charts based on keywords. Below, we search for a redis Chart:

```
$ helm search repo bitnami
```

Once you can find the chart within a repository you can deploy it on your cluster.

Deploying a Chart

```
$ helm fetch bitnami/apache --untar  
$ cd apache/  
$ ls  
Chart.lock  Chart.yaml  README.md  charts  ci  files  templates  
values.schema.json  values.yaml
```

```
$ helm install anotherweb .
```

To deploy a Chart, you can use the **helm install** command. There may be several required resources for the installation to be successful, such as available PVs to match chart PVC. Currently the only way to discover which resources need to exist is by reading the READMEs for each chart. This can be found by downloading the tarball and expanding it into the current directory. Once requirements are met and edits are made you can install using the local files.

You will be able to list the release, delete it, even upgrade it and roll back.

The output of deployment should be carefully reviewed. It often includes information on access to the applications within. If your cluster did not have a required cluster resource, the output is often the first place to begin troubleshooting.

8.4 Labs

Exercise 8.1: Working with Helm and Charts

Overview

helm allows for easy deployment of complex configurations. This could be handy for a vendor to deploy a multi-part application in a single step. Through the use of a Chart, or template file, the required components and their relationships are declared. Local agents like **Tiller** use the API to create objects on your behalf. Effectively its orchestration for orchestration.

There are a few ways to install **Helm**. The newest version may require building from source code. We will download a recent, stable version. Once installed we will deploy a Chart, which will configure **MariaDB** on our cluster.

Install Helm

- On the cp node use **wget** to download the compressed tar file. Various versions can be found here: <https://github.com/helm/helm/releases/>

```
student@cp:~$ wget https://get.helm.sh/helm-v3.15.2-linux-amd64.tar.gz
```

```
<output_omitted>
helm-v3.15.2-linux-amd64.tar.gz 100%[=====] 13.35M --.-KB/s in 0.1s
2024-07-02 13:38:09 (101 MB/s) - 'helm-v3.15.2-linux-amd64.tar.gz' saved [16624839/16624839]
```

- Uncompress and expand the file.

```
student@cp:~$ tar -xvf helm-v3.15.2-linux-amd64.tar.gz
```

```
linux-amd64/
linux-amd64/helm
linux-amd64/README.md
linux-amd64/LICENSE
```

- Copy the **helm** binary to the `/usr/local/bin/` directory, so it is usable via the shell search path.

```
student@cp:~$ sudo cp linux-amd64/helm /usr/local/bin/helm
```

- A Chart is a collection of files to deploy an application. There is a good starting repo available on <https://github.com/kubernetes/charts/tree/master/stable>, provided by vendors, or you can make your own. Search the current Charts in the Helm Hub or an instance of Monocular for available stable databases. Repos change often, so the following output may be different from what you see.

```
student@cp:~$ helm search hub database
```

URL	APP VERSION	DESCRIPTION	CHART VERSION
https://artifacthub.io/packages/helm/drycc/database	1.0.2	A PostgreSQL database used by Drycc Workflow.	1.0.2
https://artifacthub.io/packages/helm/drycc-canary	1.0.0	A PostgreSQL database used by Drycc Workflow.	1.0.0
https://artifacthub.io/packages/helm/camptocamp/postgres	0.0.6	Expose services and secret to access postgres	0.0.6
https://artifacthub.io/packages/helm/cnieg/h2-database	1.0.3 1.4.199	A helm chart to deploy h2-database	1.0.3

```
<output_omitted>
```

5. You can also add repositories from various vendors, often found by searching [artifacthub.io](#) such as ealenn, who has an echo program.

```
student@cp:~$ helm repo add ealenn https://ealenn.github.io/charts
```

```
"ealenn" has been added to your repositories
```

```
student@cp:~$ helm repo update
```

```
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "ealenn" chart repository
Update Complete. Happy Helming!
```

6. We will install the **tester** tool. The **- -debug** option will create a lot of output. The output will typically suggest ways to access the software.

```
student@cp:~$ helm upgrade -i tester ealenn/echo-server --debug
```

```
history.go:56: [debug] getting history for release tester
Release "tester" does not exist. Installing it now.
install.go:173: [debug] Original chart version: ""
install.go:190: [debug] CHART PATH: /home/student/.cache/helm/repository/echo-server-0.5.0.tgz

client.go:122: [debug] creating 4 resource(s)
NAME: tester
<output_omitted>
```

7. Ensure the newly created tester-echo-server pod is running. Fix any issues, if not.

8. Look for the newly created service. Send a **curl** to the ClusterIP. You should get a lot of information returned.

```
student@cp:~$ kubectl get svc
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP	26h
tester-echo-server	ClusterIP	10.98.252.11	<none>	80/TCP	11m

```
student@cp:~$ curl 10.98.252.11
```

```
{"host": {"hostname": "10.98.252.11", "ip": "::ffff:192.168.74.128", "ips": []}, "http": {"method": "GET", "baseUrl": "", "originalUrl": "/", "protocol": "http"}, "request": {"params": {"0": "/"}, "query": {}, "cookies": {}, "body": {}, "headers": {"host": "10.98.252.11", "user-agent": "curl/7.58.0", "accept": "*/*"}, "environment": {"PATH": "/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin", "TERM": "xterm", "HOSTNAME": "tester-echo-server-786768d9f4-4zs9", "ENABLE__HOST": "true", "ENABLE__HTTP": "true", "ENABLE__<output_omitted>
```

9. View the Chart history on the system. The use of the **-a** option will show all Charts including deleted and failed attempts.

```
student@cp:~$ helm list
```

NAME	NAMESPACE	REVISION	UPDATED
STATUS	CHART		APP VERSION
tester	default	1	2024-07-02 13:42:38.262327888 +0000 UTC
deployed	echo-server-0.5.0	0.6.0	

10. Delete the **tester** Chart. No releases of tester should be found.

```
student@cp:~$ helm uninstall tester
```

```
release "tester" uninstalled
```

```
student@cp:~$ helm list
```

NAME	NAMESPACE	REVISION	UPDATED	STATUS	CHART	APP VERSION

11. Find the downloaded chart. It should be a compressed tarball under the user's home directory. Your **echo** version may be slightly different.

```
student@cp:~$ find $HOME -name *echo*
```

```
/home/student/.cache/helm/repository/echo-server-0.5.0.tgz
```

12. Move to the archive directory and extract the tarball. Take a look at the files within.

```
student@cp:~$ cd $HOME/.cache/helm/repository ; tar -xvf echo-server-*
```

```
echo-server/Chart.yaml
echo-server/values.yaml
echo-server/templates/_helpers.tpl
echo-server/templates/configmap.yaml
echo-server/templates/deployment.yaml
<output omitted>
```

13. Examine the `values.yaml` file to see some of the values that could have been set.

```
student@cp:~/.cache/helm/repository$ cat echo-server/values.yaml
```

```
<output omitted>
```

14. You can also download and examine or edit the values file before installation. Add another repo and download the Bitnami Apache chart.

```
student@cp:~$ helm repo add bitnami https://charts.bitnami.com/bitnami
```

```
student@cp:~$ helm fetch bitnami/apache --untar
```

```
student@cp:~$ cd apache/
```

15. Take a look at the chart. You'll note it looks similar to the previous. Read through the `:values.yaml`.

```
student@cp:~$ ls
```

```
Chart.lock Chart.yaml README.md charts ci files templates  
values.schema.json values.yaml
```

```
student@cp:~$ less values.yaml
```

```
## Global Docker image parameters  
## Please, note that this will override the image parameters, including dependencies,  
→ configured....  
## Current available global Docker image parameters: imageRegistry and imagepullSecrets  
##  
# global:  
#   imageRegistry: myRegistryName  
#   imagePullSecrets:  
#     - myRegistryKeySecretName  
<output_omitted>
```

16. Use the `values.yaml` file to install the chart. Take a look at the output and ensure the pod is running.

```
student@cp:~$ helm install anotherweb .
```

```
NAME: anotherweb  
LAST DEPLOYED: Fri Jun 11 08:11:10 2021  
NAMESPACE: default  
STATUS: deployed  
REVISION: 1  
TEST SUITE: None  
<output_omitted>
```

17. Test the newly created service. You should get an HTML response saying It works! If the steps to find the service and check that it works are not familiar, you may want to make a note to review prior chapters.
18. Remove anything you have installed using **helm**. Reference earlier in the chapter if you don't remember the command. We will use **helm** again in another lab.

Chapter 9

Volumes and Data



9.1	Volumes Overview	188
9.2	Volumes	189
9.3	Persistent Volumes	193
9.4	Rook	197
9.5	Passing Data To Pods	198
9.6	ConfigMaps	201
9.7	Labs	203

9.1 Volumes Overview

Overview

- Object to save data longer than container lifetime
- Many Volume types to choose from
- Define Persistent Volumes (PV)
- Define Persistent Volume Claims (PVC)
- Create Secrets
- Create ConfigMaps

Container engines have traditionally not offered storage that outlives the container. As containers are considered transient this could lead to a loss of data, or complex exterior storage options. A Kubernetes Volume shares the Pod lifetime, not the containers within. Should a container terminate, the data would continue to be available to the new container.

A volume is a directory, possibly pre-populated, made available to containers in a Pod. The creation of the directory, the back-end storage of the data, and the contents depend on the volume type. There are many different volume types ranging from rbd to gain access to **Ceph**, to **NFS**, to dynamic volumes from a cloud provider like Google's **gcePersistentDisk**. Each has particular configuration options and dependencies.

The Container Storage Interface (CSI) adoption enables the goal of an industry standard interface for container orchestration allow access to arbitrary storage systems. Currently volume plugins are "in-tree", meaning they are compiled and built with the core Kubernetes binaries. This "out-of-tree" object will allow storage vendors to develop a single driver and allow the plugin to be containerized. This will replace the existing **Flex** plugin which requires elevated access to the host node, a large security concern.

Should you want your storage lifetime to be distinct from a Pod you can use **Persistent Volumes**. These allow for empty or pre-populated volumes to be claimed by a Pod using a **Persistent Volume Claim** then outlive the Pod. Data inside the volume could then be used by another Pod or as a means of retrieving data.

There are two API Objects which exist to provide data to a Pod already. Encoded data can be passed using a Secret and non-encoded data passed with a ConfigMap. These can be used to pass important data like SSH keys, passwords or even a configuration file like </etc/hosts>.

9.2 Volumes

Introducing Volumes

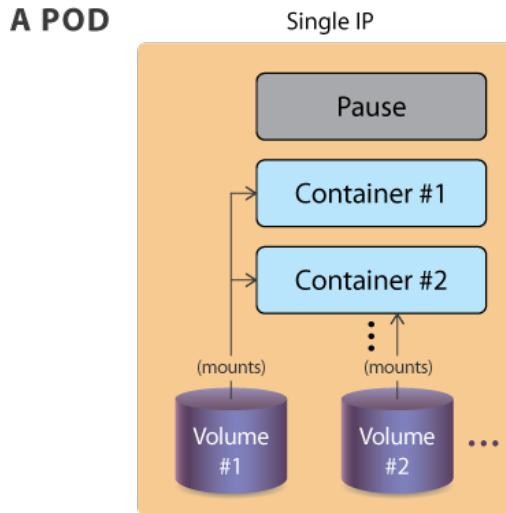


Figure 9.1: K8s Pod Volumes

A Pod specification can declare one or more volumes and where they are made available. Each requires a name, a type, and a mount point. The same volume can be made available to multiple containers within a Pod, which can be a method of container-to-container communication. A volume can also be made available to multiple Pods, with each given an access mode to write. There is no concurrency checking which means data corruption is probable unless outside locking takes place.

Part of a Pod request is a particular access mode. As a request the user may be granted more, but not less access, though a direct match is attempted first. The cluster groups volumes with the same mode together, then sorts volumes by size from smallest to largest. The claim is checked against each in that access mode group until a volume of sufficient size matches. The three access modes are `ReadWriteOnce`, which allows read-write by a single node, `ReadOnlyMany`, which allows read-only by multiple nodes, and `ReadWriteMany` which allows read-write by many nodes. Thus two pods on the same node can write to a `ReadWriteOnce`, but a third pod on a different node would not become ready due to a `FailedAttachVolume` error.

When a volume is requested the local `kubelet` uses the `kubelet_pods.go` script to map the raw devices, determine and make the mount point for the container, then create the symbolic link on the host node filesystem to associate the storage to the container. The API server makes a request for the storage to the `StorageClass` plugin, but the specifics of the requests to the back-end storage depend on the plugin in use.

If a request for a particular `StorageClass` was not made then the only parameters used will be access mode and size. The volume could come from any of the storage types available and there is no configuration to determine which of the available will be used.

Volume Spec

```
apiVersion: v1
kind: Pod
metadata:
  name: fordpinto
  namespace: default
spec:
  containers:
    - image: simpleapp
      name: gastank
      command:
        - sleep
        - "3600"
    volumeMounts:
      - mountPath: /scratch
        name: scratch-volume
  volumes:
    - name: scratch-volume
      emptyDir: {}
```

One of the many types of storage available is an `emptyDir`. The **kubelet** will create the directory in the container, but not mount any storage. Any data created is written to the shared container space. As a result it would not be persistent storage. When the Pod is destroyed the directory would be deleted along with the container.

The YAML above would create a Pod with a single container with a volume named `scratch-volume` created, which would create the `/scratch` directory inside the container.

Volume Types

- Several types possible, more being added

– awsElasticBlockStore	– gcePersistentDisk	– quobyte
– azureDisk	– gitRepo	– rbd
– azureFile	– glusterfs	– scaleIO
– cephfs	– hostPath	– secret
– csi	– iscsi	– storageos
– downwardAPI	– local	– vsphereVolume
– emptyDir	– nfs	– persistentVolumeClaim
– fc (fibre channel)	– projected	– CSIPersistentVolumeSource
– flocker	– portworxVolume	

There are several types that you can use to define volumes, each with their pros and cons. Some are local, many make use of network-based resources.

In GCE or AWS, you can use Volumes of type `GCEPersistentDisk` or `awsElasticBlockStore`, which allows you to mount GCE and EBS disks in your Pods, assuming you have already set up accounts and privileges.

`emptyDir` and `hostPath` volumes are easy to use. As mentioned, `emptyDir` is an empty directory that gets erased when the Pod dies but is recreated when the container restarts. The `hostPath` volume mounts a resource from the host node filesystem. The resource could be a directory, file socket, character, or block device. These resources must already exist on the host to be used. There are two types, `DirectoryOrCreate` and `FileOrCreate`, which create the resources on the host and use them if they don't already exist.

NFS (Network File System) and **iSCSI** (Internet Small Computer System Interface) are straightforward choices for multiple readers scenarios.

CSI allows for even more flexibility and decoupling plugins without the need to edit the core Kubernetes code. It was developed as a standard for exposing arbitrary plugins in the future.

Note: Many in-tree storage drivers are deprecated and removed in the current Kubernetes version and all operations for the in-tree deprecated volume type is redirected to the CSI driver.

Shared Volume Example

```
....  
containers:  
- name: alphacont  
  image: busybox  
  volumeMounts:  
    - mountPath: /alphadir  
      name: sharevol  
- name: betacont  
  image: busybox  
  volumeMounts:  
    - mountPath: /betadir  
      name: sharevol  
volumes:  
- name: sharevol  
  emptyDir: {}
```

The above YAML creates a pod, exampleA, with two containers both with access to one shared volume. You could use emptyDir or hostPath easily, since those types do not require any additional setup and will work in your Kubernetes cluster.

```
$ kubectl exec -ti exampleA -c betacont -- touch /betadir/foobar  
$ kubectl exec -ti exampleA -c alphacont -- ls -l /alphadir
```

```
total 0  
-rw-r--r-- 1 root root 0 Nov 19 16:26 foobar
```

Note that one container, betacont wrote, and the other container, alphacont had immediate access to the data. There is nothing to keep the containers from overwriting the other's data. Locking or versioning considerations must be part of the containerized application to avoid corruption.

9.3 Persistent Volumes

Persistent Volumes and Claims

- Useful for porting data
- Resources managed via API
- Storage abstraction
- Several phases

```
$ kubectl get pv  
$ kubectl get pvc
```

A **persistent volume (pv)** is a storage abstraction used to retain data longer than the Pod using it. Pods define a volume of type `persistentVolumeClaim` with various parameters for size and possibly the type of back-end storage known as its `StorageClass`. The cluster then attaches the `persistentVolume`.

Kubernetes will dynamically use volumes that are available irrespective of its storage type, allowing claims to any back-end storage.

There are several phases to persistent storage:

- **Provisioning** can be from PVs created in advance by the cluster administrator, or requested from a dynamic source such as the cloud provider.
- **Binding** occurs when a control loop on the cp notices the PVC, containing an amount of storage, access request, and optionally a particular `StorageClass`. The watcher locates a matching PV or waits for the `StorageClass` provisioner to create one. The PV must match at least the storage amount requested, but may provide more.
- The **use** phase begins when the bound volume is mounted for the Pod to use, which continues as long as the Pod requires.
- **Releasing** happens when the Pod is done with the volume and an API request is sent deleting the PVC. The volume remains in the state from when the claim is deleted until available to a new claim. The resident data remains depending on the `persistentVolumeReclaimPolicy`.
- The **reclaim** phase has three options: **Retain** which keeps the data intact allowing for an admin to handle the storage and data. **Delete** tells the volume plug-in to delete the API object as well as the storage behind it. The **Recycle** option runs an `rm -rf /mountpoint` then makes it available to a new claim. With the stability of dynamic provisioning the **Recycle** is planned to be deprecated.

Persistent Volume

```
kind: PersistentVolume
apiVersion: v1
metadata:
  name: 10Gpv01
  labels:
    type: local
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/somepath/data01"
```

This shows a basic declaration of a `PersistentVolume` using the `hostPath` type. Each type will have its own configuration settings. For example an already created Ceph or GCE Persistent Disks would not need to be configured but could be claimed from the provider.

Persistent volumes are not a namespaces object, but persistent volume claims are.

A beta feature of v1.13 allows for static provisioning of Raw Block Volumes, which currently supports Fibre Channel, AWS EBS, Azure Disk, and RBD plugins among others.

The use of locally attached storage has been graduated to a stable feature. This feature is often used as part of distributed file systems and databases.

Persistent Volume Claim

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: myclaim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 8Gi
```

In the Pod:

```
spec:
  containers:
  ....
  volumes:
    - name: test-volume
      persistentVolumeClaim:
        claimName: myclaim
```

With a persistent volume created in your cluster, you can then write a manifest for a claim and use that claim in your pod definition. In the Pod, the volume uses the `persistentVolumeClaim`.

The Pod configuration could also be as complex as this:

```
volumeMounts:
  - name: Cephpd
    mountPath: /data/rbd
volumes:
  - name: rbdpd
    rbd:
      monitors:
        - '10.19.14.22:6789'
        - '10.19.14.23:6789'
        - '10.19.14.24:6789'
      pool: k8s
      image: client
      fsType: ext4
      readOnly: true
      user: admin
      keyring: /etc/ceph/keyring
      imageformat: "2"
      imagefeatures: "layering"
```

Dynamic Provisioning

- Claim filled via auto-provisioning
- No need for admin to pre-create PVs
- Uses the StorageClass API object
- StorageClass defines volume plugin, or provisioner to use
- Single, default class possible via annotation

While handling volumes with a persistent volume definition and abstracting the storage provider using a claim is powerful, a cluster administrator needed to create those volumes in the first place. Starting in v1.4 Dynamic Provisioning allowed for the cluster to request storage from an exterior, pre-configured, source. API calls made by the appropriate plug-in allow for a wide range of dynamic storage use.

The StorageClass API resource allows an administrator to define a persistent volume provisioner of a certain type, passing storage-specific parameters.

With a StorageClass created, a user can request a claim which the API Server fills via auto-provisioning. The resource will also be reclaimed as configured by the provider. **AWS** and **GCE** are common choices for dynamic storage, but other options exist such as a **Ceph** cluster or **iSCSI**.

Here is an example of a StorageClass using **GCE**:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast          # Could be any name
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-ssd
```

9.4 Rook

Using Rook for Storage Orchestration

- File, block and object storage
- Multiple storage providers
- Hyper-scale or hyper-converge
- Leverages CRDs and a rook operator

In keeping with the decouple and distributed nature of Cloud technology the **Rook** project <https://rook.io> allows orchestration of storage using multiple storage providers.

As with other agents of the cluster **Rook** uses custom resource definitions (CRD) and a custom operator to provision storage according to the back-end storage type, upon API call.

Several storage providers are supported:

- Ceph
- Cassandra
- Network File System (NFS)

9.5 Passing Data To Pods

Secrets

- Leverages base64 encoding
- Is **not** encryption unless further configured
- Conversion to generic data, acceptable everywhere
- Encoded manually or via **kubectl create secret**

```
$ kubectl create secret generic mysql --from-literal=password=root
```

Pods can access local data using volumes, but there is some data you don't want readable to the naked eye. Passwords may be an example. Using the **Secret** API resource the same password could be encoded or encrypted.

A secret is **not** encrypted, only base64-encoded, by default. One must create a `EncryptionConfiguration` with a key and proper identity. Then the **kube-apiserver** needs the `--encryption-provider-config` flag set to a previously configured provider such as `aescbc` or `ksm`. Once this is enabled you need to recreate every secret as they are encrypted upon write.

Multiple keys are possible. Each key for a provider is tried during decryption. The first key of the first provider is used for encryption. To rotate keys first create a new key, restart (all) **kube-apiserver** processes, then recreate every secret.

You can see the encoded string inside the secrets with **kubectl**. The secret will be decoded and be presented as a string saved to a file. The file can be used as an environmental variable or in a new directory, similar to the presentation of a volume.

A secret can be made manually as well, then inserted into a YAML file.

Using Secrets via Environment Variables

```
...
spec:
  containers:
    - image: mysql:5.5
      name: dbpod
      env:
        - name: MYSQL_ROOT_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mysql
              key: password
```

A secret can be used as an environment variable in a Pod. Above we see one being configured.

There is not a limit to the number of Secrets used, but there is a 1MB limit to their size. Each secret occupies memory, along with other API objects, so very large numbers of secrets could deplete memory on a host.

They are stored in the **tmpfs** storage on the host node, and are only sent only to the host running Pod. All volumes requested by a Pod must be mounted before the containers within the Pod are started. So a secret must exist prior to being requested.

Mounting Secrets as Volumes

```
...
spec:
  containers:
    - image: busybox
      command:
        - sleep
        - "3600"
      volumeMounts:
        - mountPath: /mysqlpassword
          name: mysql
      name: busy
    volumes:
      - name: mysql
        secret:
          secretName: mysql
```

You can also mount secrets as files using a volume definition in a Pod manifest. The mount path will contain a file whose name will be the key of the secret created with the **kubectl create secret** step earlier.

Once the Pod is running, you can verify that the secret is indeed accessible in the container:

```
$ kubectl exec -ti busybox -- cat /mysqlpassword/password
```

```
LFTr@1n
```

9.6 ConfigMaps

Portable Data With ConfigMaps

- Decouple configuration data from container image
- Not encoded or encrypted
- Can be created from various sources
 - Multiple files in same directory
 - Individual files
 - Literal values
- Can be consumed in various ways
 - Container environmental variables
 - Use ConfigMap values in Pod commands
 - Populate Volume from ConfigMap
 - Add ConfigMap data to specific path in Volume
 - Set file names and access mode in Volume from ConfigMap data
 - Can be used by system components and controllers

A similar API resource to Secrets is a ConfigMap, except the data is not encoded. In keeping with the concept of decoupling in Kubernetes. Using a ConfigMap decouples the container image from configuration artifacts.

They store data as sets of key-value pairs or plain configuration files in any format. The data can come from a collection of files or all files in a directory. It can also be populated from a literal value.

A ConfigMap can be used in several different ways. A container can use the data as environmental variables from one or more sources. The values contained inside can be passed to commands inside the pod. A Volume or a file in a Volume can be created, including different names and particular access modes. In addition, cluster components like controllers can use the data.

Let's say you have a file on your local filesystem called `config.js`. You can create a ConfigMap. The `configmap` object will have a data section containing the content of the file:

```
$ kubectl get configmap foobar -o yaml
```

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: foobar
data:
  config.js: |
    {
    ...
}
```

Using ConfigMaps

```
env:  
- name: SPECIAL_LEVEL_KEY  
  valueFrom:  
    configMapKeyRef:  
      name: special-config  
      key: special.how  
  
volumes:  
- name: config-volume  
  configMap:  
    name: special-config
```

Like secrets, you can use ConfigMaps as environment variables or using a volume mount. They must exist prior to being used by a Pod, unless marked as optional. They also reside in a specific namespace.

In the case of environment variables, your pod manifest will use the `valueFrom` key and the `configMapKeyRef` value to read the values.

With volumes, define a volume with the `configMap` type in your Pod and mount it where it needs to be used.

9.7 Labs

Exercise 9.1: Create a ConfigMap

Overview

Container files are ephemeral, which can be problematic for some applications. Should a container be restarted the files will be lost. In addition, we need a method to share files between containers inside a Pod.

A **Volume** is a directory accessible to containers in a Pod. Cloud providers offer volumes which persist further than the life of the Pod, such that AWS or GCE volumes could be pre-populated and offered to Pods, or transferred from one Pod to another. **Ceph** is also another popular solution for dynamic, persistent volumes.

Unlike current **Docker** volumes a Kubernetes volume has the lifetime of the Pod, not the containers within. You can also use different types of volumes in the same Pod simultaneously, but Volumes cannot mount in a nested fashion. Each must have their own mount point. Volumes are declared with `spec.volumes` and mount points with `spec.containers.volumeMounts` parameters. Each particular volume type, may have other restrictions. <https://kubernetes.io/docs/concepts/storage/volumes/#types-of-volumes>

We will also work with a **ConfigMap**, which is basically a set of key-value pairs. This data can be made available so that a Pod can read the data as environment variables or configuration data. A **ConfigMap** is similar to a **Secret**, except they are not base64 byte encoded arrays. They are stored as strings and can be read in serialized form.

There are three different ways a ConfigMap can ingest data, from a literal value, from a file or from a directory of files.

1. We will create a ConfigMap containing primary colors. We will create a series of files to ingest into the ConfigMap. First, we create a directory `primary` and populate it with four files. Then we create a file in our home directory with our favorite color.

```
student@cp:~$ mkdir primary
student@cp:~$ echo c > primary/cyan
student@cp:~$ echo m > primary/magenta
student@cp:~$ echo y > primary/yellow
student@cp:~$ echo k > primary/black
student@cp:~$ echo "known as key" >> primary/black
student@cp:~$ echo blue > favorite
```

2. Now we will create the ConfigMap and populate it with the files we created as well as a literal value from the command line.

```
student@cp:~$ kubectl create configmap colors \
--from-literal=text=black \
--from-file=./favorite \
--from-file=./primary/
configmap/colors created
```

3. View how the data is organized inside the cluster. Use the `yaml` then the `json` output type to see the formatting.

```
student@cp:~$ kubectl get configmap colors
```

NAME	DATA	AGE
colors	6	30s

```
student@cp:~$ kubectl get configmap colors -o yaml
```

```
apiVersion: v1
data:
  black: |
    k
    known as key
  cyan: |
    c
  favorite: |
    blue
  magenta: |
    m
  text: black
  yellow: |
    y
kind: ConfigMap
<output omitted>
```

- Now we can create a Pod to use the ConfigMap. In this case a particular parameter is being defined as an environment variable.

```
student@cp:~$ cp /home/student/LFS458/SOLUTIONS/s_09/simpleshell.yaml .
```

```
student@cp:~$ vim simpleshell.yaml
```

YAML simpleshell.yaml

```
1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: shell-demo
5 spec:
6   containers:
7     - name: nginx
8       image: nginx
9       env:
10      - name: ilike
11        valueFrom:
12          configMapKeyRef:
13            name: colors
14            key: favorite
```

- Create the Pod and view the environmental variable. After you view the parameter, exit out and delete the pod.

```
student@cp:~$ kubectl create -f simpleshell.yaml
```

```
pod/shell-demo created
```

```
student@cp:~$ kubectl exec shell-demo -- /bin/bash -c 'echo $ilike'
```

```
blue
```

```
student@cp:~$ kubectl delete pod shell-demo
```

```
pod "shell-demo" deleted
```

6. All variables from a file can be included as environment variables as well. Comment out the previous env: stanza and add a slightly different envFrom to the file. Having new and old code at the same time can be helpful to see and understand the differences. Recreate the Pod, check all variables and delete the pod again. They can be found spread throughout the environment variable output.

```
student@cp:~$ vim simpleshell.yaml
```



simpleshell.yaml

```
1 <output_omitted>
2   image: nginx
3 #   env:
4 #     - name: ilike
5 #       valueFrom:
6 #         configMapKeyRef:
7 #           name: colors
8 #           key: favorite
9   envFrom:                      #<-- Same indent as image: line
10  - configMapRef:
11    name: colors
12
```

```
student@cp:~$ kubectl create -f simpleshell.yaml
```

```
pod/shell-demo created
```

```
student@cp:~$ kubectl exec shell-demo -- /bin/bash -c 'env'
```

```
black=k
known as key

KUBERNETES_SERVICE_PORT_HTTPS=443
cyan=c
<output_omitted>
```

```
student@cp:~$ kubectl delete pod shell-demo
```

```
pod "shell-demo" deleted
```

7. A ConfigMap can also be created from a YAML file. Create one with a few parameters to describe a car.

```
student@cp:~$ cp /home/student/LFS458/SOLUTIONS/s_09/car-map.yaml .
```

```
student@cp:~$ vim car-map.yaml
```



car-map.yaml

```

1 apiVersion: v1
2 kind: ConfigMap
3 metadata:
4   name: fast-car
5   namespace: default
6 data:
7   car.make: Ford
8   car.model: Mustang
9   car.trim: Shelby

```

8. Create the ConfigMap and verify the settings.

```
student@cp:~$ kubectl create -f car-map.yaml
configmap/fast-car created
```

```
student@cp:~$ kubectl get configmap fast-car -o yaml
```



```

1 apiVersion: v1
2 data:
3   car.make: Ford
4   car.model: Mustang
5   car.trim: Shelby
6 kind: ConfigMap
7 <output_omitted>
8

```

9. We will now make the ConfigMap available to a Pod as a mounted volume. You can again comment out the previous environmental settings and add the following new stanza. The containers: and volumes: entries are indented the same number of spaces.

```
student@cp:~$ vim simpleshell.yaml
```



simpleshell.yaml

```

1 <output_omitted>
2 spec:
3   containers:
4     - name: nginx
5       image: nginx
6       volumeMounts:
7         - name: car-vol
8           mountPath: /etc/cars
9   volumes:
10    - name: car-vol
11      configMap:
12        name: fast-car
13 <comment out rest of file>
14

```

10. Create the Pod again. Verify the volume exists and the contents of a file within. Due to the lack of a carriage return in the file your next prompt may be on the same line as the output, Shelby.

```
student@cp:~$ kubectl create -f simpleshell.yaml
```

pod "shell-demo" created

```
student@cp:~$ kubectl exec shell-demo -- /bin/bash -c 'df -ha |grep car'
```

/dev/root 9.6G 3.2G 6.4G 34% /etc/cars

```
student@cp:~$ kubectl exec shell-demo -- /bin/bash -c 'cat /etc/cars/car.trim'
```

Shelby #<-- Then your prompt

11. Delete the Pod and ConfigMaps we were using.

```
student@cp:~$ kubectl delete pods shell-demo
```

pod "shell-demo" deleted

```
student@cp:~$ kubectl delete configmap fast-car colors
```

configmap "fast-car" deleted
configmap "colors" deleted

✍ Exercise 9.2: Creating a Persistent NFS Volume (PV)

We will first deploy an NFS server. Once tested we will create a persistent NFS volume for containers to claim.

1. Install the software on your cp node.

```
student@cp:~$ sudo apt-get update && sudo \  
apt-get install -y nfs-kernel-server
```

<output_omitted>

2. Make and populate a directory to be shared. Also give it similar permissions to /tmp/

```
student@cp:~$ sudo mkdir /opt/sfw
```

```
student@cp:~$ sudo chmod 1777 /opt/sfw/
```

```
student@cp:~$ sudo bash -c 'echo software > /opt/sfw/hello.txt'
```

3. Edit the NFS server file to share out the newly created directory. In this case we will share the directory with all. You can always **snoop** to see the inbound request in a later step and update the file to be more narrow.

```
student@cp:~$ sudo vim /etc/exports
```

/opt/sfw/ *(rw,sync,no_root_squash,subtree_check)

4. Cause /etc/exports to be re-read:

```
student@cp:~$ sudo exportfs -ra
```

5. Test by mounting the resource from your **second** node.

```
student@worker:~$ sudo apt-get -y install nfs-common
```

```
<output_omitted>
```

```
student@worker:~$ showmount -e k8scp
```

```
Export list for k8scp:  
/opt/sfw *
```

```
student@worker:~$ sudo mount k8scp:/opt/sfw /mnt
```

```
student@worker:~$ ls -l /mnt
```

```
total 4  
-rw-r--r-- 1 root root 23 Aug 28 17:55 hello.txt
```

6. Return to the cp node and create a YAML file for the object with kind, `PersistentVolume`. Use the hostname of the cp server and the directory you created in the previous step. Only syntax is checked, an incorrect name or directory will not generate an error, but a Pod using the resource will not start. Note that the `accessModes` do not currently affect actual access and are typically used as labels instead.

```
student@cp:~$ cp /home/student/LFS458/SOLUTIONS/s_09/PVol.yaml .
```

```
student@cp:~$ vim PVol.yaml
```

YAML

PVol.yaml

```
1 apiVersion: v1
2 kind: PersistentVolume
3 metadata:
4   name: pvvol-1
5 spec:
6   capacity:
7     storage: 1Gi
8   accessModes:
9     - ReadWriteMany
10  persistentVolumeReclaimPolicy: Retain
11  nfs:
12    path: /opt/sfw
13    server: k8scp #<-- Edit to match cp node
14    readOnly: false
```

7. Create the persistent volume, then verify its creation.

```
student@cp:~$ kubectl create -f PVol.yaml
```

```
persistentvolume/pvvol-1 created
```

```
student@cp:~$ kubectl get pv
```

NAME	CAPACITY	ACCESS MODES	RECLAIM POLICY	STATUS	CLAIM	STORAGECLASS
→ VOLUMEATTRIBUTESCLASS	REASON	AGE				<unset>
pvvol-1	1Gi	RWX	Retain	Available		
						6s

Exercise 9.3: Creating a Persistent Volume Claim (PVC)

Before Pods can take advantage of the new PV we need to create a **Persistent Volume Claim (PVC)**.

1. Begin by determining if any currently exist.

```
student@cp:~$ kubectl get pvc
```

```
No resources found in default namespace.
```

2. Create a YAML file for the new pvc.

```
student@cp:~$ cp /home/student/LFS458/SOLUTIONS/s_09/pvc.yaml .
```

```
student@cp:~$ vim pvc.yaml
```

YAML
pvc.yaml

```
1 apiVersion: v1
2 kind: PersistentVolumeClaim
3 metadata:
4   name: pvc-one
5 spec:
6   accessModes:
7     - ReadWriteMany
8   resources:
9     requests:
10       storage: 200Mi
```

3. Create and verify the new pvc is bound. Note that the size is 1Gi, even though 200Mi was suggested. Only a volume of at least that size could be used.

```
student@cp:~$ kubectl create -f pvc.yaml
```

```
persistentvolumeclaim/pvc-one created
```

```
student@cp:~$ kubectl get pvc
```

NAME	STATUS	VOLUME	CAPACITY	ACCESS MODES	STORAGECLASS	VOLUMEATTRIBUTESCLASS	AGE
pvc-one	Bound	pvvol-1	1Gi	RWX		<unset>	7s

4. Look at the status of the pv again, to determine if it is in use. It should show a status of Bound.

```
student@cp:~$ kubectl get pv
```

NAME	CAPACITY	ACCESS MODES	RECLAIM POLICY	STATUS	CLAIM	STORAGECLASS
→ VOLUMEATTRIBUTESCLASS	REASON	AGE				<unset>
pvvol-1	1Gi	RWX	Retain	Available		6s

pvvol-1	1Gi	RWX	Retain	Bound	default/pvc-one
	↪ <unset>		3m45s		

5. Create a new deployment to use the pvc. We will copy and edit an existing deployment yaml file. We will change the deployment name then add a volumeMounts section under containers and a volumes section to the general spec. The name used must match in both places, whatever name you use. The claimName must match an existing pvc. As shown in the following example. The volumes line is the same indent as containers and dnsPolicy.

```
student@cp:~$ cp /home/student/LFS458/SOLUTIONS/s_09/nfs-pod.yaml .
```

```
student@cp:~$ vim nfs-pod.yaml
```

```

YAML nfs-pod.yaml
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   annotations:
5     deployment.kubernetes.io/revision: "1"
6   generation: 1
7   labels:
8     run: nginx
9   name: nginx-nfs           #<-- Edit name
10  namespace: default
11 spec:
12   replicas: 1
13   selector:
14     matchLabels:
15       run: nginx
16   strategy:
17     rollingUpdate:
18       maxSurge: 1
19       maxUnavailable: 1
20     type: RollingUpdate
21   template:
22     metadata:
23       creationTimestamp: null
24     labels:
25       run: nginx
26   spec:
27     containers:
28       - image: nginx
29         imagePullPolicy: Always
30         name: nginx
31         volumeMounts:          #<-- Add these three lines
32           - name: nfs-vol
33             mountPath: /opt
34         ports:
35           - containerPort: 80
36             protocol: TCP
37         resources: {}
38         terminationMessagePath: /dev/termination-log
39         terminationMessagePolicy: File
40     volumes:                  #<-- Add these four lines
41       - name: nfs-vol
42         persistentVolumeClaim:
43           claimName: pvc-one
44         dnsPolicy: ClusterFirst

```



```

45      restartPolicy: Always
46      schedulerName: default-scheduler
47      securityContext: {}
48      terminationGracePeriodSeconds: 30

```

6. Create the pod using the newly edited file.

```
student@cp:~$ kubectl create -f nfs-pod.yaml
```

```
deployment.apps/nginx-nfs created
```

7. Look at the details of the pod. You may see the daemonset pods running as well.

```
student@cp:~$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
nginx-nfs-1054709768-s8g28	1/1	Running	0	3m

```
student@cp:~$ kubectl describe pod nginx-nfs-1054709768-s8g28
```

```

Name:           nginx-nfs-1054709768-s8g28
Namespace:      default
Priority:       0
Node:          worker/10.128.0.5

<output_omitted>

Mounts:
  /opt from nfs-vol (rw)

<output_omitted>

Volumes:
  nfs-vol:
    Type:      PersistentVolumeClaim (a reference to a PersistentV...
    ClaimName:  pvc-one
    ReadOnly:   false
<output_omitted>

```

✍ Exercise 9.4: Using a ResourceQuota to Limit PVC Count and Usage

The flexibility of cloud-based storage often requires limiting consumption among users. We will use the ResourceQuota object to both limit the total consumption as well as the number of persistent volume claims.

1. Begin by deleting the deployment we had created to use NFS, the pv and the pvc.

```
student@cp:~$ kubectl delete deploy nginx-nfs
```

```
deployment.apps "nginx-nfs" deleted
```

```
student@cp:~$ kubectl delete pvc pvc-one
```

```
persistentvolumeclaim "pvc-one" deleted
```

```
student@cp:~$ kubectl delete pv pvvol-1
```

```
persistentvolume "pvvol-1" deleted
```

2. Create a yaml file for the ResourceQuota object. Set the storage limit to ten claims with a total usage of 500Mi.

```
student@cp:~$ cp /home/student/LFS458/SOLUTIONS/s_09/storage-quota.yaml .
```

```
student@cp:~$ vim storage-quota.yaml
```



storage-quota.yaml

```
1 apiVersion: v1
2 kind: ResourceQuota
3 metadata:
4   name: storagequota
5 spec:
6   hard:
7     persistentvolumeclaims: "10"
8     requests.storage: "500Mi"
```

3. Create a new namespace called small. View the namespace information prior to the new quota. Either the long name with double dashes --namespace or the nickname ns work for the resource.

```
student@cp:~$ kubectl create namespace small
```

```
namespace/small created
```

```
student@cp:~$ kubectl describe ns small
```

```
Name:           small
Labels:        <none>
Annotations:  <none>
Status:        Active
```

```
No resource quota.
```

```
No resource limits.
```

4. Create a new pv and pvc in the small namespace.

```
student@cp:~$ kubectl -n small create -f PVol.yaml
```

```
persistentvolume/pvvol-1 created
```

```
student@cp:~$ kubectl -n small create -f pvc.yaml
```

```
persistentvolumeclaim/pvc-one created
```

5. Create the new resource quota, placing this object into the `small` namespace.

```
student@cp:~$ kubectl -n small create -f storage-quota.yaml
```

```
resourcequota/storagequota created
```

6. Verify the `small` namespace has quotas. Compare the output to the same command above.

```
student@cp:~$ kubectl describe ns small
```

```
Name:          small
Labels:        <none>
Annotations:   <none>
Status:        Active

Resource Quotas
Name:           storagequota
Resource        Used   Hard
-----
-----      -----
persistentvolumeclaims  1     10
requests.storage      200Mi  500Mi

No resource limits.
```

7. Remove the namespace line from the `nfs-pod.yaml` file. Should be around line 11 or so. This will allow us to pass other namespaces on the command line.

```
student@cp:~$ vim nfs-pod.yaml
```

8. Create the container again.

```
student@cp:~$ kubectl -n small create -f nfs-pod.yaml
```

```
deployment.apps/nginx-nfs created
```

9. Determine if the deployment has a running pod.

```
student@cp:~$ kubectl -n small get deploy
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
nginx-nfs	1/1	1	1	43s

```
student@cp:~$ kubectl -n small describe deploy nginx-nfs
```

```
<output_omitted>
```

10. Look to see if the pods are ready.

```
student@cp:~$ kubectl -n small get pod
```

NAME	READY	STATUS	RESTARTS	AGE
nginx-nfs-2854978848-g3khf	1/1	Running	0	37s

11. Ensure the Pod is running and is using the NFS mounted volume. If you pass the namespace first Tab will auto-complete the pod name.

```
student@cp:~$ kubectl -n small describe pod \
    nginx-nfs-2854978848-g3khf
```

```
Name:           nginx-nfs-2854978848-g3khf
Namespace:     small
<output_omitted>

Mounts:
  /opt from nfs-vol (rw)
<output_omitted>
```

12. View the quota usage of the namespace

```
student@cp:~$ kubectl describe ns small
```

```
<output_omitted>

Resource Quotas
  Name:           storagequota
  Resource       Used   Hard
  ----
  persistentvolumeclaims  1      10
  requests.storage    200Mi  500Mi

No resource limits.
```

13. Create a 300M file inside of the `/opt/sfw` directory on the host and view the quota usage again. Note that with NFS the size of the share is not counted against the deployment.

```
student@cp:~$ sudo dd if=/dev/zero of=/opt/sfw/bigfile bs=1M count=300
```

```
300+0 records in
300+0 records out
314572800 bytes (315 MB, 300 MiB) copied, 0.196794 s, 1.6 GB/s
```

```
student@cp:~$ kubectl describe ns small
```

```
<output_omitted>
Resource Quotas
  Name:           storagequota
  Resource       Used   Hard
  ----
  persistentvolumeclaims  1      10
  requests.storage    200Mi  500Mi
<output_omitted>
```

```
student@cp:~$ du -h /opt/
```

```
301M    /opt/sfw
41M     /opt/cni/bin
41M     /opt/cni
341M    /opt/
```

14. Now let us illustrate what happens when a deployment requests more than the quota. Begin by shutting down the existing deployment.

```
student@cp:~$ kubectl -n small get deploy
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
nginx-nfs	1	1	1	11m

```
student@cp:~$ kubectl -n small delete deploy nginx-nfs
```

```
deployment.apps "nginx-nfs" deleted
```

15. Once the Pod has shut down view the resource usage of the namespace again. Note the storage did not get cleaned up when the pod was shut down.

```
student@cp:~$ kubectl describe ns small
```

```
<output_omitted>
Resource Quotas
Name:           storagequota
Resource        Used   Hard
-----
persistentvolumeclaims 1     10
requests.storage    200Mi  500Mi
```

16. Remove the pvc then view the pv it was using. Note the RECLAIM POLICY and STATUS.

```
student@cp:~$ kubectl -n small get pvc
```

NAME	STATUS	VOLUME	CAPACITY	ACCESSMODES	STORAGECLASS	AGE
pvc-one	Bound	pvvol-1	1Gi	RWX		19m

```
student@cp:~$ kubectl -n small delete pvc pvc-one
```

```
persistentvolumeclaim "pvc-one" deleted
```

```
student@cp:~$ kubectl -n small get pv
```

NAME	CAPACITY	ACCESSMODES	RECLAIMPOLICY	STATUS	CLAIM
STORAGECLASS	REASON	AGE			
pvvol-1	1Gi	RWX	Retain	Released	small/pvc-one 44m

17. Dynamically provisioned storage uses the ReclaimPolicy of the StorageClass which could be Delete, Retain, or some types allow Recycle. Manually created persistent volumes default to Retain unless set otherwise at creation. The default storage policy is to retain the storage to allow recovery of any data. To change this begin by viewing the yaml output.

```
student@cp:~$ kubectl get pv/pvvol-1 -o yaml
```

Y
A
M
L
....

```
2      path: /opt/sfw
```

YAML

```

3   server: cp
4   persistentVolumeReclaimPolicy: Retain
5   status:
6     phase: Released
7

```

18. Currently we will need to delete and re-create the object. Future development on a deleter plugin is planned. We will re-create the volume and allow it to use the Retain policy, then change it once running.

```
student@cp:~$ kubectl delete pv/pvvol-1
```

```
persistentvolume "pvvol-1" deleted
```

```
student@cp:~$ grep Retain PVol.yaml
```

```
  persistentVolumeReclaimPolicy: Retain
```

```
student@cp:~$ kubectl create -f PVol.yaml
```

```
persistentvolume "pvvol-1" created
```

19. We will use kubectl patch to change the retention policy to Delete. The yaml output from before can be helpful in getting the correct syntax.

```
student@cp:~$ kubectl patch pv pvvol-1 -p \
'{"spec":{"persistentVolumeReclaimPolicy":"Delete"}}'
```

```
persistentvolume/pvvol-1 patched
```

```
student@cp:~$ kubectl get pv/pvvol-1
```

NAME	CAPACITY	ACCESSMODES	RECLAIMPOLICY	STATUS	CLAIM
STORAGECLASS	REASON	AGE			
pvvol-1	1Gi	RWX	Delete	Available	2m

20. View the current quota settings.

```
student@cp:~$ kubectl describe ns small
```

```
....  
requests.storage 0 500Mi
```

21. Create the pvc again. Even with no pods running, note the resource usage.

```
student@cp:~$ kubectl -n small create -f pvc.yaml
```

```
persistentvolumeclaim/pvc-one created
```

```
student@cp:~$ kubectl describe ns small
```

```
....  
requests.storage      200Mi      500Mi
```

22. Remove the existing quota from the namespace.

```
student@cp:~$ kubectl -n small get resourcequota
```

NAME	CREATED AT
storagequota	2023-08-25T04:10:02Z

```
student@cp:~$ kubectl -n small delete resourcequota storagequota
```

```
resourcequota "storagequota" deleted
```

23. Edit the storagequota.yaml file and lower the capacity to 100Mi.

```
student@cp:~$ vim storage-quota.yaml
```

```
....  
2   requests.storage: "100Mi"  
3
```

24. Create and verify the new storage quota. Note the hard limit has already been exceeded.

```
student@cp:~$ kubectl -n small create -f storage-quota.yaml
```

```
resourcequota/storagequota created
```

```
student@cp:~$ kubectl describe ns small
```

```
....  
persistentvolumeclaims      1      10  
requests.storage           200Mi  100Mi  
  
No resource limits.
```

25. Create the deployment again. View the deployment. Note there are no errors seen.

```
student@cp:~$ kubectl -n small create -f nfs-pod.yaml
```

```
deployment.apps/nginx-nfs created
```

```
student@cp:~$ kubectl -n small describe deploy/nginx-nfs
```

Name:	nginx-nfs
Namespace:	small
<output_omitted>	

26. Examine the pods to see if they are actually running.

```
student@cp:~$ kubectl -n small get po
```

NAME	READY	STATUS	RESTARTS	AGE
nginx-nfs-2854978848-vb6bh	1/1	Running	0	58s

27. As we were able to deploy more pods even with apparent hard quota set, let us test to see if the reclaim of storage takes place. Remove the deployment and the persistent volume claim.

```
student@cp:~$ kubectl -n small delete deploy nginx-nfs
```

```
deployment.apps "nginx-nfs" deleted
```

```
student@cp:~$ kubectl -n small delete pvc/pvc-one
```

```
persistentvolumeclaim "pvc-one" deleted
```

28. View if the persistent volume exists. You will see it attempted a removal, but failed. If you look closer you will find the error has to do with the lack of a deleter volume plugin for NFS. Other storage protocols have a plugin.

```
student@cp:~$ kubectl -n small get pv
```

NAME	CAPACITY	ACCESSMODES	RECLAIMPOLICY	STATUS	CLAIM
STORAGECLASS		REASON	AGE		
pvvol-1	1Gi	RWX	Delete	Failed	small/pvc-one 20m

29. Ensure the deployment, pvc and pv are all removed.

```
student@cp:~$ kubectl delete pv/pvvol-1
```

```
persistentvolume "pvvol-1" deleted
```

30. Edit the persistent volume YAML file and change the `persistentVolumeReclaimPolicy`: to Recycle.

```
student@cp:~$ vim PVol.yaml
```



```
1  ....
2  persistentVolumeReclaimPolicy: Recycle
3  ....
4
```

31. Add a LimitRange to the namespace and attempt to create the persistent volume and persistent volume claim again. We can use the LimitRange we used earlier.

```
student@cp:~$ kubectl -n small create -f low-resource-range.yaml
```

```
limitrange/low-resource-range created
```

32. View the settings for the namespace. Both quotas and resource limits should be seen.

```
student@cp:~$ kubectl describe ns small
```

```
<output_omitted>
Resource Limits
  Type      Resource  Min   Max   Default Request  Default Limit  ...
  ----      -----  ---  ---  -----  -----  -----  ...
Container  cpu        -     -    500m          1           -
Container  memory     -     -   100Mi        500Mi       -
```

33. Create the persistent volume again. View the resource. Note the Reclaim Policy is Recycle.

```
student@cp:~$ kubectl -n small create -f PVol.yaml
```

```
persistentvolume/pvvol-1 created
```

```
student@cp:~$ kubectl get pv
```

NAME	CAPACITY	ACCESS MODES	RECLAIM POLICY	STATUS	...
pvvol-1	1Gi	RWX	Recycle	Available	...

34. Attempt to create the persistent volume claim again. The quota only takes effect if there is also a resource limit in effect.

```
student@cp:~$ kubectl -n small create -f pvc.yaml
```

```
Error from server (Forbidden): error when creating "pvc.yaml":  
persistentvolumeclaims "pvc-one" is forbidden: exceeded quota:  
storagequota, requested: requests.storage=200Mi, used:  
requests.storage=0, limited: requests.storage=100Mi
```

35. Edit the resourcequota to increase the requests.storage to 500mi.

```
student@cp:~$ kubectl -n small edit resourcequota
```

YAML

```
1 ....
2 spec:
3   hard:
4     persistentvolumeclaims: "10"
5     requests.storage: 500Mi
6 status:
7   hard:
8     persistentvolumeclaims: "10"
9 ....
10
```

36. Create the pvc again. It should work this time. Then create the deployment again.

```
student@cp:~$ kubectl -n small create -f pvc.yaml
```

```
persistentvolumeclaim/pvc-one created
```

```
student@cp:~$ kubectl -n small create -f nfs-pod.yaml
```

```
deployment.apps/nginx-nfs created
```

37. View the namespace settings.

```
student@cp:~$ kubectl describe ns small
```

```
<output omitted>
```

38. Delete the deployment. View the status of the pv and pvc.

```
student@cp:~$ kubectl -n small delete deploy nginx-nfs
```

```
deployment.apps "nginx-nfs" deleted
```

```
student@cp:~$ kubectl -n small get pvc
```

NAME	STATUS	VOLUME	CAPACITY	ACCESS MODES	STORAGECLASS	AGE
pvc-one	Bound	pvvol-1	1Gi	RWX		7m

```
student@cp:~$ kubectl -n small get pv
```

NAME	CAPACITY	ACCESS MODES	RECLAIM POLICY	STATUS	CLAIM	...
pvvol-1	1Gi	RWX	Recycle	Bound	small/pvc-one	...

39. Delete the pvc and check the status of the pv. It should show as Available.

```
student@cp:~$ kubectl -n small delete pvc pvc-one
```

```
persistentvolumeclaim "pvc-one" deleted
```

```
student@cp:~$ kubectl -n small get pv
```

NAME	CAPACITY	ACCESS MODES	RECLAIM POLICY	STATUS	CLAIM	STORA...
pvvol-1	1Gi	RWX	Recycle	Available		...

40. Remove the pv and any other resources created during this lab.

```
student@cp:~$ kubectl delete pv pvvol-1
```

```
persistentvolume "pvvol-1" deleted
```

✍ Exercise 9.5: Using StorageClass to Dynamically provision a volume

StorageClasses in Kubernetes simplify and automate the process of provisioning and managing storage resources, provide users with the flexibility to choose appropriate storage types for their workloads, and help administrators enforce policies and manage storage infrastructure more effectively. StorageClasses enables dynamic provisioning of storage resources. Without StorageClasses, administrators have to manually create PersistentVolumes (PVs) for each PersistentVolumeClaim (PVC) made by users. With StorageClasses, this process is automated. When a user creates a PVC and specifies a StorageClasses, the system automatically creates a corresponding PV that meets the requirements.

1. Begin by listing to see if we have any storage class available on our cluster.

```
student@cp:~$ kubectl get sc
```

No resources found

2. We don't have any StorageClass created. Before we can create the sc, we need to deploy the provisioner. Kubernetes doesn't include an internal NFS provisioner. We need to use an external provisioner to create a StorageClass for NFS. Let us deploy a nfs provisioner.

```
student@cp:~$ helm repo add nfs-subdir-external-provisioner \
https://kubernetes-sigs.github.io/nfs-subdir-external-provisioner/
```

"nfs-subdir-external-provisioner" has been added to your repositories

```
student@cp:~$ helm install nfs-subdir-external-provisioner \
nfs-subdir-external-provisioner/nfs-subdir-external-provisioner \
--set nfs.server=k8scp \
--set nfs.path=/opt/sfw/
```

NAME: nfs-subdir-external-provisioner
LAST DEPLOYED: Mon Jan 8 12:11:39 2024
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None

3. The installation also created a StorageClass for us.

```
student@cp:~$ kubectl get sc
```

NAME	PROVISIONER	RECLAIMPOLICY	VOLUMEBINDINGMODE
↪ nfs-client	cluster.local/nfs-subdir-external-provisioner	Delete	Immediate

4. List to see if there are any PV and PVC available. Clean up in previous lab should have removed all of them.

```
student@cp:~$ kubectl get pv,pvc
```

No resources found

5. Create a YAML file for the new pvc.

```
student@cp:~$ cp /home/student/LFS458/SOLUTIONS/s_09/pvc-sc.yaml .
```

```
student@cp:~$ vim pvc-sc.yaml
```



pvc-sc.yaml

```
1 apiVersion: v1
2 kind: PersistentVolumeClaim
```



```

3 metadata:
4   name: pvc-one
5 spec:
6   storageClassName: nfs-client
7   accessModes:
8     - ReadWriteMany
9   resources:
10    requests:
11      storage: 200Mi
12

```

6. Create and verify when the new pvc is created, a dynamic volume is provisioned.

```
student@cp:~$ kubectl create -f pvc-sc.yaml
persistentvolumeclaim/pvc-one created
```

```
student@cp:~$ kubectl get pv,pvc
```

NAME	POLICY	STATUS	CLAIM	STORAGECLASS	REASON	CAPACITY	ACCESS MODES	RECLAIM
persistentvolume/pvc-71149612-33f1-4b18-916d-c67f79aca797	→ Bound	Bound	default/pvc-one	nfs-client		200Mi	RWX	Delete
						28s		

NAME	STATUS	VOLUME	CAPACITY
ACCESS MODES	STORAGECLASS	AGE	
persistentvolumeclaim/pvc-one	Bound	pvc-71149612-33f1-4b18-916d-c67f79aca797	200Mi
nfs-client	28s		RWX

7. Create a new pod to use the pvc.

```
student@cp:~$ cp /home/student/LFS458/SOLUTIONS/s_09/pod-sc.yaml .
```

```
student@cp:~$ vim pod-sc.yaml
```



pod-sc.yaml

```

1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: web-server
5 spec:
6   containers:
7     - image: nginx
8       name: web-container
9       volumeMounts:
10      - name: nfs-volume
11        mountPath: /usr/share/nginx/html
12   volumes:
13     - name: nfs-volume
14       persistentVolumeClaim:
15         claimName: pvc-one

```

8. Create the pod using the file.

```
student@cp:~$ kubectl create -f pod-sc.yaml
```

```
pod/web-server created
```

9. Create a new file and copy it inside the pod.

```
student@cp:~$ echo "Welcome to the demo of storage class" > index.html  
student@cp:~$ kubectl cp index.html web-server:/usr/share/nginx/html
```

10. The file was copied on to the default location of the nginx server. Instead of the ephemeral read-write layer of the container, the file is saved on the NFS server as we have made use of the PV.

```
student@cp:~$ ls -l /opt/sfw/default-pvc-one-pvc-<Hit the Tab key>
```

```
-rw-rw-r-- 1 student student 37 Jan 8 13:08 index.html
```

11. Cleanup by deleting the pod,volume claim.

```
student@cp:~$ kubectl delete pod/web-server pvc/pvc-one
```

```
pod "web-server" deleted  
persistentvolumeclaim "pvc-one" deleted
```


Chapter 10

Services



10.1	Overview	226
10.2	Accessing Services	228
10.3	DNS	234
10.4	Labs	236

10.1 Overview

Overview

- Essential to micro-service architecture
- Connect Pods together, or outside cluster
- Service abstraction
- Load balancing
- Service types
- DNS resolution

As touched on previously Kubernetes architecture is built on the concept of transient, decoupled objects connected together. Services are the agents which connect Pods together, or provide access outside of the cluster with the idea that any particular Pod could be terminated and rebuilt. Typically using Labels, the refreshed Pod is connected and the micro-service continues to provide the expected resource via an Endpoint object. Google has been working on Extensible Service Proxy (ESP), based off the **nginx** HTTP reverse proxy server, to provide a more flexible and powerful object than Endpoints, but ESP has not been adopted much outside of Google App Engine or GKE environments.

There are several different service types, with the flexibility to add more as necessary. Each service can be exposed internally or externally to the cluster. A service can also connect internal resources to an external resource such as a third-party database.

The **kube-proxy** agent watches the Kubernetes API for new services and endpoints being created on each node. It opens random ports and listens for traffic to the ClusterIPPort:, and redirects the traffic to the randomly generated service endpoints. Services provide automatic load-balancing, matching a label-query. While there is no configuration of this option there is the possibility of session affinity via IP. As well a headless service, one without a fixed IP nor load-balancing, can be configured.

Unique IP addresses are assigned, and configured via the **etcd** database, so that Services implement **iptables** to route traffic, but could leverage other technologies to provide access to resources in the future.

Service Update Pattern

- Uses labels to match target Pods
- Rolling Deployments
- Traffic Shift

Labels are used to determine which Pods should receive traffic from a service. As we have learned labels can be dynamically updated for an object, which may affect which Pods continue to connect to a service.

The default update pattern is for a rolling deployment, where new Pods are added, with different versions of an application, and due to automatic load balancing receive traffic along with previous versions of the application.

Should there be a difference in applications deployed, such that clients would have issues communicating with different versions you may consider a more specific label for the deployment which includes a version number. When the deployment creates a new `replicaSet` for the update the label would not match. Once the new Pods have been created, and perhaps allowed to fully initialize, we would edit the labels for which the Service connects. Traffic would shift to the new and ready version, minimizing client version confusion.

10.2 Accessing Services

Accessing an Application With A Service

```
$ kubectl expose deployment/nginx --port=80 --type=NodePort  
$ kubectl get svc
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kubernetes	ClusterIP	10.0.0.1	<none>	443/TCP	18h
nginx	NodePort	10.0.0.112	<none>	80:31230/TCP	5s

```
$ kubectl get svc nginx -o yaml
```

```
apiVersion: v1  
kind: Service  
...  
spec:  
  clusterIP: 10.0.0.112  
  ports:  
    - nodePort: 31230  
  ...
```

Open browser <http://Public-IP:31230>

The basic steps to use **kubectl** to access a new service. The `kubectl expose` command created a service for the **nginx** deployment. This service used port 80 and generated a random port on all the nodes. A particular port and `targetPort` can also be passed during object creation to avoid random values. The `targetPort` defaults to the `port`, but could be set to any value including a string referring to a port on a back-end Pod. Each Pod could have a different port, but traffic is still passed via the name. Switching traffic to a different port would maintain a client connection, while changing versions of software, for example.

The `kubectl get svc` command gave you a list of all the existing services, and we saw the **nginx** service, which was created with an internal cluster IP.

The range of cluster IPs and the range of ports used for the random NodePort are configurable in the API server startup options.

Services can also be used to point to a service in a different Namespace or even a resource outside the cluster such as a legacy application not yet in Kubernetes.

Service Types

- ClusterIP
- NodePort
- LoadBalancer
- ExternalName

The **ClusterIP** service type is the default and only provides access internally (except if manually creating an external endpoint). The range of ClusterIP used is defined via an API server startup option.

The **NodePort** type is great for debugging or when a static IP address is necessary, such as opening a particular address through a firewall. NodePort range is defined in the Cluster configuration).

The **LoadBalancer** service was created to pass requests to a cloud provider like GKE or AWS. Private cloud solutions also may implement this service type if there is a Cloud provider plugin such as with **CloudStack** and **OpenStack**. Even without a cloud provider the address is made available to public traffic and packets are spread among the Pods in the deployment automatically.

A newer service is **ExternalName**, which is a bit different. It has no selectors, nor does it define ports or endpoints. It allows the return of an alias to an external service. The redirection happens at the DNS level, not via a proxy or forward. This object can be useful for services not yet brought into the Kubernetes cluster. A simple change of the type in the future would redirect traffic to the internal objects.

The **kubectl proxy** command creates a local service to access a ClusterIP. This can be useful for troubleshooting or development work.

Service Types (cont)

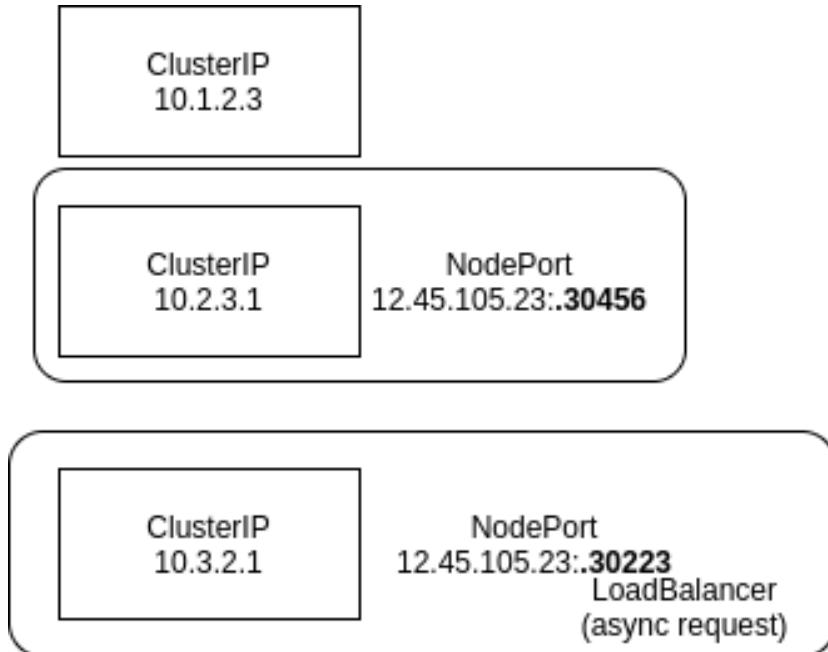


Figure 10.1: Built in services

While we have talked about three services, some build upon others. A Service is an operator running inside the `kube-controller-manager`, which sends API calls via the `kube-apiserver` to the Network Plugin (such as Cilium) and the `kube-proxy` pods running all nodes. The Service operator also creates an Endpoint operator, which queries for the ephemeral IP addresses of pods with a particular label. These agents work together to manage firewall rules using iptables or ipvs.

The **ClusterIP** service configures a persistent IP address and directs traffic sent to that address to the existing pod ephemeral addresses. This only handles traffic inside the cluster.

When a request for a **NodePort** is made, the operator first creates a **ClusterIP**. After the ClusterIP has been assigned, a high-numbered port is determined and a firewall rule is sent out so that traffic to the high-numbered port on any node will be sent to the persistent IP, which will then be sent to the pod(s).

A **LoadBalancer** does not create a load balancer. Instead, it creates a **NodePort** and makes an asynchronous request to use a load balancer. If a listener sees the request, as found when using public cloud providers, one would be created. Otherwise, the status will remain Pending as no load balancer has responded to the API call.

An ingress controller is a microservice running in a pod, listening to a high port on whichever node the pod may be running, which will send traffic to a Service based on the URL requested. It is not a built-in service, but is often used with services to centralize traffic to services. More on an ingress controller is found in a future chapter.

Services Diagram

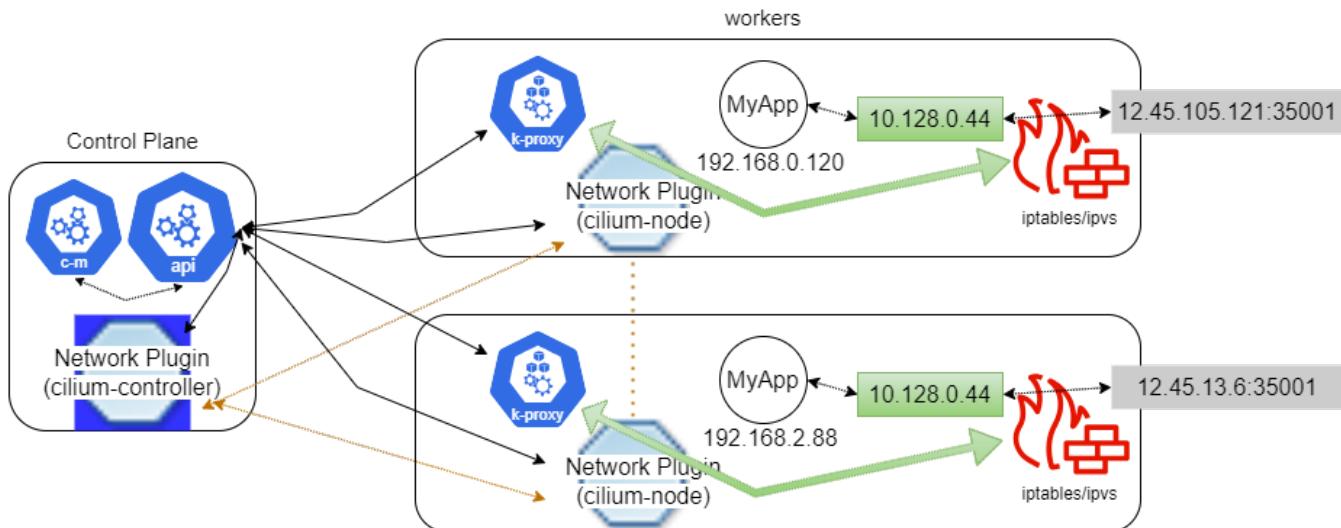


Figure 10.2: Service Traffic

The controllers of services and endpoints run inside the **kube-controller-manager** and send API calls to the **kube-apiserver**. API calls are then sent to the network plugin, such as **cilium-controller** which then communicates with agents on each node, such as **cilium-node**. Every kube-proxy is also sent an API call so that it can manage the firewall locally. The firewall is often **iptables** or **ipvs**. The **kube-proxy** mode is configured via a flag sent during initialization such as `mode=iptables` and could also be IPVS or userspace.

In the **iptables** proxy mode **kube-proxy** continues to get updates from the API server for changes in Service and Endpoint objects and updates rules for each object when created or removed.

The graphic above shows two workers each with a replica of **MyApp** running. A NodePort has been configured which will direct traffic from port 35001 to the ClusterIP and on to the ephemeral IP of the pod. All nodes use the same firewall rule. As a result you can connect to any node and Cilium will get the traffic to a node which is running the pod.

Overall Network View

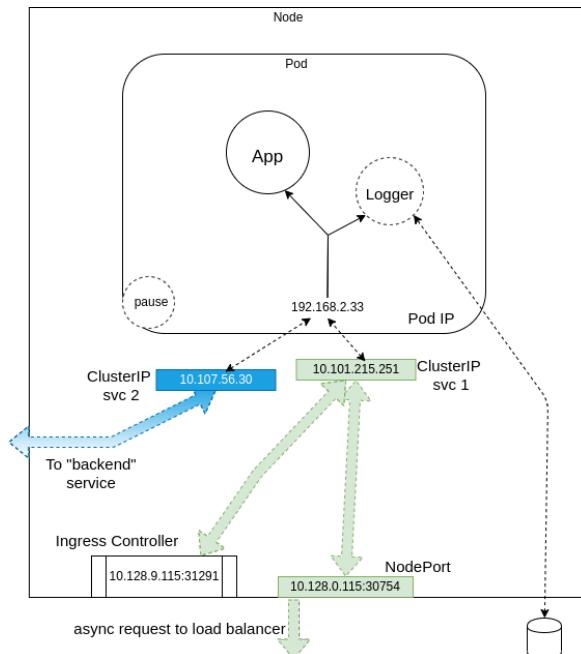


Figure 10.3: Example of Cluster Networking

An example of a multi-container pod with two services sending traffic to its ephemeral IP. The diagram also shows an ingress controller, which would typically be represented as a pod but has a different shape to show that it is listening to a high number port of an interface and is sending traffic to a service. Typically the service the ingress controller sends traffic to would be a **ClusterIP**, but the diagram shows that it would be possible to send traffic to a **NodePort** or a **LoadBalancer**.

Local Proxy For Development

- Quick way to check service
- Available on localhost
- Not exposed to Internet

```
$ kubectl proxy
```

```
Starting to serve on 127.0.0.1:8001
```

When developing an application or service, a quick way to check your service is to run a local proxy with **kubectl**. It will capture the shell unless you place it in the background. When running you can make calls to the Kubernetes API on localhost and also reach the verb:ClusterIP: services on their API URL. The IP and port where the proxy listens can be configured with command arguments.

To access a **ghost** service using the local proxy we could use this URL: <http://localhost:8001/api/v1/namespaces/default/services/ghost>.

If the service port has a name, the path will be http://localhost:8001/api/v1/namespaces/default/services/ghost:<port_name>.

10.3 DNS

DNS

- DNS provided using **CoreDNS** by default as of v1.13
- Exposed via the **kube-dns** service.
- Server started for zones served
- Configured via a configmap
- Each loads plugin chains to provide services
 - tls
 - prometheus
 - health
 - errors

The use of **CoreDNS** allows for a great amount of flexibility. Once the container starts it will run a `Server` for the zones it has been configured to serve. Then each server can load one or more `plugins` chains to provide other functionality. As with other microservices clients would access using a service, **kube-dns**.

The thirty or so in-tree plugins provide most common functionality, with an easy process to write and enable other plugins as necessary.

Common plugins can provide metrics for consumption by **Prometheus**, error logging, health reporting, and `tls` to configure certificates for TLS and gRPC servers.

More can be found here: <https://coredns.io/plugins/>.

Verifying DNS Registration

- Verify the service and the pod
- **netstat**
- **dig**
- **nc**
- Network Policy

To make sure that your DNS setup works well and that services get registered, the easiest way to do it is to run pod with a shell and network tools in the cluster, create a service to connect to the pod, then exec in it to do a DNS lookup.

Troubleshooting of DNS uses typical tools such as **nslookup**, **dig**, **nc**, **wireshark** and more. The difference is that we leverage a service to access the DNS server, so we need to check labels and selectors in addition to standard network concerns.

Other steps, similar to any DNS troubleshooting, would be to check the `/etc/resolv.conf` file of the container as well as Network Policies and firewalls. We will cover more on Network Policies in the Security chapter.

10.4 Labs

Exercise 10.1: Deploy A New Service

Overview

Services (also called **microservices**) are objects which declare a policy to access a logical set of Pods. They are typically assigned with labels to allow persistent access to a resource, when front or back end containers are terminated and replaced.

Native applications can use the Endpoints API for access. Non-native applications can use a Virtual IP-based bridge to access back end pods. ServiceTypes Type could be:

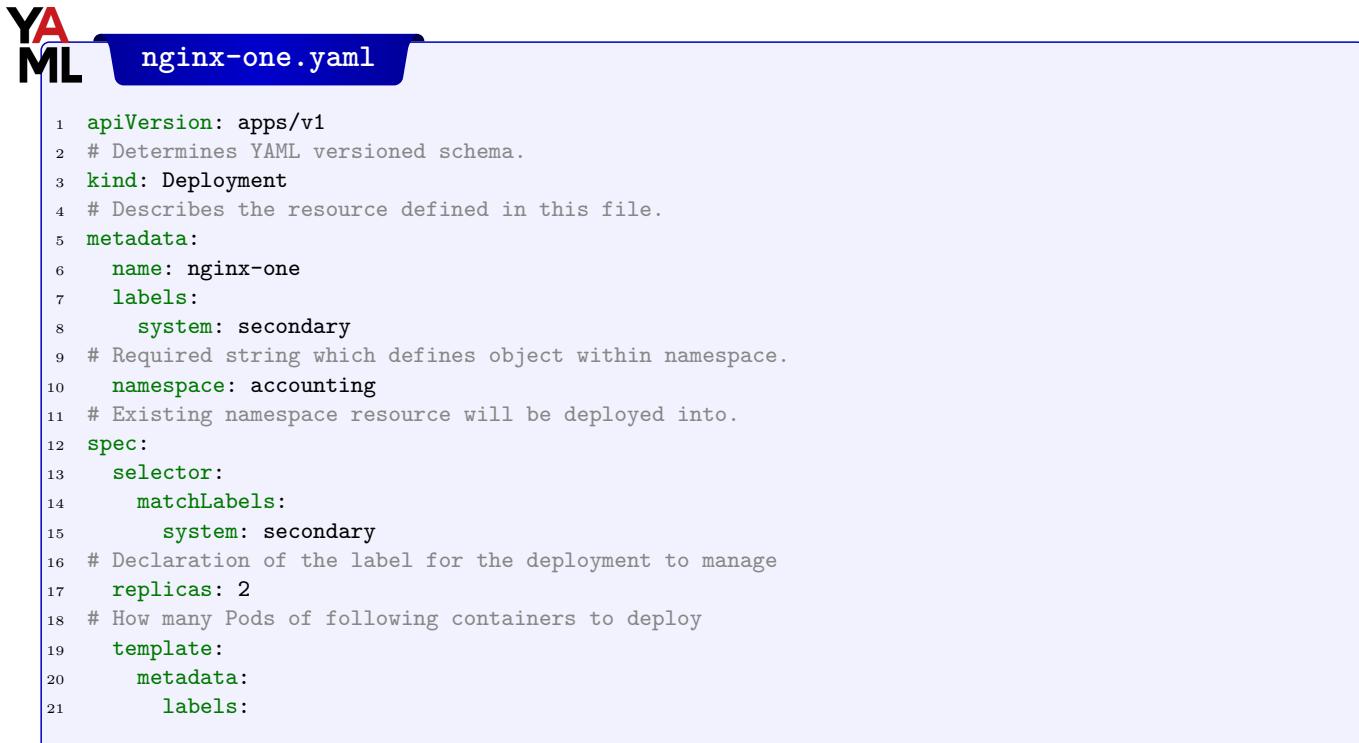
- **ClusterIP** default - exposes on a cluster-internal IP. Only reachable within cluster
- **NodePort** Exposes node IP at a static port. A ClusterIP is also automatically created.
- **LoadBalancer** Exposes service externally using cloud providers load balancer. NodePort and ClusterIP automatically created.
- **ExternalName** Maps service to contents of externalName using a CNAME record.

We use services as part of decoupling such that any agent or object can be replaced without interruption to access from client to back end application.

1. Deploy two **nginx** servers using **kubectl** and a new **.yaml** file. The kind should be Deployment and label it with **nginx**. Create two replicas and expose port 8080. What follows is a well documented file. There is no need to include the comments when you create the file. This file can also be found among the other examples in the tarball.

```
student@cp:~$ cp /home/student/LFS458/SOLUTIONS/s_10/nginx-one.yaml .
```

```
student@cp:~$ vim nginx-one.yaml
```



```

YAML nginx-one.yaml
1 apiVersion: apps/v1
2 # Determines YAML versioned schema.
3 kind: Deployment
4 # Describes the resource defined in this file.
5 metadata:
6   name: nginx-one
7   labels:
8     system: secondary
9 # Required string which defines object within namespace.
10 namespace: accounting
11 # Existing namespace resource will be deployed into.
12 spec:
13   selector:
14     matchLabels:
15       system: secondary
16 # Declaration of the label for the deployment to manage
17   replicas: 2
18 # How many Pods of following containers to deploy
19   template:
20     metadata:
21       labels:

```

YAML

```

22     system: secondary
23 # Some string meaningful to users, not cluster. Keys
24 # must be unique for each object. Allows for mapping
25 # to customer needs.
26   spec:
27     containers:
28       # Array of objects describing containerized application with a Pod.
29       # Referenced with shorthand spec.template.spec.containers
30         - image: nginx:1.20.1
31       # The Docker image to deploy
32         imagePullPolicy: Always
33         name: nginx
34       # Unique name for each container, use local or Docker repo image
35       ports:
36         - containerPort: 8080
37           protocol: TCP
38       # Optional resources this container may need to function.
39       nodeSelector:
40         system: secondOne
41       # One method of node affinity.

```

2. View the existing labels on the nodes in the cluster.

```
student@cp:~$ kubectl get nodes --show-labels
```

```
<output omitted>
```

3. Run the following command and look for the errors. Assuming there is no typo, you should have gotten an error about about the accounting namespace.

```
student@cp:~$ kubectl create -f nginx-one.yaml
```

```
Error from server (NotFound): error when creating
"nginx-one.yaml": namespaces "accounting" not found
```

4. Create the namespace and try to create the deployment again. There should be no errors this time.

```
student@cp:~$ kubectl create ns accounting
```

```
namespace/accounting" created
```

```
student@cp:~$ kubectl create -f nginx-one.yaml
```

```
deployment.apps/nginx-one created
```

5. View the status of the new pods. Note they do not show a Running status.

```
student@cp:~$ kubectl -n accounting get pods
```

NAME	READY	STATUS	RESTARTS	AGE
nginx-one-74dd9d578d-fcpmv	0/1	Pending	0	4m
nginx-one-74dd9d578d-r2d67	0/1	Pending	0	4m

6. View the node each has been assigned to (or not) and the reason, which shows under events at the end of the output.

```
student@cp:~$ kubectl -n accounting describe pod nginx-one-74dd9d578d-fcpmv
```

Name:	nginx-one-74dd9d578d-fcpmv			
Namespace:	accounting			
Node:	<none>			
<output omitted>				
Events:				
Type	Reason	Age	From
---	-----	---	----	
Warning	FailedScheduling	<unknown>	default-scheduler	
0/2 nodes are available: 2 node(s) didn't match node selector.				

7. Label the secondary node. Note the value is case sensitive. Verify the labels.

```
student@cp:~$ kubectl label node <worker_node_name> system=secondOne
```

node/worker labeled

```
student@cp:~$ kubectl get nodes --show-labels
```

NAME	STATUS	ROLES	AGE	VERSION	LABELS
cp	Ready	control-plane	14h	v1.30.1	beta.kubernetes.io/arch=amd64, beta.kubernetes.io/os=linux,kubernetes.io/arch=amd64,kubernetes.io/hostname=cp, kubernetes.io/os=linux,node-role.kubernetes.io/control-plane=, node.kubernetes.io/exclude-from-external-load-balancers=
worker	Ready	<none>	14h	v1.30.1	beta.kubernetes.io/arch=amd64, beta.kubernetes.io/os=linux,kubernetes.io/arch=amd64,kubernetes.io/hostname=worker, kubernetes.io/os=linux

8. View the pods in the accounting namespace. They may still show as Pending. Depending on how long it has been since you attempted deployment the system may not have checked for the label. If the Pods show Pending after a minute delete one of the pods. They should both show as Running after a deletion. A change in state will cause the Deployment controller to check the status of both Pods.

```
student@cp:~$ kubectl -n accounting get pods
```

NAME	READY	STATUS	RESTARTS	AGE
nginx-one-74dd9d578d-fcpmv	1/1	Running	0	10m
nginx-one-74dd9d578d-sts51	1/1	Running	0	3s

9. View Pods by the label we set in the YAML file. If you look back the Pods were given a label of app=nginx.

```
student@cp:~$ kubectl get pods -l system=secondary --all-namespaces
```

NAMESPACE	NAME	READY	STATUS	RESTARTS	AGE
accounting	nginx-one-74dd9d578d-fcpmv	1/1	Running	0	20m
accounting	nginx-one-74dd9d578d-sts51	1/1	Running	0	9m

10. Recall that we exposed port 8080 in the YAML file. Expose the new deployment.

```
student@cp:~$ kubectl -n accounting expose deployment nginx-one
```

```
service/nginx-one exposed
```

11. View the newly exposed endpoints. Note that port 8080 has been exposed on each Pod.

```
student@cp:~$ kubectl -n accounting get ep nginx-one
```

NAME	ENDPOINTS	AGE
nginx-one	192.168.1.72:8080,192.168.1.73:8080	47s

12. Attempt to access the Pod on port 8080, then on port 80. Even though we exposed port 8080 of the container the application within has not been configured to listen on this port. The **nginx** server listens on port 80 by default. A curl command to that port should return the typical welcome page.

```
student@cp:~$ curl 192.168.1.72:8080
```

```
curl: (7) Failed to connect to 192.168.1.72 port 8080: Connection refused
```

```
student@cp:~$ curl 192.168.1.72:80
```

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<output_omitted>
```

13. Delete the deployment. Edit the YAML file to expose port 80 and create the deployment again.

```
student@cp:~$ kubectl -n accounting delete deploy nginx-one
```

```
deployment.apps "nginx-one" deleted
```

```
student@cp:~$ vim nginx-one.yaml
```



nginx-one.yaml

```
1  ....
2      ports:
3          - containerPort: 8080      #<-- Edit this line
4              protocol: TCP
5  ....
```

```
student@cp:~$ kubectl create -f nginx-one.yaml
```

```
deployment.apps/nginx-one created
```

✍ Exercise 10.2: Configure a NodePort

In a previous exercise we deployed a LoadBalancer which deployed a ClusterIP andNodePort automatically. In this exercise we will deploy a NodePort. While you can access a container from within the cluster, one can use a NodePort to NAT traffic

from outside the cluster. One reason to deploy a NodePort instead, is that a LoadBalancer is also a load balancer resource from cloud providers like GKE and AWS.

1. In a previous step we were able to view the **nginx** page using the internal Pod IP address. Now expose the deployment using the `--type=NodePort`. We will also give it an easy to remember name and place it in the accounting namespace. We could pass the port as well, which could help with opening ports in the firewall.

```
student@cp:~$ kubectl -n accounting expose deployment nginx-one --type=NodePort --name=service-lab
service/service-lab exposed
```

2. View the details of the services in the accounting namespace. We are looking for the autogenerated port.

```
student@cp:~$ kubectl -n accounting describe services
```

```
...
NodePort: <unset> 32103/TCP
...
```

3. Locate the exterior facing hostname or IP address of the cluster. The lab assumes use of GCP nodes, which we access via a FloatingIP, we will first check the internal only public IP address. Look for the Kubernetes cp URL. Whichever way you access check access using both the internal and possible external IP address

```
student@cp:~$ kubectl cluster-info
```

```
Kubernetes control plane is running at https://k8scp:6443
CoreDNS is running at https://k8scp:6443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
```

4. Test access to the **nginx** web server using the combination of cp URL and NodePort.

```
student@cp:~$ curl http://k8scp:32103
```

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
```

5. Using the browser on your local system, use the public IP address you use to SSH into your node and the port. You should still see the **nginx** default page. You may be able to use `curl` to locate your public IP address.

```
student@cp:~$ curl ifconfig.io
```

```
104.198.192.84
```

Exercise 10.3: Working with CoreDNS

1. We can leverage **CoreDNS** and predictable hostnames instead of IP addresses. A few steps back we created the `service-lab` NodePort in the Accounting namespace. We will create a new pod for testing using Ubuntu. The pod name will be named `ubuntu`.

```
student@cp:~$ cp /home/student/LFS458/SOLUTIONS/s_10/nettool.yaml .
```

```
student@cp:~$ vim nettool.yaml
```



nettool.yaml

```
1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: ubuntu
5 spec:
6   containers:
7     - name: ubuntu
8       image: ubuntu:latest
9       command: [ "sleep" ]
10      args: [ "infinity" ]
11
```

2. Create the pod and then log into it.

```
student@cp:~$ kubectl create -f nettool.yaml
```

```
pod/ubuntu created
```

```
student@cp:~$ kubectl exec -it ubuntu -- /bin/bash
```



On Container

- (a) Add some tools for investigating DNS and the network. The installation will ask you the geographic area and timezone information. Someone in Austin would first answer 2. America, then 37 for Chicago, which would be central time

```
root@ubuntu:/# apt-get update ; apt-get install curl dnsutils -y
```

- (b) Use the **dig** command with no options. You should see root name servers, and then information about the DNS server responding, such as the IP address.

```
root@ubuntu:/# dig
```

```
; <>> DiG 9.16.1-Ubuntu <>>
;; global options: +cmd
;; Got answer:
;; ->>HEADER<- opcode: QUERY, status: NOERROR, id: 3394
;; flags: qr rd ra; QUERY: 1, ANSWER: 13, AUTHORITY: 0, ADDITIONAL: 1

<output_omitted>

;; Query time: 4 msec
;; SERVER: 10.96.0.10#53(10.96.0.10)
;; WHEN: Thu Aug 27 22:06:18 CDT 2020
;; MSG SIZE  rcvd: 431
```

- (c) Also take a look at the **/etc/resolv.conf** file, which will indicate nameservers and default domains to search if no using a Fully Qualified Distinguished Name (FQDN). From the output we can see the first entry is **default.svc.cluster.local..**

```
root@ubuntu:/# cat /etc/resolv.conf
```

```
nameserver 10.96.0.10
search default.svc.cluster.local svc.cluster.local cluster.local
c.endless-station-188822.internal google.internal
options ndots:5
```

- (d) Use the `dig` command to view more information about the DNS server. Use the `-x` argument to get the FQDN using the IP we know. Notice the domain name, which uses `.kube-system.svc.cluster.local.`, to match the pod namespaces instead of `default`. Also note the name, `kube-dns`, is the name of a service not a pod.

```
root@ubuntu:/# dig @10.96.0.10 -x 10.96.0.10
```

```
...  
;; QUESTION SECTION:  
.10.0.96.10.in-addr.arpa.           IN      PTR  
  
;; ANSWER SECTION:  
10.0.96.10.in-addr.arpa.  
→ 30           IN      PTR      kube-dns.kube-system.svc.cluster.local.  
  
;; Query time: 0 msec  
;; SERVER: 10.96.0.10#53(10.96.0.10)  
;; WHEN: Thu Aug 27 23:39:14 CDT 2020  
;; MSG SIZE  rcvd: 139
```

- (e) Recall the name of the service-lab service we made and the namespaces it was created in. Use this information to create a FQDN and view the exposed pod.

```
root@ubuntu:/# curl service-lab.accounting.svc.cluster.local.
```

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
    ...

```

- (f) Attempt to view the default page using just the service name. It should fail as `nettool` is in the default namespace.

```
root@ubuntu:/# curl service-lab
```

curl: (6) Could not resolve host: service-lab

- (g) Add the accounting namespaces to the name and try again. Traffic can access a service using a name, even across different namespaces.

```
root@ubuntu:/# curl service-lab.accounting
```



```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<output_omitted>
```

- (h) Exit out of the container and look at the services running inside of the `kube-system` namespace. From the output we see that the `kube-dns` service has the DNS serverIP, and exposed ports DNS uses.

```
root@ubuntu:/# exit
```

```
student@cp:~$ kubectl -n kube-system get svc
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kube-dns	ClusterIP	10.96.0.10	<none>	53/UDP,53/TCP,9153/TCP	42h

3. Examine the service in detail. Among other information notice the selector in use to determine the pods the service communicates with.

```
student@cp:~$ kubectl -n kube-system get svc kube-dns -o yaml
```

```
...
  labels:
    k8s-app: kube-dns
    kubernetes.io/cluster-service: "true"
    kubernetes.io/name: CoreDNS
...
  selector:
    k8s-app: kube-dns
    sessionAffinity: None
    type: ClusterIP
...
...
```

4. Find pods with the same labels in all namespaces. We see that infrastructure pods all have this label, including `coredns`.

```
student@cp:~$ kubectl get pod -l k8s-app --all-namespaces
```

NAMESPACE	NAME	READY	STATUS	RESTARTS	AGE
kube-system	cilium-5tv9d	1/1	Running	0	136m
kube-system	cilium-gzdk6	1/1	Running	0	54m
kube-system	coredns-5d78c9869d-44qvq	1/1	Running	0	31m
kube-system	coredns-5d78c9869d-j6tqx	1/1	Running	0	31m
kube-system	kube-proxy-lpsmq	1/1	Running	0	35m
kube-system	kube-proxy-pvl8w	1/1	Running	0	34m

5. Look at the details of one of the `coredns` pods. Read through the pod spec and find the image in use as well as any configuration information. You should find that configuration comes from a configmap.

```
student@cp:~$ kubectl -n kube-system get pod coredns-f9fd979d6-4dxpl -o yaml
```

```
...
  spec:
    containers:
      - args:
          - -conf
```

```

- /etc/coredns/Corefile
image: k8s.gcr.io/coredns:1.7.0
...
volumeMounts:
- mountPath: /etc/coredns
  name: config-volume
  readOnly: true
...
volumes:
- configMap:
  defaultMode: 420
  items:
  - key: Corefile
    path: Corefile
  name: coredns
  name: config-volume
...

```

6. View the configmaps in the kube-system namespace.

```
student@cp:~$ kubectl -n kube-system get configmaps
```

NAME	DATA	AGE
cilium-config	4	43h
coredns	1	43h
extension-apiserver-authentication	6	43h
kube-proxy	2	43h
kubeadm-config	2	43h
kubelet-config	1	43h

7. View the details of the coredns configmap. Note the cluster.local domain is listed.

```
student@cp:~$ kubectl -n kube-system get configmaps coredns -o yaml
```

```

apiVersion: v1
data:
  Corefile: |
    .:53 {
      errors
      health {
        lameduck 5s
      }
      ready
      kubernetes cluster.local in-addr.arpa ip6.arpa {
        pods insecure
        fallthrough in-addr.arpa ip6.arpa
        ttl 30
      }
      prometheus :9153
      forward . /etc/resolv.conf {
        max_concurrent 1000
      }
      cache 30
      loop
      reload
      loadbalance
    }
  kind: ConfigMap
...

```

8. It is very important to backup our resources before we make changes to it.

```
student@cp:~$ kubectl -n kube-system get configmaps coredns -o yaml > coredns-backup.yaml
```

9. While there are many options and zone files we could configure, lets start with simple edit. Add a rewrite statement such that `test.io` will redirect to `cluster.local`. More about each line can be found at [coredns.io](#).

```
student@cp:~$ kubectl -n kube-system edit configmaps coredns
```

```
apiVersion: v1
data:
  Corefile: |
    .:53 {
      rewrite name regex (.*)\.test\.io {1}.default.svc.cluster.local    #<-- Add this line
      errors
      health {
        lameduck 5s
      }
      ready
      kubernetes cluster.local in-addr.arpa ip6.arpa {
        pods insecure
        fallthrough in-addr.arpa ip6.arpa
        ttl 30
      }
      prometheus :9153
      forward . /etc/resolv.conf {
        max_concurrent 1000
      }
      cache 30
      loop
      reload
      loadbalance
    }
```

10. Delete the coredns pods causing them to re-read the updated configmap.

```
student@cp:~$ kubectl -n kube-system delete pod coredns-f9fd979d6-s4j98 coredns-f9fd979d6-xlpzf
```

```
pod "coredns-f9fd979d6-s4j98" deleted
pod "coredns-f9fd979d6-xlpzf" deleted
```

11. Create a new web server and create a ClusterIP service to verify the address works. Note the new service IP to start with a reverse lookup.

```
student@cp:~$ kubectl create deployment nginx --image=nginx
```

```
deployment.apps/nginx created
```

```
student@cp:~$ kubectl expose deployment nginx --type=ClusterIP --port=80
```

```
service/nginx expose
```

```
student@cp:~$ kubectl get svc
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP	3d15h
nginx	ClusterIP	10.104.248.141	<none>	80/TCP	7s

12. Log into the `ubuntu` container and test the URL rewrite starting with the reverse IP resolution.

```
student@cp:~$ kubectl exec -it ubuntu -- /bin/bash
```

On Container

- (a) Use the `dig` command. Note that the service name becomes part of the FQDN.

```
root@ubuntu:/# dig -x 10.104.248.141

.....
;; QUESTION SECTION:
;141.248.104.10.in-addr.arpa.      IN      PTR

;; ANSWER SECTION:
141.248.104.10.in-addr.arpa. 30    IN      PTR      nginx.default.svc.cluster.local.
....
```

- (b) Now that we have the reverse lookup test the forward lookup. The IP should match the one we used in the previous step.

```
root@ubuntu:/# dig nginx.default.svc.cluster.local.

.....
;; QUESTION SECTION:
;nginx.default.svc.cluster.local. IN      A

;; ANSWER SECTION:
nginx.default.svc.cluster.local. 30    IN      A      10.104.248.141
....
```

- (c) Now test to see if the rewrite rule for the `test.io` domain we added resolves the IP. Note the response uses the original name, not the requested FQDN.

```
root@ubuntu:/# dig nginx.test.io

.....
;; QUESTION SECTION:
;nginx.test.io.          IN      A

;; ANSWER SECTION:
nginx.default.svc.cluster.local. 30    IN      A      10.104.248.141
....
```

13. Exit out of the container then edit the configmap to add an answer section.

```
student@cp:~$ kubectl -n kube-system edit configmaps coredns
```

```
.....
data:
  Corefile: |
    .:53 {
      rewrite stop {                      #<-- Edit this and following two lines
        name regex (*.)\.test\.io {1}.default.svc.cluster.local
        answer name (*.)\.default\.svc\.\cluster\.local {1}.test.io
      }
```

```

    errors
    health {
      ....

```

14. Delete the coredns pods again to ensure they re-read the updated configmap.

```
student@cp:~$ kubectl -n kube-system delete pod coredns-f9fd979d6-fv9qn coredns-f9fd979d6-lnxn5
```

```

pod "coredns-f9fd979d6-fv9qn" deleted
pod "coredns-f9fd979d6-lnxn5" deleted

```

15. Log into the ubuntu container again. This time the response should show the FQDN with the requested FQDN.

```
student@cp:~$ kubectl exec -it ubuntu -- /bin/bash
```

On Container

```

root@ubuntu:/# dig nginx.test.io

.....
;; QUESTION SECTION:
;nginx.test.io.           IN      A
;; ANSWER SECTION:
nginx.test.io.          30       IN      A      10.104.248.141
.....

```

16. Exit then delete the DNS test tools container to recover the resources.

```
student@cp:~$ kubectl delete -f nettool.yaml
```

✍ Exercise 10.4: Use Labels to Manage Resources

1. Try to delete all Pods with the system=secondary label, in all namespaces.

```
student@cp:~$ kubectl delete pods -l system=secondary \
--all-namespaces

pod "nginx-one-74dd9d578d-fcpmv" deleted
pod "nginx-one-74dd9d578d-sts51" deleted
```

2. View the Pods again. New versions of the Pods should be running as the controller responsible for them continues.

```
student@cp:~$ kubectl -n accounting get pods
```

NAME	READY	STATUS	RESTARTS	AGE
nginx-one-74dd9d578d-ddt5r	1/1	Running	0	1m
nginx-one-74dd9d578d-hfzml	1/1	Running	0	1m

3. We also gave a label to the deployment. View the deployment in the accounting namespace.

```
student@cp:~$ kubectl -n accounting get deploy --show-labels
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE	LABELS
nginx-one	2/2	2	2	10m	system=secondary

4. Delete the deployment using its label.

```
student@cp:~$ kubectl -n accounting delete deploy -l system=secondary
```

```
deployment.apps "nginx-one" deleted
```

5. Remove the label from the secondary node. Note that the syntax is a minus sign directly after the key you want to remove, or `system` in this case.

```
student@cp:~$ kubectl label node worker system-
```

```
node/worker unlabeled
```

Chapter 11

Ingress



11.1	Overview	250
11.2	Ingress Controller	251
11.3	Ingress Rules	256
11.4	Service Mesh	258
11.5	Labs	259

11.1 Overview

Ingress Overview

- Single entrypoint to cluster
- Manage external access to services in cluster
 - HTTP
 - Load-balancing
 - SSL termination
 - Name-based virtual hosting

In an earlier chapter we learned about using a Service to expose a containerized application outside of the cluster. We use Ingress Controllers and Rules to do the same function. The difference is efficiency. Instead of lots of services, such as LoadBalancer, you can route traffic based on request host or path. This allows for centralization of many services to a single point.

An Ingress Controller is different than most controllers as it does not run as part of **kube-controller-manager** binary. You can deploy multiple controllers, each with unique configurations. A controller uses Ingress Rules to handle traffic to and from outside the cluster.

There are many ingress controllers such as GKE, nginx, Traefik, Contour, Envoy to name a few. Any tool capable of reverse proxy should work. These agents consume rules and listen for associated traffic. An Ingress Rule is an API resource that you can create with **kubectl**. When you create that resource, it re-programs and re-configures your Ingress Controller to allow traffic to flow from the outside to an internal service. You can leave a service as a ClusterIP type and define how the traffic gets routed to that internal service using an Ingress Rule.

11.2 Ingress Controller

Ingress Controller

- Allow inbound traffic access to services
- Used instead of NodePort, or other services
- Proxy reconfigured according to rules

The Ingress

is collection of rules that allow inbound connections to reach the cluster services

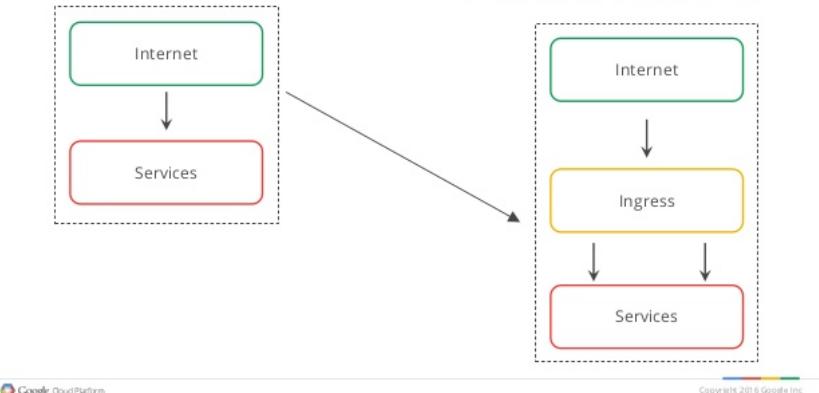


Figure 11.1: Ingress Controller for inbound connections

An Ingress Controller is a daemon running in a Pod which watches the `/ingresses` endpoint on the API Server, which is found under `networking.k8s.io/v1` group, for new objects. When a new endpoint is created the daemon uses the configured set of rules to allow inbound connection to a service, most often HTTP traffic. This allows easy access to a service through an edge router to Pods regardless of where the Pod is deployed.

Multiple Ingress controllers can be deployed. Traffic should use annotations to select the proper controller. The lack of a matching annotation will cause every controller to attempt to satisfy the ingress traffic.

nginx

- Easy integration with RBAC
- Uses the annotation
`kubernetes.io/ingress.class: "nginx"`
- L7 traffic requires the `proxy-real-ip-cidr` setting
- Bypasses kube-proxy to allow session affinity
- Does not use conntrack entries for iptables DNAT
- TLS requires `host` field to be defined

Deployment of an `nginx` controller has been made easy through the use of provided YAML files which can be found here:
<https://github.com/kubernetes/ingress-nginx/tree/master/deploy>

This page has configuration files to configure `nginx` on several platforms such as AWS, GKE, Azure, and bare-metal among others.

As with any Ingress Controller there are some configuration requirements for proper deployment. Customization can be done via a `ConfigMap`, `Annotations`, or for detailed configuration a `Custom template`.

Google Load Balancer Controller (GLBC)

- Controller called **glbc**, must be created and started first
- Also create
 - Replication Controller with single replica
 - Three services for application Pod
 - Ingress with two hostnames and three endpoints for each service.
- Backend is group of virtual machine instances, Instance Group
- Multi-pool path:
 - Global Forwarding Rule -> Target HTTP Proxy -> URL map
 - > Backend Service -> Instance Group
- Each pool checks next hop pool
- Not aware of GCE quota settings

There are several objects which need to be created to deploy the GCE Ingress Controller. YAML files are available to make the process easy. Be aware that several objects would be created for each service, and currently quotas are not evaluated prior to creation.

Each path for traffic uses a group of like objects referred to as a pool. Each pool regularly checks the next hop up to ensure connectivity.

Currently the TLS Ingress only supports port 443 and assumes TLS termination. It does not support SNI, only using the first cert. The TLS secret must contain keys named `tls.crt` and `tls.key`

Ingress API Resources

- Now part of networking.k8s.io API
- POST to API server
- Manage like other resources
- Able to expose low-number ports

```
$ kubectl get ingress
$ kubectl delete ingress <ingress_name>
$ kubectl edit ingress <ingress_name>
```

Ingress objects now part of the networking.k8s.io API, but still a beta object. A typical Ingress object that you can POST to the API server is:

YAML

```
1 apiVersion: networking.k8s.io/v1
2 kind: Ingress
3 metadata:
4 name: ghost
5 spec:
6 rules:
7   - host: ghost.192.168.99.100.nip.io
8     http:
9       paths:
10         - backend:
11           service
12             name: ghost
13             port:
14               number: 2368
15             path: /
16             pathType: ImplementationSpecific
17
```

You can manage Ingress resources like you do pods, deployments, services, etc

Deploy Ingress Controller

- Several options available
- Firewall rules may stop traffic
- Default backend serves 404 pages

```
$ kubectl create -f backend.yaml
```

To deploy an Ingress Controller, it can be as simple as creating it with **kubectl**. The source for a sample controller deployment is available on GitHub: <https://github.com/kubernetes/ingress-nginx/tree/master/deploy>

The result will be a set of pods managed by a replication controller and some internal services. You will notice a default HTTP backend which serves 404 pages.

```
$ kubectl get pods,rc,svc
```

NAME	READY	STATUS	RESTARTS	AGE
po/default-http-backend-xvep8	1/1	Running	0	4m
po/nginx-ingress-controller-fkshm	1/1	Running	0	4m
<hr/>				
NAME	DESIRED	CURRENT	READY	AGE
rc/default-http-backend	1	1	0	4m
<hr/>				
NAME	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
svc/default-http-backend	10.0.0.212	<none>	80/TCP	4m
svc/kubernetes	10.0.0.1	<none>	443/TCP	77d

11.3 Ingress Rules

Creating an Ingress Rule

- Run deployment and expose port
- POST rules entry to controller

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
rules:
- host: ghost.192.168.99.100.nip.io
  http:
    paths:
    - backend:
        service
        name: ghost
        port:
          number: 2368
    path: /
  pathType: ImplementationSpecific
```

To get exposed with ingress quickly, you can go ahead and try to create a similar rule as mentioned on the previous page. First, start a Ghost deployment and expose it with an internal ClusterIP service.

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

With the deployment exposed and the Ingress rules in place you should be able to access the application from outside the cluster.

Multiple Rules

- Multiple Services can use multiple rules

```
- host: ghost.192.168.99.100.nip.io
  http:
    paths:
      - backend:
          service
            name: external
            port:
              number: 80
    ....
    - host: ghost.192.168.99.100.nip.io
      http:
        paths:
          - backend:
              service
                name: internal
                port:
                  number: 8080
    ....
```

On the previous page we defined a single rule. If you have multiple services, you can define multiple rules in the same ingress, each rule forwarding traffic to a specific service.

11.4 Service Mesh

Intelligent Connected Proxies

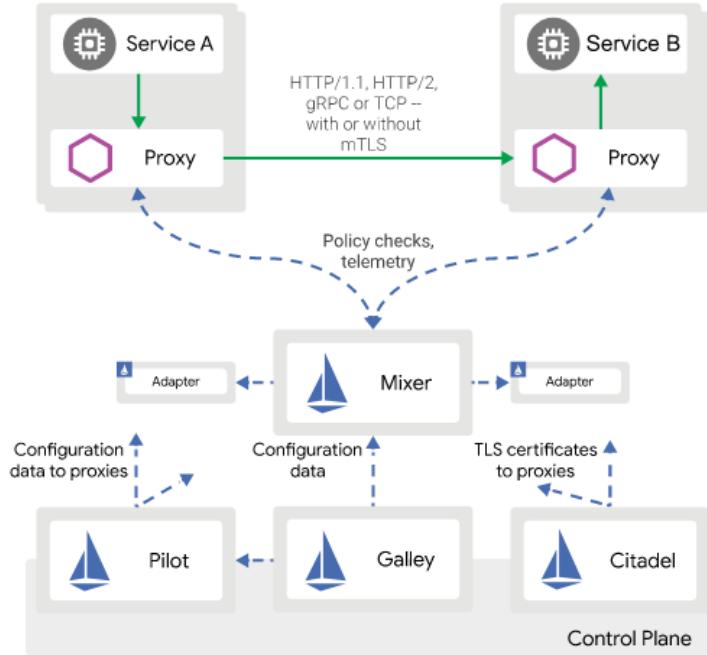


Figure 11.2: **Istio Service Mesh**

For more complex connections or resources such as service discovery, rate limiting, traffic management and advanced metrics you may want to implement a **service mesh**.

A **service mesh** consists of edge and embedded proxies communicating with each other and handling traffic based on rules from a control plane. Various options are available including **Envoy**, **Istio**, and **linkerd**.

- **Envoy** - a modular and extensible proxy favored due to modular construction, open architecture and dedication to remaining un-monetized. Often used as a data plane under other tools of a service mesh. envoyproxy.io
- **Istio** - a powerful tool set which leverages Envoy proxies via a multi-component control plane. Built to be platform independent it can be used to make the service mesh flexible and feature filled.
- **linkerd** - Another service mesh purpose built to be easy to deploy, fast, and ultralight. <https://linkerd.io/>

a

^aImage downloaded from <https://istio.io/docs/concepts/what-is-istio/> 23-sep-2019

11.5 Labs

Exercise 11.1: Service Mesh

If you have a large number of services to expose outside of the cluster, or to expose a low-number port on the host node you can deploy an ingress controller. While nginx and GCE have controllers mentioned a lot in Kubernetes.io, there are many to chose from. Even more functionality and metrics come from the use of a service mesh, such as Istio, Linkerd, Contour, Aspen, or several others.

1. We will install linkerd using their own scripts. There is quite a bit of output. Instead of showing all of it the output has been omitted. Look through the output and ensure that everything gets a green check mark. Some steps may take a few minutes to complete. Each command is listed here to make install easier. As well these steps are in the `setupLinkerd.txt` file.

```
student@cp:~$ curl -sL run.linkerd.io/install-edge | sh

student@cp:~$ export PATH=$PATH:/home/student/.linkerd2/bin

student@cp:~$ echo "export PATH=$PATH:/home/student/.linkerd2/bin" >> $HOME/.bashrc

student@cp:~$ linkerd check --pre

student@cp:~$ linkerd install --crds | kubectl apply -f -

student@cp:~$ linkerd install | kubectl apply -f -

student@cp:~$ linkerd check

student@cp:~$ linkerd viz install | kubectl apply -f -

student@cp:~$ linkerd viz check

student@cp:~$ linkerd viz dashboard &
```

2. By default the GUI is on available on the localhost. We will need to edit the service and the deployment to allow outside access, in case you are using a cloud provider for the nodes. Edit to remove all characters after equal sign for `-enforced-host`, which is around line 59.

```
student@cp:~$ kubectl -n linkerd-viz edit deploy web
```



```
spec:
  containers:
  - args:
    - -linkerd-controller-api-addr=linkerd-controller-api.linkerd.svc.cluster.local:8085
    - -linkerd-metrics-api-addr=metrics-api.linkerd-viz.svc.cluster.local:8085
    - -cluster-domain=cluster.local
    - -grafana-addr=grafana.linkerd-viz.svc.cluster.local:3000
    - -controller-namespace=linkerd
    - -viz-namespace=linkerd-viz
    - -log-level=info
    - -enforced-host=                         #<-- Comment the line by adding #
  image: cr.l15d.io/linkerd/web:stable-2.11.1
  imagePullPolicy: IfNotPresent
```

3. Now edit the http nodePort and type to be a NodePort.

```
student@cp:~$ kubectl edit svc web -n linkerd-viz
```

```
YAML
1 ....
2 ports:
3   - name: http
4     nodePort: 31500
5     port: 8084
6 ....
7 sessionAffinity: None
8 type: NodePort
9 status:
10 loadBalancer: {}
11 ....
12
```

4. Test access using a local browser to your public IP. Your IP will be different than the one shown below.

```
student@cp:~$ curl ifconfig.io
```

104.197.159.20

5. From your local system open a browser and go to the public IP and the high-number nodePort. Be aware the look of the web page may look slightly different as the software is regularly updated, for example Grafana is not longer fully integrated.

The screenshot shows the Linkerd Viz interface with the following details:

- HTTP metrics:**

Namespace	Meshed	Success Rate	RPS	P50 Latency	P95 Latency	P99 Latency	Grafana
accounting	0/2	---	---	---	---	---	---
default	0/2	---	---	---	---	---	---
kube-node-lease	0/0	---	---	---	---	---	---
kube-public	0/0	---	---	---	---	---	---
kube-system	0/11	---	---	---	---	---	---
linkerd	9/9	100.00% ●	6.47	4 ms	67 ms	93 ms	●
low-usage-limit	0/1	---	---	---	---	---	---
- TCP metrics:**

Namespace	Meshed	Connections	Read Bytes / sec	Write Bytes / sec	Grafana
accounting	0/2	---	---	---	---
default	0/2	---	---	---	---
kube-node-lease	0/0	---	---	---	---

Figure 11.3: Main Linkerd Page

6. In order for linkerd to pay attention to an object we need to add an annotation. The **linkerd inject** command will do this for us. Generate YAML and pipe it to **linkerd** then pipe again to **kubectl**. Expect an error about how the object was created, but the process will work. The command can run on one line if you omit the back-slash. Recreate the nginx-one deployment we worked with in a previous lab exercise.

```
student@cp:~$ kubectl get ns accounting      ## Verify namespace exists
student@cp:~$ kubectl label node worker<TAB> system=secondOne      ## Re-label the node
student@cp:~$ vim nginx-one.yaml            ## Validate or correct containerPort: 80 (not 8080)
student@cp:~$ kubectl apply -f nginx-one.yaml      ## Re-deploy nginx-one application
student@cp:~$ kubectl -n accounting get deploy nginx-one -o yaml | \
    linkerd inject - | kubectl apply -f -
<output_omitted>
```

7. Check the GUI, you should see that the accounting namespaces and pods are now meshed, and the name is a link.
 8. Generate some traffic to the pods, and watch the traffic via the GUI. Use the service-lab service.

```
student@cp:~$ kubectl -n accounting get svc
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
nginx-one	ClusterIP	10.107.141.227	<none>	8080/TCP	5h15m
service-lab	NodePort	10.102.8.205	<none>	80:30759/TCP	5h14m

```
student@cp:~$ curl 10.102.8.205
```

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<output_omitted>
```

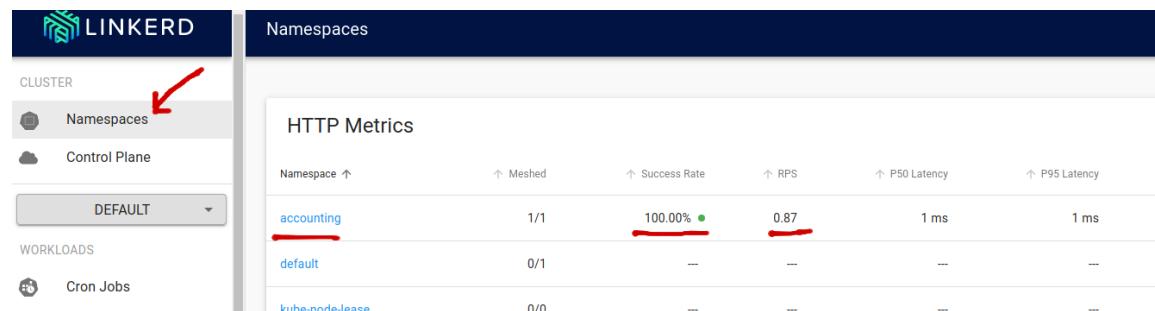


Figure 11.4: Now shows meshed

9. Scale up the nginx-one deployment. Generate traffic to get metrics for all the pods.

```
student@cp:~$ kubectl -n accounting scale deploy nginx-one --replicas=5
```

```
deployment.apps/nginx-one scaled
```

```
student@cp:~$ curl 10.102.8.205 #Several times
```

10. Explore some of the other information provided by the GUI. Note that the initial view is of the default namespaces. Change to accounting to see details of the nginx-one deployment.

TCP Metrics

Namespace ↑	↑ Meshed	↑ Connections	↑ Read Bytes / sec	↑ Write Bytes / s
accounting	5/5	5	150B/s	1.401kB
default	0/1	--	--	--

Figure 11.5: Five meshed pods

✍ Exercise 11.2: Ingress Controller

We will use the **Helm** tool we learned about earlier to install an ingress controller.

1. Create two deployments, `web-one` and `web-two`, one running `httpd`, the other `nginx`. Expose both as ClusterIP services. Use previous content to determine the steps if you are unfamiliar. Test that both ClusterIPs work before continuing to the next step.
2. Linkerd does not come with an ingress controller, so we will add one to help manage traffic. We will leverage a **Helm** chart to install an ingress controller. Search the hub to find that there are many available.

```
student@cp:~$ helm search hub ingress
```

URL	APP VERSION	DESCRIPTION	CHART VERSION
https://artifacthub.io/packages/helm/k8s-as-hel...	v1.0.0	Helm Chart representing a single Ingress Kubern...	1.0.2
https://artifacthub.io/packages/helm/openstack-...	v0.32.0	OpenStack-Helm Ingress Controller	0.2.1
<output_omitted>			
https://artifacthub.io/packages/helm/api/ingres...	0.45.0	Ingress controller for Kubernetes using NGINX a...	3.29.1
https://artifacthub.io/packages/helm/wener/ingr...	0.46.0	Ingress controller for Kubernetes using NGINX a...	3.31.0
<output_omitted>	1.11.2	NGINX Ingress Controller	0.9.2
<output_omitted>			

3. We will use a popular ingress controller provided by **NGINX**.

```
student@cp:~$ helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx
```

```
"ingress-nginx" has been added to your repositories
```

```
student@cp:~$ helm repo update
```

```
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "ingress-nginx" chart repository
Update Complete. -Happy Helming!-
```

4. Download and edit the `values.yaml` file and change it to use a DaemonSet instead of a Deployment. This way there will be a pod on every node to handle traffic.

```
student@cp:~$ helm fetch ingress-nginx/ingress-nginx --untar
student@cp:~$ cd ingress-nginx
student@cp:~/ingress-nginx$ ls
CHANGELOG.md  Chart.yaml  OWNERS  README.md  ci  templates  values.yaml
```

```
student@cp:~/ingress-nginx$ vim values.yaml
```



values.yaml

```
1  ....
2  ## DaemonSet or Deployment
3  ##
4  kind: DaemonSet           #<-- Change to DaemonSet, around line 204
5
6  ## Annotations to be added to the controller Deployment or DaemonSet
7  ....
8
```

5. Now install the controller using the chart. Note the use of the dot (.) to look in the current directory.

```
student@cp:~/ingress-nginx$ helm install myingress .
NAME: myingress
LAST DEPLOYED: Thu Aug 23 13:47:16 2023
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
The ingress-nginx controller has been installed.
It may take a few minutes for the LoadBalancer IP to be available.
You can watch the status by running
'kubectl --namespace default get services -o wide -w myingress-ingress-nginx-controller'

An example Ingress that makes use of the controller:
<output_omitted>
```

6. We now have an ingress controller running, but no rules yet. View the resources that exist. Use the `-w` option to watch the ingress controller service show up. After it is available use `ctrl-c` to quit and move to the next command.

```
student@cp:~$ kubectl get ingress --all-namespaces
```

```
No resources found
```

```
student@cp:~$ kubectl --namespace default get services -o wide myingress-ingress-nginx-controller
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP
PORT(S)	AGE	SELECTOR	
myingress-ingress-nginx-controller	LoadBalancer	10.104.227.79	<pending>
80:32558/TCP,443:30219/TCP	47s	app.kubernetes.io/component=controller, app.kubernetes.io/instance=myingress,app.kubernetes.io/name=ingress-nginx	

```
student@cp:~$ kubectl get pod --all-namespaces |grep nginx
```

default	myingress-ingress-nginx-controller-mrq5	1/1	Running	0	20s
default	myingress-ingress-nginx-controller-pkdxm	1/1	Running	0	62s
default	nginx-b68dd9f75-h6ww7	1/1	Running	0	21h

7. Now we can add rules which match HTTP headers to services.

```
student@cp:~$ cp /home/student/LFS458/SOLUTIONS/s_11/ingress.yaml .
```

```
student@cp:~$ vim ingress.yaml
```

YAML

ingress.yaml

```

1 apiVersion: networking.k8s.io/v1
2 kind: Ingress
3 metadata:
4   name: ingress-test
5   annotations:
6     nginx.ingress.kubernetes.io/service-upstream: "true"
7   namespace: default
8 spec:
9   ingressClassName: nginx
10  rules:
11    - host: www.external.com
12      http:
13        paths:
14          - backend:
15            service:
16              name: web-one
17              port:
18                number: 80
19              path: /
20              pathType: ImplementationSpecific

```

8. Create then verify the ingress is working. If you don't pass a matching header you should get a 404 error.

```
student@cp:~$ kubectl create -f ingress.yaml
```

```
ingress.networking.k8s.io/ingress-test created
```

```
student@cp:~$ kubectl get ingress
```

NAME	CLASS	HOSTS	ADDRESS	PORTS	AGE
ingress-test	nginx	www.external.com		80	5s

```
student@cp:~$ kubectl get pod -o wide |grep myingress
```

```
myingress-ingress-nginx-controller-mrqt5  1/1    Running   0        8m9s   192.168.219.118
  cp <none>           <none>
myingress-ingress-nginx-controller-pkdxm  1/1    Running   0        8m9s   192.168.0.250
  worker <none>          <none>
```

student@cp:~/ingress-nginx\$ curl 192.168.219.118

```
<html>
<head><title>404 Not Found</title></head>
<body>
<center><h1>404 Not Found</h1></center>
<hr><center>nginx</center>
</body>
</html>
```

- Check the ingress service and expect another 404 error, don't use the admission controller.

student@cp:~/ingress-nginx\$ kubectl get svc |grep ingress

myingress-ingress-nginx-controller	LoadBalancer	10.104.227.79	<pending>
80:32558/TCP,443:30219/TCP	10m		
myingress-ingress-nginx-controller-admission	ClusterIP	10.97.132.127	<none>
443/TCP	10m		

student@cp:~/ingress-nginx\$ curl 10.104.227.79

```
<html>
<head><title>404 Not Found</title></head>
<body>
<center><h1>404 Not Found</h1></center>
<hr><center>nginx</center>
</body>
</html>
```

- Now pass a header which matches a URL to one of the services we exposed in an earlier step. You should see the default nginx or httpd web server page.

student@cp:~/ingress-nginx\$ curl -H "Host: www.external.com" http://10.104.227.79

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
<output_omitted>
```

- We can add an annotation to the ingress pods for Linkerd. You will get some warnings, but the command will work.

student@cp:~/ingress-nginx\$ kubectl get ds myingress-ingress-nginx-controller -o yaml |\\
linkerd inject --ingress - | kubectl apply -f -

```
daemonset "myingress-ingress-nginx-controller" injected
```

```
Warning: resource daemonsets/myingress-ingress-nginx-controller is missing the
kubectl.kubernetes.io/last-applied-configuration annotation which is required
by kubectl apply. kubectl apply should only be used on resources created
```

declaratively by either `kubectl create --save-config` or `kubectl apply`. The missing annotation will be patched automatically.
`daemonset.apps/myingress-ingress-nginx-controller` configured

12. Go to the Top page, change the namespace to default and the resource to `daemonset/myingress-ingress-nginx-controller`. Press start then pass more traffic to the ingress controller and view traffic metrics via the GUI. Let top run so we can see another page added in an upcoming step.

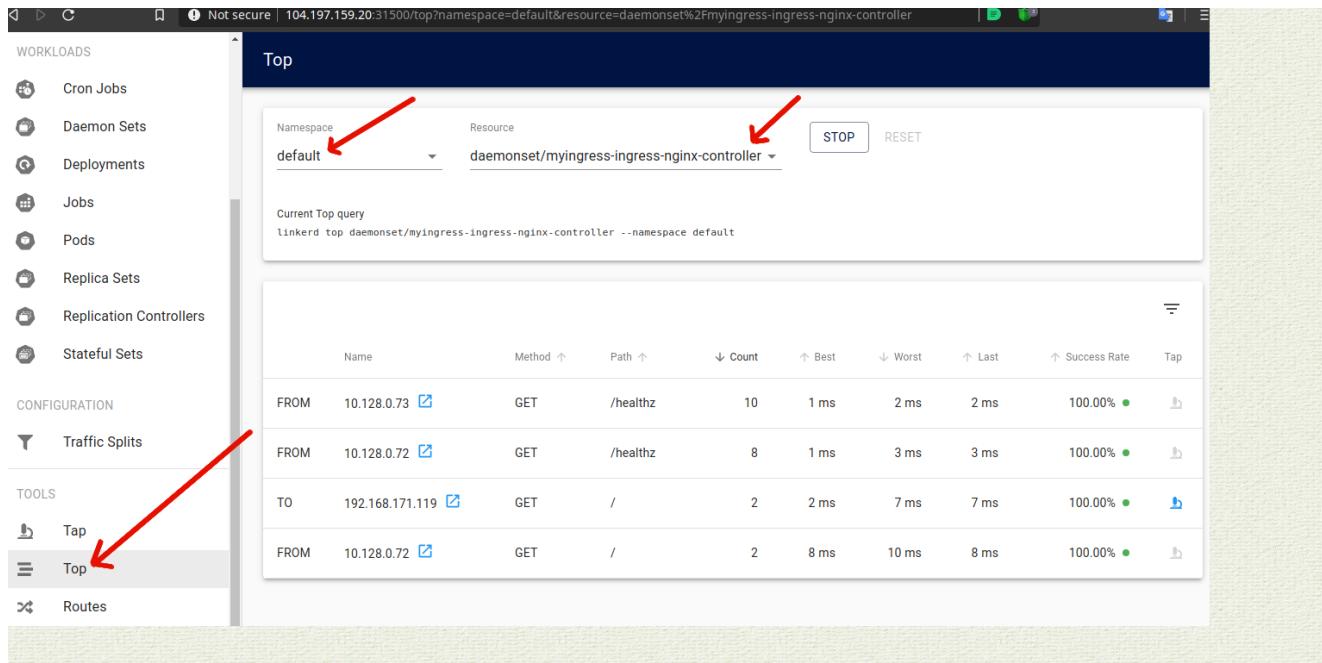


Figure 11.6: Ingress Traffic

13. At this point we would keep adding more and more servers. We'll configure one more, which would then could be a process continued as many times as desired.

Customize the web-two (or whichever deployment is running nginx) welcome page. Run a bash shell inside the web-two pod. Your pod name will end differently. Install `vim` or an editor inside the container then edit the `index.html` file of nginx so that the title of the web page will be Internal Welcome Page. Much of the command output is not shown below.

```
student@cp:~$ kubectl exec -it web-two-<Tab> -- /bin/bash
```

On Container

```
root@web-two-...-# apt-get update
root@web-two-...-# apt-get install vim -y
root@web-two-...-# vim /usr/share/nginx/html/index.html

<!DOCTYPE html>
<html>
<head>
<title>Internal Welcome Page</title>      #<-- Edit this line
<style>
<output_omitted>
```



```
root@thirdpage:/$ exit
```

Edit the ingress rules to point the thirdpage service. It may be easiest to copy the existing host stanza and edit the host and name.

14. student@cp:~\$ kubectl edit ingress ingress-test



ingress-test

```

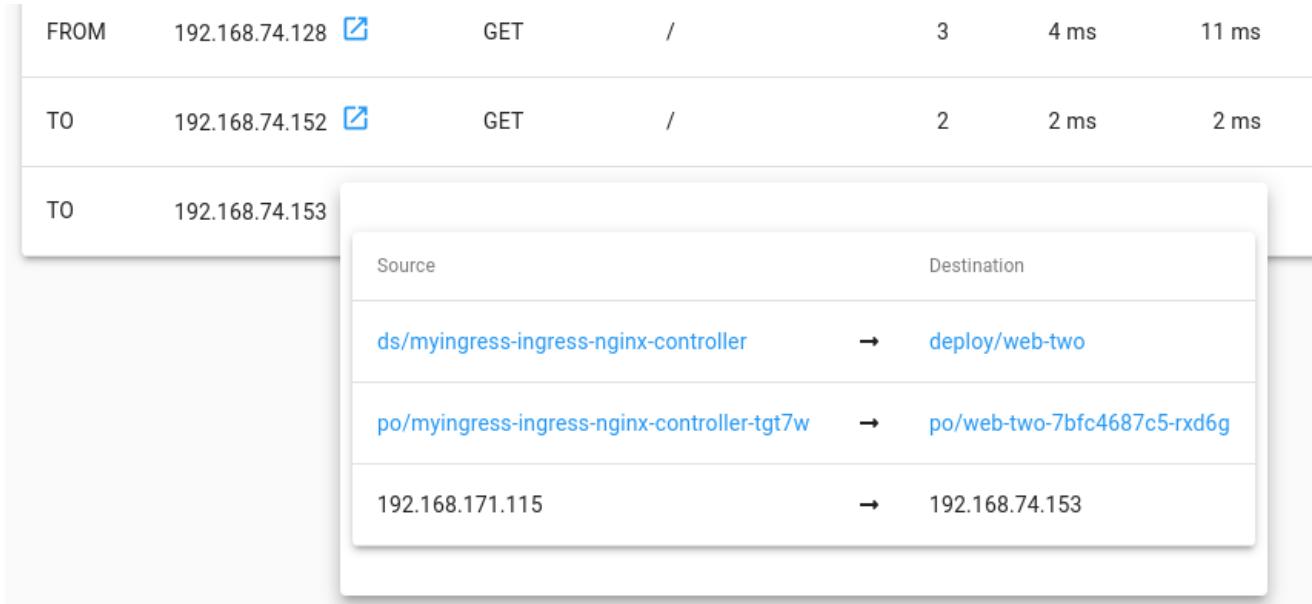
1  ....
2 spec:
3   rules:
4     - host: internal.org
5       http:
6         paths:
7           - backend:
8             service:
9               name: web-two
10              port:
11                number: 80
12              path: /
13              pathType: ImplementationSpecific
14 - host: www.external.com
15   http:
16     paths:
17       - backend:
18         service:
19           name: web-one
20           port:
21             number: 80
22           path: /
23           pathType: ImplementationSpecific
24 status:
25 ....
26
```

15. Test the second Host: setting using **curl** locally as well as from a remote system, be sure the <title> shows the non-default page. Use the main IP of either node. The Linkerd GUI should show a new T0 line, if you select the small blue box with an arrow you will see the traffic is going to internal.org.

student@cp:~\$ curl -H "Host: internal.org" http://10.128.0.7/

```

<!DOCTYPE html>
<html>
<head>
<title>Internal Welcome Page</title>
<style>
<output_omitted>
```

Figure 11.7: **Linkerd Top Metrics**

Chapter 12

Scheduling



12.1	Overview	270
12.2	Scheduler Settings	271
12.3	Pod Specification	276
12.4	Affinity Rules	278
12.5	Taints and Tolerations	283
12.6	Labs	286

12.1 Overview

kube-scheduler

- Topology-aware algorithm to determine Pod placement
- Pod priority and preemption
- Labels used for each method
- **podAffinity** label encourage Pod on specific nodes
- Avoid via **podAntiAffinity**
- Node **taints** to repel specific Pods
- Pod **tolerations** of node **taints**
- **nodeSelector** to force specific Pods
- Require, prefer and evict

The larger and more diverse a Kubernetes deployment becomes the more administration of scheduling can be important. The **kube-scheduler** determines which nodes will run a Pod.

Users can set the priority of a pod, which will allow preemption of lower priority pods. The eviction of lower priority pods would then allow the higher priority pod to be scheduled.

The scheduler tracks the set of nodes in your cluster, filters and scores to determine on which node each Pod should be scheduled. The Pod spec as part of a request is sent to the **kubelet** on the node for creation.

The default scheduling decision can be affected through the use of Labels on nodes or Pods. Labels of **podAffinity**, **taints**, and **pod bindings** allow for configuration from the Pod or the node perspective. Some like **tolerations** allow a Pod to work with a Node, even when the Node has a **taint** that would otherwise preclude a Pod being scheduled.

Not all labels are drastic. Affinity settings may encourage a Pod to be deployed on a node, but would deploy the Pod elsewhere if the node was not available. Sometimes documentation may use the term require, but practice shows the setting to be more of a request. As beta features expect the specifics to change. Some settings will evict Pods from a node should the required condition no longer be true such as `requiredDuringSchedulingRequiredDuringExecution`.

Others options, like a custom scheduler, need to be programmed and deployed into your Kubernetes cluster.

12.2 Scheduler Settings

Node selection in kube-scheduler

- **Filtering**
- **Scoring**

The **Filtering stage**, identifies the set of Nodes where the Pod can be scheduled.

For example, the PodFitsResources filter determines whether a prospective Node has sufficient resources available to satisfy a Pod's particular resource requirements (requests & limits). Any appropriate Nodes that were found after this phase are included in the node list. It is not possible to schedule that pod if the list is empty, it remains unscheduled.

The **Scoring stage**, the scheduler rates the remaining nodes to determine the best Pod placement. Each Node that made it through filtering is given a score by the scheduler, which is based on the default scheduler configuration.

The Pod is given to the Node with the highest ranking by **kube-scheduler**.

Note: The filtering and scoring behavior of the scheduler can be configured using scheduling configuration profiles.

Scheduling Configuration

- **Customize the behavior**
- **Scheduling Profiles**
 - Extension points
 - Scheduling plugins
 - Multiple profiles

You can customize the behavior of the **kube-scheduler** by writing a configuration file and passing its path as a command line argument. A scheduling Profile allows you to configure the different stages of scheduling in the **kube-scheduler**. Each stage is exposed in an extension point. Plugins provide scheduling behaviors by implementing one or more of these extension points.

We can configure the scheduler via the use of **scheduling profiles**. These profiles allow the configuration of extension points at which plugins can be used.

An **extension point** is one of the twelve stages of scheduling, at which point a **plugin** can be used to modify how that state of a scheduler works

Extension Points

- queueSort
- preFilter
- filter
- postFilter
- preScore
- score
- reserve
- permit
- preBind
- bind
- postBind
- multiPoint

queueSort: These plugins provide an ordering function that is used to sort pending Pods in the scheduling queue. Exactly one queue sort plugin may be enabled at a time.

preFilter: These plugins are used to pre-process or check information about a Pod or the cluster before filtering. They can mark a pod as unschedulable.

filter: These plugins are the equivalent of Predicates in a scheduling Policy and are used to filter out nodes that can not run the Pod. Filters are called in the configured order. A pod is marked as unschedulable if no nodes pass all the filters.

postFilter: These plugins are called in their configured order when no feasible nodes were found for the pod. If any postFilter plugin marks the Pod schedulable, the remaining plugins are not called.

preScore: This is an informational extension point that can be used for doing pre-scoring work.

score: These plugins provide a score to each node that has passed the filtering phase. The scheduler will then select the node with the highest weighted scores sum.

More information can be found here: <https://kubernetes.io/docs/reference/scheduling/config/#profiles>

Scheduling plugins

- ImageLocality
- TaintToleration
- NodeName
- NodePorts
- NodeAffinity
- PodTopologySpread
- NodeUnschedulable
- NodeResourcesFit
- NodeResourcesBalancedAllocation
- VolumeBinding
- VolumeRestrictions
- VolumeZone
- NodeVolumeLimits
- EBSLimits
- GCEPDLimits
- AzureDiskLimits
- InterPodAffinity
- PrioritySort
- DefaultBinder
- DefaultPreemption

The following plugins, enabled by default, implement one or more of these extension points

ImageLocality: Favors nodes that already have the container images that the Pod runs. Extension points: score.

TaintToleration: Implements taints and tolerations. Implements extension points: filter, preScore, score.

NodeName: Checks if a Pod spec node name matches the current node. Extension points: filter.

NodePorts: Checks if a node has free ports for the requested Pod ports. Extension points: preFilter, filter.

NodeAffinity: Implements node selectors and node affinity. Extension points: filter, score.

PodTopologySpread: Implements Pod topology spread. Extension points: preFilter, filter, preScore, score.

NodeUnschedulable: Filters out nodes that have .spec.unschedulable set to true. Extension points: filter.

NodeResourcesFit: Checks if the node has all the resources that the Pod is requesting.

NodeResourcesBalancedAllocation: Favors nodes that would obtain a more balanced resource usage if the Pod is scheduled there. Extension points: score.

More information can be found here: <https://kubernetes.io/docs/reference/scheduling/config/#scheduling-plugins>

Multiple profiles

- Configure more than one profile
 - Example config

```
apiVersion: kubescheduler.config.k8s.io/v1
kind: KubeSchedulerConfiguration
profiles:
  - schedulerName: default-scheduler
  - schedulerName: custom-scheduler
    plugins:
      preFilter:
        disabled:
          - name: '*'
      filter:
        disabled:
          - name: '*'
      postFilter:
        disabled:
          - name: '*'
```

kube-scheduler can be configured to run more than one profile. Each profile should have an associated scheduler name and also can run a different set of plugins.

In the above example, the scheduler will run with two profiles: one with the default plugins and one with all filtering plugins disabled.

Pods that want to be scheduled according to a specific profile can include the corresponding scheduler name in its `.spec.schedulerName`.

By default, one profile with the scheduler name **default-scheduler** is created. This profile includes the default plugins described above. When declaring more than one profile, a unique **scheduler name** for each of them is required.

Note: If a scheduler name is not specified in the pod spec, kube-apiserver will set it to default-scheduler. Therefore, a profile with this scheduler name should exist to get those pods scheduled.

12.3 Pod Specification

Pod Specification

- Pod Specification fields
 - nodeName
 - nodeSelector
 - affinity
 - tolerations
 - schedulerName

Most scheduling decisions can be made as part of the Podspec. The `nodeName` and `nodeSelector` options allow a Pod to be assigned to a single node or a group of nodes with particular labels.

Affinity and anti-affinity can be used to require or prefer which node is used by the scheduler. If using a preference instead a matching node is chosen first, but other nodes would be used if no match is present.

The use of `taints` allows a node to be labeled such that Pods would not be scheduled for some reason, such as the `cp` node after initialization. A `toleration` allows a Pod to ignore the taint and be scheduled assuming other requirements are met.

Should none of these options meet the needs of the cluster there is also the ability to deploy a custom scheduler. Each Pod could then include a `schedulerName` to choose which schedule to use.

Specifying Node Label

- Match a label with `nodeSelector`
- Pod remains in Pending state until a suitable Node is found

```
spec:  
  containers:  
    - name: redis  
      image: redis  
  nodeSelector:  
    net: fast
```

The `nodeSelector` field in a pod specification provides a straightforward way to target a node or set of nodes, using one or more key-value pairs.

Setting the `nodeSelector` tells the scheduler to place the pod on a node that matches the labels. All listed selectors must be met, but the node could have more labels. In the above example any node with a key of `net` set to `fast` would be a candidate for scheduling. Remember that labels are admin created tags, with no tie to actual resources. This node could have a slow network.

The pod would remain Pending until a node is found with the matching labels.

The use of affinity/anti-affinity should be able to express every feature as `nodeSelector`.

12.4 Affinity Rules

Pod Affinity Rules

- Uses In, NotIn, Exists, and DoesNotExist operators
- requiredDuringSchedulingIgnoredDuringExecution
- preferredDuringSchedulingIgnoredDuringExecution
- affinity
 - podAffinity
 - podAntiAffinity

Pods which may communicate a lot or share data may operate best if co-located, which would be a form of affinity. For greater fault tolerance you may want Pods to be as separate as possible, which would be anti-affinity. These settings are used by the scheduler based on labels of Pods already running. As a result the scheduler must interrogate each node and track the labels of running Pods. Clusters larger than several hundred nodes may see significant performance loss.

The use of requiredDuringSchedulingIgnoredDuringExecution means that the Pod will not be scheduled on a node unless the following operator is true. If the operator changes to become false in the future the Pod will continue to run. This could be seen as a hard rule.

Similar is preferredDuringSchedulingIgnoredDuringExecution which will choose a node with the desired setting before those without. Should no properly labeled nodes be available the Pod will execute anyway. This is more of a soft settings which declares a preference instead of a requirement.

With the use of podAffinity the scheduler will try to schedule Pods together. The use of podAntiAffinity would cause the scheduler to keep Pods on different nodes.

podAffinity Example

```
spec:  
  affinity:  
    podAffinity:  
      requiredDuringSchedulingIgnoredDuringExecution:  
        - labelSelector:  
            matchExpressions:  
              - key: security  
                operator: In  
                values:  
                  - S1
```

An example of affinity and podAffinity settings. This also requires a particular label to be matched when the Pod starts, but not required if the label is later removed.

The Pod can be scheduled on a node running a Pod with a key label of `security` and a value of `S1`. If this requirement is not met the Pod will remain in a Pending state.

podAntiAffinity Example

```
podAntiAffinity:  
  preferredDuringSchedulingIgnoredDuringExecution:  
    - weight: 100  
      podAffinityTerm:  
        labelSelector:  
          matchExpressions:  
            - key: security  
              operator: In  
              values:  
                - S2
```

With `podAntiAffinity` we can prefer to avoid nodes with a particular label. In this case the scheduler will prefer to avoid a node running a pod that has a key label of `security` and value of `S2`.

In a large, varied, environment there may be multiple situations to be avoided. As a preference this settings tries to avoid certain labels, but will still schedule the Pod on some node. As the Pod will still run we can provide a weight to a particular rule. The weights can be declared in a value from 1 to 100. The scheduler then tries to choose, or avoid, the node with the greatest combined value.

Node Affinity Rules

- Similar to Pod affinity rules, declared with `affinity`
- Uses `In`, `NotIn`, `Exists`, and `DoesNotExist` operators
- `requiredDuringSchedulingIgnoredDuringExecution`
- `preferredDuringSchedulingIgnoredDuringExecution`
- Planned for future
 - `requiredDuringSchedulingRequiredDuringExecution`

Where Pod affinity/anti-affinity has to do with other Pods, the use of `nodeAffinity` allows Pod scheduling based of node labels. This is similar, and will some day replace, use of the `nodeSelector` setting. The scheduler will not look at other Pods on the system, but the labels of the nodes. This should have much less performance impact on the cluster, even with a large number of nodes.

Until `nodeSelector` has been fully deprecated both the selector and required labels must be met for a Pod to be scheduled.

Node Affinity Example

```
spec:  
affinity:  
nodeAffinity:  
preferredDuringSchedulingIgnoredDuringExecution:  
- weight: 1  
preference:  
matchExpressions:  
- key: diskspeed  
operator: In  
values:  
- quick  
- fast
```

The `nodeAffinity` prefers a node with the following rule, but the pod would be scheduled even if there were no matching nodes. The rule gives extra weight to nodes with a key of `diskspeed` with a value of `fast` or `quick`.

12.5 Taints and Tolerations

Taint

- Expressed as key=value:**effect**
- key and value value created by admin
- Effect must be **NoSchedule**, **PreferNoSchedule**, or **NoExecute**
- Only nodes can be tainted currently
- Multiple taints possible on a node

A node with a particular taint will repel Pods without tolerations for that taint.

The key and value used can be any legal string, while allows flexibility to prevent Pods from running on nodes based off of any need. If a Pod does not have an existing toleration the scheduler will not consider the tainted node.

There are three effects, or ways to handle Pod scheduling.

- **NoSchedule** The scheduler will not schedule a Pod on this node unless the Pod has this toleration. Existing Pods continue to run, regardless of toleration.
- **PreferNoSchedule** The scheduler will avoid using this node unless there are no untainted nodes for the Pods toleration. Existing Pods are unaffected.
- **NoExecute** This taint will cause existing Pods to be evacuated and no future Pods scheduled. Should an existing Pod have a toleration it will continue to run. If the Pod tolerationSeconds value is set the Pod will remain for that many seconds then be evicted. Certain node issues will cause **kubelet** to add 300 second tolerations to avoid unnecessary evictions.

If a node has multiple taints the scheduler ignores those with matching tolerations. The remaining un-ignored taints have their typical effect.

The use of TaintBasedEvictions is still an alpha feature. The **kubelet** uses taints to rate-limit evictions when the node has problems.

Tolerations

- Pod setting to run on tainted nodes
- Same **effects** as node taints
- Two operators
 - Exists
 - Equal which requires value

tolerations:

```
- key: "server"
  operator: "Equal"
  value: "ap-east"
  effect: "NoExecute"
  tolerationSeconds: 3600
```

Setting tolerations on a node are used to schedule on tainted nodes. This provides an easy way to avoid Pods using the node. Only those with a particular toleration would be scheduled.

An operator can be included in a Pod spec, defaulting to Equal if not declared. The use of operator Equal requires a value to match. The Exists operator should not be specified. If an empty key uses the Exists operator it will tolerate every taint. If there is no effect, but a key and operator are declared all effects are matched with the declared key.

In the above example the Pod will remain on the server with a key of server and value of ap-east for 3600 seconds after the node has been tainted with NoExecute. When the time runs out the Pod will be evicted.

Custom Scheduler

- Create and deploy custom scheduler as a container
- Run multiple schedulers simultaneously
- Pods can declare which scheduler to use
- Scheduler must be running or Pod remains Pending
- View scheduler and other information with **kubectl get events**

If the default scheduling mechanisms are not flexible enough for your needs you can write your own scheduler. The programming of a custom scheduler is outside the scope of this course, but you may want to start with the existing scheduler code which can be found here: <https://github.com/kubernetes/kubernetes/tree/master/pkg/scheduler>

If a Pod spec does not declare which scheduler to use the standard scheduler is used by default. If the Pod declares a scheduler and that container is not running the Pod would remain in Pending state forever.

The end result of the scheduling process is that a pod gets a **binding** that specifies which node it should run on. A binding is a Kubernetes API primitive in the **api/v1** group. Technically without any scheduler running, you could still schedule a pod on a node, by specifying a binding for that pod.

12.6 Labs

Exercise 12.1: Assign Pods Using Labels

Overview

While allowing the system to distribute Pods on your behalf is typically the best route, you may want to determine which nodes a Pod will use. For example you may have particular hardware requirements to meet for the workload. You may want to assign VIP Pods to new, faster hardware and everyone else to older hardware.

In this exercise we will use labels to schedule Pods to a particular node. Then we will explore taints to have more flexible deployment in a large environment.

1. Begin by getting a list of the nodes. They should be in the ready state and without added labels or taints.

```
student@cp:~$ kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
cp	Ready	control-plane	44h	v1.30.1
worker	Ready	<none>	43h	v1.30.1

2. View the current labels and taints for the nodes.

```
student@cp:~$ kubectl describe nodes |grep -A5 -i label
```

```
Labels:          beta.kubernetes.io/arch=amd64
                beta.kubernetes.io/os=linux
                kubernetes.io/arch=amd64
                kubernetes.io/hostname=scp
                kubernetes.io/os=linux
                node-role.kubernetes.io/control-plane=
--
Labels:          beta.kubernetes.io/arch=amd64
                beta.kubernetes.io/os=linux
                kubernetes.io/arch=amd64
                kubernetes.io/hostname=worker
                kubernetes.io/os=linux
                system=secondOne
```

```
student@cp:~$ kubectl describe nodes |grep -i taint
```

```
Taints:          <none>
Taints:          <none>
```

3. Get a count of how many containers are running on both the cp and worker nodes. There are about 24 containers running on the cp in the following example, and eight running on the worker. There are status lines which increase the **wc** count. You may have more or less, depending on previous labs and cleaning up of resources. Take note of the number of containers, and then notice the numbers change due to scheduling. The change between nodes is the important information, not the particular number. If you are using **cri-o** you can view containers using **crlctl ps**.

```
student@cp:~$ kubectl get deployments --all-namespaces
```

NAMESPACE	NAME	READY	UP-TO-DATE	AVAILABLE	AGE
accounting	nginx-one	1/1	1	1	19h
default	anotherweb-apache	1/1	1	1	8h
default	web-one	1/1	1	1	45m
default	web-two	1/1	1	1	45m

```
kube-system cilium-operator      1/1     1          1      35h
<output_omitted>
```

```
student@cp:~$ sudo crictl ps | wc -l
```

```
24
```

```
student@worker:~$ sudo crictl ps | wc -l
```

```
21
```

4. For the purpose of the exercise we will assign the cp node to be VIP hardware and the secondary node to be for others.

```
student@cp:~$ kubectl label nodes cp status=vip
```

```
node/cp labeled
```

```
student@cp:~$ kubectl label nodes worker status=other
```

```
node/worker labeled
```

5. Verify your settings. You will also find there are some built in labels such as hostname, os and architecture type. The output below appears on multiple lines for readability.

```
student@cp:~$ kubectl get nodes --show-labels
```

NAME	STATUS	ROLES	AGE	VERSION	LABELS
cp	Ready	control-plane	35h	v1.30.1	beta.kubernetes.io/arch=amd64, beta.kubernetes.io/os=linux,kubernetes.io/arch=amd64,kubernetes.io/hostname=cp, kubernetes.io/os=linux,node-role.kubernetes.io/control-plane=,node-role.kubernetes.io/master=, node.kubernetes.io/exclude-from-external-load-balancers=,status=vip
worker	Ready	<none>	35h	v1.30.1	beta.kubernetes.io/arch=amd64, beta.kubernetes.io/os=linux,kubernetes.io/arch=amd64,kubernetes.io/hostname=worker, kubernetes.io/os=linux,status=other,system=secondOne

6. Create `vip.yaml` to spawn four busybox containers which sleep the whole time. Include the `nodeSelector` entry.

```
student@cp:~$ vim vip.yaml
```



`YAML`

```
1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: vip
5 spec:
6   containers:
7     - name: vip1
8       image: busybox
9       args:
10      - sleep
11      - "1000000"
```



```

12   - name: vip2
13     image: busybox
14     args:
15       - sleep
16       - "1000000"
17   - name: vip3
18     image: busybox
19     args:
20       - sleep
21       - "1000000"
22   - name: vip4
23     image: busybox
24     args:
25       - sleep
26       - "1000000"
27   nodeSelector:
28     status: vip

```

7. Deploy the new pod. Verify the containers have been created on the cp node. It may take a few seconds for all the containers to spawn. Check both the cp and the secondary nodes. From this point forward use **crictl** where the step lists **docker** if you have deployed your cluster with cri-o.

```
student@cp:~$ kubectl create -f vip.yaml
```

```
pod/vip created
```

```
student@cp:~$ sudo crictl ps |wc -l
```

```
28
```

```
student@worker:~$ sudo crictl ps |wc -l
```

```
21
```

8. Delete the pod then edit the file, commenting out the `nodeSelector` lines. It may take a while for the containers to fully terminate.

```
student@cp:~$ kubectl delete pod vip
```

```
pod "vip" deleted
```

```
student@cp:~$ vim vip.yaml
```

```
....  
# nodeSelector:  
#   status: vip
```

9. Create the pod again. Containers can now be spawning on either of the node. You may see pods for the daemonsets as well.

```
student@cp:~$ kubectl get pods
```

```
<output_omitted>
```

```
student@cp:~$ kubectl create -f vip.yaml
```

```
pod/vip created
```

10. Determine where the new containers have been deployed. They should be more evenly spread this time. Again, the numbers may be different, the change in numbers is what we are looking for. Due to lack of nodeSelector they could go to either node.

```
student@cp:~$ sudo crictl ps |wc -l
```

```
24
```

```
student@worker:~$ sudo crictl ps |wc -l
```

```
25
```

11. Create another file for other users. Change the names from vip to others, and uncomment the nodeSelector lines.

```
student@cp:~$ cp vip.yaml other.yaml
```

```
student@cp:~$ sed -i s/vip/other/g other.yaml
```

```
student@cp:~$ vim other.yaml
```



other.yaml

```
1  ....
2  nodeSelector:
3    status: other
4
```

12. Create the other containers. Determine where they deploy.

```
student@cp:~$ kubectl create -f other.yaml
```

```
pod/other created
```

```
student@cp:~$ sudo crictl ps |wc -l
```

```
24
```

```
student@worker:~$ sudo crictl ps |wc -l
```

```
25
```

13. Shut down both pods and verify they terminated. Only our previous pods should be found.

```
student@cp:~$ kubectl delete pods vip other
```

```
pod "vip" deleted
pod "other" deleted
```

```
student@cp:~$ kubectl get pods
```

```
<output omitted>
```

Exercise 12.2: Using Taints to Control Pod Deployment

Use taints to manage where Pods are deployed or allowed to run. In addition to assigning a Pod to a group of nodes, you may also want to limit usage on a node or fully evacuate Pods. Using taints is one way to achieve this. You may remember that the cp node begins with a NoSchedule taint. We will work with three taints to limit or remove running pods.

1. Create a deployment which will deploy eight **nginx** containers. Begin by creating a YAML file.

```
student@cp:~$ vim taint.yaml
```

YAML

taint.yaml

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: taint-deployment
5 spec:
6   replicas: 8
7   selector:
8     matchLabels:
9       app: nginx
10    template:
11      metadata:
12        labels:
13          app: nginx
14    spec:
15      containers:
16        - name: nginx
17          image: nginx:1.20.1
18        ports:
19          - containerPort: 80
```

2. Apply the file to create the deployment.

```
student@cp:~$ kubectl apply -f taint.yaml
```

```
deployment.apps/taint-deployment created
```

3. Determine where the containers are running. In the following example three have been deployed on the cp node and five on the secondary node. Remember there will be other housekeeping containers created as well. Your numbers may be different, the actual number is not important, we are tracking the change in numbers.

```
student@cp:~$ sudo crictl ps |grep nginx
```

```
00c1be5df1e7      nginx@sha256:e3456c851a152494c3e.....
<output_omitted>
```

```
student@cp:~$ sudo crictl ps |wc -l
```

27

```
student@worker:~$ sudo crictl ps |wc -l
```

17

- Delete the deployment. Verify the containers are gone.

```
student@cp:~$ kubectl delete deployment taint-deployment
```

```
deployment.apps "taint-deployment" deleted
```

```
student@cp:~$ sudo crictl ps |wc -l
```

21

- Now we will use a taint to affect the deployment of new containers. There are three taints, NoSchedule, PreferNoSchedule and NoExecute. The taints having to do with schedules will be used to determine newly deployed containers, but will not affect running containers. The use of NoExecute will cause running containers to move.

Taint the secondary node, verify it has the taint then create the deployment again. We will use the key of bubba to illustrate the key name is just some string an admin can use to track Pods.

```
student@cp:~$ kubectl taint nodes worker \
    bubba=value:PreferNoSchedule
```

```
node/worker tainted
```

```
student@cp:~$ kubectl describe node |grep Taint
```

```
Taints:          bubba=value:PreferNoSchedule
Taints:          <none>
```

```
student@cp:~$ kubectl apply -f taint.yaml
```

```
deployment.apps/taint-deployment created
```

- Locate where the containers are running. We can see that more containers are on the cp, but there still were some created on the secondary. Delete the deployment when you have gathered the numbers.

```
student@cp:~$ sudo crictl ps |wc -l
```

21

```
student@worker:~$ sudo crictl ps |wc -l
```

23

```
student@cp:~$ kubectl delete deployment taint-deployment
deployment.apps "taint-deployment" deleted
```

7. Remove the taint, verify it has been removed. Note that the key is used with a minus sign appended to the end.

```
student@cp:~$ kubectl taint nodes worker bubba-
node/worker untainted
```

```
student@cp:~$ kubectl describe node |grep Taint
```

Taints:	<none>
Taints:	<none>

8. This time use the NoSchedule taint, then create the deployment again. The secondary node should not have any new containers, with only daemonsets and other essential pods running.

```
student@cp:~$ kubectl taint nodes worker \
bubba=value:NoSchedule
node/worker tainted
```

```
student@cp:~$ kubectl apply -f taint.yaml
deployment.apps/taint-deployment created
```

```
student@cp:~$ sudo crictl ps |wc -l
```

21

```
student@worker:~$ sudo crictl ps |wc -l
```

23

9. Remove the taint and delete the deployment. When you have determined that all the containers are terminated create the deployment again. Without any taint the containers should be spread across both nodes.

```
student@cp:~$ kubectl delete deployment taint-deployment
deployment.apps "taint-deployment" deleted
```

```
student@cp:~$ kubectl taint nodes worker bubba-
node/worker untainted
```

```
student@cp:~$ kubectl apply -f taint.yaml
```

```
deployment.apps/taint-deployment created
```

```
student@cp:~$ sudo crictl ps |wc -l
```

```
27
```

```
student@worker:~$ sudo crictl ps |wc -l
```

```
17
```

10. Now use the NoExecute to taint the secondary (**worker**) node. Wait a minute then determine if the containers have moved. The DNS containers can take a while to shutdown. Some containers will remain on the worker node to continue communication from the cluster.

```
student@cp:~$ kubectl taint nodes worker \
    bubba=value:NoExecute
```

```
node "worker" tainted
```

```
student@cp:~$ sudo crictl ps |wc -l
```

```
37
```

```
student@worker:~$ sudo crictl ps |wc -l
```

```
5
```

11. Remove the taint. Wait a minute. Note that all of the containers did not return to their previous placement.

```
student@cp:~$ kubectl taint nodes worker bubba-
```

```
node/worker untainted
```

```
student@cp:~$ sudo crictl ps |wc -l
```

```
32
```

```
student@worker:~$ sudo crictl ps |wc -l
```

```
6
```

12. Remove the deployment a final time to free up resources.

```
student@cp:~$ kubectl delete deployment taint-deployment
```

```
deployment.apps "taint-deployment" deleted
```

13. **CHALLENGE STEP** Use your knowledge of deployments and scaling items to deploy multiple `httpd` pods across the nodes and examine the typical spread and spread after using taints.

Chapter 13

Logging and Troubleshooting



13.1	Overview	296
13.2	Troubleshooting Flow	297
13.3	Basic Start Sequence	299
13.4	Monitoring	300
13.5	Plugins	301
13.6	Logging	305
13.7	Troubleshooting Resources	306
13.8	Labs	307

13.1 Overview

Overview

- **Linux** troubleshooting via shell
- Turn on basic monitoring
- Set up cluster-wide logging
- External products **Fluentd**, **Prometheus** helpful.
- Internal **Metrics Server** and API

Kubernetes relies on API calls and is sensitive to network issues. Standard **Linux** tools and processes are the best method for troubleshooting your cluster. If a shell, such as **bash** is not available in an affected Pod, consider deploying another similar pod with a shell, like **busybox**. DNS configuration files and tools like **dig** are a good place to start. For more difficult challenges you may need to install other tools like **tcpdump**.

Large and diverse workloads can be difficult to track, so monitoring of usage is essential. Monitoring is about collecting key metrics such as CPU, memory, and disk usage, and network bandwidth on your nodes, as well as monitoring key metrics in your applications. These features are being ingested into Kubernetes with the **Metric Server**, which is a cut-down version of the now deprecated **Heapster**. Once installed the **Metrics Server** exposes a standard API which can be consumed by other agents such as autoscalers. Once installed this endpoint can be found here on the cp server: </apis/metrics/k8s.io/>

Logging activity across all the nodes is another feature not part of Kubernetes. Using **Fluentd** can be a useful data collector for a unified logging layer. Having aggregated logs can help visualize the issue and provides the ability to search all logs. A good place to start when local network troubleshooting does not expose the root cause. It can be downloaded from <http://www.fluentd.org>

Another project from cncf.io combines logging, monitoring, and alerting called **Prometheus** can be found here <https://prometheus.io>. It provides a time-series database as well as integration with **Grafana** for visualization and dashboards.

We are going to review some of the basic **kubectl** commands that you can use to debug what is happening, and we will walk you through the basic steps to be able to debug your containers, your pending containers, and also the systems in Kubernetes.

13.2 Troubleshooting Flow

Basic Steps

- Errors from command line
- Pod logs and state of the Pod
- Use shell to troubleshoot Pod DNS and network
- Check node logs for errors. Enough resources
- RBAC, SELinux or AppArmor security settings
- API calls to and from controllers to **kube-apiserver**
- Enable auditing
- Inter-node network issues, DNS and firewall
- Control Plane server controllers.
 - Control Pods in pending or error state
 - Errors in log files
 - Enough resources

The flow of troubleshooting should start with the obvious. If there are errors from the command line investigate them first. The symptoms of the issue will probably determine the next step to check. Working from the application running inside a container to the cluster as a whole may be a good idea. The application may have a shell you can use, for example:

```
$ kubectl create deploy busybox --image=busybox --command sleep 3600  
$ kubectl exec -ti <busybox_pod> -- /bin/sh
```

If the Pod is running use **kubectl logs pod-name** to view standard out of the container. Without logs you may consider deploying a sidecar container in the Pod to generate and handle logging. The next place to check is networking including DNS, firewalls and general connectivity using standard **Linux** commands and tools.

Security settings can also be a challenge. RBAC, covered in the security chapter, provides mandatory or discretionary access control in a granular manner. SELinux and AppArmor are also common issues, especially with network-centric applications.

A newer feature of Kubernetes is the ability to enable auditing for the **kube-apiserver**, which can allow a view into actions after the API call has been accepted.

The issues found with a decoupled system like Kubernetes are similar to those of a traditional datacenter, plus the added layers of Kubernetes controllers.

Ephemeral Containers

- Add a tool-filled container to a running pod
- Alpha with v1.16 release

```
kubectl debug buggypod --image debian --attach
```

A feature new to the 1.16 version is the ability to add a container to a running pod. This would allow a feature-filled container to be added to an existing pod without having to terminate and re-create. Intermittent and difficult to determine problems may take a while to reproduce, or not exist with the addition of another container.

As an Alpha stability feature, it may change or be removed at any time. As well they will not be restarted automatically, and several resources such as ports or resources are not allowed.

These containers are added via the `ephemeralcontainers` handler via an API call, not via the `podSpec`. As a result the use of `kubectl edit` is not possible.

You may be able to use the `kubectl attach` command to join an existing process within the container. This can be helpful instead of `kubectl exec` which executes a new process. The functionality of the attached process depends entirely on what you are attaching to.

13.3 Basic Start Sequence

Cluster Start Sequence

- **systemd** starts `kubelet.service`
 - Uses `/etc/systemd/system/kubelet.service.d/10-kubeadm.conf`
 - Uses `/var/lib/kubelet/config.yaml` config file
 - **staticPodPath** set to `/etc/kubernetes/manifests/`
- **kubelet** creates all pods from `*.yaml` in directory
 - `kube-apiserver`
 - `etcd`
 - `kube-controller-manager`
 - `kube-scheduler`
- `kube-controller-manager` control loops use `etcd` data to start rest

The cluster startup sequence begins with **systemd** if you built the cluster using **kubeadm**. Other tools may leverage a different method. Use `systemctl status kubelet.service` to see the current state and configuration files used to run the **kubelet** binary.

Inside of the `config.yaml` file you will find several settings for the binary including the **staticPodPath** which indicates the directory where **kubelet** will read every yaml file and start every pod. If you put a yaml file in this directory it is a way to troubleshoot the scheduler, as the pod is created with any requests to the scheduler.

The four default yaml files will start the base pods necessary to run the cluster. Once the watch loops and controllers from **kube-controller-manager** run using `etcd` data the rest of the configured objects will be created.

13.4 Monitoring

Monitoring

- Enable the add-ons
- **Metrics Server** and API
- **Prometheus**
- **Jaeger**
- **OpenTelemetry**
- API Server Tracing

Monitoring is about collecting metrics from the infrastructure, as well as applications.

The long used and now deprecated **Heapster** has been replaced with an integrated **Metrics Server**. Once installed and configured the server exposes a standard API which other agents can use to determine usage. It can also be configured to expose custom metrics, which then could also be used by autoscalers to determine if an action should take place.

Prometheus

Prometheus is part of the **Cloud Native Computing Foundation (CNCF)**. As a Kubernetes plugin, it allows one to scrape resource usage metrics from Kubernetes objects across the entire cluster. It also has several client libraries which allow you to instrument your application code in order to collect application level metrics.

Other CNCF projects such as **OpenTelemetry** which allows for adding instrumentation to code, and **Jaeger** for consuming the telemetry are popular to use when tracking distributed issues. An alpha feature is API Server Tracing, which leverages **OpenTelemetry**. More can be found here: <https://kubernetes.io/blog/2021/09/03/api-server-tracing/>.

13.5 Plugins

Using Krew

- Install the software using steps at <https://krew.dev>

```
$ kubectl krew help
krew is the kubectl plugin manager.
You can invoke krew through kubectl: "kubectl krew [command]..."
```

Usage:

```
krew [command]
```

Available Commands:

help	Help about any command
info	Show information about a kubectl plugin
install	Install kubectl plugins
list	List installed kubectl plugins
search	Discover kubectl plugins
uninstall	Uninstall plugins
update	Update the local copy of the plugin index
upgrade	Upgrade installed plugins to newer versions
version	Show krew version and diagnostics

We have been using the **kubectl** command throughout the course. The basic commands can be used together in a more complex manner extending what can be done. There are over seventy and growing plugins available to interact with Kubernetes objects and components.

At the moment plugins cannot overwrite existing **kubectl** commands. Nor can it add sub-commands to existing commands. Writing new plugins should take into account the command line runtime package and a Go library for plugin authors.

As a plugin the declaration of options such as namespace or container to use must come after the command.

```
$ kubectl sniff bigpod-abcd-123 -c mainapp -n accounting
```

Plugins can be distributed in many ways. The use of **krew** allows for cross-platform packaging and a helpful plugin index, which makes finding new plugins easy.

More information can be found here: <https://kubernetes.io/docs/tasks/extend-kubectl/kubectl-plugins/>

Managing Plugins

- Add paths to plugins to \$PATH variable
- View current plugins with **kubectl plugin list**
- Find new plugins with **kubectl krew search**
- Installing using **kubectl krew install new-plugin**
- Once installed use as **kubectl** sub-command
- **upgrade** and **uninstall** also available

The `help` option explains basic operation. After installation ensure the `$PATH` includes the plugins. `krew` should allow easy installation and use after that.

```
$ export PATH="${KREW_ROOT:-$HOME/.krew}/bin:$PATH"
$ kubectl krew search
```

NAME	DESCRIPTION	INSTALLED
access-matrix	Show an RBAC access matrix for server resources	no
advise-psp	Suggests PodSecurityPolicies for cluster.	no
....		

```
$ kubectl krew install tail
```

```
Updated the local copy of plugin index.
Installing plugin: tail
Installed plugin: tail
 \
| Use this plugin:

....
| | Usage:
| |
| | # match all pods
| | $ kubectl tail
| |
| | # match pods in the 'frontend' namespace
| | $ kubectl tail --ns staging
....
```

Sniffing Traffic With Wireshark

- Read installation output for basic usage
- Note some require extra packages and configuration.
- **sniff** requires **Wireshark** and ability to export graphical display
- Pass the command the pod and container to use

```
$ kubectl krew install sniff nginx-123456-abcd -c webcont
```

Cluster network traffic is encrypted making troubleshooting of possible network issues more complex. Using the **sniff** plugin you can view the traffic from within.

The **sniff** command will use the first found container unless you pass the **-c** option to declare which container in the pod to use for traffic monitoring.

13.6 Logging

Logging Tools

- No cluster-wide logging in Kubernetes
- Often aggregated and digested by outside tools like **ElasticSearch**
- **Fluentd**
- **Kibana**

Logging, like monitoring, is a vast subject in IT. It has many tools that you can use as part of your arsenal.

Typically, logs are collected locally and aggregated before being ingested by a search engine and displayed via a dashboard which can use the search syntax. While there are many software stacks that you can use for logging, the **Elasticsearch**, **Logstash**, and **Kibana Stack (ELK)** has become quite common.

In Kubernetes, the **kubelet** writes container logs to local files (via the **Docker** logging driver). The command `kubectl logs` allows you to retrieve these logs.

Cluster-wide, you can use **Fluentd** to aggregate logs: <http://www.fluentd.org/>. Check the cluster administration logging concepts for a detailed description: <https://kubernetes.io/docs/concepts/cluster-administration/logging/>.

Fluentd is part of the **Cloud Native Computing Foundation (CNCF)** and, together with **Prometheus**, makes a nice combination for monitoring and logging. You can find a detailed walk-through of running **Fluentd** on Kubernetes here: <https://kubernetes.io/docs/tasks/debug-application-cluster/logging-elasticsearch-kibana>.

Setting up **Fluentd** for Kubernetes logging can be a good exercise in understanding **DaemonSets**. **Fluentd** agents run on each node via **DaemonSet**, they aggregate the logs, and feed them to an **Elasticsearch** instance prior to visualization in a **Kibana** dashboard.

13.7 Troubleshooting Resources

More Resources

- Official documentation
- Major vendor pages
- Github pages
- Kubernetes **Slack** channel

Other URLs to view for more troubleshooting information:

- General guidelines and instructions:
<https://kubernetes.io/docs/tasks/debug-application-cluster/troubleshooting/>
- Troubleshooting applications:
<https://kubernetes.io/docs/tasks/debug-application-cluster/debug-application>
- Troubleshooting clusters:
<https://kubernetes.io/docs/tasks/debug-application-cluster/debug-cluster>
- Debugging pods:
<https://kubernetes.io/docs/tasks/debug-application-cluster/debug-pod-replication-controller>
- Debugging services:
<https://kubernetes.io/docs/tasks/debug-application-cluster/debug-service/>
- Github Site for issue and bug tracking
<https://github.com/kubernetes/kubernetes/issues>
- Kubernetes Slack channel
kubernetes.slack.com

13.8 Labs

Exercise 13.1: Review Log File Locations

Overview

In addition to various logs files and command output, you can use **journalctl** to view logs from the node perspective. We will view common locations of log files, then a command to view container logs. There are other logging options, such as the use of a **sidecar** container dedicated to loading the logs of another container in a pod.

Whole cluster logging is not yet available with Kubernetes. Outside software is typically used, such as **Fluentd**, part of <http://fluentd.org/>, which is another member project of **CNCF.io**, like Kubernetes.

Take a quick look at the following log files and web sites. As server processes move from node level to running in containers the logging also moves.

1. If using a **systemd**-based Kubernetes cluster, view the node level logs for **kubelet**, the local Kubernetes agent. Each node will have different contents as this is node specific.

```
student@cp:~$ journalctl -u kubelet |less
```

```
<output_omitted>
```

2. Major Kubernetes processes now run in containers. You can view them from the container or the pod perspective. Use the **find** command to locate the **kube-apiserver** log. Your output will be different, but will be very long.

```
student@cp:~$ sudo find / -name "*apiserver*log"
```

```
/var/log/containers/kube-apiserver-cp_kube-system_kube-apiserver-423d25701998f68b503e64d41dd786e657fc09504f13278044934d79a4019e3c.log
```

3. Take a look at the log file.

```
student@cp:~$ sudo less /var/log/containers/kube-apiserver-cp_kube-system_kube-apiserver-423d25701998f68b503e64d41dd786e657fc09504f13278044934d79a4019e3c.log
```

```
<output_omitted>
```

4. Search for and review other log files for coredns, kube-proxy, and other cluster agents.

5. If **not** on a Kubernetes cluster using **systemd** which collects logs via **journalctl** you can view the text files on the cp node.

- (a) **/var/log/kube-apiserver.log**
Responsible for serving the API
- (b) **/var/log/kube-scheduler.log**
Responsible for making scheduling decisions
- (c) **/var/log/kube-controller-manager.log**
Controller that manages replication controllers

6. **/var/log/containers**

Various container logs

7. `/var/log/pods/`

More log files for current Pods.

8. Worker Nodes Files (on non-**systemd** systems)(a) `/var/log/kubelet.log`

Responsible for running containers on the node

(b) `/var/log/kube-proxy.log`

Responsible for service load balancing

9. More reading: <https://kubernetes.io/docs/tasks/debug-application-cluster/debug-service/> and <https://kubernetes.io/docs/tasks/debug-application-cluster/determine-reason-pod-failure/> **Exercise 13.2: Viewing Logs Output**

Container standard out can be seen via the **kubectl logs** command. If there is no standard out, you would not see any output. In addition, the logs would be destroyed if the container is destroyed.

1. View the current Pods in the cluster. Be sure to view Pods in all namespaces.

```
student@cp:~$ kubectl get po --all-namespaces
```

NAMESPACE	NAME	READY	STATUS	RESTARTS	AGE
kube-system	cilium-operator-788c7d7585-jgn6s	1/1	Running	0	13m
kube-system	cilium-5tv9d	1/1	Running	0	6d1h
....					
kube-system	etcd-cp	1/1	Running	2	44h
kube-system	kube-apiserver-cp	1/1	Running	2	44h
kube-system	kube-controller-manager-cp	1/1	Running	2	44h
kube-system	kube-scheduler-cp	1/1	Running	2	44h
....					

2. View the logs associated with various infrastructure pods. Using the **Tab** key you can get a list and choose a container. Then you can start typing the name of a pod and use **Tab** to complete the name.

```
student@cp:~$ kubectl -n kube-system logs <Tab><Tab>
```

```
cilium-operator-788c7d7585-jgn6s
cilium-5tv9d
coredns-5644d7b6d9-k7kts
coredns-5644d7b6d9-rnr2v
etcd-cp
kube-apiserver-cp
kube-controller-manager-cp
kube-proxy-qhc4f
kube-proxy-s56hl
kube-scheduler-f-cp
traefik-ingress-controller-hw5tv
traefik-ingress-controller-mcn47
```

```
student@cp:~$ kubectl -n kube-system logs \
    kube-apiserver-cp
```

```
Flag --insecure-port has been deprecated, This flag will be removed in a future version.
I1119 02:31:14.933023      1 server.go:623] external host was not specified, using 10.128.0.3
```

```
I1119 02:31:14.933356      1 server.go:149] Version: v1.29.1
I1119 02:31:15.595131      1 plugins.go:158] Loaded 11 mutating admission controller(s)
successfully in the following order: NamespaceLifecycle,LimitRanger,ServiceAccount,
NodeRestriction,TaintNodesByCondition,Priority,DefaultTolerationSeconds,DefaultStorageClass,
StorageObjectInUseProtection,MutatingAdmissionWebhook,RuntimeClass.
I1119 02:31:15.595357      1 plugins.go:161] Loaded 7 validating admission controller(s)
successfully in the following order: LimitRanger,ServiceAccount,Priority,
PersistentVolumeClaimResize,ValidatingAdmissionWebhook,RuntimeClass,
ResourceQuota.
<output_omitted>
```

- View the logs of other Pods in your cluster.

Exercise 13.3: Adding tools for monitoring and metrics

With the deprecation of **Heapster** the new, integrated **Metrics Server** has been further developed and deployed. The **Prometheus** project of **CNCF.io** has matured from incubation to graduation, is commonly used for collecting metrics, and should be considered as well.

Configure Metrics

- Begin by cloning the software. The **git** command should be installed already. Install it if not found.

```
student@cp:~$ git clone \
  https://github.com/kubernetes-incubator/metrics-server.git
<output_omitted>
```

- As the software may have changed it is a good idea to read the **README.md** file for updated information.

```
student@cp:~$ cd metrics-server/ ; less README.md
<output_omitted>
```

- Create the necessary objects. Be aware as new versions are released there may be some changes to the process and the created objects. Use the **components.yaml** to create the objects. The backslash is not necessary if you type it all on one line.

```
student@cp:~$ kubectl create -f \
  https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml
serviceaccount/metrics-server created
clusterrole.rbac.authorization.k8s.io/system:aggregated-metrics-reader created
clusterrole.rbac.authorization.k8s.io/system:metrics-server created
rolebinding.rbac.authorization.k8s.io/metrics-server-auth-reader created
clusterrolebinding.rbac.authorization.k8s.io/metrics-server:system:auth-delegator created
clusterrolebinding.rbac.authorization.k8s.io/system:metrics-server created
service/metrics-server created
deployment.apps/metrics-server created
apiservice.apiregistration.k8s.io/v1beta1.metrics.k8s.io created
```

4. View the current objects, which are created in the kube-system namespace. All should show a Running status. You will notice the metrics server pod is in not ready state. Allow the deployment to run insecure TLS and pod will start accepting the traffic.

```
student@cp:~$ kubectl -n kube-system get pods
```

<output omitted>				
kube-proxy-1d2hb	1/1	Running	0	2d21h
kube-scheduler-u16-1-13-1-2f8c	1/1	Running	0	2d21h
metrics-server-fc6d4999b-b9rjj	0/1	Running	0	42s

5. Edit the metrics-server deployment to allow insecure TLS. The default certificate is x509 self-signed and not trusted by default. In production you may want to configure and replace the certificate. You may encounter other issues as this software is fast-changing. The need for the kubelet-preferred-address-types line has been reported on some platforms.

```
student@cp:~$ kubectl -n kube-system edit deployment metrics-server
```

Y
ML

```
.....
2   spec:
3     containers:
4       - args:
5         - --cert-dir=/tmp
6         - --secure-port=4443
7         - --kubelet-insecure-tls           #<-- Add this line
8         - --kubelet-preferred-address-types=InternalIP,ExternalIP,Hostname #<--May be needed
9       image: k8s.gcr.io/metrics-server/metrics-server:v0.3.7
10    ....
11
```

6. Test that the metrics server pod is running and does not show errors. At first you should see a few lines showing the container is listening. As the software changes these messages may be slightly different.

```
student@cp:~$ kubectl -n kube-system logs metrics-server<TAB>
```

I0207 14:08:13.383209	1 serving.go:312] Generated self-signed cert
(/tmp/apiserver.crt, /tmp/apiserver.key)	
I0207 14:08:14.078360	1 secure_serving.go:116] Serving securely on
[::]:4443	

7. Test that the metrics working by viewing pod and node metrics. Your output may have different pods. It can take an minute or so for the metrics to populate and not return an error.

```
student@cp:~$ sleep 120 ; kubectl top pod --all-namespaces
```

NAMESPACE	NAME	CPU(cores)	MEMORY(bytes)
kube-system	cilium-kube-controllers-7b9dcdcc5-qg6zd	2m	6Mi
kube-system	cilium-node-dr279	23m	22Mi
kube-system	cilium-node-xtvfd	21m	22Mi
kube-system	coredns-5644d7b6d9-k7kts	2m	6Mi
kube-system	coredns-5644d7b6d9-rnr2v	3m	6Mi
<output omitted>			

```
student@cp:~$ kubectl top nodes
```

NAME	CPU(cores)	CPU%	MEMORY(bytes)	MEMORY%
cp	228m	11%	2357Mi	31%

worker	76m	3%	1385Mi	18%
--------	-----	----	--------	-----

8. Using keys we generated in an earlier lab we can also interrogate the API server. Your server IP address will be different.

```
student@cp:~$ curl --cert ./client.pem \
--key ./client-key.pem --cacert ./ca.pem \
https://k8scp:6443/apis/metrics.k8s.io/v1beta1/nodes

{
  "kind": "NodeMetricsList",
  "apiVersion": "metrics.k8s.io/v1beta1",
  "metadata": {
    "selfLink": "/apis/metrics.k8s.io/v1beta1/nodes"
  },
  "items": [
    {
      "metadata": {
        "name": "u16-1-13-1-2f8c",
        "selfLink": "/apis/metrics.k8s.io/v1beta1/nodes/u16-1-13-1-2f8c",
        "creationTimestamp": "2023-08-10T20:27:00Z"
      },
      "timestamp": "2023-08-10T20:26:18Z",
      "window": "30s",
      "usage": {
        "cpu": "215675721n",
        "memory": "2414744Ki"
      }
    },
    <output_omitted>
```

Configure the Dashboard

While the dashboard looks nice it has not been a common tool in use. Those that could best develop the tool tend to only use the CLI, so it may lack full wanted functionality.

The first commands do not have the details. Refer to earlier content as necessary.

1. Copy the dashboard yaml from the tarball and deploy the dashboard.

```
student@cp:~$ cp /home/student/LFS458/SOLUTIONS/s_13/dashboard.yaml .
```

```
student@cp:~$ kubectl create -f dashboard.yaml
```

2. We will give the dashboard full admin rights, which may be more than one would in production. The dashboard is running in the kubernetes-dashboard namespace. kubernetes-dashboard is the name of the service account.

There is more on service account in the Security chapter.

```
student@cp:~$ kubectl get sa -n kubernetes-dashboard
```

NAME	SECRETS	AGE
default	0	4m25s
kubernetes-dashboard	0	4m25s

```
student@cp:~$ kubectl create clusterrolebinding dashaccess \
--clusterrole=cluster-admin \
--serviceaccount=kubernetes-dashboard:kubernetes-dashboard
```

```
clusterrolebinding.rbac.authorization.k8s.io/dashaccess created
```

3. On your local system open a browser and navigate to an HTTPS URL made of the Public IP and the high-numbered port. You will get a message about an insecure connection. Select the **Advanced** button, then **Add Exception...**, then **Confirm Security Exception**. Some browsers won't even give you to option. If nothing shows up try a different browser. The page should then show the Kubernetes Dashboard. You may be able to find the public IP address using **curl**.

```
student@cp:~$ curl ifconfig.io
```

```
35.231.8.178
```

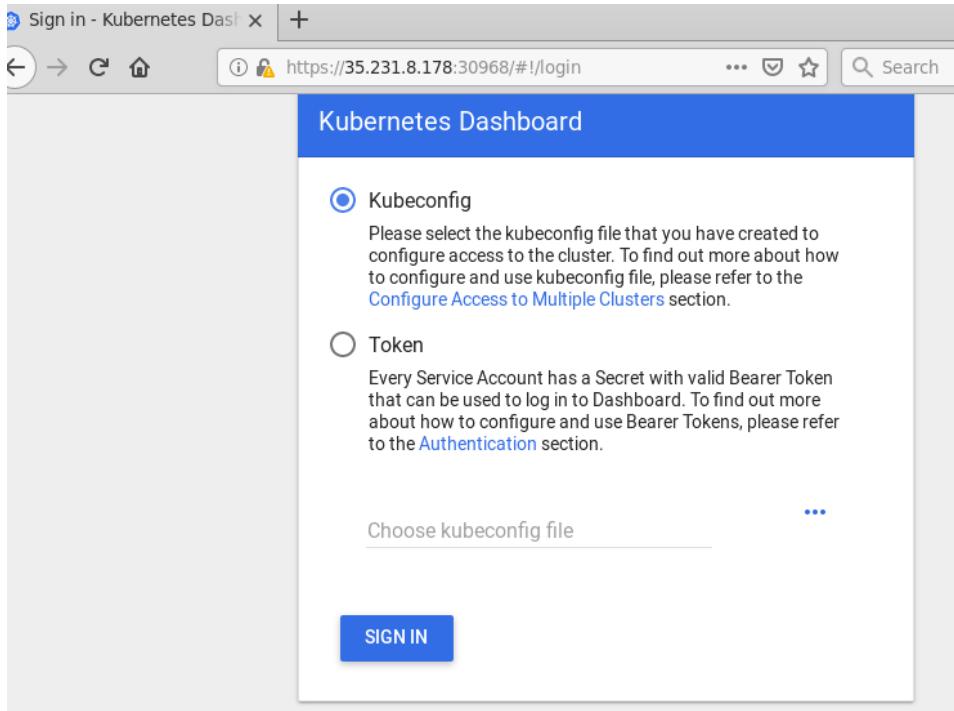


Figure 13.1: External Access via Browser

4. We will use the Token method to access the dashboard. With RBAC we need to use the proper token, the `kubernetes-dashboard-token` in this case. Find the token, copy it then paste into the login page. The **Tab** key can be helpful to complete the secret name instead of finding the hash.

```
student@cp:~$ kubectl create token kubernetes-dashboard -n kubernetes-dashboard
```

```
eyJlxvezoLAilithbGci0iJSUzI1NiIsImtpZCI6IiJ9.eyJpc3MiOiJrdWJlcm5ldGVzL3N1cnZpY2VhY2NvdW50Iiwia3ViZXJuZXRLcy5pbv9zZXJ2aWN1YWNgjb3VudC9uYW11c3BhY2UiOiJrdWJlcm5c3R1bSIsImt1YmVybmVOZXMuaW8vc2VydmljZWfjY291bnQvc2Vjcm5hbwUiOiJrdWJlcm5ldGVzLWRhc2hib2FyZC10b2t1bi1wbw04NCIsImt1YmVybmVOZXMuaW8vc2VydmljZWfjY291bnQvc2VydmljZS1hY2NvdW50Lm5hbWUiOiJrdWJlcm5ldGVzLWRhc2hib2FyZCIsImt1YmVybmVOZXMuaW8vc2VydmljZWfjY291bnQvc2VydmljZS1hY2NvdW50LnVpZCI6IjE5MDY4ZD1zLTE1MTctMTFl0S1hZmMyLTQyMDEwYTh1MDAwMyIsInN1YiI6InN5c3R1bTpzZXJ2aWN1YWNgjb3VudDprdWJlcm51dGVzLWRhc2hib2FyZCJ9.aYTUMWr290pj5i32rb8qXpq4nn3hLhvz6yLSYexgRd6NYsygVUyqnkRsFE1trg9i1ftNKKJdzkY5kQzN3AcpUTvyj_BvJgzNh3JM9p7QMjI8LHTz4TrRZrvwJVWitrEn4VnTQuFvCAdFD_rKB9FyI_gvT_QiW5fQm24ygTIgf0Yd44263oakG8sL64q7UfQNw2t5S0orMUtyb0mx4CXNUYM8G44ejEtV9GW50sVjEmLIGaoEMX7fctwUN_XCyPdzCg2W0xRHahBJmbCuLz2SSWL52q4nXQmhTq_L8VDDpt6LjEqXW6LtDJZGjVCs2MnBLerQz-ZAgsvaubbQ
```

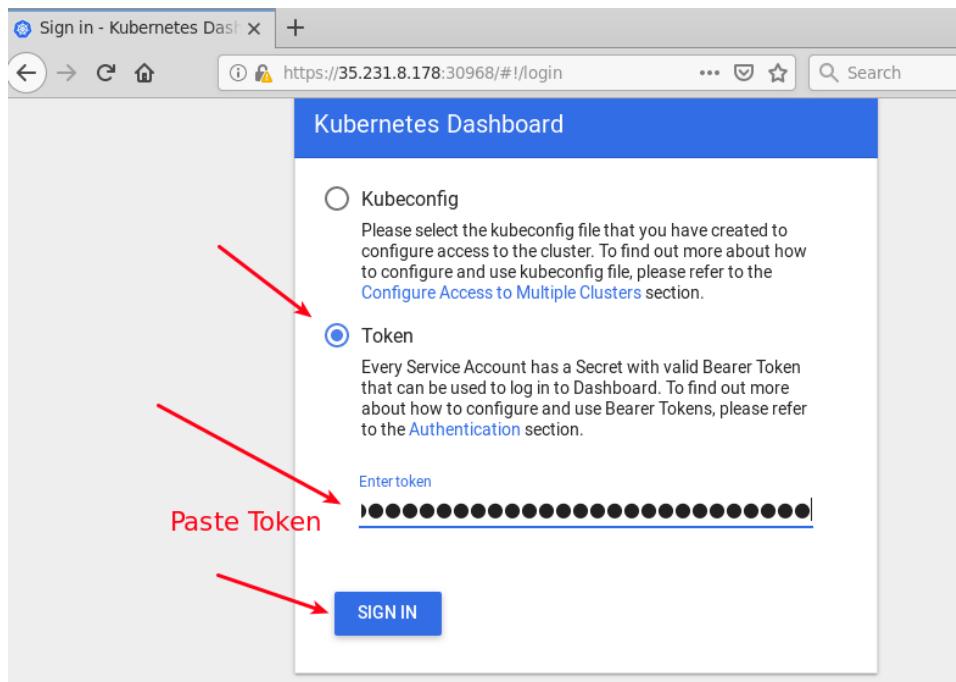


Figure 13.2: External Access via Browser

5. Navigate around the various sections and use the menu to the left as time allows. As the pod view is of the default namespace, you may want to switch over to the kube-system namespace or create a new deployment to view the resources via the GUI. Scale the deployment up and down and watch the responsiveness of the GUI.

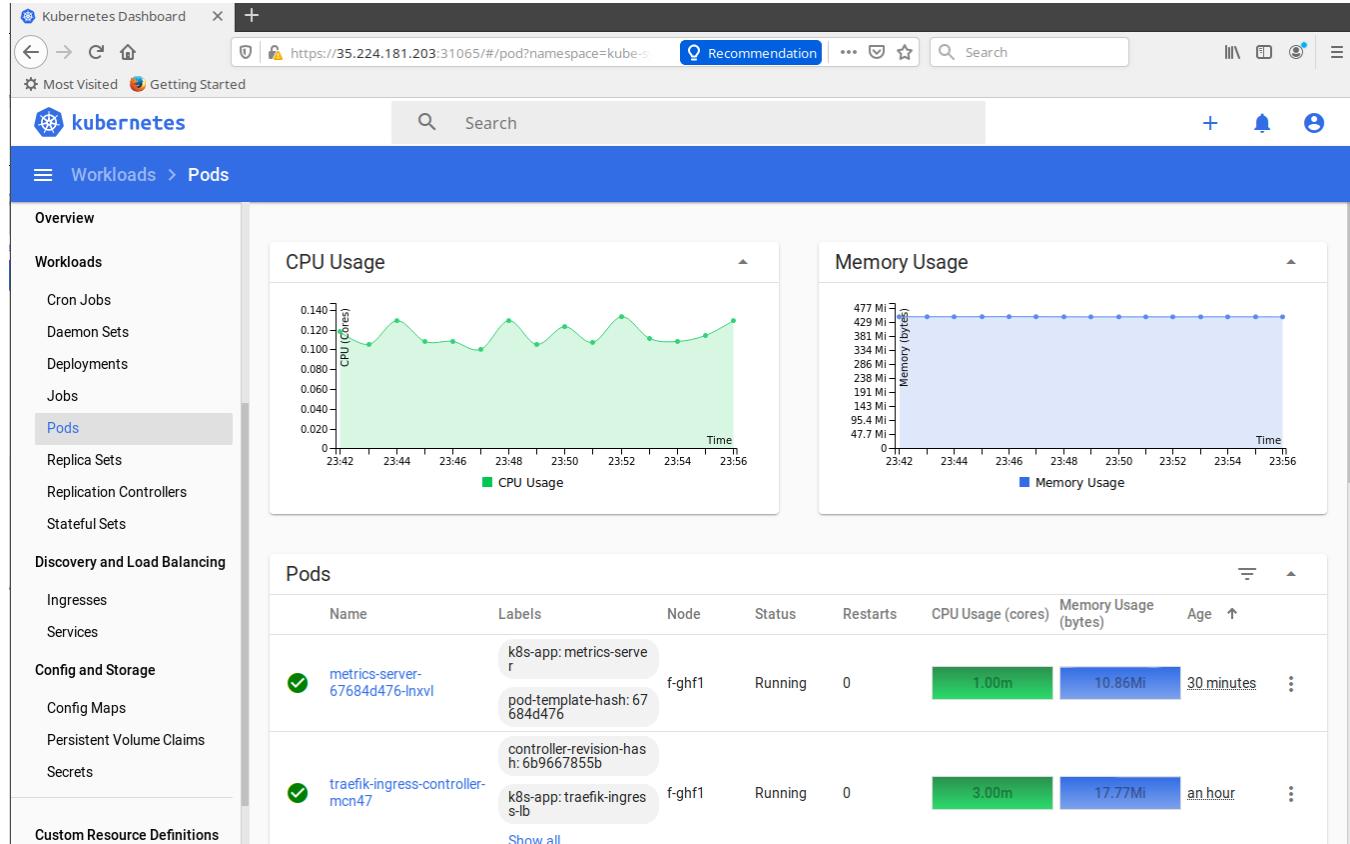


Figure 13.3: External Access via Browser

Chapter 14

Custom Resource Definition



14.1	Overview	316
14.2	Custom Resource Definitions	317
14.3	Aggregated APIs	321
14.4	Labs	322

14.1 Overview

Custom Resources

- Flexible to meet changing needs
- Create your own API objects
- Manage your API objects via **kubectl**
- Custom Resource Definitions (CRD)
- Aggregated APIs (AA)

We have been working with built-in resources, or API endpoints. The flexibility of Kubernetes allows for dynamic addition of new resources as well. Once these Custom Resources have been added the objects can be created and accessed using standard calls and commands like **kubectl**. The creation of a new object stores new structured data in the **etcd** database and allows access via **kube-apiserver**.

To make a new, custom resource part of a declarative API there needs to be a controller to retrieve the structured data continually and act to meet and maintain the declared state. This controller, or operator, is an agent to create and manage one or more instances of a specific stateful application. We have worked with built-in controllers such for Deployments, DaemonSets and other resources.

The functions encoded into a custom operator should be all the tasks a human would need to perform if deploying the application outside of Kubernetes. The details of building a custom controller are outside the scope of this course and not included.

There are two ways to add custom resources to your Kubernetes cluster. The easiest, but less flexible, way is by adding a Custom Resource Definition to the cluster. The second which is more flexible is the use of Aggregated APIs which requires a new API server to be written and added to the cluster.

Either way of a new object to the cluster, as distinct from a built-in resource, is called a Custom Resource.

If you are using RBAC for authorization you probably will need to grant access to the new CRD resource and controller. If using an Aggregated API, you can use the same or different authentication process.

14.2 Custom Resource Definitions

Custom Resource Definitions

- Easy to deploy
- Typically does not require programming
- Does not require another API server
- Namespaced or cluster-scoped

As we have learned decoupled nature of Kubernetes depends on a collection of watcher loops, or controllers, interrogating the **kube-apiserver** to determine if a particular configuration is true. If the current state does not match the declared state the controller makes API calls to modify the state until they do match. If you add a new API object and controller you can use the existing **kube-apiserver** to monitor and control the object. The addition of a Custom Resource Definition will be added to the cluster API path, currently under [apiextensions.k8s.io/v1](#).

While this is the easiest way to add a new object to the cluster it may not be flexible enough for your needs. Only the existing API functionality can be used. Objects must respond to REST requests and have their configuration state validated and stored in the same manner as built-in objects. They would also need to exist with the protection rules of built-in objects.

A CRD allows the resource to be deployed in a namespace or available in the entire cluster. The YAML file sets this with the scope: parameter which can be set to Namespaced or Cluster.

Prior to v1.8 there was a resource type called ThirdPartyResource (TPR). This has been deprecated and is no longer available. All resources will need to be rebuilt as CRD. After upgrade existing TPRs will need to be removed and replaced by CRDs such that the API URL points to functional object.

Configuration Example

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: backups.stable.linux.com
spec:
  group: stable.linux.com
  versions: v1
  scope: Namespaced
  names:
    plural: backups
    singular: backup
    shortNames:
      - bks
    kind: BackUp
```

apiVersion:

Should match the current level of stability, currently apiextensions.k8s.io/v1

kind: CustomResourceDefinition

The object type being inserted by the **kube-apiserver**.

name: backups.stable.linux.com

The name must match the spec field declared later. The syntax must be <plural name>.<group>.

group: stable.linux.com

The group name will become part of the REST API under /apis/<group>/<version> or /apis/stable/v1 in this case with the versions set to v1.

scope

Determines if the object exists in a single namespace or is cluster-wide.

plural

Defines the last part of the API URL such as apis/stable/v1/backups.

singular and shortNames

represent the name with displayed and make CLI usage easier.

kind

A CamelCased singular type used in resource manifests.

New Object Configuration

```
apiVersion: "stable.linux.com/v1"
kind: BackUp
metadata:
  name: a-backup-object
spec:
  timeSpec: "* * * * */5"
  image: linux-backup-image
  replicas: 5
```

Note that the `apiVersion` and `kind` match the CRD we created in a previous step. The `spec` parameters depend on the controller.

The object will be evaluated by the controller. If the syntax, such as `timeSpec` does not match the expected value you will receive an error, should validation be configured. Without validation only the existence of the variable is checked, not its details.

Optional Hooks

- Finalizer

```
metadata:  
  finalizers:  
    - finalizer.stable.linux.com
```

- Validation

```
validation:  
  openAPIV3Schema:  
    properties:  
      spec:  
        properties:  
          timeSpec:  
            type: string  
            pattern: '^(\d+|*)(/\d+)?(\s+(\d+|*)(/\d+)?){4}$'  
          replicas:  
            type: integer  
            minimum: 1  
            maximum: 10
```

Just as with built-in objects you can use an asynchronous pre-delete hook known as a `Finalizer`. If an API delete request is received the object metadata field `metadata.deletionTimestamp` is updated. The controller then triggers whichever finalizer has been configured. When the finalizer completes it is removed from the list. The controller continues to compete and remove finalizers until the string is empty. Then the object itself is deleted.

A feature in beta starting with v1.9 allows for validation of custom objects via the OpenAPI v3 schema. This will check various properties of the object configuration being passed the API server. In the example above the `timeSpec` must be a string matching a particular pattern and the number of allowed replicas is between one and 10. If the validation does not match the error returned is the failed line of validation.

14.3 Aggregated APIs

Understanding Aggregated APIs (AA)

- Usually requires non-trivial programming
- More control over API behavior
- Subordinate API server behind primary
- Primary acts as proxy
- Leverages extension resource
- Mutual TLS auth between API servers
- RBAC rule to allow addition of API service objects

The use of Aggregated APIs allows adding additional Kubernetes-style API servers to the cluster. The added server acts as a subordinate to **kube-apiserver** which as of v1.7 runs the aggregation layer in-process. When an extension resource is registered the aggregation layer watches a passed URL path and proxy any requests to the newly registered API service.

The aggregation layer is easy to enable. Edit the flags passed during startup of the **kube-apiserver** to include `--enable-aggregator-routing=true`. Some vendors enable this feature by default.

The creation of the exterior can be done via YAML configuration files or APIs. Configuring TLS auth between components and RBAC rules for various new objects is also required. A sample API server is available on github here: <https://github.com/kubernetes/sample-apiserver>. A project currently in incubation stage is an API server builder which should handle much of the security and connection configuration. It can be found here: <https://github.com/kubernetes-incubator/apiserver-builder>

14.4 Labs

Exercise 14.1: Create a Custom Resource Definition

Overview

The use of CustomResourceDefinitions (CRD), has become a common manner to deploy new objects and operators. Creation of a new operator is beyond the scope of this course, basically it is a watch-loop comparing a spec to the current status, and making changes until the states match. A good discussion of creating a operators can be found here: <https://operatorframework.io/>.

First we will examine an existing CRD, then make a simple CRD, but without any particular action. It will be enough to find the object ingested into the API and responding to commands.

1. View the existing CRDs.

```
student@cp:~$ kubectl get crd --all-namespaces
```

NAME	CREATED AT
NAME	CREATED AT
authorizationpolicies.policy.linkerd.io	2023-08-28T11:30:34Z
ciliumcidrgroups.cilium.io	2023-08-28T08:58:54Z
ciliumclusterwidernetworkevents.cilium.io	2023-08-28T08:58:57Z
<output_omitted>	

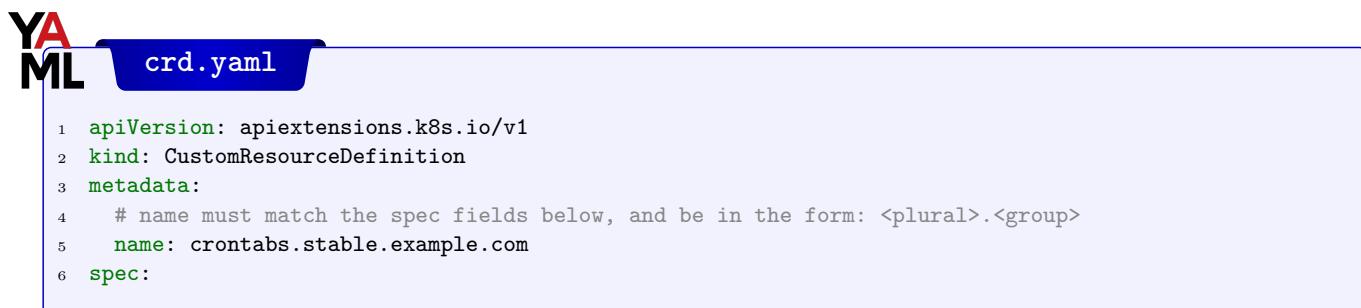
2. We can see from the names that these CRDs are all working on Cilium, our network plugin. View the `cilium-cni.yaml` file we used when we initialized the cluster to see how these objects were created, and some CRD templates to review.

```
student@cp:~$ less cilium-cni.yaml
student@cp:~$ kubectl describe crd ciliumcidrgroups.cilium.io
```

```
<output_omitted>
---
Name: ciliumcidrgroups.cilium.io
Namespace:
Labels: io.cilium.k8s.crd.schema.version=1.26.10
Annotations: <none>
API Version: apiextensions.k8s.io/v1
Kind: CustomResourceDefinition
Metadata:
<output_omitted>
```

3. Now that we have seen some examples, we will create a new YAML file.

```
student@cp:~$ vim crd.yaml
```



```
YAML crd.yaml
1 apiVersion: apiextensions.k8s.io/v1
2 kind: CustomResourceDefinition
3 metadata:
4   # name must match the spec fields below, and be in the form: <plural>.<group>
5   name: crontabs.stable.example.com
6 spec:
```

YAML

```

7  # group name to use for REST API: /apis/<group>/<version>
8  group: stable.example.com
9  # list of versions supported by this CustomResourceDefinition
10 versions:
11   - name: v1
12     # Each version can be enabled/disabled by Served flag.
13     served: true
14     # One and only one version must be marked as the storage version.
15     storage: true
16     schema:
17       openAPIV3Schema:
18         type: object
19         properties:
20           spec:
21             type: object
22             properties:
23               cronSpec:
24                 type: string
25               image:
26                 type: string
27               replicas:
28                 type: integer
29   # either Namespaced or Cluster
30   scope: Namespaced
31   names:
32     # plural name to be used in the URL: /apis/<group>/<version>/<plural>
33     plural: crontabs
34     # singular name to be used as an alias on the CLI and for display
35     singular: crontab
36     # kind is normally the CamelCased singular type. Your resource manifests use this.
37     kind: CronTab
38     # shortNames allow shorter string to match your resource on the CLI
39     shortNames:
40       - ct
41

```

4. Add the new resource to the cluster.

```
student@cp:~$ kubectl create -f crd.yaml
```

```
customresourcedefinition.apiextensions.k8s.io/crontabs.stable.example.com created
```

5. View and describe the resource. The new line may be in the middle of the output. You'll note the **describe** output is unlike other objects we have seen so far.

```
student@cp:~$ kubectl get crd
```

NAME	CREATED AT
<output_omitted>	
crontabs.stable.example.com	2023-08-13T03:18:07Z
<output_omitted>	

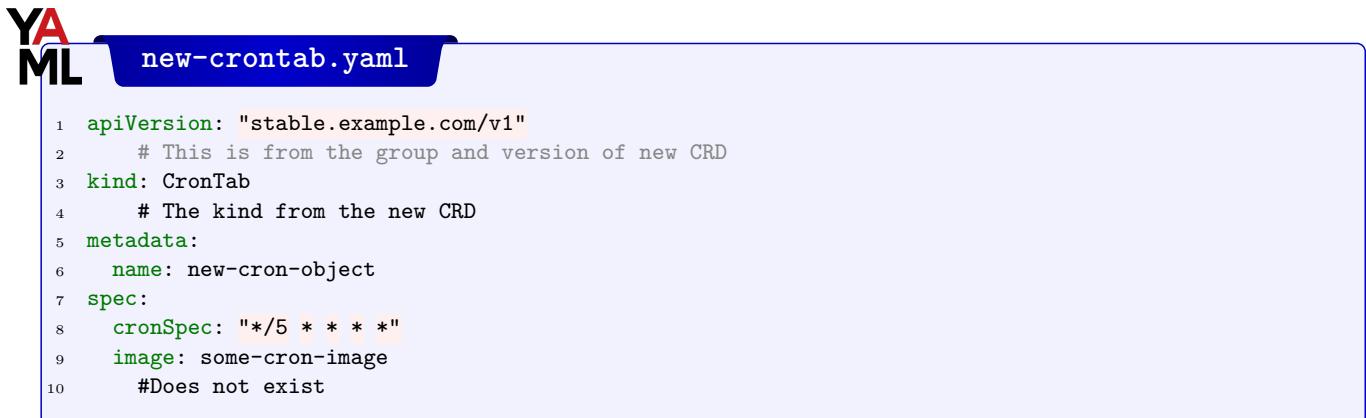
```
student@cp:~$ kubectl describe crd crontab<Tab>
```

Name:	crontabs.stable.example.com
Namespace:	
Labels:	<none>

```
Annotations: <none>
API Version: apiextensions.k8s.io/v1
Kind: CustomResourceDefinition
<output_omitted>
```

6. Now that we have a new API resource we can create a new object of that type. In this case it will be a crontab-like image, which does not actually exist, but is being used for demonstration.

```
student@cp:~$ vim new-crontab.yaml
```



The terminal window shows the creation of a new CronTab object named 'new-cron-object'. The YAML file 'new-crontab.yaml' defines the object with a cronSpec of */5 * * * * and an image of 'some-cron-image'. The command 'kubectl create -f new-crontab.yaml' is run, and the output shows the object was created successfully.

```
YAML new-crontab.yaml
1 apiVersion: "stable.example.com/v1"
2     # This is from the group and version of new CRD
3 kind: CronTab
4     # The kind from the new CRD
5 metadata:
6     name: new-cron-object
7 spec:
8     cronSpec: "*/5 * * * *"
9     image: some-cron-image
10    #Does not exist
```

7. Create the new object and view the resource using short and long name.

```
student@cp:~$ kubectl create -f new-crontab.yaml
```

```
crontab.example.com/new-cron-object created
```

```
student@cp:~$ kubectl get CronTab
```

NAME	AGE
new-cron-object	22s

```
student@cp:~$ kubectl get ct
```

NAME	AGE
new-cron-object	29s

```
student@cp:~$ kubectl describe ct
```

```
Name:      new-cron-object
Namespace: default
Labels:    <none>
Annotations: <none>
API Version: stable.example.com/v1
Kind:      CronTab

<output_omitted>

Spec:
  Cron Spec:  */5 * * * *
  Image:      some-cron-image
  Events:    <none>
```

8. To clean up the resources we will delete the CRD. This should delete all of the endpoints and objects using it as well.

```
student@cp:~$ kubectl delete -f crd.yaml
```

```
customresourcedefinition.apiextensions.k8s.io "crontabs.stable.example.com" deleted
```


Chapter 15

Security



15.1	Overview	328
15.2	Accessing the API	330
15.3	Authentication and Authorization	331
15.4	Admission Controller	335
15.5	Network Policies	337
15.6	Labs	341

15.1 Overview

Overview

- Explain how API requests go through the system.
- Compare different types of authentication.
- Configure authorization rules.
- Configure pod policies to control what containers are allowed to do.
- Restrict network traffic using network policies.

Security is a big and complex topic, especially in a distributed system like Kubernetes. Thus, we are just going to cover some of the concepts that deal with security in the context of Kubernetes. In-depth Cloud and Kubernetes security is covered in detail in the Kubernetes Security Fundamentals (LFS460) course <https://training.linuxfoundation.org/training/kubernetes-security-fundamentals-lfs460/>

Then we are going to focus on the authentication aspect of the API server and will dive into authorization, looking at things like RBAC, which is now the default configuration when you bootstrap a Kubernetes cluster with **kubeadm**.

We are going to look at the admission control system, which lets you look at and possibly modify the requests that are coming in, and do a final deny or accept those requests.

Following that we're going to look at a few other concepts, including how you can secure your Pods more tightly using security contexts and pod security policies, which are full-fledged API objects in Kubernetes.

Finally, we will look at network policies. By default, we tend to not turn on network policies, which lets any traffic flow through all our pods, in all the different namespaces. Using network policies, we can actually define Ingress rules so that we can restrict the Ingress traffic between the different namespaces. The network tool in use, such as **flannel** or **Calico** will determine if a network policy can be implemented. As Kubernetes becomes more mature, this will become a strongly suggested configuration.

Cloud Security Considerations

- Ongoing cycle of security
- Image and binary supply chain
- Hardening the operating system
- Securing the kube-apiserver
- Network considerations
- Static and runtime workload analysis
- Issue detection

Keeping a cloud environment secure is an ongoing, and wide ranging task. As more moves to the cloud we must look at more than just Kubernetes towards the hardware, software, and configuration options for the entire environment. Starting in the design phase care must be taken to secure safe hardware, firmware and operating system binaries.

Once the platform is hardened the kube-apiserver has a list of considerations, tools, and settings to limit access and formalized access in an easy to understand manner.

As a network intensive environment it becomes important to secure the network both inside Kubernetes as done with a `NetworkPolicy` as well as traditional firewall tools and pod to pod encryption.

Minimizing base images, insisting on container immutability, and static and runtime analysis of tools is also an important part of security which often begins with developers and is implemented in the CI/CD pipeline prior to an image being used in a production cluster. Tools like AppArmor and SELinux should also be used to further protect the environment from malicious containers.

Security is more than just a settings and configuration. It is an ongoing process of issue detection using intrusion detection tools and behavioral analytics. There needs to be an ongoing process of assessment, prevention, detection, and reaction following written and often updated policies.

15.2 Accessing the API

Accessing the API

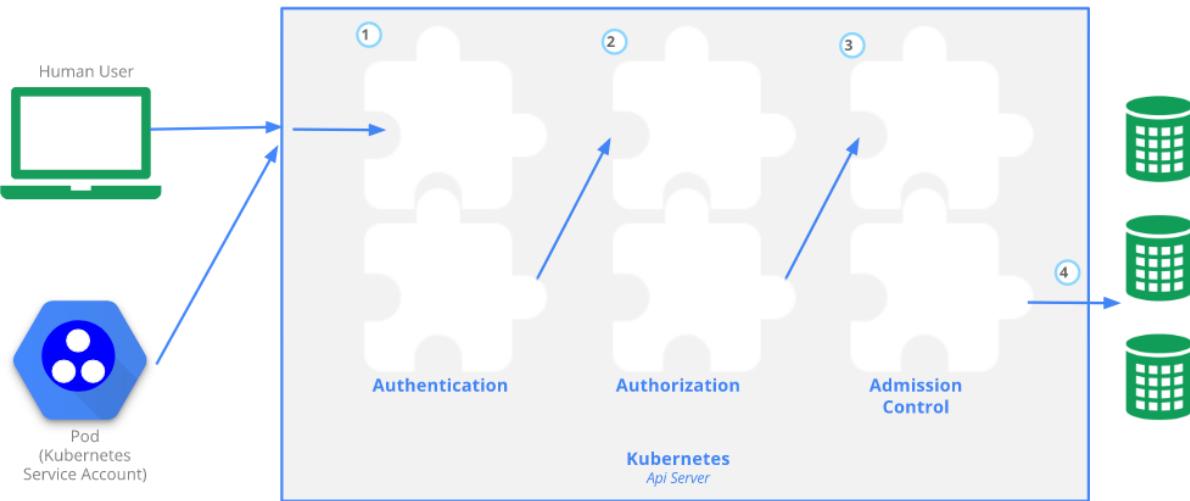


Figure 15.1: Accessing the API from [kubernetes.io](https://kubernetes.io/docs/admin/accessing-the-api/)

To perform any action in a Kubernetes cluster, you need to access the API and go through three main steps:

- Authentication (token)
- Authorization (RBAC)
- Admission Controllers

These steps are described in more detail in the official documentation about controlling access to the API at <https://kubernetes.io/docs/admin/accessing-the-api/>, and illustrated by the picture.

Once a request reaches the API server securely, it will first go through any authentication module that has been configured. The request can be rejected if authentication fails or it gets authenticated and passed to the authorization step.

At the authorization step, the request will be checked against existing policies. It will be authorized if the user has the permissions to perform the requested actions. Then, the requests will go through the last step of admission. In general, admission controllers will check the actual content of the objects being created and validate them before admitting the request.

In addition to these steps, the requests reaching the API server over the network are encrypted using TLS. This needs to be properly configured using SSL certificates. If you use **kubeadm** this configuration is done for you; otherwise, follow this guide: <https://github.com/kelseyhightower/kubernetes-the-hard-way> or the API server configuration options: <https://kubernetes.io/docs/admin/kube-apiserver/>.

15.3 Authentication and Authorization

Authentication

- One or more **Authenticator Modules** used
 - x509 Client Certs
 - Static Token, Bearer or Bootstrap Token
 - Static Password File
 - Service Account and OpenID Connect Tokens
- Each tried until success, order not guaranteed
- Anonymous access can be enabled. Otherwise 401 response.
- Users are not created by the API, should be managed by an external system.
- Use of proxy or webhook to LDAP, SAML, Kerberos, alternate x509 etc. instead.
- System accounts are used by processes to access the API

There are three main points to remember with authentication in Kubernetes. In its straightforward form, authentication is done with certificates, tokens or basic authentication (i.e. username and password). Users are not created by the API, but should be managed by an external system. System accounts are used by processes to access the API: <https://kubernetes.io/docs/tasks/configure-pod-container/configure-service-account/>.

There are two more advanced authentication mechanisms: **Webhooks** which can be used to verify bearer tokens; and connection with an external **OpenID** provider.

The type of authentication used is defined in the **kube-apiserver** startup options. Below are four examples of a subset of configuration options that would need to be set depending on what choice of authentication mechanism you choose. For example:

- --basic-auth-file
- --oidc-issuer-url
- --token-auth-file
- --authorization-webhook-config-file

To learn more about authentication, see the official documentation at: <https://kubernetes.io/docs/admin/authentication/>.

Authorization

- **RBAC**
- **WebHook**

Once a request is authenticated, it needs to be authorized to be able to proceed through the Kubernetes system and perform its intended action.

There are three main authorization modes and two global Deny/Allow settings.

They can be configured as **kube-apiserver** startup options:

- --authorization-mode=RBAC
- --authorization-mode=Webhook
- --authorization-mode=AlwaysDeny
- --authorization-mode=AlwaysAllow

The authorization modes implement policies to allow requests. Attributes of the requests are checked against the policies (e.g. user, group, namespace, verb).

RBAC

- Resources and Operations (verbs)
- Rules
- Roles and ClusterRoles
- Subjects
- RoleBindings and ClusterRoleBindings

RBAC stands for **R**ole **B**ased **A**ccess **C**ontrol: <https://kubernetes.io/docs/admin/authorization/rbac>.

All resources are modeled API objects in Kubernetes from Pods to Namespaces. They also belong to API Groups such as core and apps. These resources allow operations such as Create, Read, Update, and Delete (CRUD) which are called operations, which we have been working with so far. Operations are called verbs inside YAML files. Adding to these basic components we will add more elements of the API, which can then be managed via RBAC.

Rules are operations which can act upon an API group. **Roles** are a group of rules which affect, or scope, a single namespace, whereas **ClusterRoles** have a scope of the entire cluster.

Each operation can act upon one of three **subjects** which are User Accounts which don't exist as API objects, Service Accounts, and Groups which are known as `clusterrolebinding` when using `kubectl`.

RBAC is then writing rules to allow or deny operations by users, roles or groups upon resources.

RBAC Process Overview

- Determine or create namespace
- Create certificate credentials for user
- Set the credentials for the user to the namespace using a **context**
- Create a role for the expected task set
- Bind the user to the role
- Verify the user has limited access

While **RBAC** can be complex the basic flow is to create a certificate for a user. As a user is not a API object of Kubernetes, requiring outside authentication such as OpenSSL certificates. After generation of the certificate against the cluster certificate authority we can set that credential for the user using a **context**.

Roles can then be used to configure an association of `apiGroups`, `resources`, and the `verbs` allowed to them. The user can then be bound to a `role` limiting what and where they can work the the cluster.

15.4 Admission Controller

Admission Controller

- Access content of objects created
- Modify or validate content

The last step in letting an API request into Kubernetes is **Admission Control**.

Admission controllers are pieces of software that can access the content of the objects being created by the requests. They can modify the content or validate it, and potentially deny the request.

Admission controllers are needed for certain features to work properly. Controllers have been added as Kubernetes has matured. As of the v1.12 release the **kube-apiserver** uses a compiled in set of controllers. Instead of passing a list we can enable or disable particular controllers. If you want to use a controller not available by default you would need to download source and compile.

The first controller is **Initializers** which will allow dynamic modification of the API request, providing great flexibility. Each admission controller functionality is explained in the documentation. For example, the **ResourceQuota** controller will ensure that the object created does not violate any of the existing quotas.

Security Contexts

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  securityContext:
    runAsNonRoot: true
  containers:
    - image: nginx
      name: nginx
```

Pods and containers within Pods can be given specific security constraints to limit what processes running in containers can do. For example, the UID of the process, the **Linux** capabilities, and the file system group can be limited.

This security limitation is called a **security context**. It can be defined for the entire Pod or per container and is represented as additional sections in the resources manifests. The notable difference is that **Linux** capabilities are set at the container level.

For example, if you want to enforce a policy that containers cannot run their process as the **root** user, you can add a Pod security context like the one above.

Then, when you create this pod, you will see a warning that the container is trying to run as **root** and that it is not allowed. Hence, the Pod will never run:

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
nginx	0/1	container has runAsNonRoot and image will run as root	0	10s

Read more about security contexts to give proper constraints to your containers at:
<https://kubernetes.io/docs/tasks/configure-pod-container/security-context/>.

15.5 Network Policies

Network Security Policies

- All traffic allowed by default
- Networking add-ons must support network policies
- Can be limited to a namespace
- Egress policy type now available
- Can match PodSelector, ingress labels and egress labels.

By default all Pods can reach each other; all ingress and egress traffic is allowed. This has been a high-level networking requirement in Kubernetes. However, network isolation can be configured and traffic to pods can be blocked. In newer versions of Kubernetes egress traffic can also be blocked. This is done by configuration of a `NetworkPolicy`. As all traffic is allowed you may want to implement a policy that drops all traffic, then other policies which allow desired ingress and egress traffic.

The spec of the policy can narrow down effect to a particular namespace, which can be handy. Further settings include a `podSelector`, or label, to narrow down which Pods are affected. Further `ingress` and `egress` settings declare `to` and `from` IP addresses and ports.

Not all network providers support the `NetworkPolicies` kind. A non-exhaustive list of providers with support includes **Calico**, **Romana**, **Cilium**, **Kube-router**, and **WeaveNet**.

In previous versions of Kubernetes there was a requirement to annotate a namespace as part of network isolation, specifically the `net.beta.kubernetes.io/network-policy=` value. Some network plugins may still require this setting.

Following is an example of a `NetworkPolicy` recipe. More recipes can be found here: <https://github.com/ahmetb/kubernetes-network-policy-recipes>

Network Security Policy Example

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: ingress-egress-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
    - Ingress
    - Egress
  ingress:
    - from:
        - ipBlock:
            cidr: 172.17.0.0/16
            except:
              - 172.17.1.0/24
<continued_on_next_slide>
```

The use of policies has become stable, noted with the v1 apiVersion. The example above narrows down the policy to affect the default namespace.

Only Pods with the label of role db: will be affected by this policy, and has both Ingress and Egress settings.

The ingress setting includes a 172.17 network, with a smaller range of 172.17.1.0 IPs being excluded from this traffic.

Network Security Policy Example Cont.

```

- namespaceSelector:
  matchLabels:
    project: myproject
- podSelector:
  matchLabels:
    role: frontend
  ports:
  - protocol: TCP
    port: 6379
  egress:
  - to:
    - ipBlock:
      cidr: 10.0.0.0/24
  ports:
  - protocol: TCP
    port: 5978

```

These rules change the namespace for the following settings to be labeled `project myproject`. The affected Pods also would need to match the label `role frontend`. Finally TCP traffic on port 6379 would be allowed from these Pods.

The egress rules have to settings, in this case the 10.0.0.0/24 range TCP traffic to port 5978.

The use of empty ingress or egress rules denies all type of traffic for the included Pods, though not suggested. Use another dedicated NetworkPolicy instead.

There can also be complex `matchExpressions` statements in the spec, but this may change as NetworkPolicy matures.

```

podSelector:
  matchExpressions:
  - {key: inns, operator: In, values: ["yes"]}

```

Default Policy Example

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny
spec:
  podSelector: {}
  policyTypes:
    - Ingress
```

The empty braces will match all Pods not selected by other NetworkPolicy will not allow ingress traffic. Egress traffic would be unaffected by this policy.

With the potential for complex ingress and egress rules it may be helpful to create multiple objects which include simple isolation rules and use easy to understand names and labels.

Some network plugins, such as **WeaveNet** may require annotation of the Namespace. The following shows the setting of a DefaultDeny for the myns namespace:

```
kind: Namespace
apiVersion: v1
metadata:
  name: myns
  annotations:
    net.beta.kubernetes.io/network-policy: |
      {
        "ingress": {
          "isolation": "DefaultDeny"
        }
      }
```

15.6 Labs

Exercise 15.1: Working with TLS

Overview

We have learned that the flow of access to a cluster begins with TLS connectivity, then authentication followed by authorization, finally an admission control plug-in allows advanced features prior to the request being fulfilled. The use of Initializers allows the flexibility of a shell-script to dynamically modify the request. As security is an important, ongoing concern, there may be multiple configurations used depending on the needs of the cluster.

Every process making API requests to the cluster must authenticate or be treated as an anonymous user.

While one can have multiple cluster root Certificate Authorities (CA) by default each cluster uses their own, intended for intra-cluster communication. The CA certificate bundle is distributed to each node and as a secret to default service accounts. The **kubelet** is a local agent which ensures local containers are running and healthy.

- View the **kubelet** on both the cp and secondary nodes. The **kube-apiserver** also shows security information such as certificates and authorization mode. As **kubelet** is a **systemd** service we will start looking at that output.

```
student@cp:~$ systemctl status kubelet.service
```

```
kubelet.service - kubelet: The Kubernetes Node Agent
   Loaded: loaded (/lib/systemd/system/kubelet.service; enabled; vendor preset: en
   Drop-In: /etc/systemd/system/kubelet.service.d
             |_ 10-kubeadm.conf
<output omitted>
```

- Look at the status output. Follow the CGroup and kubelet information, which is a long line where configuration settings are drawn from, to find where the configuration file can be found.

```
CGroup: /system.slice/kubelet.service
--19523 /usr/bin/kubelet .... --config=/var/lib/kubelet/config.yaml ..
```

- Take a look at the settings in the `/var/lib/kubelet/config.yaml` file. Among other information we can see the `/etc/kubernetes/pki/` directory is used for accessing the **kube-apiserver**. Near the end of the output it also sets the directory to find other pod spec files.

```
student@cp:~$ sudo less /var/lib/kubelet/config.yaml
```



config.yaml

```
1 <output omitted>
2 rotateCertificates: true
3 runtimeRequestTimeout: 0s
4 shutdownGracePeriod: 0s
5 shutdownGracePeriodCriticalPods: 0s
6 staticPodPath: /etc/kubernetes/manifests
7 streamingConnectionIdleTimeout: 0s
8 syncFrequency: 0s
9 volumeStatsAggPeriod: 0s
10
```

- Other agents on the cp node interact with the **kube-apiserver**. View the configuration files where these settings are made. This was set in the previous YAML file. Look at one of the files for cert information.

```
student@cp:~$ sudo ls /etc/kubernetes/manifests/
```

etcd.yaml	kube-controller-manager.yaml
kube-apiserver.yaml	kube-scheduler.yaml

```
student@cp:~$ sudo less /etc/kubernetes/manifests/kube-controller-manager.yaml
```

<output_omitted>

5. The use of **tokens** has become central to authorizing component communication. The tokens are kept as **secrets**. Take a look at the current secrets in the `kube-system` namespace. Note: Token is valid only for 24 hours and then the secret is removed, If you dont see the secret please eexecute, `sudo kubeadm token create`

```
student@cp:~$ kubectl -n kube-system get secrets
```

NAME	TYPE
DATA AGE	
bootstrap-token-i3r13t	bootstrap.kubernetes.io/token
7 5d	
<output_omitted>	

6. Take a closer look at one of the secrets and the token within. The `bootstrap-token` could be one to look at. The use of the Tab key can help with long names. Long lines have been truncated in the output below.

```
student@cp:~$ kubectl -n kube-system get secrets bootstrap-token<Tab> -o yaml
```

YAML

```
1 apiVersion: v1
2 data:
3   auth-extra-groups: c3lzdGVtOmJvb3RzdHJhcHBlcM6a3ViZWFKbTpkZWZhdWx0LW5vZGUtdG9rZW4=
4   expiration: MjAyMy0wNi0wM1QzMjowOTowMlo=
5   token-id: NXBvMGo3
6   token-secret: MXhzMDgx0G1rcTFyeDQxbg==
7   usage-bootstrap-authentication: dHJ1ZQ==
8   usage-bootstrap-signing: dHJ1ZQ==
9 kind: Secret
10 metadata:
11   creationTimestamp: "2023-08-01T12:09:02Z"
12   name: bootstrap-token-5po0j7
13   namespace: kube-system
14   resourceVersion: "209"
15   uid: 98219199-4876-4cfa-a4be-586cda27cc6b
16   type: bootstrap.kubernetes.io/token
17
```

7. The **kubectl config** command can also be used to view and update parameters. When making updates this could avoid a typo removing access to the cluster. View the current configuration settings. The keys and certs are redacted from the output automatically.

```
student@cp:~$ kubectl config view
```

apiVersion: v1
clusters:
- cluster:
certificate-authority-data: REDACTED
<output_omitted>

8. View the options, such as setting a password for the admin instead of a key. Read through the examples and options.

```
student@cp:~$ kubectl config set-credentials -h
```

```
Sets a user entry in kubeconfig  
<output_omitted>
```

9. Make a copy of your access configuration file. Later steps will update this file and we can view the differences.

```
student@cp:~$ cp $HOME/.kube/config $HOME/cluster-api-config
```

10. Explore working with cluster and security configurations both using **kubectl** and **kubeadm**. Among other values, find the name of your cluster. You will need to become **root** to work with **kubeadm**.

```
student@cp:~$ kubectl config <Tab><Tab>
```

```
current-context      get-contexts      set-context      view
delete-cluster      rename-context   set-credentials
delete-context      set              unset
get-clusters        set-cluster     use-context
```

```
student@cp:~$ sudo kubeadm token -h
```

```
<output_omitted>
```

```
student@cp:~$ sudo kubeadm config -h
```

```
<output_omitted>
```

11. Review the cluster default configuration settings. There may be some interesting tidbits to the security and infrastructure of the cluster.

```
student@cp:~$ sudo kubeadm config print init-defaults
```

```
apiVersion: kubeadm.k8s.io/v1beta2
bootstrapTokens:
- groups:
  - system:bootstrappers:kubeadm:default-node-token
    token: abcdef.0123456789abcdef
    ttl: 24h0m0s
    usages:
<output_omitted>
```

Exercise 15.2: Authentication and Authorization

Kubernetes clusters have two types of users service accounts and normal users, but normal users are assumed to be managed by an outside service. There are no objects to represent them and they cannot be added via an API call, but service accounts can be added.

We will use **RBAC** to configure access to actions within a namespace for a new contractor, Developer Dan who will be working on a new project.

- Create two namespaces, one for production and the other for development.

```
student@cp:~$ kubectl create ns development
```

```
namespace/development created
```

```
student@cp:~$ kubectl create ns production
```

```
namespace/production created
```

2. View the current clusters and context available. The context allows you to configure the cluster to use, namespace and user for **kubectl** commands in an easy and consistent manner.

```
student@cp:~$ kubectl config get-contexts
```

CURRENT	NAME	CLUSTER	AUTHINFO	NAMESPACE
*	kubernetes-admin@kubernetes	kubernetes	kubernetes-admin	

3. Create a new user DevDan and assign a password of lftr@in.

```
student@cp:~$ sudo useradd -s /bin/bash DevDan
```

```
student@cp:~$ sudo passwd DevDan
```

```
Enter new UNIX password: lftr@in
Retype new UNIX password: lftr@in
passwd: password updated successfully
```

4. Generate a private key then Certificate Signing Request (CSR) for DevDan. On some Ubuntu 18.04 nodes a missing file may cause an error with random number generation. The **touch** command should ensure one way of success.

```
student@cp:~$ openssl genrsa -out DevDan.key 2048
```

```
Generating RSA private key, 2048 bit long modulus
.....+++
.....+++
e is 65537 (0x10001)
```

```
student@cp:~$ touch $HOME/.rnd
```

```
student@cp:~$ openssl req -new -key DevDan.key \
-out DevDan.csr -subj "/CN=DevDan/O=development"
```

5. Using the newly created request generate a self-signed certificate using the x509 protocol. Use the CA keys for the Kubernetes cluster and set a 45 day expiration. You'll need to use **sudo** to access to the inbound files.

```
student@cp:~$ sudo openssl x509 -req -in DevDan.csr \
-CA /etc/kubernetes/pki/ca.crt \
-CAkey /etc/kubernetes/pki/ca.key \
-CAcreateserial \
-out DevDan.crt -days 45
```

```
Signature ok
subject=/CN=DevDan/O=development
Getting CA Private Key
```

6. Update the access config file to reference the new key and certificate. Normally we would move them to a safe directory instead of a non-root user's home.

```
student@cp:~$ kubectl config set-credentials DevDan \
--client-certificate=/home/student/DevDan.crt \
--client-key=/home/student/DevDan.key
```

User "DevDan" set.

- View the update to your credentials file. Use **diff** to compare against the copy we made earlier.

```
student@cp:~$ diff cluster-api-config .kube/config
```

```
16a,19d15
> - name: DevDan
>   user:
>     as-user-extra: {}
>     client-certificate: /home/student/DevDan.crt
>     client-key: /home/student/DevDan.key
```

- We will now create a context. For this we will need the name of the cluster, namespace and CN of the user we set or saw in previous steps.

```
student@cp:~$ kubectl config set-context DevDan-context \
--cluster=kubernetes \
--namespace=development \
--user=DevDan
```

Context "DevDan-context" created.

- Attempt to view the Pods inside the DevDan-context. Be aware you will get an error.

```
student@cp:~$ kubectl --context=DevDan-context get pods
```

```
Error from server (Forbidden): pods is forbidden: User "DevDan"
cannot list pods in the namespace "development"
```

- Verify the context has been properly set.

```
student@cp:~$ kubectl config get-contexts
```

CURRENT	NAME	CLUSTER	AUTHINFO	NAMESPACE
	DevDan-context	kubernetes	DevDan	development
*	kubernetes-admin@kubernetes	kubernetes	kubernetes-admin	

- Again check the recent changes to the cluster access config file.

```
student@cp:~$ diff cluster-api-config .kube/config
```

<output_omitted>

- We will now create a YAML file to associate RBAC rights to a particular namespace and Role.

```
student@cp:~$ vim role-dev.yaml
```



role-dev.yaml

```

1 kind: Role
2 apiVersion: rbac.authorization.k8s.io/v1
3 metadata:
4   namespace: development
5   name: developer
6 rules:
7 - apiGroups: [ "", "extensions", "apps" ]
8   resources: [ "deployments", "replicasets", "pods" ]
9   verbs: [ "list", "get", "watch", "create", "update", "patch", "delete" ]
10 # You can use ["*"] for all verbs
11

```

13. Create the object. Check white space and for typos if you encounter errors.

```
student@cp:~$ kubectl create -f role-dev.yaml
```

```
role.rbac.authorization.k8s.io/developer created
```

14. Now we create a RoleBinding to associate the Role we just created with a user. Create the object when the file has been created.

```
student@cp:~$ vim rolebind.yaml
```



rolebind.yaml

```

1 kind: RoleBinding
2 apiVersion: rbac.authorization.k8s.io/v1
3 metadata:
4   name: developer-role-binding
5   namespace: development
6 subjects:
7 - kind: User
8   name: DevDan
9   apiGroup: ""
10 roleRef:
11   kind: Role
12   name: developer
13   apiGroup: ""

```

```
student@cp:~$ kubectl create -f rolebind.yaml
```

```
rolebinding.rbac.authorization.k8s.io/developer-role-binding created
```

15. Test the context again. This time it should work. There are no Pods running so you should get a response of No resources found.

```
student@cp:~$ kubectl --context=DevDan-context get pods
```

```
No resources found in development namespace.
```

16. Create a new pod, verify it exists, then delete it.

```
student@cp:~$ kubectl --context=DevDan-context \
    create deployment nginx --image=nginx
```

deployment.apps/nginx created

```
student@cp:~$ kubectl --context=DevDan-context get pods
```

NAME	READY	STATUS	RESTARTS	AGE
nginx-7c87f569d-7gb9k	1/1	Running	0	5s

```
student@cp:~$ kubectl --context=DevDan-context delete \
    deploy nginx
```

deployment.apps "nginx" deleted

17. We will now create a different context for production systems. The Role will only have the ability to view, but not create or delete resources. Begin by copying and editing the Role and RoleBindings YAML files.

```
student@cp:~$ cp role-dev.yaml role-prod.yaml
```

```
student@cp:~$ vim role-prod.yaml
```

Y
A
M
L

role-prod.yaml

```
1 kind: Role
2 apiVersion: rbac.authorization.k8s.io/v1
3 metadata:
4   namespace: production      #<-- This line
5   name: dev-prod            #<-- and this line
6 rules:
7 - apiGroups: [ "", "extensions", "apps"]
8   resources: [ "deployments", "replicasets", "pods" ]
9   verbs: [ "get", "list", "watch" ] #<-- and this one
10
```

```
student@cp:~$ cp rolebind.yaml rolebindprod.yaml
```

```
student@cp:~$ vim rolebindprod.yaml
```

Y
A
M
L

rolebindprod.yaml

```
1 kind: RoleBinding
2 apiVersion: rbac.authorization.k8s.io/v1
3 metadata:
4   name: production-role-binding #<-- Edit to production
5   namespace: production        #<-- Also here
6 subjects:
7 - kind: User
8   name: DevDan
9   apiGroup: ""
10 roleRef:
11   kind: Role
12   name: dev-prod             #<-- Also this
13   apiGroup: ""
```

18. Create both new objects.

```
student@cp:~$ kubectl create -f role-prod.yaml
role.rbac.authorization.k8s.io/dev-prod created
```

```
student@cp:~$ kubectl create -f rolebindprod.yaml
rolebinding.rbac.authorization.k8s.io/production-role-binding created
```

19. Create the new context for production use.

```
student@cp:~$ kubectl config set-context ProdDan-context \
--cluster=kubernetes \
--namespace=production \
--user=DevDan
```

```
Context "ProdDan-context" created.
```

20. Verify that user DevDan can view pods using the new context.

```
student@cp:~$ kubectl --context=ProdDan-context get pods
No resources found in production namespace.
```

21. Try to create a Pod in production. The developer should be Forbidden.

```
student@cp:~$ kubectl --context=ProdDan-context create \
deployment nginx --image=nginx
Error from server (Forbidden): deployments.apps is forbidden:
User "DevDan" cannot create deployments.apps in the
namespace "production"
```

22. View the details of a role.

```
student@cp:~$ kubectl -n production describe role dev-prod
Name:           dev-prod
Labels:         <none>
Annotations:   kubectl.kubernetes.io/last-applied-configuration=
               {"apiVersion":"rbac.authorization.k8s.io/v1","kind":"Role"
               , "metadata": {"annotations": {}, "name": "dev-prod", "namespace":
               "production"}, "rules": [{"api...
PolicyRule:
  Resources      Non-Resource URLs  Resource Names  Verbs
  -----          -----          -----          -----
  deployments    []                []            [get list watch]
  deployments.apps    []                []            [get list watch]
<output_omitted>
```

23. Experiment with other subcommands in both contexts. They should match those listed in the respective roles.

- 24. OPTIONAL CHALLENGE STEP:** Become the DevDan user. Solve any missing configuration errors. Try to create a deployment in the development and the production namespaces. Do the errors look the same? Configure as necessary to only have two contexts available to DevDan.

```
DevDan@cp:~$ kubectl config get-contexts
```

CURRENT	NAME	CLUSTER	AUTHINFO	NAMESPACE
*	DevDan-context	kubernetes	DevDan	development
	ProdDan-context	kubernetes	DevDan	production

📝 Exercise 15.3: Admission Controllers

The last stop before a request is sent to the API server is an admission control plug-in. They interact with features such as setting parameters like a default storage class, checking resource quotas, or security settings. A newer feature (v1.7.x) is dynamic controllers which allow new controllers to be ingested or configured at runtime.

- View the current admission controller settings. Unlike earlier versions of Kubernetes the controllers are now compiled into the server, instead of being passed at run-time. Instead of a list of which controllers to use we can enable and disable specific plugins.

```
student@cp:~$ sudo grep admission \
/etc/kubernetes/manifests/kube-apiserver.yaml
- --enable-admission-plugins=NodeRestriction
```


Chapter 16

High Availability



16.1	Overview	352
16.2	Stacked Database	353
16.3	External Database	354
16.4	Labs	355

16.1 Overview

Cluster High Availability

- Uses load balancer
- Three nodes for database quorum
- Stacked control planes and **etcd** nodes
- External **etcd** nodes
- Use FQDN for SSL connections

A newer feature of **kubeadm** is the integrated ability to join multiple cp nodes with co-located **etcd** databases. This allows for higher redundancy and fault tolerance. As long as the database services the cluster will continue to run and catch up with **kubelet** information should the cp node go down and be brought back online.

Three instances are required for **etcd** to be able to determine quorum (50% if the data is accurate or corrupt the database could become unavailable. Once **etcd** is able to determine quorum it will elect a leader and return to functioning as it had before failure.

One can either co-locate the database with control planes or use an external **etcd** database cluster. The **kubeadm** command makes the co-located deployment easier to use.

To ensure that workers and other control planes continue to have access it is a good idea to use a load balancer. The default configuration leverages SSL, so you may need to configure the load balancer as a TCP pass through unless you want the extra work of certificate configuration. As the certificates will be decoded only for particular node names it is a good idea to use a FQDN instead of an ip address, although there are many possible ways to handle access.

16.2 Stacked Database

Collocated Databases

- Default database location
- Simpler to deploy and manager
- Requires at least three nodes
- Instance failure would remove essential services and database

The easiest way to gain higher availability is to use the **kubeadm** command and join at least two more cp servers to the cluster. The command is almost the same as a worker join except an additional --control-plane flag and a certificate-key. The key will probably need to be generated unless the other cp nodes are added within two hours of the cluster initialization.

Should a node fail you would lose both a control plane and a database. As the database is the one object that cannot be rebuilt this may not be an important issue.

16.3 External Database

Non-collocated Databases

- Configure HA **etcd** cluster
- Manually copy PKI certificates
- Configure first control plane
- Add more control planes
- Add worker nodes

Using an external cluster of **etcd** allows for less interruption should a node fail. Creating a cluster in this manner requires a lot more equipment to properly spread out services and takes more work to configure.

The external **etcd** cluster needs to be configured first. The **kubeadm** command has options to configure this cluster, or other options are available. Once the **etcd** cluster is running the certificates need to be manually copied to the intended first control plane node.

The `kubeadm-config.yaml` file needs to be populated with the **etcd** set to `external`, `endpoints`, and the certificate locations.

Once the first control plane is fully initialized the redundant control planes need to be added one at a time, each fully initialized before the next is added.

16.4 Labs

Exercise 16.1: High Availability Steps

Overview

In this lab we will add two more control planes to our cluster, change taints and deploy an application to a particular node, and test that we can access it from outside the cluster. The nodes will handle various infrastructure services and the **etcd** database and should be sized accordingly.

The steps are presented in two ways. First the general steps for those interested in more of a challenge. Following that will be the detailed steps found in previous labs.

You will need three more nodes. One to act as a load balancer, the other two will act as cp nodes for quorum. Log into each and use the **ip** command to fill in the table with the IP addresses of the primary interface of each node. If using **GCE** nodes it would be **ens4**, yours may be different. You may need to install software such an editor on the nodes.

Proxy Node	
Second Control Plane	
Third Control Plane	

As the prompts may look similar you may want to change the terminal color or other characteristics to make it easier to keep them distinct. You can also change the prompt using something like: **PS1="ha-proxy\$ "**, which may help to keep the terminals distinct.

High level steps:

1. Deploy a load balancer configured to pass through traffic on your new proxy node. HAProxy is easy to deploy using online documentation. Start with forwarding traffic of the cp alias to just the working cp.
2. Install the Kubernetes software on the second and third cp nodes.
3. Use **kubeadm join** on the second cp, adding it to the cluster as another control plane using the node name.
4. Join the third cp as another control plane to the cluster using the node name.
5. Update the proxy to use all three cps backend IPs.
6. Temporarily shut down the first cp and monitor traffic.

Exercise 16.2: Detailed Steps

Deploy a Load Balancer

While there are many options, both software and hardware, we will be using an open source tool **HAProxy** to configure a load balancer.

1. Deploy HAProxy. Log into the proxy node. Update the repos then install a the HAProxy software. Answer yes, should you the installation ask if you will allow services to restart.

```
student@ha-proxy:~$ sudo apt-get update ; sudo apt-get install -y haproxy vim
```

```
<output_omitted>
```

2. Edit the configuration file and add sections for the front-end and back-end servers. We will comment out the second and third cp node until we are sure the proxy is forwarding traffic to the known working cp.

```
student@ha-proxy:~$ sudo vim /etc/haproxy/haproxy.cfg

.....
defaults
    log global          #<-- Edit these three lines, starting around line 23
    option tcplog
    mode tcp
.....
    errorfile 503 /etc/haproxy/errors/503.http
    errorfile 504 /etc/haproxy/errors/504.http

frontend proxynode          #<-- Add the following lines to bottom of file
    bind *:80
    bind *:6443
    stats uri /proxystats
    default_backend k8sServers

backend k8sServers
    balance roundrobin
    server cp 10.128.0.24:6443 check #<-- Edit these with your IP addresses, port, and hostname
#    server secondcp 10.128.0.30:6443 check #<-- Comment out until ready
#    server thirdcp 10.128.0.66:6443 check #<-- Comment out until ready
listen stats
    bind :9999
    mode http
    stats enable
    stats hide-version
    stats uri /stats
```

3. Restart the haproxy service and check the status. You should see the frontend and backend proxies report being started.

```
student@ha-proxy:~$ sudo systemctl restart haproxy.service
student@ha-proxy:~$ sudo systemctl status haproxy.service
```

```
<output_omitted>
Aug 08 18:43:08 ha-proxy systemd[1]: Starting HAProxy Load Balancer...
Aug 08 18:43:08 ha-proxy systemd[1]: Started HAProxy Load Balancer.
Aug 08 18:43:08 ha-proxy haproxy-systemd-wrapper[13602]: haproxy-systemd-wrapper:
Aug 08 18:43:08 ha-proxy haproxy[13603]: Proxy proxynode started.
Aug 08 18:43:08 ha-proxy haproxy[13603]: Proxy proxynode started.
Aug 08 18:43:08 ha-proxy haproxy[13603]: Proxy k8sServers started.
Aug 08 18:43:08 ha-proxy haproxy[13603]: Proxy k8sServers started.
```

4. **On the cp** Edit the `/etc/hosts` file and comment out the old and add a new k8scp alias to the IP address of the proxy server.

```
student@cp:~$ sudo vim /etc/hosts
```

```
10.128.0.64 k8scp      #<-- Add alias to proxy IP
#10.128.0.24 k8scp    #<-- Comment out the old alias, in case its needed
127.0.0.1 localhost
....
```

5. Use a local browser to navigate to the public IP of your proxy server. The `http://34.69.XX.YY:9999/stats` is an example your IP address would be different. Leave the browser up and refresh as you run following steps. You can find your public ip using `curl`. Your IP will be different than the one shown below.

```
ha-proxy$ curl ifconfig.io
```

34.69.73.159

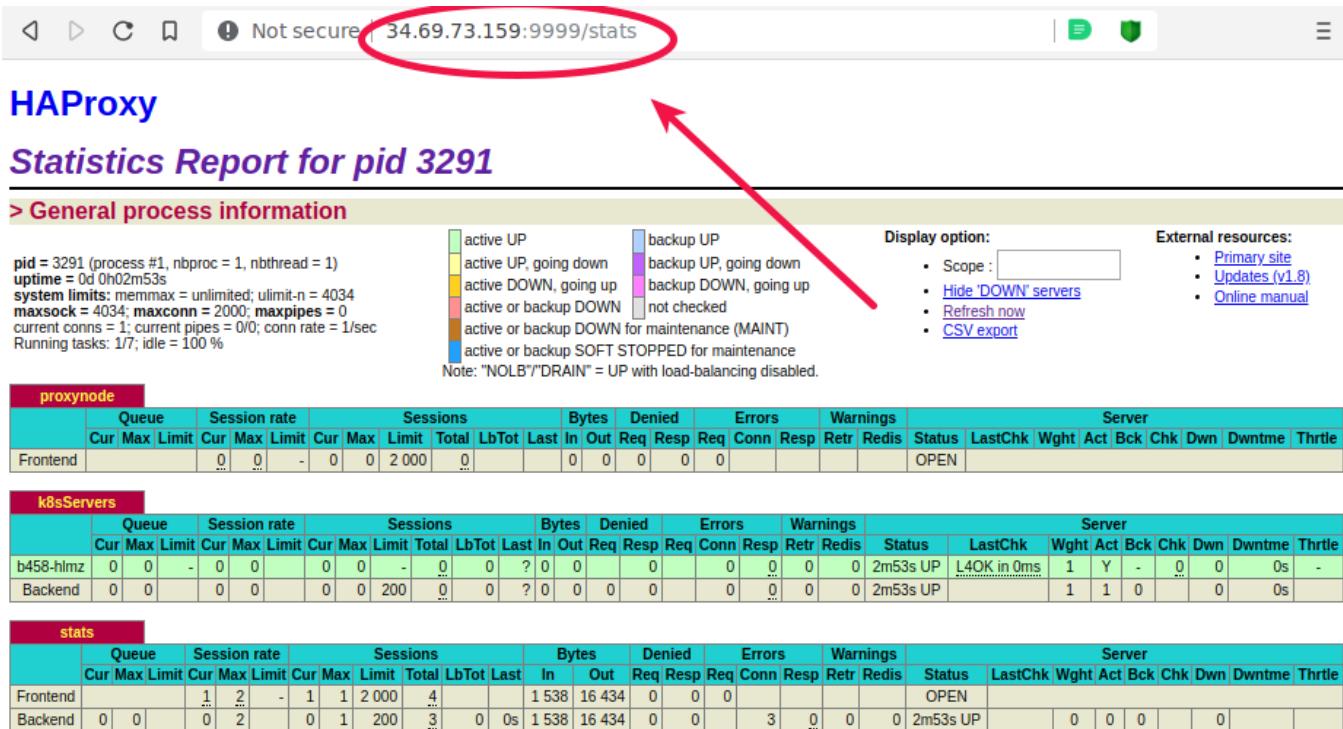


Figure 16.1: Initial HAProxy Status

6. Check the node status from the cp node then check the proxy statistics. You should see the byte traffic counter increase.

```
student@cp:~$ kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
cp	Ready	control-plane	2d6h	v1.30.1
worker	Ready	<none>	2d3h	v1.30.1

Install Software

We will add two more control planes with stacked **etcd** databases for cluster quorum. You may want to open up two more PuTTY or SSH sessions and color code the terminals to keep track of the nodes.

Initialize the second cp before adding the third cp

- Configure and install the Kubernetes software on the **second cp**. Use the same steps as when we first set up the cluster, earlier in the course. You may want to copy and paste from earlier commands in your **history** to make these steps easier. **All the steps up to but not including kubeadm init or kubeadm join** A script `k8sWorker.sh` has been included in the course tarball to make this process go faster, if you would like. View and edit the script to be the correct version before running it.
- Install the software on the **third cp** using the same commands.

Join Control Plane Nodes

1. Edit the `/etc/hosts` file **ON ALL NODES** to ensure the alias of k8scp is set on each node to the proxy IP address. Your IP address may be different.

```
student@cp:~$ sudo vim /etc/hosts
```

```
10.128.0.64 k8scp
#10.128.0.24 k8scp
127.0.0.1 localhost
....
```

2. On the **first cp** create the tokens and hashes necessary to join the cluster. These commands may be in your **history** and easier to copy and paste.

3. Create a new token.

```
student@cp:~$ sudo kubeadm token create
```

```
jasg79.fdh4p2791320cz1g
```

4. Create a new SSL hash.

```
student@cp:~$ openssl x509 -pubkey \
-in /etc/kubernetes/pki/ca.crt | openssl rsa \
-pubin -outform der 2>/dev/null | openssl dgst \
-sha256 -hex | sed 's/^.* //'
```

```
f62bf97d4fba6876e4c3ff645df3fca969c06169dee3865aab9d0bca8ec9f8cd
```

5. Create a new cp certificate to join as a cp instead of as a worker.

```
student@cp:~$ sudo kubeadm init phase upload-certs --upload-certs
```

```
[upload-certs] Storing the certificates in Secret "kubeadm-certs" in the "kube-system" Namespace
[upload-certs] Using certificate key:
5610b6f73593049acddee6b59994360aa4441be0c0d9277c76705d129ba18d65
```

6. On the **second cp** use the previous output to build a **kubeadm join** command. Please be aware that multi-line copy and paste from Windows and some MacOS has paste issues. If you get unexpected output copy one line at a time.

```
student@secondcp:~$ sudo kubeadm join k8scp:6443 \
--token jasg79.fdh4p2791320cz1g \
--discovery-token-ca-cert-hash sha256:f62bf97d4fba6876e4c3ff645df3fca969c06169dee3865aab9d0bca8ec9f8cd \
--control-plane --certificate-key \
5610b6f73593049acddee6b59994360aa4441be0c0d9277c76705d129ba18d65
```

```
[preflight] Running pre-flight checks
[WARNING IsDockerSystemdCheck]: detected "cgroupfs" as the Docker cgroup driver. The recommended
→   driver \
    is "systemd". Please follow the guide at https://kubernetes.io/docs/setup/cri/
<output_omitted>
```

7. Return to the first cp node and check to see if the node has been added and is listed as a cp.

```
student@cp:~$ kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
cp	Ready	control-plane	2d6h	v1.30.1
secondcp	Ready	control-plane	10m	v1.30.1
worker	Ready	<none>	2d3h	v1.30.1

8. Copy and paste the **kubeadm join** command to the third cp. Then check that the third cp has been added.

```
student@cp:~$ kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
cp	Ready	control-plane	2d6h	v1.30.1
secondcp	Ready	control-plane	13m	v1.30.1
thirdcp	Ready	control-plane	3m	v1.30.1
worker	Ready	<none>	2d3h	v1.30.1

9. Copy over the configuration file as suggested in the output at the end of the join command. Do this on both newly added cp nodes.

```
student@secondcp:~$ mkdir -p $HOME/.kube
student@secondcp:~$ sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
student@secondcp:~$ sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

10. On the **Proxy node**. Edit the proxy to include all three cp nodes then restart the proxy.

```
student@ha-proxy:~$ sudo vim /etc/haproxy/haproxy.cfg
```

```
.....
backend k8sServers
    balance roundrobin
    server cp      10.128.0.24:6443 check
    server secondcp 10.128.0.30:6443 check #<-- Edit/Uncomment these lines
    server thirdcp  10.128.0.66:6443 check #<--
....
```

```
student@ha-proxy:~$ sudo systemctl restart haproxy.service
```

11. View the proxy statistics. When it refreshes you should see three new back-ends. As you check the status of the nodes using **kubectl get nodes** you should see the byte count increase on each node indicating each is handling some of the requests.

proxynode			Queue			Session rate			Sessions						Bytes		Denied	
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp		
Frontend				0	5	-	4	6	262 121	34			57 679	333 984	0	0		

k8sServers			Queue			Session rate			Sessions						Bytes		Denied	
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp		
cp	0	0	-	0	2		2	3	-	12	12	2s	19 280	89 042		0		
secondcp	0	0	-	0	2		1	2	-	11	11	3s	19 147	115 928		0		
thirdcp	0	0	-	0	2		1	2	-	11	11	3s	19 252	129 014		0		
Backend	0	0		0	5		4	5	262 213	34	34	2s	57 679	333 984	0	0		

stats			Queue			Session rate			Sessions						Bytes		Denied	
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp		
Frontend				0	3		-	1	4	262 121	11				4 916	119 887	0	0
Backend	0	0		0	1			0	1	262 213	7	0	0s	4 916	119 887	0	0	

Figure 16.2: Multiple HAProxy Status

12. View the logs of the newest **etcd** pod. Leave it running, using the **-f** option in one terminal while running the following commands in a different terminal. As you have copied over the cluster admin file you can run **kubectl** on any cp.

```
student@cp:~$ kubectl -n kube-system get pods |grep etcd
```

```
etcd-cp          1/1    Running   0           2d12h
etcd-secondcp   1/1    Running   0           22m
etcd-thirdcp    1/1    Running   0           18m
```

```
student@cp:~$ kubectl -n kube-system logs -f etcd-thirdcp
```

```
.....
2023-08-09 01:58:03.768858 I | mvcc: store.index: compact 300473
2023-08-09 01:58:03.770773 I | mvcc: finished scheduled compaction at 300473 (took 1.286565ms)
2023-08-09 02:03:03.766253 I | mvcc: store.index: compact 301003
2023-08-09 02:03:03.767582 I | mvcc: finished scheduled compaction at 301003 (took 995.775µs)
2023-08-09 02:08:03.785807 I | mvcc: store.index: compact 301533
2023-08-09 02:08:03.787058 I | mvcc: finished scheduled compaction at 301533 (took 913.185µs)
```

13. Log into one of the **etcd** pods and check the cluster status, using the IP address of each server and port 2379. Your IP addresses may be different. Exit back to the node when done.

```
student@cp:~$ kubectl -n kube-system exec -it etcd-cp -- /bin/sh
```

etcd pod

```
/ # ETCDCTL_API=3 etcdctl -w table \
--endpoints 10.128.0.66:2379,10.128.0.24:2379,10.128.0.30:2379 \
--cacert /etc/kubernetes/pki/etcd/ca.crt \
--cert /etc/kubernetes/pki/etcd/server.crt \
--key /etc/kubernetes/pki/etcd/server.key \
endpoint status
```



ENDPOINT → INDEX	ID	VERSION	DB SIZE	IS LEADER	RAFT TERM	RAFT
10.128.0.66:2379 2331065cd4fb02ff 3.5.7 24 MB true 11						
→ 392573						
10.128.0.24:2379 d2620a7d27a9b449 3.5.7 24 MB false 11						
→ 392573						
10.128.0.30:2379 ef44cc541c5f37c7 3.5.7 24 MB false 11						
→ 392573						

Test Failover

Now that the cluster is running and has chosen a leader we will shut down containerd, which will stop all containers on that node. This will emulate an entire node failure. We will then view the change in leadership and logs of the events.

1. Shut down the service on the node which shows IS LEADER set to true.

```
student@cp:~$ sudo systemctl stop containerd.service
```

If you chose cri-o as the container engine then the cri-o service and common processes are distinct. It may be easier to reboot the node and refresh the HAProxy web page until it shows the node is down. It may take a while for the node to finish the boot process. The second and third cp should work the entire time.

```
student@cp:~$ sudo reboot
```

2. You will probably note the **logs** command exited when the service shut down. Run the same command and, among other output, you'll find errors similar to the following. Note the messages about losing the leader and electing a new one, with an eventual message that a peer has become inactive.

```
student@cp:~$ kubectl -n kube-system logs -f etcd-thirdcp
```

```
....  
2023-08-09 02:11:39.569827 I | raft: 2331065cd4fb02ff [term: 9] received a MsgVote message with  
→ higher \  
          term from ef44cc541c5f37c7 [term: 10]  
2023-08-09 02:11:39.570130 I | raft: 2331065cd4fb02ff became follower at term 10  
2023-08-09 02:11:39.570148 I | raft: 2331065cd4fb02ff [logterm: 9, index: 355240, vote: 0] cast  
→ MsgVote \  
          for ef44cc541c5f37c7 [logterm: 9, index: 355240] at term 10  
2023-08-09 02:11:39.570155 I | raft: raft.node: 2331065cd4fb02ff lost leader d2620a7d27a9b449 at  
→ term 10  
2023-08-09 02:11:39.572242 I | raft: raft.node: 2331065cd4fb02ff elected leader ef44cc541c5f37c7  
→ at \  
          term 10  
2023-08-09 02:11:39.682319 W | rafthttp: lost the TCP streaming connection with peer  
→ d2620a7d27a9b449 \  
          (stream Message reader)  
2023-08-09 02:11:39.682635 W | rafthttp: lost the TCP streaming connection with peer  
→ d2620a7d27a9b449 \  
          (stream MsgApp v2 reader)  
2023-08-09 02:11:39.706068 E | rafthttp: failed to dial d2620a7d27a9b449 on stream MsgApp v2 \  
          (peer d2620a7d27a9b449 failed to find local node 2331065cd4fb02ff)  
2023-08-09 02:11:39.706328 I | rafthttp: peer d2620a7d27a9b449 became inactive (message send to  
→ peer failed)  
....
```

3. View the proxy statistics. The proxy should show the first cp as down, but the other cp nodes remain up.

k8s Servers																			Serv					
	Queue			Session rate			Sessions						Bytes		Denied		Errors		Warnings		Status	LastChk	Wght	
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis			
cp	0	0	-	0	2		2	3	-	23	23	1m50s	40 091	518 686	0	0	0	0	0	0	0	2s DOWN	* L4TOUT in 2001ms	1
secondcp	0	0	-	0	2		1	2	-	23	23	1m50s	48 861	302 977	0	0	0	0	0	0	0	41m10s UP	L4OK in 0ms	1
thirdcp	0	0	-	0	2		1	2	-	22	22	1m54s	38 446	257 768	0	0	0	0	0	0	0	41m18s UP	L4OK in 0ms	1
Backend	0	0		0	5		4	7	26 213	68	68	1m50s	127 398	1 079 431	0	0	0	0	0	0	0	41m18s UP		2

stats																			Serv						
	Queue			Session rate			Sessions						Bytes		Denied		Errors		Warnings		Status	LastChk	Wght		
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis				
Frontend				0	3	-	1	4		262 121	24		11 228	282 167	0	0	8					OPEN			
Backend	0	0		0	1		0	1		26 213	15	0	0s	11 228	282 167	0	0		15	0	0	0	41m18s UP		0

Figure 16.3: HAProxy Down Status

4. View the status using **etcdctl** from within one of the running **etcd** pods. You should get an error for the endpoint you shut down and a new leader of the cluster.

```
student@secondcp:~$ kubectl -n kube-system exec -it etcd-secondcp -- /bin/sh
```

etcd pod

```
/ # ETCDCTL_API=3 etcdctl -w table \
--endpoints 10.128.0.66:2379,10.128.0.24:2379,10.128.0.30:2379 \
--cacert /etc/kubernetes/pki/etcd/ca.crt \
--cert /etc/kubernetes/pki/etcd/server.crt \
--key /etc/kubernetes/pki/etcd/server.key \
endpoint status

Failed to get the status of endpoint 10.128.0.66:2379 (context deadline exceeded)
+-----+-----+-----+-----+-----+
| ENDPOINT | ID | VERSION | DB SIZE | IS LEADER | RAFT TERM | RAFT
+-----+-----+-----+-----+-----+
| 10.128.0.24:2379 | d2620a7d27a9b449 | 3.5.7 | 24 MB | true | 12 |
| ↵ 395729 |
| 10.128.0.30:2379 | ef44cc541c5f37c7 | 3.5.7 | 24 MB | false | 12 |
| ↵ 395729 |
+-----+-----+-----+-----+-----+
```

5. Turn the containerd service back on. You should see the peer become active and establish a connection.

```
student@cp:~$ sudo systemctl start containerd.service
```

```
student@cp:~$ kubectl -n kube-system logs -f etcd-thirdcp
```

```
....  
2023-08-09 02:45:11.337669 I | rafthttp: peer d2620a7d27a9b449 became active  
2023-08-09 02:45:11.337710 I | rafthttp: established a TCP streaming connection with peer\  
d2620a7d27a9b449 (stream MsgApp v2 reader)  
....
```

6. View the **etcd** cluster status again. Experiment with how long it takes for the **etcd** cluster to notice failure and choose a new leader with the time you have left.

Chapter 17

Closing and Evaluation Survey



17.1 Evaluation Survey	363
------------------------------	-----

17.1 Evaluation Survey

Evaluation Survey

Thank you for taking this course brought to you by **The Linux Foundation**.

Your comments are important to us and we take them seriously, both to measure how well we fulfilled your needs and to help us improve future sessions.

- Please Evaluate your training using the link your instructor will provide.
- Please be sure to check the spelling of your name and use correct capitalization as this is how your name will appear on the certificate.
- This information will be used to generate your **Certificate of Completion**, which will be sent to the email address you have supplied, so make sure it is correct.

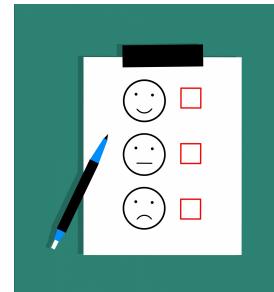


Figure 17.1: Course Survey

Appendices

Appendix A

Domain Review



A.1	CKA Exam	368
A.2	Exam Domain Review	369

A.1 CKA Exam

Domain Review

- Review the Candidate Handbook
- Find the proper version of the Curriculum Overview
- Write out each bullet point and steps
- Recognize good YAML examples in documentation
- Practice at speed

Navigate to the <https://www.cncf.io/certification/cka/> page and scroll down to look at the Exam Resources section. Read through each of the resources as you prepare for the CKA exam. Begin by looking through the **Candidate Handbook**. You can either view it as several pages, or download the entire handbook as a pdf file. Read through each section. Take note of the Resources allowed during exam section as this will list the URLs and browser you can use for the exam. Read the rest of the document carefully.

Also navigate to the **Exam Curriculum** <https://github.com/cncf/curriculum> page. You should find PDF files for the CKA, CKAD, and the CKS exam. There may be more than one PDF. Ensure the PDF you use matches the version of the exam you are going to take. These update on a regular basis. It is your responsibility to revisit and ensure you are looking at the correct version of the file.

Write out each of the knowledge, skills and abilities listed in the curriculum. Ensure you know the command line steps to complete each of the items. **Warning:** The word understand means more than you have a general idea. It means you are able to create, integrate, troubleshoot, and properly remove that object.

The exam now uses a secure browser, so you will not be able to use your own bookmarks. It is important you are familiar with the allowed documentation, so that you can use known, working YAML with minimal searching. Test each for the version of Kubernetes the exam is using. If the version changes you must re-check each YAML file to ensure it still works. It is better to know it works than discover it during the exam and have to troubleshoot the issue.

Many people simply run out of time during the exam. Use a timer and practice the curriculum at the pace necessary to complete each item during the exam, and have enough time to verify and troubleshoot your own work.

A.2 Exam Domain Review

Exercise A.1: Are you Ready?

1. If you have not searched for working and tested YAML examples, and can easily search for them again for all of the subjects the domain review mentions for the exam, you may:
 - a. Run out of time while searching for good YAML examples.
 - b. Make more mistakes.
 - c. Use YAML that isn't proper for the Kubernetes version of the exam and waste time trying to troubleshoot the issue.
 - d. All of the above.
2. Answer all that apply. In the context of the Curriculum Overview the term **Understand** when stating what a candidate should be able to do means:
 - a. You only have a general idea what the object does.
 - b. You can create the object.
 - c. You can configure and integrate the object with other objects.
 - d. You can properly update and test the object.
 - e. You can troubleshoot the object.
3. Have you practiced creating, integrating, and troubleshooting all of the domain review items **at speed**?
 - a. No. I kind of did the exercises. So I'm good.
 - b. No. I did the labs twice over a two week period.
 - c. Yes. I know the exam is intense and practiced with a clock running to make sure I can get everything done and also check my work.

Solution A.1

Are You Ready?

1. d.
2. b, c, d, e.
3. Hopefully c.

Exercise A.2: Preparing for the CKA Exam



Very Important

The source pages and content in this review could change at any time. **IT IS YOUR RESPONSIBILITY TO CHECK THE CURRENT INFORMATION.**

Before Taking Exam

Use this exercise as a resource after you complete the course but before you take the exam. Review the resources, know what good YAML looks like, and practice creating and working with objects at exam speed to assist with review and preparation.

1. Using a browser go to <https://www.cncf.io/certification/cka/> and read through the program description.

2. In the **Exam Resources** section open the **Curriculum Overview** and **Candidate-handbook** in new tabs. Both of these should be read and understood prior to sitting for the exam.
3. Navigate to the **Curriculum Overview** tab. You should see links for domain information for various versions of the exam. Select the latest version, such as **CKA_Curriculum_V1.25.pdf**. The versions you see may be different. You should see a new page showing a PDF.
4. Read through the document. Be aware that the term **Understand**, such as **Understand Services**, is more than just knowing they exist. In this case expect it to also mean create, configure, update, and troubleshoot.
5. Using only the exam-allowed URLs and sub-domains search for YAML examples for each domain or skill item. Ensure it works for the version of the exam you are taking, as the YAML may not have been re-tested after a new release. Become familiar with out to find each good example again, so you can find the page again during the exam.
6. Using a timer see how long it takes you to create and verify the objects listed below. Write down the time. Try it again and see how much faster you can complete and test each step.

"Practice until you get it right. Then practice until you can't get it wrong" -Unknown

Domain Review Items

This list is copied from competency domains found on the PDF. Again, **it remains your responsibility to check the web page for any changes to this list**.

- **Cluster Architecture, Installation & Configuration**
 - Manage role based access control (RBAC)
 - Use Kubeadm to install a basic cluster
 - Manage a highly-available Kubernetes cluster
 - Provision underlying infrastructure to deploy a Kubernetes cluster
 - Perform a version upgrade on a Kubernetes cluster using Kubeadm
 - Implement etcd backup and restore
- **Workloads & Scheduling**
 - Understand deployments and how to perform rolling updates and rollbacks
 - Use ConfigMaps and Secrets to configure applications
 - Know how to scale applications
 - Understand the primitives used to create robust, self-healing, application deployments
 - Understand how resource limits can affect Pod scheduling
 - Awareness of manifest management and common templating tools
- **Services & Networking**
 - Understand host networking configuration on the cluster nodes
 - Understand connectivity between Pods
 - Understand ClusterIP, NodePort, LoadBalancer service types and endpoints
 - Know how to use Ingress controllers and Ingress resources
 - Know how to configure and use CoreDNS
 - Choose an appropriate container network interface plugin
- **Storage**
 - Understand storage classes, persistent volumes

- Understand volume mode, access modes and reclaim policies for volumes
- Understand persistent volume claims primitive
- Know how to configure applications with persistent storage

- **Troubleshooting**

- Evaluate cluster and node logging
- Understand how to monitor applications
- Manage container stdout & stderr logs
- Troubleshoot application failure
- Troubleshoot cluster component failure
- Troubleshoot networking

Exercise A.3: Practicing Skills

This exercise is to help you practice your skills. It does not cover all the items listed in the domain review guide. You should develop your own steps to build a full list of skill tests and steps.

Also note that all the detailed steps are not included. You should be able to complete these steps without being told what to type.

In a work or exam environment you may not be told exactly what to do or how to do it. The following steps are meant to get you used to thinking about solutions when the exact need isn't clear.

1. Find and use the `review1.yaml` file included in the course tarball. Use the `find` output and copy the YAML file to your home directory. Use `kubectl create` to create the object. Determine if the pod is running. Fix any errors you may encounter. The use of `kubectl describe` may be helpful.

```
student@cp:~$ find ~ -name review1.yaml
student@cp:~$ cp <copy-paste-from-above> .
student@cp:~$ kubectl create -f review1.yaml
```

2. After you get the pod running remove any pods or services you may have created as part of the review before moving on to the next section. For example:

```
student@cp:~$ kubectl delete -f review1.yaml
```

3. Use the `review2.yaml` file to create a non-working deployment. Fix the deployment such that both containers are running and in a READY state. The web server listens on port 80, and the proxy listens on port 8080.

4. View the default page of the web server. When successful verify the GET activity logs in the container log. The message should look something like the following. Your time and IP may be different.

```
192.168.124.0 -- [3/Dec/2020:03:30:31 +0000] "GET / HTTP/1.1" 200 612 "-"
"curl/7.58.0" "-"
```

5. Find and use the `review4.yaml` file to create a pod, and verify it's running
6. Edit the pod such that it only runs on your worker node using the `nodeSelector` label.
7. Determine the CPU and memory resource requirements of `design-pod1`.
8. Edit the pod resource requirements such that the CPU limit is exactly twice the amount requested by the container. (Hint: subtract .22)
9. Increase the memory resource limit of the pod until the pod shows a Running status. This may require multiple edits and attempts. Determine the minimum amount necessary for the Running status to persist at least a minute.

10. Use the `review5.yaml` file to create several pods with various labels.
11. Using **only** the `--selector` value `tux` to delete only those pods. This should be half of the pods. Hint, you will need to view pod settings to determine the key value as well.
12. Create a new cronjob which runs busybox and the `sleep 30` command. Have the cronjob run every three minutes. View the job status to check your work. Change the settings so the pod runs 10 minutes from the current time, every week. For example, if the current time was 2:14PM, I would configure the job to run at 2:24PM, every Monday.
13. Delete any objects created during this review. You may want to delete all but the cronjob if you'd like to see if it runs in 10 minutes. Then delete that object as well.
14. Create a new secret called `specialofday` using the key `entree` and the value `meatloaf`.
15. Create a new deployment called `foodie` running the `nginx` image.
16. Add the `specialofday` secret to pod mounted as a volume under the `/food/` directory.
17. Execute a bash shell inside a `foodie` pod and verify the secret has been properly mounted.
18. Update the deployment to use the `nginx:1.12.1-alpine` image and verify the new image is in use.
19. Roll back the deployment and verify the typical, current stable version of nginx is in use again.
20. Create a new 200M NFS volume called `reviewvol` using the NFS server configured earlier in the lab.
21. Create a new PVC called `reviewpvc` which will uses the `reviewvol` volume.
22. Edit the deployment to use the PVC and mount the volume under `/newvol`
23. Execute a bash shell into the nginx container and verify the volume has been mounted.
24. Delete any resources created during this review.
25. Create a new deployment which uses the `nginx` image.
26. Create a new LoadBalancer service to expose the newly created deployment. Test that it works.
27. Create a new NetworkPolicy called `netblock` which blocks all traffic to pods in this deployment only. Test that all traffic is blocked to deployment.
28. Create a pod running nginx and ensure traffic can reach that deployment.
29. Update the `netblock` policy to allow traffic to the pod on port 80 only. Test that you can now access the default nginx web page.
30. Find and use the `review6.yaml` file to create a pod.

```
student@cp:~$ kubectl create -f review6.yaml
```

31. View the status of the pod.
32. Use the following commands to figure out why the pod has issues.

```
student@cp:~$ kubectl get pod securityreview
student@cp:~$ kubectl describe pod securityreview
student@cp:~$ kubectl logs securityreview
```

33. After finding the errors, log into the container and find the proper id of the nginx user.
34. Edit the pod such that the `securityContext` is in place and allows the web server to read the proper configuration files.
35. Create a new serviceAccount called `securityaccount`.
36. Create a ClusterRole named `secrole` which only allows create, delete, and list of pods in all apiGroups.

37. Bind the new clusterRole to the new serviceAccount.
38. Locate the token of the securityaccount. Create a file called `/tmp/securitytoken`. Put only the value of `token:` is equal to, a long string that may start with eyJh and be several lines long. Careful that only that string exists in the file.
39. Remove any resources you have added during this review
40. Create a new pod called `webone`, running the `nginx` service. Expose port 80.
41. Create a new service named `webone-svc`. The service should be accessible from outside the cluster.
42. Update both the pod and the service with selectors so that traffic for to the service IP shows the web server content.
43. Change the type of the service such that it is only accessible from within the cluster. Test that exterior access no longer works, but access from within the node works.
44. Deploy another pod, called `webtwo`, this time running the `wlniao/website` image. Create another service, called `webtwo-svc` such that only requests from within the cluster work. Note the default page for each server is distinct.
45. Test DNS names and verify CoreDNS is properly functioning.
46. Install and configure an ingress controller such that requests for `webone.com` see the `nginx` default page, and requests for `webtwo.org` see the `wlniao/website` default page. It does not matter which ingress controller you use.
47. Remove any resources created in this review.
48. Install a new cluster using an recent, previous version of Kubernetes. Backup etcd, then properly upgrade the entire cluster.
49. Create a pod running `busybox` without the scheduler being consulted.
50. Continue to create objects, integrate them with other objects and troubleshoot until each domain item has been covered.