LFD5461

Kubernetes Administration and Development, Capital One Custom Course

Version 1.20

THE LINUX FOUNDATION | Training & Certification

Version 1.20

Published by:

the **Linux Foundation**
https://www.linuxfoundation.org

Contents

List of Figures

Chapter 1

Introduction

We lay out our objectives and describe the **Linux Foundation** and our target audience.

1.1 Objectives

- By the end of this course, you should be able to:
  - Containerize and deploy a new **Python** script.
  - Configure the deployment with `ConfigMaps`, `Secrets`, and `SecurityContexts`.
  - Understand multi-container Pod design.
  - Configure probes for pod health.
  - Update and roll back an application.
  - Implement services and `NetworkPolicies`.
  - Use `PersistentVolumeClaims` for state persistence.

1.2 Who You Are

- This course is for developers looking to learn how to deploy, configure, and test their containerized applications on a multi-node **Kubernetes** cluster.
- For a successful learning experience, basic **Linux** command line and file editing skills are required.

- Familiarity with using a programming language (such as **Python**, **Node.js**, **Go**) and Cloud Native application concepts and architectures is helpful.
- Our free **LFS158x: Introduction to Kubernetes MOOC** on edX.org is a useful preparation for this course, and can be found at: https://www.edx.org/course/introduction-to-kubernetes.
- You might be doing this just for fun, but more likely, this task is part of your job. The purpose here is to ease your path and perhaps shorten the amount of time it takes to reach the level of competence required.
- How much you get out of this and how bug-free, efficient, and optimized your code will be, depends on how good a programmer you were before you started with the present material. There is no intent or time here to teach you the elements of good programming or the **Linux** operating system design in great detail.

1.3 The Linux Foundation

- The **Linux Foundation** is the non-profit consortium dedicated to fostering the growth of **Linux** and many other **Open Source Software** (**OSS**) projects and communities.
- The **Linux Foundation** supports the creation of sustainable **OSS** ecosystems by providing:
  - Financial and intellectual resources and services
  - Training
  - Events
- Founded in 2000, it sponsors the work of **Linux** creator **Linus Torvalds** and is supported by leading technology companies and developers from around the world.
- Today it has grown well beyond that mission to support a host of other **OSS** projects and communities, with the list of hosted foundations and organizations growing daily.
- 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.

  Over 85,000 open source technologists and leaders worldwide gather at **Linux Foundation** events annually to share ideas, learn and collaborate. **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.

  The **Linux Foundation** hosts an ever increasing number of events each year. Some are held in multiple locations yearly, such as North America, Europe and Asia. Here is a partial list:

  - Open Source Summit
  - Embedded Linux Conference
  - 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.

- The **Linux Foundation** also provides services to the **Linux** development community, including an open source developer travel fund, legal support and other administrative assistance. Through its work groups, members and developers can collaborate on key technical areas which can include everything from Cloud Computing to Printing in **Linux**. Finally, users can access **Linux** training through its technical training program. (https://training.linuxfoundation.org)
- The **Linux Foundation** website (https://www.linuxfoundation.org) contains fuller descriptions of the ongoing work it is involved in.

1.4 Linux Foundation Training

- 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
- To get more information about specific courses offered by the **Linux Foundation**:
  - **Linux Programming & Development Training**:
    https://training.linuxfoundation.org/linux-courses/development-training
  - **Enterprise IT & Linux System Administration Courses**:
    https://training.linuxfoundation.org/linux-courses/system-administration-training
  - **Open Source Compliance Courses**:
    https://training.linuxfoundation.org/linux-courses/open-source-compliance-courses

1.5 Certification Programs and Digital Badging

Certification Programs

- The **Linux Foundation** offers a two-level **Certification Program**:

LFCS (**L**inux **F**oundation **C**ertified **S**ysadmin)

LFCE (**L**inux **F**oundation **C**ertified **E**ngineer)

- 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** and **LFCE** exams, the **Linux Foundation** is currently associated with other open source certification programs involving its collaborative projects, for which it has created (or is currently creating) the exams. These include:

- **Certified Kubernetes Administrator** (**CKA**)
- **Certified Kubernetes Application Developer** (**CKAD**)
- **Certified Kubernetes Application Security Specialist** (**CKAS**)
- **Linux Foundation Certified IT Associate** (**LFCA**)
- **FinOps Certified Practitioner** (**FOCP**)
- **Certified Hyperledger Fabric Administrator** (**CHFA**)
- **Certified Hyperledger Sawtooth Administrator** (**CHSA**)
- **OpenJS Node.js Services Developer** (**JSNSD**)
- **OpenJS Node.js Application Developer** (**JSNAD**)
- **Cloud Foundry Certified Developer** (**CFCD**)
- **Certified ONAP Professional** (**COP**)

• For additional information, including technical requirements and other logistics, see https://training.linuxfoundation.org.

Linux Foundation Digital Badges

THE LINUX FOUNDATION CERTIFIED IT ASSOCIATE
THE LINUX FOUNDATION CERTIFIED SYSADMIN
THE LINUX FOUNDATION CERTIFIED ENGINEER
CLOUD FOUNDRY CERTIFIED DEVELOPER
CERTIFIED kubernetes ADMINISTRATOR
CERTIFIED kubernetes APPLICATION DEVELOPER
CERTIFIED kubernetes SECURITY SPECIALIST
FinOps Certified Practitioner
CERTIFIED HYPERLEDGER FABRIC ADMINISTRATOR
CERTIFIED HYPERLEDGER FABRIC DEVELOPER
CERTIFIED HYPERLEDGER SAWTOOTH ADMINISTRATOR
CERTIFIED ONAP PROFESSIONAL
OPENJS NODE.JS SERVICES DEVELOPER
OPENJS NODE.JS APPLICATION DEVELOPER

Figure 1.1: **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 **Linux Foundation** courses and certifications
- Details at https://training.linuxfoundation.org/badges/

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.

1.6 Preparing Your System

- You will need a computer installed with a current **Linux** distribution, with the important developer tools (for compiling, etc.) properly deployed.
- Once you have installed your favorite OS, go to https://training.linuxfoundation.org/cm/prep/ and download a copy of the `ready-for.sh` script, either by downloading through your browser, or using **wget**:

```
$ wget https://training.linuxfoundation.org/cm/prep/ready-for.sh
$ chmod 755 ready-for.sh
$ ./ready-for.sh LFD5461
```

- If there are missing dependencies, you can do:

```
$ ./ready-for.sh --install LFD5461
```

1.7 Course Registration

- Please go to: https://training.linuxfoundation.org/surveys/registration/, and using the unique **key** your instructor provides you, register for this class.
- This will be used to generate your **Certificate of Completion** at the end of the class. 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.
- At the end of the class you will be asked to submit an evaluation survey, at: https://training.linuxfoundation.org/surveys/evaluation/, using the same key and email address you used for registration.
- Thank you!

1.8 Labs

Exercise 1.1: Obtaining Some System Information

- Open up a command line terminal window; there are a number of ways to do this as we shall discuss later. If you do not see it under `Applications->Accessories` or `Applications->System Tools`, do `Alt-F2` and type in **gnome-terminal**.
- The **uname** utility will provide some basics. If you type

```
$ uname --help
```

you will get:

```
$ uname --help
```

```
Usage: uname [OPTION]...
Print certain system information.  With no OPTION, same as -s.

  -a, --all                print all information, in the following order,
                             except omit -p and -i if unknown:
  -s, --kernel-name        print the kernel name
  -n, --nodename           print the network node hostname
  -r, --kernel-release     print the kernel release
  -v, --kernel-version     print the kernel version
  -m, --machine            print the machine hardware name
  -p, --processor          print the processor type or "unknown"
  -i, --hardware-platform  print the hardware platform or "unknown"
  -o, --operating-system   print the operating system
      --help     display this help and exit
      --version  output version information and exit
```

Try some of the options.

- Other basic commands you might try are:

```
$ df
$ free
$ less /proc/meminfo
$ more /proc/cpuinfo
$ cat  /proc/version
```

Chapter 2

Kubernetes Architecture

In this chapter we are going to cover a little bit of the history of Kubernetes. This helps to understand the why things are where they are. Then we'll talk about `master` nodes and the main agents that work to make Kubernetes what it is. We'll also talk about `minion` or `worker` nodes. These are nodes that use API calls back to the master node to understand what the configuration should be, and report back status. We'll also talk about CNI network plugin configuration, which continues to be the future of Kubernetes networking. And we'll cover some common terms so as you learn more about Kubernetes, you'll understand major components and how they work with the other components in our decoupled and transient environment.

2.1 What Is Kubernetes?

- Running a container on a laptop is relatively simple. Deploying and connecting containers across multiple hosts, scaling them, deploying applications without downtime, and service discovery among several aspects can be complex.
- 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 operators allows easy customization for various production needs. The ability to add multiple schedulers and multiple API servers adds to this customization.

- According to the https://kubernetes.io website, **Kubernetes** is:

  " an open-source system for automating deployment, scaling, and management of containerized applications"

- 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.

Figure 2.1: **Kubernetes Logo**

In Greek, κνβερνητης means the Helmsman, or pilot of the ship. Keeping with the maritime theme of Docker containers, Kubernetes is the pilot of a ship of containers. Due to the 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**.

- Kubernetes can be an integral part of **Continuous Integration/Continuous Delivery (CI/CD)**, as it offers many of the necessary components.
  - **Continuous Integration** - A consistent way to build and test software. Deploying new packages with code written each day, or every hour, instead of quarterly. Tools like **Helm** and **Jenkins** are often part of this with Kubernetes.
  - **Continuous Delivery** - An automated way to test and deploy software into various environments. Kubernetes handles the life cycle of containers and connection of infrastructure resources to make rolling updates and rollbacks easy, among other deployment schemes.
  - There are several options and possible configurations when building a CI/CD pipeline. Tools such as Jenkins, Spinnaker, GitHub, GitLab, and Helm among others may be part of your particular pipeline. To learn more about CI/CD take a look at **DevOps and SRE Fundamentals: Implementing Continuous Delivery (LFS261)** at https://training.linuxfoundation.org/training/devops-and-sre-fundamentals-implementing-continuous-delivery-lfs261/

2.2 Components of Kubernetes

- Deploying containers and using Kubernetes may require a change in the development and the 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, a 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 web servers, or **microservices** . The server and client sides of the application expect that there are one or more microservices, called `replicas`, available to respond to a request. It is also important that clients expect server processes to be terminated and eventually be replaced, leading to a transient server deployment. Instead of a large tightly-coupled Apache web server with several **httpd** daemons responding to page requests, there would be many **nginx** servers, each responding without knowledge of each other.

- 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 and API calls. A service ties traffic from one agent to another (for example, a frontend web server to a backend database) and handles new IP or other information, should either one die and be replaced.

- Developers new to Kubernetes sometimes assume it is another virtual-machine manager, similar to what they have been using for decades and continue to develop applications in the same way as prior to using Kubernetes. **This is a mistake.** The decoupled, transient, microservice architecture is not the same. Most legacy applications will need to be rewritten to optimally run in a cloud. In this diagram we see the legacy deployment strategy on the left with a monolithic applications deployed to nodes. On the right we see an example of the same functionality, on the same hardware, using multiple microservices.

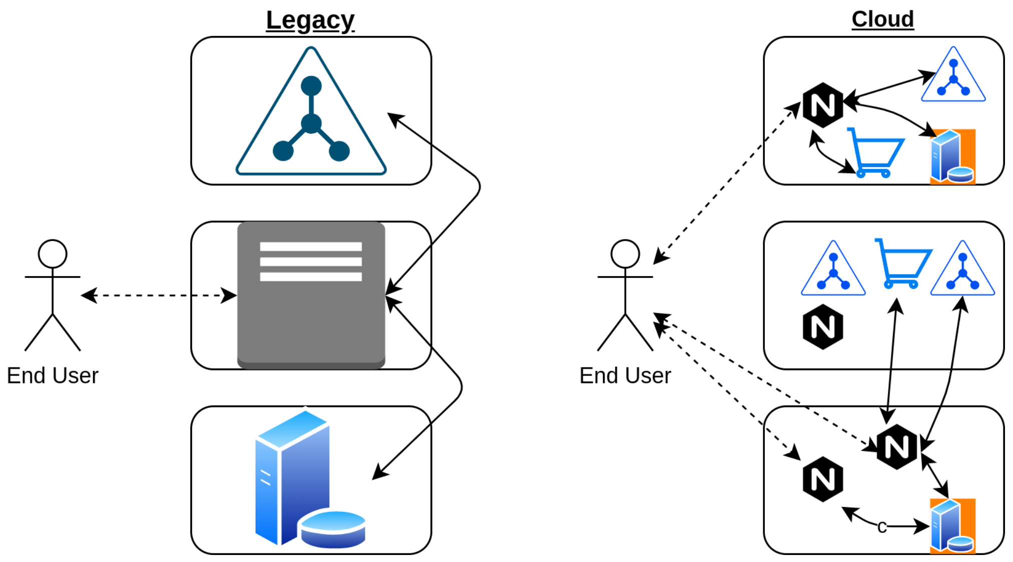


Figure 2.2: **Legacy vs Cloud Architecture**

- Communication to, as well as internally, between components is API call-driven, which allows for flexibility. Configuration information is stored in a JSON format, but is most often written in YAML. Kubernetes agents convert the YAML to JSON prior to persistence to the database.

> 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 Challenges

- Containers have seen a huge rejuvenation. They provide a great way to package, ship, and run applications - that is the **Docker** motto, now provided by many tools.
- 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 registries, and providing a powerful user experience to manage containers. Now several tools such as **Buildah**, **Podman**, **cri-o**, **containerd**, **frakti**, and others allow for easy container creation and management.
- However, managing containers at scale and architecting a distributed application based on microservices' principles is still challenging.
- You first need a continuous integration pipeline to build your container images, test them, and verify them. 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, and watch over them when things fail and self-heal. You must be able to perform rolling updates and rollbacks, and eventually tear down the resource when no longer needed.

- All of these actions require flexible, scalable, and easy-to-manage 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.

Please Note

Would users notice if you ran **Chaos Monkey**, which terminates `any` container randomly? If so you may have more work making containers and applications more decoupled and transient.

2.4 The Borg Heritage

- 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, part of the current growth in Kubernetes is making it easier to work with and handle workloads not found in a Google data center.

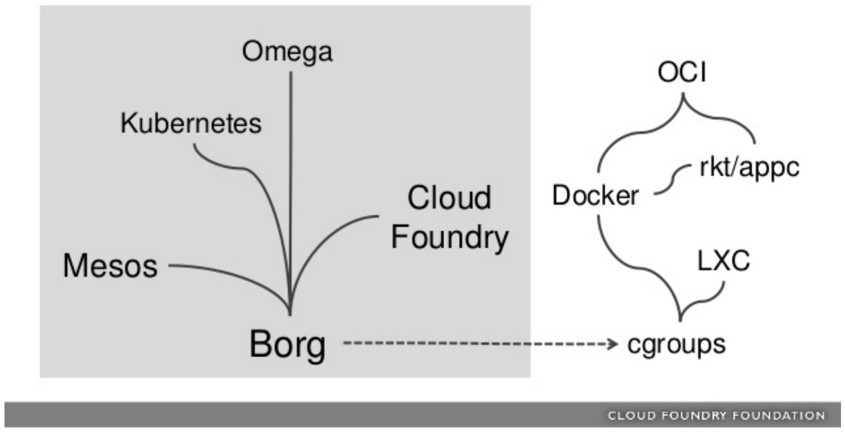


Figure 2.3: **Kubernetes Lineage**

(The above figure was obtained from
https://www.slideshare.net/chipchilders/cloud-foundry-the-platform-for-forging-cloud-native-applications.)

- To learn more about the ideas behind Kubernetes, you can read the ***Large-scale cluster management at Google with Borg paper*** at https://ai.google/research/pubs/pub43438.
- 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.
- The Cloud Foundry Foundation embraces the 12 factor application principles at https://12factor.net/.
- 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.

2.5 Kubernetes Architecture

- To quickly demystify Kubernetes, let's have a look at a Kubernetes architecture graphic, which shows a high-level architecture diagram of the system components.

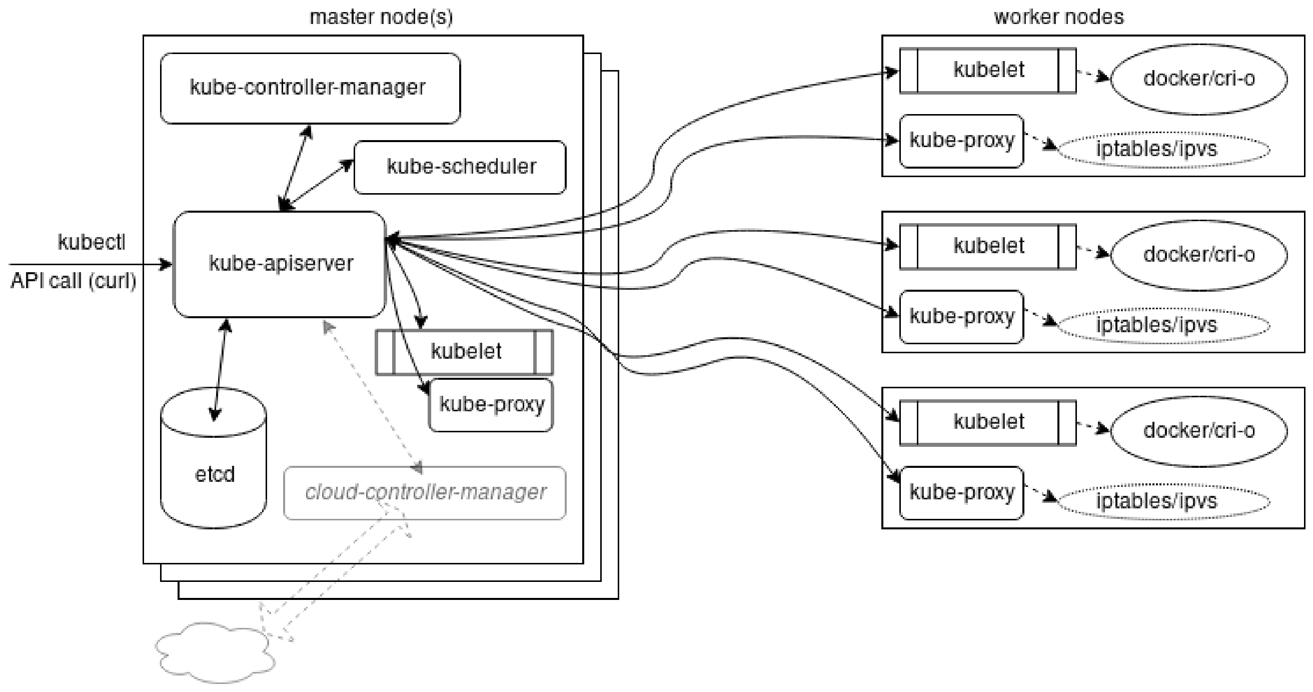


Figure 2.4: **Kubernetes Architecture**

- In its simplest form, Kubernetes is made of one or more central managers (aka master) and worker nodes (we will see in a follow-on chapter how you can actually run everything on a single node for testing purposes). The manager runs an API server, a scheduler, various operators and a datastore to keep the state of the cluster, container settings, and the 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. The **kube-scheduler** sees the API requests for running a new container and finds a suitable node to run that container. Each node in the cluster runs two containers: **kubelet** and **kube-proxy**. The **kubelet** container receives spec information for container configuration, downloads and manages any necessary resources and works with the container engine on the local node to ensure the container runs or is restarted upon failure. The **kube-proxy** container creates and manages local firewall rules and networking configuration to expose containers on the network.

2.6 Terminology

- 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.
- Orchestration is managed through a series of watch-loops, also known as **operators** or controllers. Each operator interrogates the **kube-apiserver** for a particular object state, modifying the object until the declared state matches the current state. The default, newest, and feature-filled operator for containers is a **Deployment**. A Deployment deploys and manages a different operator called a **ReplicaSet**. A replicaSet is an operator which deploys multiple pods, each with the same spec information. These are called replicas. The Kubernetes architecture is made up of many operators such as **Jobs** and **CronJobs** to handle single or recurring tasks, or custom resource definitions and purpose built operators.

- There are several other API objects which can be used to deploy pods, other than ensuring a certain number of replicas is running somewhere. A **DaemonSet** will ensure that a single pod is deployed on every **node**. These are often used for logging, metrics, and security Pods. A **StatefulSet** can be used to deploy Pods in a particular order, such that following pods are only deployed if previous pods report a ready status. This is useful for legacy applications which are not cloud-friendly.
- To easily manage thousands of Pods across hundreds of nodes can be a difficult. To make management easier, we can use **labels**, arbitrary strings which become part of the object metadata. These are selectors which can then be used when checking or changing the state of objects without having to know individual names or UIDs. Nodes can have **taints**, an arbitrary string in the node metadata, to inform the scheduler on Pod assignments used along with **toleration** in Pod metadata, indicating it should be scheduled on a node with the particular taint.
- There is also space in metadata for **annotations** which remain with the object but cannot be used as a selector, but could be leveraged by other objects or Pods.
- While using lots of smaller, commodity hardware could allow every user to have their very own cluster, often multiple users and teams share access to one or more clusters. This is referred to as **multi-tenancy**. Some form of isolation is necessary in a multi-tenant cluster, using a combination of the following, which we introduce here but will learn more about in the future:
  - **namespace** - A segregation of resources, upon which resource quotas and permissions can be applied. Kubernetes objects may be created in a namespace or cluster-scoped. Users can be limited by the object verbs allowed per namespace. Also the `LimitRange` admission controller constrains resource usage in that namespace. Two objects cannot have the same `Name:` value in the same namespace.
  - **context** - A combination of user, cluster name and namespace. A convenient way to switch between combinations of permissions and restrictions. For example you may have a development cluster and a production cluster, or may be part of both the operations and architecture namespaces. This information is referenced from `~/.kube/config`.
  - **Resource Limits** - A way to limit the amount of resources consumed by a pod, or to request a minimum amount of resources reserved, but not necessarily consumed, by a pod. Limits can also be set per-namespaces, which have priority over those in the PodSpec.
  - **Pod Security Policies** - A policy to limit the ability of pods to elevate permissions or modify the node upon which they are scheduled. This wide-ranging limitation may prevent a pod from operating properly. The use of PSPs may be replaced by `Open Policy Agent` in the future.
  - **Network Policies** - The ability to have an inside-the-cluster firewall. Ingress and Egress traffic can be limited according to namespaces and labels as well as typical network traffic characteristics.

2.7 Master Node

- The Kubernetes master runs various server and manager processes for the cluster. Among the components of the master node are the **kube-apiserver**, the **kube-scheduler**, and the **etcd** database. 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 **Digital Ocean** 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 resource monitoring.

kube-apiserver

- 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 agent which connects to the **etcd** database. As a result, it acts as a master process for the entire cluster, and acts as a front end of the cluster's shared state. Each API call goes through three steps: authentication, authorization, and several admission controllers.

kube-scheduler

- 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 available CPU) to, and then assign the Pod based on availability and success. The scheduler uses pod-count by default, but complex configuration is often done if cluster-wide metrics are collected.
- There are several ways you can affect the algorithm, or a custom scheduler could be used simultaneously instead. A Pod can also be assigned bind to a particular node in the pod spec, though the Pod may remain in a pending if the node or other declared resource is unavailable.
- One of the first configurations referenced during creation is if the Pod can be deployed within the current quota restrictions. If so, then the taints and tolerations, and labels of the Pods are used along with those of the nodes to determine the proper placement. Some is done as an admission controller in the kube-apiserver, the rest is done by the chosen scheduler. You can find more details about the scheduler can be found on GitHub at https://github.com/kubernetes/kubernetes/blob/master/pkg/scheduler/scheduler.go.

Etcd Database

- 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 finding and changing an entry, values are always appended to the end. Previous copies of the data are then marked for future removal by a compaction process. It works with **curl** and other HTTP libraries, and provides reliable watch queries.
- Simultaneous requests to update a particular 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 as found in the object, 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 **master** database along with possible **followers**. They communicate with each other on an ongoing basis to determine which will be **master**, and determine another in the event of failure. While very fast and potentially durable, there have been some hiccups with some features like whole cluster upgrades. The **kubeadm** cluster creation tool allows easy deployment of a multi-master cluster with stacked **etcd** or an external database cluster.

Other Agents

- 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** as of v1.16, 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.

2.8 Minion (Worker) Nodes

- All worker nodes run the **kubelet** and **kube-proxy**, as well as the container engine, such as **Docker** or **cri-o** https://cri-o.io. Other management daemons are deployed to watch these agents or provide services not yet included with Kubernetes.
- The **kubelet** interacts with the underlying Docker 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 the **userspace** mode, in which it monitors Services and Endpoints using a random high-number port to proxy traffic. Use of **ipvs** can be enabled, with the expectation it becoming the default, replacing **iptables**.

- Kubernetes does not have cluster-wide logging yet. Instead, another CNCF project is used, called **Fluentd** (https://www.fluentd.org/). When implemented, it provides a unified logging layer for the cluster, which filters, buffers, and routes messages.
- Cluster-wide metrics is not quite fully mature, so **Prometheus** (https://Prometheus.io) is also often deployed to gather metrics from nodes and perhaps some applications.

Kubelet

- The **kubelet** agent is the heavy lifter for changes and configuration on worker nodes. It accepts the API calls for Pod specifications (a **PodSpec** is a `JSON` or `YAML` file that describes a pod). 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.

2.9 Pods

- 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 by default. As a result, there is no way to determine which container becomes available first inside a pod. The use of `initContainers` can be used to ensure some containers are ready before others in a pod. 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, with most network plugins. (**HPE Labs** made a plugin which allows more than one IP per pod) As a result if there is more than one container, they must share the IP. To communicate with each other they can 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.
- Pods, and other objects, can be created in several ways. They can be created by using a generator, which historically has changed with each release:

  ```
  kubectl run newpod --image=nginx  --generator=run-pod/v1
  ```

  Or they can be created and deleted using a properly formatted JSON or YAML file:

  ```
  kubectl create -f newpod.yaml

  kubectl delete -f newpod.yaml
  ```

  Other objects will be created by operators/watch-loops to ensure the spec and current status are the same.

2.10 Services

- With every object and agent decoupled we need a flexible and scalable operator which connects resources together and will reconnect, should something die and a replacement is spawned. Each Service is a microservice handling a particular bit of traffic, such as a single **NodePort** or a **LoadBalancer** to distribute inbound requests among many Pods.
- A **Service** also handles access policies for inbound requests, useful for resource control, as well as for security.
- A service, as well as **kubectl**, uses a **selector** in order to know which objects to connect. There are two selectors currently supported:

- **equality-based** - Filters by label keys and their values. Three operators can be used such as `=`, `==`, and `!=`. If multiple values or keys are used all must be included for a match.
- **set-based** - Filters according to a set of values. The operators are `in`, `notin`, and `exists`. For example the use of

  ```
  status notin (dev, test, maint)
  ```

  would select resources with the key of `status` which did not have a value of `dev`, `test`, nor `maint`.

2.11 Operators

- An important concept for orchestration is the use of operators. These are also known as watch-loops and controllers. These query the current state, compare it against the spec, and execute code based on how they differ.

  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 receives 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 specification.

- The **Informer** which uses the API server as a source requests the state of an object via an 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` operators each manage the eponymous resources for Pods.

2.12 Single IP per Pod

- 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 diagram below shows a pod with two containers, MainApp and Logger, and two data volumes, made available under two mount points. Containers MainApp and Logger 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. You won't see this container from the Kubernetes perspective, but you would by running **sudo docker ps**. The volumes are shown for completeness and will be discussed later.

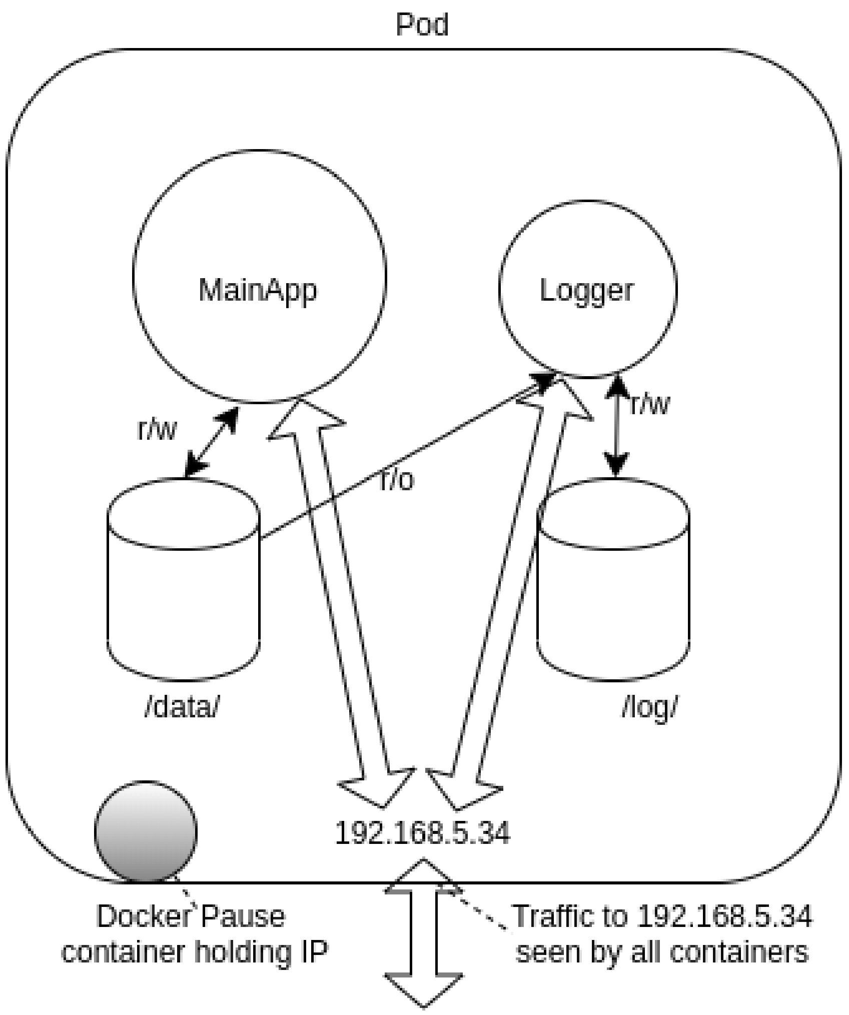

Pod
MainApp
Logger
r/w
r/o
r/w
/data/
/log/
192.168.5.34
Docker Pause container holding IP
Traffic to 192.168.5.34 seen by all containers

Figure 2.5: **Single IP, multiple volumes**

- To communicate with each other, Pods can use the loopback interface, write to files on a common filesystem, or via inter-process communication (IPC). As a result co-locating applications in the same pod may have issues. There is a network plugin which will allow more than one IP address, but so far has only been used within HPE labs.
- Support for dual-stack, IPv4 and IPv6, continues to increase with each release. For example, in a recent release **kube-proxy** `iptables` supports both stacks simultaneously.

2.13 Networking Setup

- 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 set up properly, as well.
- A detailed explanation about the Kubernetes networking model can be seen on the Cluster Networking page in the Kubernetes documentation 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 of physical hosts. The network needs to assign IP addresses to pods, and needs to provide traffic routes between all pods on any nodes.

- The three main networking challenges to solve in a container orchestration system are:
  - Coupled container-to-container communications (solved by the pod concept)
  - Pod-to-pod communications
  - External-to-pod communications
- Kubernetes expects the network configuration to enable pod-to-pod communications to be available; it will not do it for you.
- Pods are assigned an IP address prior to application containers being started. The **service** object is used to connect pods within the network using **ClusterIP** addresses, from outside of the cluster using **NodePort** addresses, and using a load balancer if configured with a **LoadBalancer** service.
- A **ClusterIP** is used for traffic within the cluster. A **NodePort** first creates a **ClusterIP** then associates a port of the node to that new **ClusterIP**. If you create a **LoadBalancer** service it will first create a **ClusterIP**, then a **NodePort** and then make an asynchronous request for an external load balancer. If one is not configured to respond the `EXTERNAL-IP` will remain in `pending` state for the life of the service.

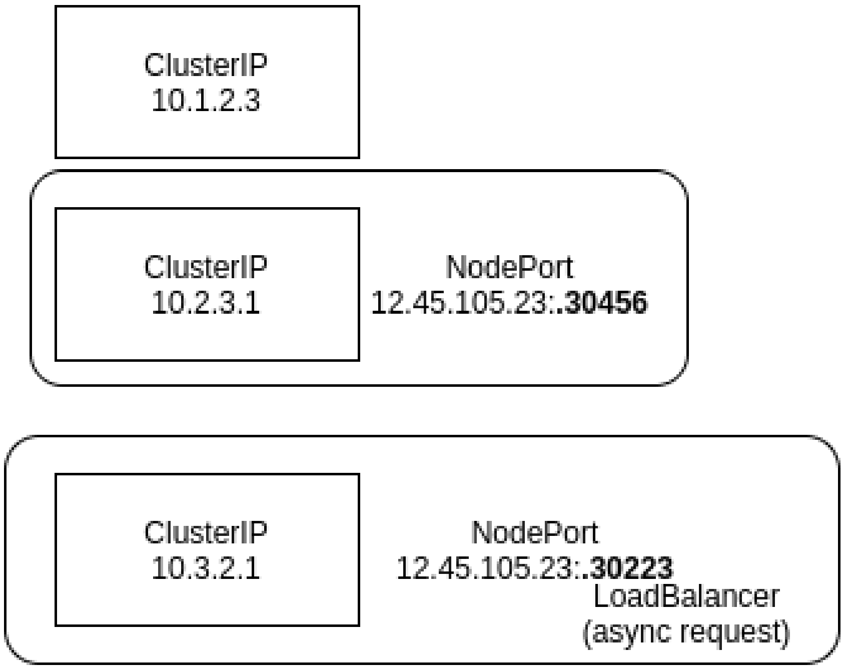


Figure 2.6: **Service Relationships**

- An **Ingress Controller** or a service mesh like **Istio** can also be used to connect traffic to a pod. This image shows a multi-container pod, two services with one for internal traffic only, and a an ingress controller. The sidecar container, acting as a logger, is shown writing out storage, just to show a more complete pod. The `pause` container, which is only used to retrieve the namespace and IP address is also shown.

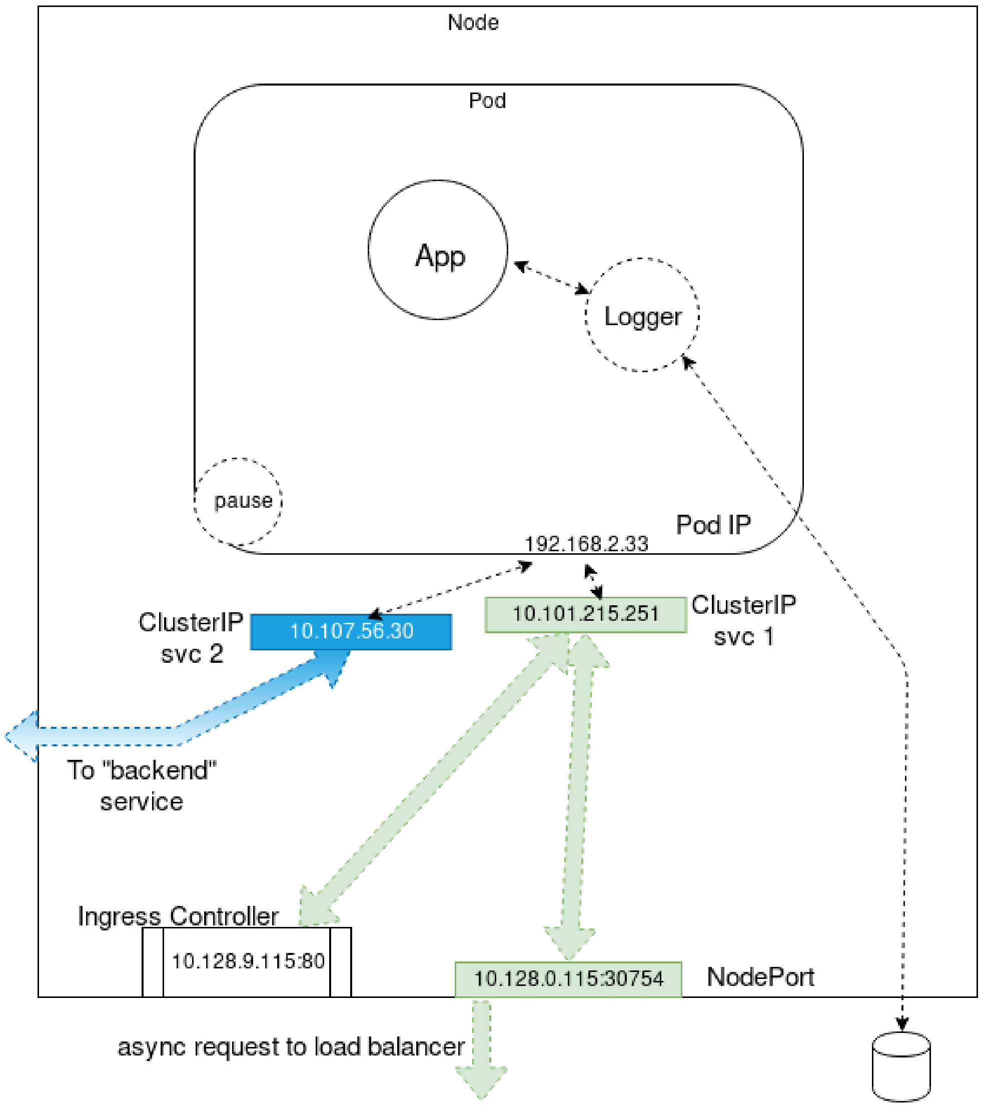


Figure 2.7: **Pod and Services Example**

- Another possible view of a cluster with multiple pods and services. This graphic shows the **Calico** pod running on each node and communicating with the `BIRD` protocol. There are also three **ClusterIP** services and on **LoadBalancer** service trying to show how the front end may communicate with other pods. The pods could have been on any worker, and are shown on one only as an example. Note this graphic is not an expansion of the previous graphic.

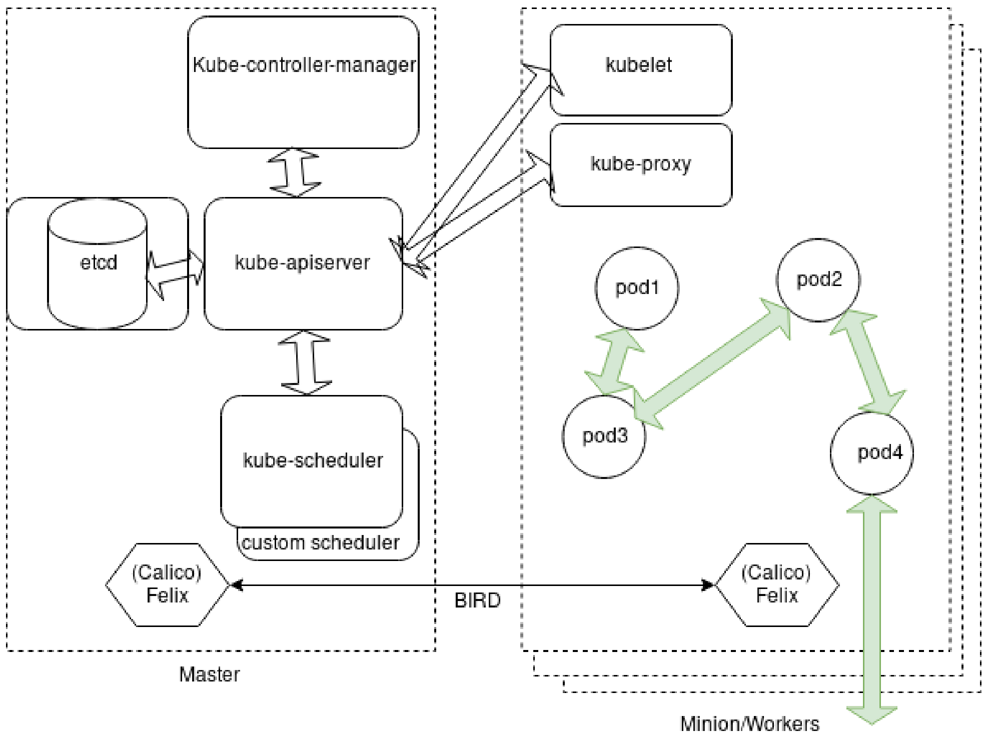


Figure 2.8: **Another Cluster Example**

- Tim Hockin, one of the lead Kubernetes developers, has created a very useful slide deck to understand the Kubernetes networking An Illustrated Guide to Kubernetes Networking at https://speakerdeck.com/thockin/illustrated-guide-to-kubernetes-networking

2.14 CNI Network Configuration File

- To provide container networking, Kubernetes is standardizing on the Container Network Interface (CNI) specification at https://github.com/containernetworking/cni. As of v1.6.0, **kubeadm** (the Kubernetes cluster bootstrapping tool) uses CNI as the default network interface mechanism.
- 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 specification is language-agnostic, there are many plugins from Amazon ECS, to SR-IOV, to Cloud Foundry, and more.
- With CNI, you can write a 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" }
             ]
    }
}
```

- 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.

2.15 Pod-to-Pod Communication

- 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 communication across nodes.
- The early requirement from Kubernetes was the following:
  - All pods can communicate with each other across nodes.
  - All nodes can communicate with all pods.
  - No Network Address Translation (NAT).
- 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:
  - **Weave**
    https://www.weave.works/oss/net/
  - **Flannel**
    https://github.com/coreos/flannel#flannel
  - **Calico**
    https://www.projectcalico.org/
  - **Romana**
    https://romana.io/
- Most network plugins now support the use of `Network Policies` which act as an internal firewall, limiting ingress and egress traffic.
- See the cluster networking documentation page (https://kubernetes.io/docs/concepts/cluster-administration/networking/) or the list of networking add-ons (https://kubernetes.io/docs/concepts/cluster-administration/addons/ for a more complete list.

2.16 Cloud Native Computing Foundation

Figure 2.9: **Cloud Native Computing Foundation (CNCF)**

- Kubernetes is an open source software with 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 is known as 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.

2.17 Resource Recommendations

- If you want to go beyond this general introduction to Kubernetes, here are a few things we recommend:
  - Read the Borg paper.
    https://ai.google/research/pubs/pub43438
  - Listen to John Wilkes talking about Borg and Kubernetes.
    https://www.gcppodcast.com/post/episode-46-borg-and-k8s-with-john-wilkes/
  - Add the Kubernetes community hangout to your calendar, and attend at least once.
    https://github.com/kubernetes/community
  - Join the community on **Slack** and go in the `#kubernetes-users` channel.
    https://github.com/kubernetes/community

2.18 Labs

Exercise 2.1: Overview and Preliminaries

We will create a two-node **Ubuntu 18.04** cluster. Using two nodes allows an understanding of some issues and configurations found in a production environment. Currently 2 vCPU and 8G of memory allows for quick labs. Other Linux distributions should work in a very similar manner, but have not been tested.

Very Important

Regardless of the platform used (**VirtualBox**, **VMWare**, **AWS**, **GCE** or even bare metal) please remember that security software like **SELinux**, **AppArmor**, and firewall configurations can prevent the labs from working. While not something to do in production consider disabling the firewall and security software.

GCE requires a new `VPC` to be created and a rule allowing all traffic to be included. The use of **wireshark** can be a helpful place to start with troubleshooting network and connectivity issues if you're unable to open all ports.

The **kubeadm** utility currently requires that swap be turned off on every node. The **swapoff -a** command will do this until the next reboot, with various methods to disable swap persistently. Cloud providers typically deploy instances with swap disabled.

Download shell scripts and YAML files

To assist with setting up your cluster please download the tarball of shell scripts and YAML files. The `k8sMaster.sh` and `k8sSecond.sh` scripts deploy a Kubernetes cluster using **kubeadm** and use `Project Calico` for networking. Should the file not be found you can always use a browser to investigate the parent directory.

```
$ wget https://training.linuxfoundation.org/cm/LFD5461/LFD5461_V1.20_SOLUTIONS.tar.xz \
              --user=LFtraining --password=Penguin2014

$ tar -xvf LFD5461_V1.20_SOLUTIONS.tar.xz
```

(Note: depending on your software, 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!)

Exercise 2.2: Deploy a New Cluster

Deploy a Master Node using Kubeadm

1. Log into your nodes using **PuTTY** or using **SSH** from a terminal window. Unless the instructor tells you otherwise the user name to use will be **student**. You may need to change the permissions on the `pem` or `ppk` file as shown in the following commands. Your file and node IP address will probably be different.

```
localTerm:~$ chmod 400 LFD459.pem
localTerm:~$ ssh -i LFD459.pem student@WW.XX.YY.ZZ
```

```
student@master:~$
```

2. Review the script to install and begin the configuration of the master kubernetes server. You may need to change the **find** command search directory which uses tilde for your home directory depending on how and where you downloaded the tarball.

A **find** command is shown if you want to locate and copy to the current directory instead of creating the file. Mark the command for reference as it may not be shown for future commands.

```
student@master:~$ find $HOME -name <YAML File>
student@master:~$ cp LFD5461/<Some Path>/<YAML File> .
```

```
student@master:~$ find $HOME -name k8sMaster.sh
```

```
student@master:~$ more LFD5461/SOLUTIONS/s_02/k8sMaster.sh
```

SH **k8sMaster.sh**

```
#!/bin/bash -x
## TxS 01-2021
## v1.20.1 CKAD
echo "This script is written to work with Ubuntu 18.04"
sleep 3
echo
echo "Disable swap until next reboot"
echo
sudo swapoff -a

echo "Update the local node"
sudo apt-get update && sudo apt-get upgrade -y
echo
echo "Install Docker"
sleep 3

sudo apt-get install -y docker.io
echo
echo "Install kubeadm, kubelet, and kubectl"
sleep 3

sudo sh -c
↪  "echo 'deb http://apt.kubernetes.io/ kubernetes-xenial main' >> /etc/apt/sources.list.d/kubernetes.list"

sudo sh -c "curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add -"

sudo apt-get update

sudo apt-get install -y kubeadm=1.20.1-00 kubelet=1.20.1-00 kubectl=1.20.1-00
```

```sh
sudo apt-mark hold kubelet kubeadm kubectl

echo
echo "Installed - now to get Calico Project network plugin"

## If you are going to use a different plugin you'll want
## to use a different IP address, found in that plugins
## readme file.

sleep 3

sudo kubeadm init --kubernetes-version 1.20.1 --pod-network-cidr 192.168.0.0/16

sleep 5

echo "Running the steps explained at the end of the init output for you"

mkdir -p $HOME/.kube

sleep 2

sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config

sleep 2

sudo chown $(id -u):$(id -g) $HOME/.kube/config

echo "Apply Calico network plugin from ProjectCalico.org"
echo "If you see an error they may have updated the yaml file"
echo "Use a browser, navigate to the site and find the updated file"

kubectl apply -f https://docs.projectcalico.org/manifests/calico.yaml

echo
echo
sleep 3
echo "You should see this node in the output below"
echo "It can take up to a mintue for node to show Ready status"
echo
kubectl get node
echo
echo
echo "Script finished. Move to the next step"
```

3. Run the script as an argument to the **bash** shell. You will need the `kubeadm join` command shown near the end of the output when you add the worker/minion node in a future step. Use the **tee** command to save the output of the script, in case you cannot scroll back to find the `kubeadm join` in the script output. Please note the following is one command and then its output.

   Using **Ubuntu 18** you will be asked questions during the installation. Allow restarts and use the local, installed software if asked during the update, usually option 2.

   Copy files to your home directory first.

```
student@master:~$ cp LFD5461/SOLUTIONS/s_02/k8sMaster.sh .

student@master:~$ bash k8sMaster.sh | tee $HOME/master.out
```

```
<output_omitted>

```

```
Your Kubernetes master has initialized successfully!

      To start using your cluster, you need to run the
      following as a regular user:

  mkdir -p $HOME/.kube
  sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
  sudo chown $(id -u):$(id -g) $HOME/.kube/config

      You should now deploy a pod network to the cluster.
      Run \verb?kubectl apply -f [podnetwork].yaml? with one
      of the options listed at:
      https://kubernetes.io/docs/concepts/cluster-administration/addons/

      You can now join any number of machines by running the
      following on each node as root:

  kubeadm join 10.128.0.3:6443 --token 69rdjq.2x20l2j9ncexy37b
  --discovery-token-ca-cert-hash

sha256:72143e996ef78301191b9a42184124416aebcf0c7f363adf9208f9fa599079bd

<output_omitted>

NAME               STATUS     ROLES                   AGE       VERSION
master             NotReady   control-plane,master    19s       v1.20.1


Script finished. Move to the next step
```

Deploy a Minion Node

4. Open a separate terminal into your **second node**. Having both terminal sessions allows you to monitor the status of the cluster while adding the second node. Change the color or other characteristic of the second terminal to make it visually distinct from the first. This will keep you from running commands on the incorrect instance, which probably won't work.

   Use the previous **wget** command to download the tarball to the second node. Extract the files with **tar** as before. Find and copy the `k8sSecond.sh` file to `student`'s home directory then view it. You should see the same early steps as on the master system.

```
student@worker:~$ more k8sSecond.sh
```

SH **k8sSecond.sh**

```
#!/bin/bash -x
## TxS 01-2021
## CKAD for 1.20.1
##
echo "  This script is written to work with Ubuntu 18.04"
echo
sleep 3
echo "  Disable swap until next reboot"
echo
sudo swapoff -a

echo "  Update the local node"
sleep 2
sudo apt-get update && sudo apt-get upgrade -y
echo
```

```sh
sleep 2

echo "  Install Docker"
sleep 3
sudo apt-get install -y docker.io

echo
echo "  Install kubeadm, kubelet, and kubectl"
sleep 2
sudo sh -c
↪  "echo 'deb http://apt.kubernetes.io/ kubernetes-xenial main' >> /etc/apt/sources.list.d/kubernetes.list"

sudo sh -c "curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add -"

sudo apt-get update

sudo apt-get install -y kubeadm=1.20.1-00 kubelet=1.20.1-00 kubectl=1.20.1-00

sudo apt-mark hold kubelet kubeadm kubectl
echo
echo "  Script finished. You now need the kubeadm join command"
echo "  from the output on the master node"
echo
```

5. Run the script on the **second node**. Again please note you may have questions during the update. Allow daemons to restart and use the local installed version, usually option 2.

```
student@worker:~$ bash k8sSecond.sh
```

```
<output_omitted>
```

6. When the script is done the minion node is ready to join the cluster. The `kubeadm join` statement can be found near the end of the `kubeadm init` output on the master node. It should also be in the file `master.out` as well. Your nodes will use a different IP address and hashes than the example below. You'll need to pre-pend **sudo** to run the script copied from the master node. Also note that some non-Linux operating systems and tools insert extra characters when multi-line samples are copied and pasted. Copying one line at a time solves this issue.

```
student@worker:~$ sudo kubeadm join --token 118c3e.83b49999dc5dc034 \
 10.128.0.3:6443 --discovery-token-ca-cert-hash \
sha256:40aa946e3f53e38271bae24723866f56c86d77efb49aedeb8a70cc189bfe2e1d
```

```
<output_omitted>
```

Configure the Master Node

7. Return to the master node. Install a text editor. While the lab uses **vim**, any text editor such as **emacs** or **nano** will work. Be aware that Windows editors may have issues with special characters. Also install the **bash-completion** package, if not already installed. Use the locally installed version of a package if asked.

```
student@master:~$ sudo apt-get install bash-completion vim -y
```

```
<output_omitted>
```

8. We will configure command line completion and verify both nodes have been added to the cluster. The first command will configure completion in the current shell. The second command will ensure future shells have completion. You may need to exit the shell and log back in for command completion to work without error.

```
student@master:~$ source <(kubectl completion bash)

student@master:~$ echo "source <(kubectl completion bash)" >> $HOME/.bashrc
```

9. Verify that both nodes are part of the cluster. And show a `Ready` state.

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

```
NAME        STATUS   ROLES                    AGE         VERSION
master      Ready    control-plane,master     4m11s       v1.20.1
worker      Ready    <none>                   61s         v1.20.1
```

10. We will use the **kubectl** command for the majority of work with Kubernetes. Review the help output to become familiar with commands options and arguments.

```
student@master:~$ kubectl --help
```

```
kubectl controls the Kubernetes cluster manager.

Find more information at:
  https://kubernetes.io/docs/reference/kubectl/overview/

Basic Commands (Beginner):
  create         Create a resource from a file or from stdin.
  expose         Take a replication controller, service,
 deployment or pod and expose it as a new Kubernetes Service
  run            Run a particular image on the cluster
  set            Set specific features on objects

Basic Commands (Intermediate):
<output_omitted>
```

11. With more than 40 arguments, you can explore each also using the `--help` option. Take a closer look at a few, starting with `taint` for example.

```
student@master:~$ kubectl taint --help
```

```
Update the taints on one or more nodes.

  * A taint consists of a key, value, and effect. As an argument
   here, it is expressed as key=value:effect.
  * The key must begin with a letter or number, and may contain
   letters, numbers, hyphens, dots, and underscores, up to
   253 characters.
  * Optionally, the key can begin with a DNS subdomain prefix
   and a single '/',
like example.com/my-app
<output_omitted>
```

12. By default the master node will not allow general containers to be deployed for security reasons. This is via a `taint`. Only containers which tolerate this taint will be scheduled on this node. As we only have two nodes in our cluster we will remove the taint, allowing containers to be deployed on both nodes. This is not typically done in a production environment for security and resource contention reasons. The following command will remove the taint from all nodes, so you should see one success and one `not found` error. The worker/minion node does not have the taint to begin with. Note the **minus sign** at the end of the command, which removes the preceding value.

```
student@master:~$  kubectl describe nodes | grep -i Taint
```

```
Taints:             node-role.kubernetes.io/master:NoSchedule
Taints:             <node>
```

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

```
node/master untainted
error: taint "node-role.kubernetes.io/master:" not found
```

13. Check that both nodes are without a `Taint`. If they both are without taint the `nodes` should now show as `Ready`. It may take a minute or two for all infrastructure pods to enter `Ready` state, such that the `nodes` will show a `Ready` state.

```
student@master:~$ kubectl describe nodes | grep -i taint
```

```
Taints:             <none>
Taints:             <none>
```

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

```
NAME                 STATUS   ROLES    AGE     VERSION
master               Ready    master   6m1s    v1.20.1
worker               Ready    <none>   5m31s   v1.20.1
```

Exercise 2.3: Create a Basic Pod

1. The smallest unit we directly control with Kubernetes is the pod. We will create a pod by creating a minimal YAML file. First we will get a list of current API objects and their `APIGROUP`. If value is not shown it may not exist, as with `SHORTNAMES`. Note that `pods` does not declare an `APIGROUP`. At the moment this indicates it is part of the stable `v1` group.

```
student@master:~$ kubectl api-resources
```

```
NAME                SHORTNAMES   APIVERSION   NAMESPACED   KIND
bindings                         v1           true         Binding
componentstatuses   cs           v1           false        ComponentStatus
configmaps          cm           v1           true         ConfigMap
endpoints           ep           v1           true         Endpoints
.....
pods                po           v1           true         Pod
....
```

2. From the output we see most are `v1` which is used to denote a stable object. With that information we will add the other three required sections for pods such as `metadata`, with a `name`, and `spec` which declares which container image to use and a `name` for the container. We will create an eight line YAML file. White space and indentation matters. Don't use **Tabs**. There is a `basic.yaml` file available in the tarball, as well as `basic-later.yaml` which shows what the file will become and can be helpful for figuring out indentation.

YAML **basic.yaml**

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: basicpod
spec:
  containers:
  - name: webcont
    image: nginx
```

3. Create the new pod using the recently created YAML file.

```
student@master:~$ kubectl create -f basic.yaml
```

```
pod/basicpod created
```

4. Make sure the pod has been created then use the **describe** sub-command to view the details. Among other values in the output you should be about to find the image and the container name.

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

```
NAME       READY   STATUS    RESTARTS   AGE
basicpod   1/1     Running   0          23s
```

```
student@master:~$ kubectl describe pod basicpod
```

```
Name:                 basicpod
Namespace:            default
Priority:             0
<output_omitted>
```

5. Shut down the pod and verify it is no longer running.

```
student@master:~$ kubectl delete pod basicpod
```

```
pod "basicpod" deleted
```

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

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

6. We will now configure the pod to expose port 80. This configuration does not interact with the container to determine what port to open. We have to know what port the process inside the container is using, in this case port 80 as a web server. Add two lines to the end of the file. Line up the indentation with the `image` declaration.

```
student@master:~$ vim basic.yaml
```

basic.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: basicpod
spec:
  containers:
  - name: webcont
    image: nginx
    ports:                          #<--Add this and following line
    - containerPort: 80
```

7. Create the pod and verify it is running. Use the `-o wide` option to see the internal IP assigned to the pod, as well as `NOMINATED NODE`, which is used by the scheduler and `READINESS GATES` which show if experimental features are enabled. Using **curl** and the pods IP address you should get the default nginx welcome web page.

```
student@master:~$ kubectl create -f basic.yaml
```

```
pod/basicpod created
```

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

```
NAME     READY STATUS  RESTARTS AGE  IP           NODE
  NOMINATED NODE  READINESS GATES
basicpod 1/1   Running 0        9s   192.168.1.3  master
  <none>          <none>
```

```
student@master:~$ curl http://192.168.1.3
```

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

<output_omitted>
```

```
student@master:~$ kubectl delete pod basicpod
```

```
pod "basicpod" deleted
```

8. We will now create a simple service to expose the pod to other nodes and pods in the cluster. The `service` YAML will have the same four sections as a pod, but different `spec` configuration and the addition of a `selector`.

```
student@master:~$ vim basicservice.yaml
```

basicservice.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
    name: basicservice
spec:
  selector:
    type: webserver
  ports:
  - protocol: TCP
    port: 80
```

9. We will also add a `label` to the pod and a `selector` to the service so it knows which object to communicate with.

```
student@master:~$ vim basic.yaml
```

basic.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: basicpod
  labels:                 #<-- Add this line
    type: webserver       #<-- and this line which matches selector
spec:
....
```

10. Create the new pod and service. Verify both have been created.

```
student@master:~$ kubectl create -f basic.yaml
```

```
pod/basicpod created
```

```
student@master:˜$ kubectl create -f basicservice.yaml
```

```
service/basicservice created
```

```
student@master:˜$ kubectl get pod
```

```
NAME       READY   STATUS    RESTARTS   AGE
basicpod   1/1     Running   0          110s
```

```
student@master:˜$ kubectl get svc
```

```
NAME          TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)     AGE
basicservice  ClusterIP   10.96.112.50    <none>        80/TCP      14s
kubernetes    ClusterIP   10.96.0.1       <none>        443/TCP     4h
```

11. Test access to the web server using the `CLUSTER-IP` for the `basicservice`.

```
student@master:˜$ curl http://10.96.112.50
```

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

<output_omitted>
```

12. We will now expose the service to outside the cluster as well. Delete the service, edit the file and add a `type` declaration.

```
student@master:˜$ kubectl delete svc basicservice
```

```
service "basicservice" deleted
```

```
student@master:˜$  vim basicservice.yaml
```

basicservice.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
    name: basicservice
spec:
  selector:
    type: webserver
  type: NodePort       #<--Add this line
  ports:
  - protocol: TCP
    port: 80
```

13. Create the service again. Note there is a different `TYPE` and `CLUSTER-IP` and also a high-numbered port.

```
student@master:˜$  kubectl create -f basicservice.yaml
```

```
service/basicservice created
```

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

```
NAME         TYPE      CLUSTER-IP     EXTERNAL-IP PORT(S)       AGE
basicservice NodePort  10.100.139.155 <none>      80:31514/TCP  3s
kubernetes   ClusterIP 10.96.0.1      <none>      443/TCP       47h
```

14. Using the public IP address of the node and the high port you should be able to test access to the webserver. In the example below the public IP is `35.238.3.83`, yours will be different. The high port will also probably be different. Note that testing from within a GCE or AWS node will not work. Use a local to you terminal or web browser to test.

```
local$ curl http://35.238.3.83:31514
```

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

Exercise 2.4: Multi-Container Pods

Using a single container per pod allows for the most granularity and decoupling. There are still some reasons to deploy multiple containers, sometimes called `composite containers`, in a single pod. The secondary containers can handle logging or enhance the primary, the `sidecar` concept, or acting as a proxy to the outside, the `ambassador` concept, or modifying data to meet an external format such as an `adapter`. All three concepts are secondary containers to perform a function the primary container does not.

1. We will add a second container to the pod to handle logging. Without going into details of how to use **fluentd** we will add a logging container to the exiting pod from its own repository. The second container would act as a `sidecar`. At this state we will just add the second container and verify it is running. In the **Deployment Configuration** chapter we will continue to work on this pod by adding persistent storage and configure **fluentd** via a `configMap`.

   Edit the YAML file and add a **fluentd** container. The dash should line up with the previous container dash. At this point a name and image should be enough to start the second container.

```
student@master:~$ vim basic.yaml
```

basic.yaml

```yaml
....
  containers:
  - name: webcont
    image: nginx
    ports:
    - containerPort: 80
  - name: fdlogger
    image: fluent/fluentd
```

2. Delete and create the pod again. The commands can be typed on a single line, separated by a semicolon. This time you should see `2/2` under the `READY` column. You should also find information on the **fluentd** container inside of the `kubectl describe` output.

```
student@master:~$ kubectl delete pod basicpod ; kubectl create -f basic.yaml
```

```
pod "basicpod" deleted
pod/basicpod created
```

```
student@master:˜$ kubectl get pod
```

```
NAME       READY   STATUS    RESTARTS   AGE
basicpod   2/2     Running   0          2m8s
```

```
student@master:˜$ kubectl describe pod basicpod
```

```
Name:                basicpod
Namespace:           default
Priority:            0
Node:                master/10.128.0.11

....

  fdlogger:
    Container ID:    docker://f0649457217f00175ce9aec35022d0b238b9b....
    Image:           fluent/fluentd

....
```

3. For now shut down the pod. We will use it again in a future exercise.

```
student@master:˜$ kubectl delete pod basicpod
```

```
pod "basicpod" deleted
```

Exercise 2.5: Create a Simple Deployment

Creating a pod does not take advantage of orchestration abilities of Kubernetes. We will now create a `Deployment` which gives us scalability, reliability, and updates.

1. Now run a containerized webserver **nginx**. Use **kubectl create** to create a simple, single replica deployment running the nginx web server. It will create a single pod as we did previously but with new controllers to ensure it runs as well as other features.

```
student@master:˜$  kubectl create deployment firstpod --image=nginx
```

```
deployment.apps/firstpod created
```

2. Verify the new deployment exists and the desired number of pods matches the current number. Using a comma, you can request two resource types at once. The **Tab** key can be helpful. Type enough of the word to be unique and press the **Tab** key, it should complete the word. The deployment should show a number `1` for each value, such that the desired number of pods matches the up-to-date and running number. The pod should show zero restarts.

```
student@master:˜$ kubectl get deployment,pod
```

```
NAME                       READY UP-TO-DATE AVAILABLE AGE
deployment.apps/firstpod   1/1   1          1         2m42s

NAME                           READY STATUS  RESTARTS AGE
pod/firstpod-7d88d7b6cf-lrsbk  1/1   Running 0        2m42s
```

3. View the details of the deployment, then the pod. Work through the output slowly. Knowing what a healthy deployment and looks like can be helpful when troubleshooting issues. Again the **Tab** key can be helpful when using long auto-generated object names. You should be able to type firstpod**Tab** and the name will complete when viewing the pod.

```
student@master:˜$ kubectl describe deployment firstpod
```

```
Name:                firstpod
Namespace:           default
CreationTimestamp:   Wed, 15 Arp 2020 17:17:25 +0000
Labels:              app=firstpod
Annotations:         deployment.kubernetes.io/revision=1
Selector:            app=firstpod
Replicas:            1 desired | 1 updated | 1 total | 1 available....
StrategyType:        RollingUpdate
MinReadySeconds:     0
<output_omitted>
```

```
student@master:~$ kubectl describe pod firstpod-6bb4574d94-rqk76
```

```
Name:               firstpod-6bb4574d94-rqk76
Namespace:          default
Priority:           0
PriorityClassName:  <none>
Node:               master/10.128.0.2
Start Time:         Wed, 15 Apr 2020 17:17:25 +0000
Labels:             pod-template-hash=2660130850
                    app=firstpod
Annotations:        cni.projectcalico.org/podIP: 192.168.200.65/32
Status:             Running
IP:                 192.168.200.65
Controlled By:      ReplicaSet/firstpod-6bb4574d94

<output_omitted>
```

4. Note that the resources are in the default namespace. Get a list of available namespaces.

```
student@master:~$ kubectl get namespaces
```

```
NAME              STATUS     AGE
default           Active     20m
kube-node-lease   Active     20m
kube-public       Active     20m
kube-system       Active     20m
```

5. There are four default namespaces. Look at the pods in the `kube-system` namespace.

```
student@master:~$ kubectl get pod -n kube-system
```

```
NAME                                 READY   STATUS     RESTARTS    AGE
calico-node-5ftrr                    2/2     Running    0           24m
calico-node-f7zrw                    2/2     Running    0           21m
coredns-fb8b8dccf-cmkds              1/1     Running    0           24m
coredns-fb8b8dccf-grltk              1/1     Running    0           24m
etcd-v141-r24p                       1/1     Running    0           23m
<output_omitted>
```

6. Now look at the pods in a namespace that does not exist. Note you do not receive an error.

```
student@master:~$ kubectl get pod -n fakenamespace
```

```
No resources found in fakenamespaces namespace.
```

7. You can also view resources in all namespaces at once. Use the `--all-namespaces` options to select objects in all namespaces at once.

```
student@master:˜$ kubectl get pod --all-namespaces
```

```
NAMESPACE     NAME                            READY   STATUS    RESTARTS   AGE
default       firstpod-69cfdfd8d9-kj6ql       1/1     Running   0          44m
kube-system   calico-node-5ftrr               2/2     Running   0          92m
kube-system   calico-node-f7zrw               2/2     Running   0          89m
kube-system   coredns-fb8b8dccf-cmkds         1/1     Running   0          92m
<output_omitted>
```

8. View several resources at once. Note that most resources have a short name such as `rs` for ReplicaSet, `po` for Pod, `svc` for Service, and `ep` for endpoint. Note the endpoint still exists after we deleted the pod.

```
student@master:˜$ kubectl get deploy,rs,po,svc,ep
```

```
NAME                      READY  UP-TO-DATE  AVAILABLE   AGE
deployment.apps/firstpod  1/1    1           1           4m

NAME                                     DESIRED   CURRENT   READY....
replicaset.apps/firstpod-6bb4574d94-rqk76   1         1         1 ....

NAME                          READY  STATUS    RESTARTS   AGE
pod/firstpod-6bb4574d94-rqk76 1/1    Running   0          4m

NAME                 TYPE       CLUSTER-IP    EXTERNAL-IP PORT(S)      AGE
service/basicservice NodePort   10.108.147.76 <none>      80:31601/TCP 21m
service/kubernetes   ClusterIP  10.96.0.1     <none>      443/TCP      21m

NAME                   ENDPOINTS         AGE
endpoints/basicservice <none>            21m
endpoints/kubernetes   10.128.0.3:6443   21m
```

9. Delete the `ReplicaSet` and view the resources again. Note that the age on the `ReplicaSet` and the pod it controls is now less than a minute of age. The deployment operator started a new `ReplicaSet` operator when we deleted the existing one. The new `ReplicaSet` started another pod when the desired spec did not match the current status.

```
student@master:˜$ kubectl delete rs firstpod-6bb4574d94-rqk76
```

```
replicaset.apps "firstpod-6bb4574d94-rqk76" deleted
```

```
student@master:˜$ kubectl get deployment,rs,po,svc,ep
```

```
NAME                          READY  UP-TO-DATE AVAILABLE AGE
deployment.apps/firstpod  1/1    1          1         7m

NAME                                          DESIRED   CURRENT....
replicaset.apps/firstpod-6bb4574d94-rqk76   1         1     ....

NAME                           READY     STATUS    RESTARTS   AGE
pod/firstpod-7d99ffc75-p9hbw   1/1       Running   0          12s

NAME                 TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
service/kubernetes   ClusterIP   10.96.0.1    <none>        443/TCP   24m

NAME                   ENDPOINTS         AGE
endpoints/kubernetes   10.128.0.2:6443   80m
endpoints/basicservice   <none>            21m
```

10. This time delete the top-level controller. After about 30 seconds for everything to shut down you should only see the cluster service and endpoint remain for the cluster and the service we created.

```
student@master:˜$ kubectl delete deployment firstpod
```

```
deployment.apps "firstpod" deleted
```

```
student@master:˜$ kubectl get deployment,rs,po,svc,ep
```

```
NAME                 TYPE      CLUSTER-IP    EXTERNAL-IP PORT(S)      AGE
service/basicservice NodePort  10.108.147.76 <none>      80:31601/TCP 35m
kubernetes           ClusterIP 10.96.0.1     <none>      443/TCP      24m

NAME                   ENDPOINTS        AGE
endpoints/basicservice <none>           21m
kubernetes             10.128.0.3:6443  24m
```

11. As we won't need it for a while, delete the `basicservice` service as well.

```
student@master:˜$ kubectl delete svc basicservice
```

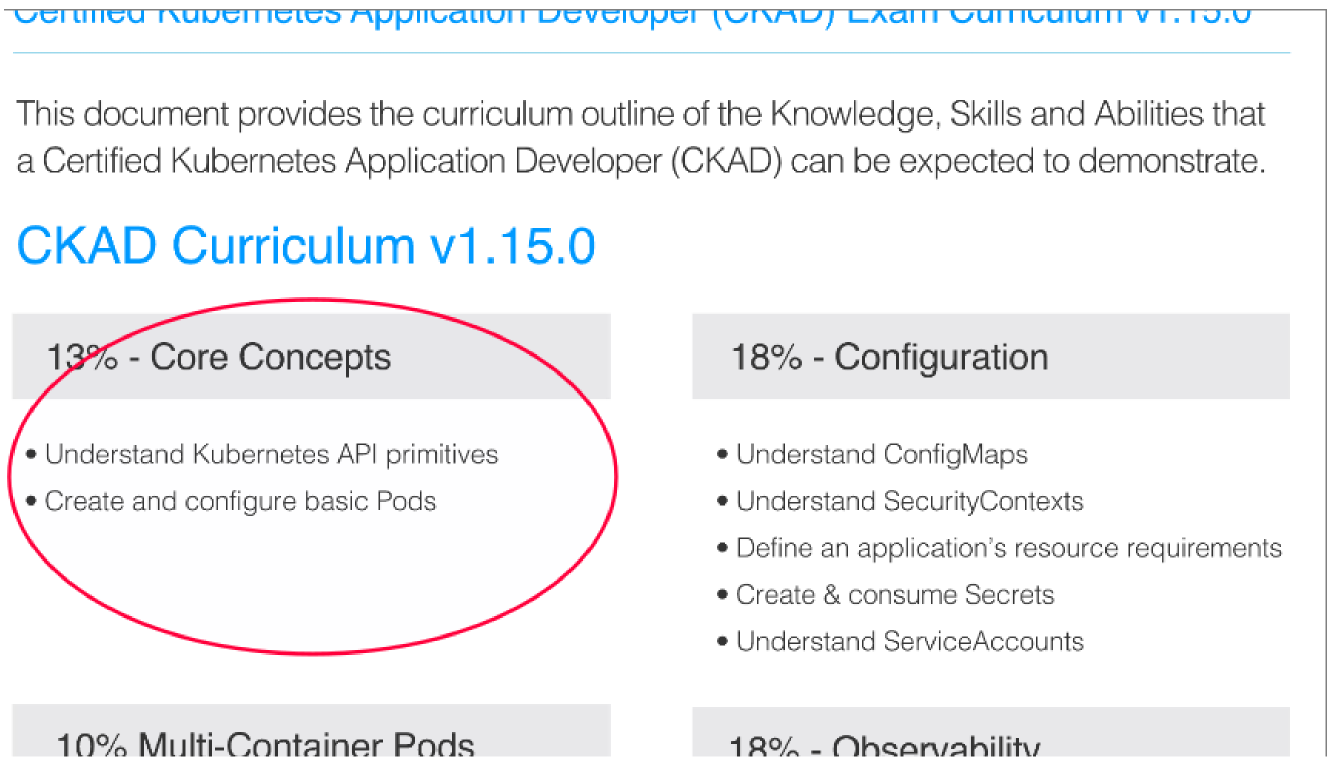
```
service "basicservice" deleted
```

Exercise 2.6: Domain Review

Very Important

The source pages and content in this review could change at any time. **IT IS YOUR RESPONSIBILITY TO CHECK THE CURRENT INFORMATION**.

1. Using a browser go to https://www.cncf.io/certification/ckad/ 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 **CKAD_Curriculum_V1.20.1.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, update, and troubleshoot.
5. Locate the **Core Concepts** section. If you review the lab, you will see we have covered these steps. Again, please note this document will change, distinct from this book. **It remains your responsibility to check for changes in the online document**. They may change on an irregular and unannounced basis.


Certified Kubernetes Application Developer (CKAD) Exam Curriculum v1.15.0

This document provides the curriculum outline of the Knowledge, Skills and Abilities that a Certified Kubernetes Application Developer (CKAD) can be expected to demonstrate.

CKAD Curriculum v1.15.0

13% - Core Concepts

- Understand Kubernetes API primitives
- Create and configure basic Pods

18% - Configuration

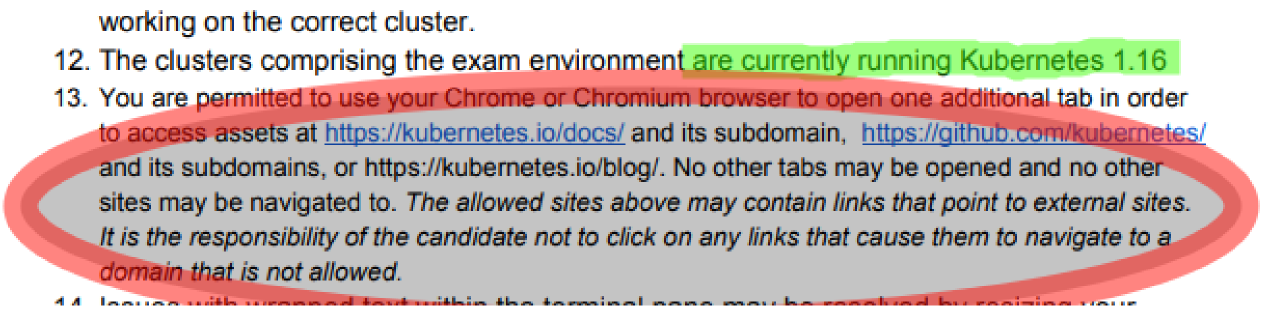
- Understand ConfigMaps
- Understand SecurityContexts
- Define an application's resource requirements
- Create & consume Secrets
- Understand ServiceAccounts

10% Multi-Container Pods

18% - Observability

Figure 2.10: **Core Concepts Domain**

6. Navigate to the Candidate-handbook tab. You are strongly encourage to read and understand this entire document prior to taking the exam. Again, please note this document will change, distinct from this book. **It remains your responsibility to check for changes in the online document**. They may change on an irregular and unannounced basis.
7. Find the **Guidelines and Tips for Use of the Linux server terminal**" section in the document.
8. Among other points you will note the current exam version and three (at the time this was written) domains and subdomains you can use, with some stated conditions.


working on the correct cluster.

12. The clusters comprising the exam environment are currently running Kubernetes 1.16
13. You are permitted to use your Chrome or Chromium browser to open one additional tab in order to access assets at https://kubernetes.io/docs/ and its subdomain, https://github.com/kubernetes/ and its subdomains, or https://kubernetes.io/blog/. No other tabs may be opened and no other sites may be navigated to. *The allowed sites above may contain links that point to external sites. It is the responsibility of the candidate not to click on any links that cause them to navigate to a domain that is not allowed.*

Figure 2.11: **Exam Handbook Guidelines and Tips**

9. Using only the allowed browser, URLs, and subdomains search for and bookmark a YAML example to create and configure a basic pod. Ensure it works for the version of the exam you are taking. URLs may change, plan on checking each book mark prior to taking the exam.
10. Using a timer and bookmarked YAML files see how long it takes you to create and verify. Try it again and see how much faster you can complete and test each step:
    - A new pod with the **nginx** image. Showing all containers running and a `Ready` status.
    - A new service exposing the pod as a **nodePort**, which presents a working webserver configured in the previous step.

- Update the pod to run the **nginx:1.11-alpine** image and re-verify you can view the webserver via a nodePort.

11. Find and use the `architecture-review1.yaml` file included in the course tarball. Your path, such as course number, may be different than the one in the example below. Use the **find** output. Determine if the pod is running. Fix any errors you may encounter. The use of **kubectl describe** may be helpful.

```
student@master:~$ find $HOME -name architecture-review1.yaml
```

```
/home/student/LFD259/SOLUTIONS/s_02/architecture-review1.yaml
```

```
student@master:~$ cp <copy-paste-from-above>  .

student@master:~$ kubectl create -f architecture-review1.yaml
```

12. Remove any pods or services you may have created as part of the review before moving on to the next section. For example:

```
student@master:~$ kubectl delete -f architecture-review1.yaml
```

Chapter 3

Build

In this chapter, we are going to deploy our new Python application into our Kubernetes cluster. We'll first containerize it, using Docker commands, then create and use our own local registry. After testing that the registry is available to all of our nodes, we will deploy our application in a multi-container pod. In keeping with the ideas of decoupled applications that are transient, we will also use readinessProbes, to make sure that the application is fully running prior to accepting traffic from the cluster, and then, livenessProbes, to ensure that the container and the application continues to run in a healthy state, and startupProbes to allow for a slow-starting application.

3.1 Container Options

- There are many competing organizations working with containers. As an orchestration tool, Kubernetes is being developed to work with many of them, with the overall community moving toward open standards and easy interoperability. The early and strong presence of Docker meant that historically, this was not the focus. As Docker evolved, spreading their vendor-lock characteristics through the container creation and deployment life cycle, new projects and features have become popular. As other container engines become mature Kubernetes continues to become more open and independent.

- A **container runtime** is the component which runs the containerized application upon request. Docker Engine remains the default for Kubernetes, though **cri-o** and others are gaining community support.

- The containerized image is moving from Docker to one that is not bound to higher-level tools and that is more portable across operating systems and environments. The **Open Container Initiative** (**OCI**) was formed to help with this. Docker donated their **libcontainer** project to form a new code base called runC to support these goals. More information about runC can be found on GitHub at https://github.com/opencontainers/runc.

- Where Docker was once the only real choice for developers, the trend toward open specifications and flexibility indicates that building with vendor-neutral features is a wise choice.

Docker

- Launched in 2013, Docker has become synonymous with running containerized applications. Docker made containerizing, deploying, and consuming applications easy. As a result, it became the default option in production. With an open registry of images, Docker Hub (https://hub.docker.com/), you can download and deploy vendor or individual-created images on multiple architectures with a single and easy to use tool set. This ease meant it was the sensible default choice for any developer as well. Kubernetes defaults to using the Docker engine to run containers.
- Over the past few years, Docker has continued to grow and add features, including orchestration, which they call `Swarm`. These added features addressed some of the pressing needs in production, but also increased the vendor-lock and size of the product. This has lead to an increase in open tools and specifications such as **cri-o**. A developer looking toward the future would be wise to work with mostly open tools for containers and Kubernetes, but he or she should understand that Docker is still the production tool of choice outside of a Red Hat environment at the moment.

Container Runtime Interface (CRI)

- The goal of the Container Runtime Interface (CRI) is to allow easy integration of container runtimes with kubelet. By providing a protobuf method for API, specifications and libraries, new runtimes can easily be integrated without needing deep understanding of kubelet internals.
- The project is in early stage, with lots of development in action. Now that Docker-CRI integration is done, new runtimes should be easily added and swapped out. At the moment, **cri-o**, **rktlet** and **frakti** are listed as work-in-progress.

rkt

After Red Hat bought CoreOS work on **rkt** stopped, in favor of **cri-o** The **rkt** project has become the first `archived` project of CNCF. While the code is available to be forked and worked on, no one has done so at this point.

- The rkt (https://github.com/rkt/rkt) runtime, pronounced rocket, provides a CLI for running containers. Announced by CoreOS in 2014, it is now part of the Cloud Native Computing Foundation family of projects. Learning from early Docker issues, it is focused on being more secure, open and inter-operable. Many of its features have been met by Docker improvements. It is not quite an easy drop-in replacement for Docker, but progress has been made. **rkt** uses the **appc** specification, and can run Docker, **appc** and **OCI** images. It deploys immutable pods.
- There has been a lot of attention to the project and it was expected to be the leading replacement for Docker until cri-o became part of the official Kubernetes Incubator.

CRI-O

- This project is currently in incubation as part of Kubernetes. It uses the Kubernetes Container Runtime Interface with OCI-compatible runtimes, thus the name CRI-O (https://github.com/kubernetes-sigs/cri-o). Currently, there is support for runC (default) and Clear Containers, but a stated goal of the project is to work with any OCI-compliant runtime.
- While newer than Docker or rkt, this project has gained major vendor support due to its flexibility and compatibility.

Containerd

- The intent of the **containerd** project is not to build a user-facing tool; instead, it is focused on exposing highly-decoupled low-level primitives:
  - Defaults to runC to run containers according to the OCI Specifications
  - Intended to be embedded into larger systems
  - Minimal CLI, focused on debugging and development.
- With a focus on supporting the low-level, or backend plumbing of containers, this project is better suited to integration and operation teams building specialized products, instead of typical build, ship, and run application.

3.2 Containerizing an Application

- To containerize an application, you begin by creating your application. Not all applications do well with containerization. The more stateless and transient the application, the better. Also, remove any environmental configuration, as those can be provided using other tools, like `ConfigMaps` and `secrets`. Develop the application until you have a single build artifact which can be deployed to multiple environments without needing to be changed, using decoupled configuration instead. Many legacy applications become a series of objects and artifacts, residing among multiple containers.
- The use of **Docker** has been the industry standard. Large companies such as **Red Hat** are moving to other open source tools. While currently new one may expect them to become the new standard.
  - **buildah** - This tool is focused on creating Open Container Initiative (OCI) images. It allows for creating images with `and without a Dockerfile`. It also does not require super-user privilege. A growing `golang` based API allows for easy integration with other tools. More can be found here: https://github.com/containers/buildah
  - **podman** - This "pod manager" tool allows for the life cycle management of a container, including creating, starting, stopping, and updating. One could consider this a replacement for `docker run`. More can be found here: https://github.com/containers/libpod

3.3 Creating the Dockerfile

- Create a directory to hold application files. The **docker build** process will pull everything in the directory when it creates the image. Move the scripts and files for the containerized application into the directory.
- Also, in the directory, create a `Dockerfile`. The name is important, as it must be those ten characters, beginning with a capital `D`; newer versions allow a different filename to be used after `-f <filename>`. This is the expected file for the **docker build** to know how to create the image.
- Each instruction is iterated by the Docker daemon in sequence. The instructions are not case-sensitive, but are often written in uppercase to easily distinguish from arguments. This file must have the `FROM` instruction first. This declares the base image for the container. Following can be a collection of `ADD`, `RUN` and a `CMD` to add resources and run commands to populate the image. More details about the Dockerfile reference at https://docs.docker.com/engine/reference/builder can be found in the Docker documentation.
- Test the image by verifying that you can see it listed among other images, and use `docker run <app-name>` to execute it. Once you find the application performs as expected, you can push the image to a local repository in Docker Hub, after creating an account. Here is process summary:
  - Create Dockerfile
  - Build the container
    ```
    sudo docker build -t simpleapp
    ```
  - Verify the image
    ```
    sudo docker images
    sudo docker run simpleapp
    ```
  - Push to the repository
    ```
    sudo docker push
    ```

3.4 Hosting a Local Repository

- While Docker has made it easy to upload images to their Hub, these images are then public and accessible to the world. A common alternative is to create a local repository and push images there. While it can add administrative overhead, it can save downloading bandwidth, while maintaining privacy and security.
- Once the local repository has been configured, you can use **docker tag**, then **push** to populate the repository with local images. Consider setting up an insecure repository until you know it works, then configuring TLS access for greater security.

3.5 Creating a Deployment

- Once you can push and pull images using the **docker** command, try to run a new deployment inside Kubernetes using the image. The string passed to the `--image` argument includes the repository to use, the name of the application, then the version.
- Use **kubectl create** to test the image:

```
kubectl create deployment <Deploy-Name> --image=<repo>/<app-name>:<version>
```

```
kubectl create deployment time-date --image=10.110.186.162:5000/simpleapp:v2.2
```

- Be aware that the string "latest" is just that: a string, and has no reference to actually being the latest. Only use this string if you have a process in place to name and rename versions of the application as they become available. If this is not understood, you could be using the "latest" release, which is several releases older then the most current.
- Verify the Pod shows a running status and that all included containers are running as well.

```
kubectl get pods
```

- Test that the application is performing as expected, running whichever tempest and QA testing the application has. Terminate a pod and test that the application is as transient as designed.

3.6 Running Commands in a Container

- Part of the testing may be to execute commands within the Pod. What commands are available depend on what was included in the base environment when the image was created. In keeping to a decoupled and lean design, it's possible that there is no shell, or that the Bourne shell is available instead of bash. After testing, you may want to revisit the build and add resources necessary for testing and production.
- Use the `-it` options for an interactive shell instead of the command running without interaction or access.
- If you have more than one container, declare which container:

```
kubectl exec -it <Pod-Name> -- /bin/bash
```

3.7 Multi-Container Pod

- It may not make sense to recreate an entire image to add functionality like a shell or logging agent. Instead, you could add another container to the pod, which would provide the necessary tools.
- Each container in the pod should be transient and decoupled. If adding another container limits the scalability or transient nature of the original application, then a new build may be warranted.

- With most network plugins every container in a pod shares a single IP address and namespace. Each container has equal potential access to storage given to the pod. Kubernetes does not provide any locking, so your configuration or application should be such that containers do not have conflicting writes.

  One container could be read only while the other writes, you could configure the containers to write to different directories in the volume, or the application could have built in locking. Without these protections there would be no way to control the order of containers writing to the storage.

- There are three terms often used for multi-container pods, ambassador, adapter, and sidecar. Based off of the documentation you may think there is some setting for each. There is not. Each term is an expression of what a secondary pod is intended to do. All are just multi-container pods.

  – **ambassador** - This type of secondary container would be used to communicate with outside resources, often outside the cluster. Using a proxy, like **Envoy** or other, you can embed a proxy instead of using one provided by the cluster. Helpful if you are unsure of the cluster configuration.
  – **adapter** - This type of secondary container is useful to modify the data the primary container generates. For example the Microsoft version of ASCII is distinct from everyone else. You may need to modify a datastream for proper use.
  – **sidecar** - Similar to a sidecar on a motorcycle, it does not provide the main power, but it does help carry stuff. A sidecar is a secondary container which helps or provides a service not found in the primary application. Logging containers are a common sidecar.

3.8 readinessProbe

- Oftentimes, our application may have to initialize or be configured prior to being ready to accept traffic. As we scale up our application, we may have containers in various states of creation. Rather than communicate with a client prior to being fully ready, we can use a `readinessProbe`. Kubelet will continue to test the container every `periodSeconds` and will not direct a service to send it traffic until the test succeeds.

- With the **exec** statement, the container is not considered ready until a command returns a zero exit code. As long as the return is non-zero, the container is considered not ready and kubelet will continue to test.

- Another type of probe uses an HTTP GET request (`httpGet`). Using a defined header to a particular port and path, the container is not considered healthy until the web server returns a code `200-399`. Any other code indicates failure, and the probe will try again.

- The TCP Socket probe (`tcpSocket`) will attempt to open a socket on a predetermined port, and keep trying based on periodSeconds. Once the port can be opened, the container is considered healthy.

3.9 livenessProbe

- In addition to deciding if a container is available for traffic you may want a container to be restarted if it fails. In this case you can use a `livenessProbe`. It shares the same syntax as a readinessProbe, and only differs in the name.

- Both probes are often configured into a deployment to ensure applications are ready for traffic and are restarted if unhealthy.

3.10 startupProbe

- Becoming beta recently we can also use the `startupProbe`. This probe is useful for testing application which takes a long time to start.

- If kubelet uses a `startupProbe` it will disable liveness and readiness checks until the application passes the test. The duration until the container is considered failed is `failureThreshold` times `periodSeconds`. For example if your `periodSeconds` was set to five seconds, and your `failureThreshold` was set to ten, kubelet would check the application every five seconds until it succeeds or is considered failed after a total of 50 seconds. If you set the `periodSeconds` to 60 seconds, kubelet would keep testing for 300 seconds, or five minutes, before considering the container failed.

3.11 Testing

- With the decoupled and transient nature and great flexibility, there are many possible combinations of deployments. Each deployment would have its own method for testing. No matter which technology is implemented, the goal is the end user getting what is expected. Building a test suite for your newly deployed application will help speed up the development process and limit issues with the Kubernetes integration.
- In addition to overall access, building tools which ensure the distributed application functions properly, especially in a transient environment, is a good idea.
- While custom-built tools may be best at testing a deployment there are some built-in **kubectl** arguments to begin the process. The first is the **describe** and the next would be **logs**

  You can see the details, conditions, volumes and events for an object with **describe**. The `Events` at the end of the output can be helpful during testing as it presents a chronological and node view of cluster actions and associated messages. A simple `nginx` pod may show the following output.

  ```
  kubectl describe pod test1
  ```

  ```
  ....
  Events:
    Type    Reason     Age   From               Message
    ----    ------     ----  ----               -------
    Normal  Scheduled  18s   default-scheduler  Successfully assigned default/test1 to master
    Normal  Pulling    17s   kubelet, master    Pulling image "nginx"
    Normal  Pulled     16s   kubelet, master    Successfully pulled image "nginx"
    Normal  Created    16s   kubelet, master    Created container nginx
    Normal  Started    16s   kubelet, master    Started container nginx
  ```

- A next step in testing may be to look at the output of containers within a pod. Not all applications will generate logs, so it could be difficult to know if the lack of output is due to error or configuration.

  ```
  kubectl logs test1
  ```

  ```

  ```

  A different pod may be configured to show lots of log output, such as the **etcd** pod:

  ```
  kubectl -n kube-system logs etcd-master    ##<-- Use tab to complete for your etcd pod
  ```

  ```
  ....
  2020-01-15 22:08:31.613174 I | mvcc: store.index: compact 446329
  2020-01-15 22:08:31.627117 I | mvcc: finished scheduled compaction at 446329 (took 13.540866ms)
  2020-01-15 22:13:31.622615 I | mvcc: store.index: compact 447019
  2020-01-15 22:13:31.638184 I | mvcc: finished scheduled compaction at 447019 (took 15.122934ms)
  2020-01-15 22:18:31.626845 I | mvcc: store.index: compact 447709
  2020-01-15 22:18:31.640592 I | mvcc: finished scheduled compaction at 447709 (took 13.309975ms)
  2020-01-15 22:23:31.637814 I | mvcc: store.index: compact 448399
  2020-01-15 22:23:31.651318 I | mvcc: finished scheduled compaction at 448399 (took 13.170176ms)
  ```

3.12 Labs

Exercise 3.1: Deploy a New Application

Overview

In this lab we will deploy a very simple **Python** application, test it using Docker, ingest it into Kubernetes and configure probes to ensure it continues to run. This lab requires the completion of the previous lab, the installation and configuration of a Kubernetes cluster.

Working with Python

1. Install python on your master node. It may already be installed, as is shown in the output below.

```
student@master:˜$ sudo apt-get -y install python
```

```
Reading package lists... Done
Building dependency tree
Reading state information... Done
python is already the newest version (2.7.12-1˜16.04).
python set to manually installed.
0 upgraded, 0 newly installed, 0 to remove and 5 not upgraded.
```

2. Locate the python binary on your system.

```
student@master:˜$ which python
```

```
/usr/bin/python
```

3. Create and change into a new directory. The Docker build process pulls everything from the current directory into the image file by default. Make sure the chosen directory is empty.

```
student@master:˜$ mkdir app1
student@master:˜$ cd app1
student@master:˜/app1$ ls -l
```

```
total 0
```

4. Create a simple python script which prints the time and hostname every 5 seconds. There are six commented parts to this script, which should explain what each part is meant to do. The script is included with others in the course tar file, though you are encouraged to create the file by hand if not already familiar with the process. While the command shows **vim** as an example other text editors such as **nano** work just as well.

```
student@master:˜/app1$ vim simple.py
```

simple.py

```
#!/usr/bin/python
## Import the necessary modules
import time
import socket

## Use an ongoing while loop to generate output
while True :

## Set the hostname and the current date
  host = socket.gethostname()
  date = time.strftime("%Y-%m-%d %H:%M:%S")

## Convert the date output to a string
  now = str(date)

## Open the file named date in append mode
## Append the output of hostname and time
```

```
    f = open("date.out", "a" )
    f.write(now + "\n")
    f.write(host + "\n")
    f.close()

  ## Sleep for five seconds then continue the loop
    time.sleep(5)
```

5. Make the file executable and test that it works. Use `Ctrl-C` to interrupt the `while` loop after 20 or 30 seconds. The output will be sent to a newly created file in your current directory called `date.out`.

```
student@master:~/app1$ chmod +x simple.py
student@master:~/app1$ ./simple.py
```

```
^CTraceback (most recent call last):
  File "./simple.py", line 42, in <module>
    time.sleep(5)
KeyboardInterrupt
```

6. and timedate stamps.

```
student@master:~/app1$ cat date.out
```

```
2018-03-22 15:51:38
master
2018-03-22 15:51:43
master
2018-03-22 15:51:48
master
<output_omitted>
```

7. Create a text file named `Dockerfile`.

Very Important

The name is important: it cannot have a suffix.

We will use three statements, `FROM` to declare which version of Python to use, `ADD` to include our script and `CMD` to indicate the action of the container. Should you be including more complex tasks you may need to install extra libraries, shown commented out as `RUN pip install` in the following example.

```
student@master:~/app1$ vim Dockerfile
```

Dockerfile

```
FROM python:2
ADD simple.py /
## RUN pip install pystrich
CMD [ "python", "./simple.py" ]
```

8. Build the container. The output below shows mid-build as necessary software is downloaded. You will need to use **sudo** in order to run this command. After the three step process completes the last line of output should indicate success. Note the dot (.) at the end of the command indicates the current directory.

```
student@master:~/app1$ sudo docker build -t simpleapp .
```

```
Sending build context to Docker daemon 3.072 kB
Step 1/3 : FROM python:2
2: Pulling from library/python
4176fe04cefe: Pull complete
851356ecf618: Pull complete
6115379c7b49: Pull complete
aaf7d781d601: Extracting [================>         ] 54.03 MB/135 MB
40cf661a3cc4: Download complete
c582f0b73e63: Download complete
6c1ea8f72a0d: Download complete
7051a41ae6b7: Download complete
<output_omitted>
Successfully built c4e0679b9c36
```

9. Verify you can see the new image among others downloaded during the build process, installed to support the cluster, or you may have already worked with. The newly created `simpleapp` image should be listed first.

```
student@master:~/app1$ sudo docker images
```

```
REPOSITORY                 TAG      IMAGE ID       CREATED         SIZE
simpleapp                  latest   ba54e4910397   6 seconds ago   902MB
k8s.gcr.io/kube-proxy      v1.20.1  4e68534e24f6   7 days ago      117MB
k8s.gcr.io/kube-apiserver  v1.20.1  a595af0107f9   7 days ago      173MB
<output_omitted>
```

10. Use **sudo docker** to run a container using the new image. While the script is running you won't see any output and the shell will be occupied running the image in the background. After 30 seconds use **ctrl-c** to interrupt. The local `date.out` file will not be updated with new times, instead that output will be a file of the container image.

```
student@master:~$ sudo docker run simpleapp
```

```
^CTraceback (most recent call last):
  File "./simple.py", line 24, in <module>
    time.sleep(5)
KeyboardInterrupt
```

11. Locate the newly created `date.out` file. The following command should show two files of this name, the one created when we ran simple.py and another under `/var/lib/docker` when run via a Docker container.

```
student@master:~/app1$ sudo find / -name date.out
```

```
/home/student/app1/date.out
/var/lib/docker/overlay2/ee814320c900bd24fad0c5db4a258d3c2b78a19cde
629d7de7d27270d6a0c1f5/diff/date.out
```

12. View the contents of the `date.out` file created via Docker. Note the need for **sudo** as Docker created the file this time, and the owner is `root`. The long name is shown on several lines in the example, but would be a single line when typed or copied.

```
student@master:~/app1$ sudo tail \
  /var/lib/docker/overlay2/ee814320c900bd24fad0c5db4a258d3c2b78a19cde
629d7de7d27270d6a0c1f5/diff/date.out
```

```
2018-03-22 16:13:46
53e1093e5d39
2018-03-22 16:13:51
53e1093e5d39
2018-03-22 16:13:56
53e1093e5d39
```

Exercise 3.2: Configure A Local Docker Repo

While we could create an account and upload our application to hub.docker.com, thus sharing it with the world, we will instead create a local repository and make it available to the nodes of our cluster.

1. We'll need to complete a few steps with special permissions, for ease of use we'll become root using **sudo**.

```
student@master:~/app1$ cd
student@master:~$ sudo -i
```

2. Install the **docker-compose** software and utilities to work with the **nginx** server which will be deployed with the registry.

```
root@master:~# apt-get install -y docker-compose apache2-utils
```

```
<output_omitted>
```

3. Create a new directory for configuration information. We'll be placing the repository in the root filesystem. A better location may be chosen in a production environment.

```
root@master:~# mkdir -p /localdocker/data
root@master:~# cd /localdocker/
```

4. Create a Docker compose file. Inside is an entry for the **nginx** web server to handle outside traffic and a registry entry listening to loopback port 5000 for running a local Docker registry.

```
root@master:/localdocker# vim docker-compose.yaml
```

docker-compose.yaml

```yaml
nginx:
  image: "nginx:1.17"
  ports:
    - 443:443
  links:
    - registry:registry
  volumes:
    - /localdocker/nginx/:/etc/nginx/conf.d
registry:
  image: registry:2
  ports:
    - 127.0.0.1:5000:5000
  environment:
    REGISTRY_STORAGE_FILESYSTEM_ROOTDIRECTORY: /data
  volumes:
    - /localdocker/data:/data
```

5. Use the **docker-compose up** command to create the containers declared in the previous step YAML file. This will capture the terminal and run until you use **ctrl-c** to interrupt. There should be five `registry_1` entries with info messages about memory and which port is being listened to. Once we're sure the Docker file works we'll convert to a Kubernetes tool. **Let it run. You will use ctrl-c in a few steps.**

```
root@master:/localdocker# docker-compose up
```

```
Pulling nginx (nginx:1.17)...
1.17: Pulling from library/nginx
2a72cbf407d6: Pull complete
f37cbdc183b2: Pull complete
```

```
78b5ad0b466c: Pull complete
Digest: sha256:edad623fc7210111e8803b4359ba4854e101bcca1fe7f46bd1d35781f4034f0c
Status: Downloaded newer image for nginx:1.17
Creating localdocker_registry_1
Creating localdocker_nginx_1
Attaching to localdocker_registry_1, localdocker_nginx_1
registry_1  | time="2018-03-22T18:32:37Z" level=warning msg="No HTTP secret provided - generated ran
<output_omitted>
```

6. Test that you can access the repository. Open a second terminal to the master node. Use the **curl** command to test the repository. It should return {}, but does not have a carriage-return so will be on the same line as the following prompt. You should also see the GET request in the first, captured terminal, without error. Don't forget the trailing slash. You'll see a "Moved Permanently" message if the path does not match exactly.

```
student@master:~/localdocker$ curl http://127.0.0.1:5000/v2/
```

```
{}student@master:~/localdocker$
```

7. Now that we know **docker-compose** format is working, ingest the file into Kubernetes using **kompose**. Use **ctrl-c** to stop the previous **docker-compose** command.

```
^CGracefully stopping... (press Ctrl+C again to force)
Stopping localdocker_nginx_1 ... done
Stopping localdocker_registry_1 ... done
```

8. Download the kompose binary and make it executable. The command can run on a single line. Note that the option following the dash is the letter as in **o**utput. Also that is a zero, not capital O (ohh) in the short URL. The short URL goes here: https://github.com/kubernetes/kompose/releases/download/v1.1.0/kompose-linux-amd64

```
root@master:/localdocker# curl -L https://bit.ly/2tN0bEa -o kompose
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   609    0   609    0     0   1963      0 --:--:-- --:--:-- --:--:--  1970
100 45.3M  100 45.3M    0     0  16.3M      0  0:00:02  0:00:02 --:--:-- 25.9M
```

```
root@master:/localdocker# chmod +x kompose
```

9. Move the binary to a directory in our $PATH. Then return to your non-root user.

```
root@master:/localdocker# mv ./kompose /usr/local/bin/kompose
root@master:/localdocker# exit
```

10. Create two physical volumes in order to deploy a local registry for Kubernetes. 200Mi for each should be enough for each of the volumes. Use the **hostPath** `storageclass` for the volumes.

More details on how persistent volumes and persistent volume claims are covered in an upcoming chapter, `Deployment Configuration`.

```
student@master:~$ vim vol1.yaml
```

YAML **vol1.yaml**

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  labels:
    type: local
  name: task-pv-volume
spec:
  accessModes:
  - ReadWriteOnce
  capacity:
    storage: 200Mi
  hostPath:
    path: /tmp/data
  persistentVolumeReclaimPolicy: Retain
```

```
student@master:˜$ vim vol2.yaml
```

vol2.yaml

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  labels:
    type: local
  name: registryvm
spec:
  accessModes:
  - ReadWriteOnce
  capacity:
    storage: 200Mi
  hostPath:
    path: /tmp/nginx
  persistentVolumeReclaimPolicy: Retain
```

11. Create both volumes.

```
student@master:˜$ kubectl create -f vol1.yaml
```

```
persistentvolume/task-pv-volume created
```

```
student@master:˜$ kubectl create -f vol2.yaml
```

```
persistentvolume/registryvm created
```

12. Verify both volumes have been created. They should show an `Available` status.

```
student@master:˜$ kubectl get pv
```

```
NAME             CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS
  CLAIM     STORAGECLASS   REASON     AGE
registryvm       200Mi      RWO            Retain           Available
                                   27s
task-pv-volume   200Mi      RWO            Retain           Available
                                   32s
```

13. Go to the configuration file directory for the local Docker registry.

```
student@master:˜$ cd /localdocker/
student@master:˜/localdocker$ ls
```

```
data  docker-compose.yaml  nginx
```

14. Convert the Docker file into a single YAML file for use with Kubernetes. Not all objects convert exactly from Docker to **kompose**, you may get errors about the mount syntax for the new volumes. They can be safely ignored.

```
student@master:~/localdocker$ sudo kompose convert -f docker-compose.yaml -o localregistry.yaml
```

```
WARN Volume mount on the host "/localdocker/nginx/" isn't supported - ignoring path on the host
WARN Volume mount on the host "/localdocker/data" isn't supported - ignoring path on the host
```

15. Review the file. You'll find that multiple Kubernetes objects will have been created such as `Services`, `Persistent Volume Claims` and `Deployments` using environmental parameters and volumes to configure the container within.

```
student@master:/localdocker$ less localregistry.yaml
```

```
apiVersion: v1
items:
- apiVersion: v1
  kind: Service
  metadata:
    annotations:
      kompose.cmd: kompose convert -f docker-compose.yaml -o localregistry.yaml
      kompose.version: 1.1.0 (36652f6)
    creationTimestamp: null
    labels:
<output_omitted>
```

16. View the cluster resources prior to deploying the registry. Only the cluster service and two available persistent volumes should exist in the default namespace.

```
student@master:~/localdocker$ kubectl get pods,svc,pvc,pv,deploy
```

```
NAME           TYPE        CLUSTER-IP    EXTERNAL-IP   PORT(S)   AGE
kubernetes    ClusterIP   10.96.0.1       <none>            443/TCP   4h

NAME                             CAPACITY   ACCESS MODES   RECLAIM POLICY
STATUS      CLAIM     STORAGECLASS   REASON   AGE
persistentvolume/registryvm      200Mi      RWO            Retain
 Available                                  15s
persistentvolume/task-pv-volume 200Mi      RWO            Retain
 Available                                  17s
```

17. To illustrate the fast changing nature of Kubernetes you will show that the API has changed for `Deployments`. With each new release of Kubernetes you may want to plan on a YAML review. First determine new object settings and configuration, then compare and contrast to your existing YAML files. Edit and test for the new configurations.

    Another more common way to find YAML issues is to attempt to create an object using previous YAML in a new version of Kubernetes and track down errors. Not suggested, but what often happens instead of the following process.

    To view the current cluster requirements use the `--dry-run` option for the **kubectl create** command to see what the API now uses. We can compare the current values to our existing (previous version) YAML files. This will help determine what to edit for the local registry in an upcoming step.

Very Important

If you are not familiar with Linux you may want to skip the following couple steps and copy and use the `edited-localregistry.yaml` file included in the course tarball. If so skip to the step which says to: `Use kubectl to create the local docker registry.`

```
student@master:~/localdocker$ kubectl create deployment drytry --image=nginx --dry-run=client -o yaml
```

drytry

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    app: drytry
  name: drytry
spec:
  replicas: 1
  selector:
    matchLabels:
      app: drytry
  strategy: {}
  template:
<output_omitted>
```

18. From the previous command output and comparing line by line to objects in the existing `localregistry.yaml` file output we can see that the `apiVersion` of the `Deployment` object has changed, and we need to add `selector`, add `matchLabels`, and a `label` line. The three lines to add will be part of the replicaSet information, right after the `replicas` line, with `selector` the same indentation as `replicas`.

    Following is a **diff** output, a common way to compare two files to each other, before and after an edit. Use the **man** page to decode the output if you are not already familiar with the command.

```
student@master:~/localdocker$ sudo cp localregistry.yaml old-localregistry.yaml

student@master:~/localdocker$ sudo vim localregistry.yaml
<make edits>

student@master:~/localdocker$ diff localregistry.yaml old-localregistry.yaml


41c41
< - apiVersion: apps/v1
---
> - apiVersion: extensions/v1beta1
53,55d52
<     selector:
<       matchLabels:
<         io.kompose.service: nginx
93c90
< - apiVersion: apps/v1
---
> - apiVersion: extensions/v1beta1
105,107d101
<     selector:
<       matchLabels:
<         io.kompose.service: registry
```

19. Use **kubectl** to create the local docker registry.

```
student@master:~/localdocker$ kubectl create -f localregistry.yaml
```

```
service/nginx created
service/registry created
deployment.apps/nginx created
persistentvolumeclaim/nginx-claim0 created
deployment.apps/registry created
persistentvolumeclaim/registry-claim0 created
```

20. View the newly deployed resources. The persistent volumes should now show as `Bound`. Be aware that due to the manner that volumes are bound it is possible that the registry claim may not to be bound to the registry volume. Find the `service IP` for the registry. It should be sharing port 5000. In the example below the IP address is 10.110.186.162, yours may be different.

```
student@master:~/localdocker$ kubectl get pods,svc,pvc,pv,deploy
```

```
NAME                             READY     STATUS    RESTARTS   AGE
pod/nginx-6b58d9cdfd-95zxq       1/1       Running   0          1m
pod/registry-795c6c8b8f-b8z4k    1/1       Running   0          1m

NAME                   TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE
service/kubernetes     ClusterIP   10.96.0.1        <none>        443/TCP    1h
service/nginx          ClusterIP   10.106.82.218    <none>        443/TCP    1m
service/registry       ClusterIP   10.110.186.162    <none>       5000/TCP   1m

NAME                                    STATUS     VOLUME
 CAPACITY   ACCESS MODES   STORAGECLASS   AGE
persistentvolumeclaim/nginx-claim0      Bound      registryvm
 200Mi      RWO                           1m
persistentvolumeclaim/registry-claim0   Bound      task-pv-volume
 200Mi      RWO                           1m

NAME                             CAPACITY   ACCESS MODES   RECLAIM POLICY
   STATUS    CLAIM    STORAGECLASS   REASON    AGE
persistentvolume/registryvm      200Mi      RWO            Retain
   Bound
default/nginx-claim0                                 5m
persistentvolume/task-pv-volume 200Mi      RWO            Retain
  Bound
default/registry-claim0                              6m

NAME                             READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/nginx      1/1     1            1            12s
deployment.apps/registry   1/1     1            1            12s
```

21. Verify you get the same {} response using the Kubernetes deployed registry as we did when using **docker-compose**. Note you must use the trailing slash after `v2`. Please also note that if the connection hangs it may be due to a firewall issue. If running your nodes using GCE ensure your instances are using VPC setup and all ports are allowed. If using AWS also make sure all ports are being allowed.

    Edit the IP address to that of your `registry` service.

```
student@master:~/localdocker$ curl http://10.110.186.162:5000/v2/
```

```
{}student@master:~/localdocker$
```

22. Edit the Docker configuration file to allow insecure access to the registry. In a production environment steps should be taken to create and use TLS authentication instead. Use the IP and port of the registry you verified in the previous step.

```
student@master:~$ sudo vim /etc/docker/daemon.json
```

```
{ "insecure-registries":["10.110.186.162:5000"] }
```

23. Restart docker on the local system. It can take up to a minute for the restart to take place. Ensure the service is `active`. It should report that the service recently became status as well.

```
student@master:~$ sudo systemctl restart docker.service
student@master:~$ sudo systemctl status docker.service | grep Active
```

```
   Active: active (running) since Tue 2019-09-24 15:24:36 UTC; 40s ago
```

24. Download and tag a typical image from hub.docker.com. Tag the image using the IP and port of the registry. We will also use the `latest` tag.

```
student@master:~$ sudo docker pull ubuntu
```

```
Using default tag: latest
latest: Pulling from library/ubuntu
<output_omitted>
Digest: sha256:9ee3b83bcaa383e5e3b657f042f4034c92cdd50c03f73166c145c9ceaea9ba7c
Status: Downloaded newer image for ubuntu:latest
```

```
student@master:~$ sudo docker tag ubuntu:latest 10.110.186.162:5000/tagtest
```

25. Push the newly tagged image to your local registry. If you receive an error about an HTTP request to an HTTPS client check that you edited the `/etc/docker/daemon.json` file correctly and restarted the service.

```
student@master:~$ sudo docker push 10.110.186.162:5000/tagtest
```

```
The push refers to a repository [10.110.186.162:5000/tagtest]
db584c622b50: Pushed
52a7ea2bb533: Pushed
52f389ea437e: Pushed
88888b9b1b5b: Pushed
a94e0d5a7c40: Pushed
latest: digest: sha256:0847cc7fed1bfafac713b0aa4ddfb8b9199a99092ae1fc4e718cb28e8528f65f size: 1357
```

26. We will test to make sure we can also pull images from our local repository. Begin by removing the local cached images.

```
student@master:~$ sudo docker image remove ubuntu:latest
```

```
Untagged: ubuntu:latest
Untagged: ubuntu@sha256:e348fbbea0e0a0e73ab0370de151e7800684445c509d46195aef73e090a49bd6
```

```
student@master:~$ sudo docker image remove 10.110.186.162:5000/tagtest
```

```
Untagged: 10.110.186.162:5000/tagtest:latest
<output_omitted>
```

27. Pull the image from the local registry. It should report the download of a newer image.

```
student@master:~$ sudo docker pull 10.110.186.162:5000/tagtest
```

```
Using default tag: latest
latest: Pulling from tagtest
Digest: sha256:0847cc7fed1bfafac713b0aa4ddfb8b9199a99092ae1fc4e718cb28e8528f65f
Status: Downloaded newer image for 10.110.186.162:5000/tagtest:latest
```

28. Use docker tag to assign the `simpleapp` image and then push it to the local registry. The image and dependent images should be pushed to the local repository.

```
student@master:~$ sudo docker tag simpleapp 10.110.186.162:5000/simpleapp
student@master:~$ sudo docker push 10.110.186.162:5000/simpleapp
```

```
The push refers to a repository [10.110.186.162:5000/simpleapp]
321938b97e7e: Pushed
ca82a2274c57: Pushed
de2fbb43bd2a: Pushed
4e32c2de91a6: Pushed
6e1b48dc2ccc: Pushed
ff57bdb79ac8: Pushed
6e5e20cbf4a7: Pushed
86985c679800: Pushed
8fad67424c4e: Pushed
latest: digest: sha256:67ea3e11570042e70cdcbad684a1e2986f59aaf53703e51725accdf5c70d475a size: 2218
```

29. Configure the worker (second) node to use the local registry running on the master server. Connect to the worker node. Edit the Docker `daemon.json` file with the same values as the master node and restart the service. Ensure it is `active`.

```
student@worker:~$ sudo vim /etc/docker/daemon.json
```

```
{ "insecure-registries":["10.110.186.162:5000"] }
```

```
student@worker:~$ sudo systemctl restart docker.service
```

```
student@worker:~$ sudo systemctl status docker.service
```

30. From the worker node, pull the recently pushed image from the registry running on the master node.

```
student@worker:~$ sudo docker pull 10.110.186.162:5000/simpleapp
```

```
Using default tag: latest
latest: Pulling from simpleapp
f65523718fc5: Pull complete
1d2dd88bf649: Pull complete
c09558828658: Pull complete
0e1d7c9e6c06: Pull complete
c6b6fe164861: Pull complete
45097146116f: Pull complete
f21f8abae4c4: Pull complete
1c39556edcd0: Pull complete
85c79f0780fa: Pull complete
Digest: sha256:67ea3e11570042e70cdcbad684a1e2986f59aaf53703e51725accdf5c70d475a
Status: Downloaded newer image for 10.110.186.162:5000/simpleapp:latest
```

31. Return to the master node and deploy the `simpleapp` in Kubernetes with several replicas. We will name the deployment `try1`. Scale to have six replicas. Multiple replicas the scheduler should run some containers on each node.

```
student@master:~$ kubectl create deployment try1 --image=10.110.186.162:5000/simpleapp
```

```
deployment.apps/try1 created
```

```
student@master:~$  kubectl scale deployment try1 --replicas=6
```

```
deployment.apps/try1 scaled
```

32. View the running pods. You should see six replicas of `simpleapp` as well as two running the locally hosted image repository.

```
student@master:~$ kubectl get pods
```

```
NAME                       READY     STATUS     RESTARTS   AGE
nginx-6b58d9cdfd-j6jm6     1/1       Running    1          13m
registry-795c6c8b8f-5jnpn  1/1       Running    1          13m
try1-857bdcd888-6klrr      1/1       Running    0          25s
try1-857bdcd888-9pwnp      1/1       Running    0          25s
try1-857bdcd888-9xkth      1/1       Running    0          25s
try1-857bdcd888-tw58z      1/1       Running    0          25s
try1-857bdcd888-xj9lk      1/1       Running    0          25s
try1-857bdcd888-znpm8      1/1       Running    0          25s
```

33. On the `second node` use **sudo docker ps** to verify containers of `simpleapp` are running. The scheduler will usually balance pod count across nodes. As the master already has several pods running the new pods may be on the worker.

```
student@worker:~$ sudo docker ps | grep simple
```

```
3ae4668d71d8     \
  10.110.186.162:5000/simpleapp@sha256:67ea3e11570042e70cdcbad684a1e2986f59aaf53703e51725accdf5c70d475a \
            "python ./simple.py"     48 seconds ago      Up 48 seconds \
             k8s_try1_try1-857bdcd888-9xkth_default_2e94b97e-322a-11e8-af56-42010a800004_0
ef6448764625 \
  10.110.186.162:5000/simpleapp@sha256:67ea3e11570042e70cdcbad684a1e2986f59aaf53703e51725accdf5c70d475a \
            "python ./simple.py"     48 seconds ago      Up 48 seconds \
             k8s_try1_try1-857bdcd888-znpm8_default_2e99f356-322a-11e8-af56-42010a800004_0
```

34. Return to the `master node`. Save the `try1` deployment as YAML.

```
student@master:~/app1$ cd ~/app1/
student@master:~/app1$ kubectl get deployment try1 -o yaml > simpleapp.yaml
```

35. Delete and recreate the `try1` deployment using the YAML file. Verify the deployment is running with the expected six replicas.

```
student@master:~$ kubectl delete deployment try1
```

```
deployment.apps "try1" deleted
```

```
student@master:~/app1$ kubectl create -f simpleapp.yaml
```

```
deployment.apps/try1 created
```

```
student@master:~/app1$ kubectl get deployment
```

```
NAME        READY   UP-TO-DATE   AVAILABLE   AGE
nginx       1/1     1            1           15m
registry    1/1     1            1           15m
try1        6/6     6            6           5s
```

Exercise 3.3: Configure Probes

When large datasets need to be loaded or a complex application launched prior to client access, a `readinessProbe` can be used. The pod will not become available to the cluster until a test is met and returns a successful exit code. Both `readinessProbes` and `livenessProbes` use the same syntax and are identical other than the name. Where the `readinessProbe` is checked prior to being ready, then not again, the `livenessProbe` continues to be checked.

There are three types of liveness probes: a command returns a zero exit value, meaning success, an HTTP request returns a response code in the 200 to 399 range, and the third probe uses a TCP socket. In this example we'll use a command, **cat**, which will return a zero exit code when the file `/tmp/healthy` has been created and can be accessed.

1. Edit the YAML deployment file and add the stanza for a `readinessprobe`. Remember that when working with YAML whitespace matters. Indentation is used to parse where information should be associated within the stanza and the entire file. Do not use tabs. If you get an error about validating data, check the indentation. It can also be helpful to paste the file to this website to see how indentation affects the JSON value, which is actually what Kubernetes ingests: https://www.json2yaml.com/ An edited file is also included in the tarball, but requires the image name to be edited to match your registry IP address.

```
student@master:~/app1$ vim simpleapp.yaml
```

simpleapp.yaml

```yaml
....
    spec:
      containers:
      - image: 10.111.235.60:5000/simpleapp:latest
        imagePullPolicy: Always
        name: simpleapp
        readinessProbe:         #<--This line and next five
          periodSeconds: 5
          exec:
            command:
            - cat
            - /tmp/healthy
        resources: {}
....
```

2. Delete and recreate the `try1` deployment.

```
student@master:~/app1$ kubectl delete deployment try1
```

```
deployment.apps "try1" deleted
```

```
student@master:~/app1$ kubectl create -f simpleapp.yaml
```

```
deployment.apps/try1 created
```

3. The new `try1` deployment should reference six pods, but show zero available. They are all missing the `/tmp/healthy` file.

```
student@master:~/app1$ kubectl get deployment
```

```
NAME       READY   UP-TO-DATE   AVAILABLE   AGE
nginx      1/1     1            1           19m
registry   1/1     1            1           19m
try1       0/6     6            0           15s
```

4. Take a closer look at the pods. Choose one of the `try1` pods as a test to create the health check file.

```
student@master:~/app1$ kubectl get pods
```

```
NAME                          READY     STATUS    RESTARTS   AGE
nginx-6b58d9cdfd-g7lnk        1/1       Running   1          40m
registry-795c6c8b8f-7vwdn     1/1       Running   1          40m
try1-9869bdb88-2wfnr          0/1       Running   0          26s
try1-9869bdb88-6bknl          0/1       Running   0          26s
try1-9869bdb88-786v8          0/1       Running   0          26s
try1-9869bdb88-gmvs4          0/1       Running   0          26s
try1-9869bdb88-lfvlx          0/1       Running   0          26s
try1-9869bdb88-rtchc          0/1       Running   0          26s
```

5. Run the bash shell interactively and touch the `/tmp/healthy` file.

```
student@master:~/app1$ kubectl exec  -it try1-9869bdb88-rtchc -- /bin/bash

root@try1-9869bdb88-rtchc:/# touch /tmp/healthy

root@try1-9869bdb88-rtchc:/# exit
```

```
exit
```

6. Wait at least five seconds, then check the pods again. Once the probe runs again the container should show available quickly. The pod with the existing `/tmp/healthy` file should be running and show `1/1` in a `READY` state. The rest will continue to show `0/1`.

```
student@master:˜/app1$ kubectl get pods
```

```
NAME                           READY     STATUS    RESTARTS   AGE
nginx-6b58d9cdfd-g7lnk         1/1       Running   1          44m
registry-795c6c8b8f-7vwdn      1/1       Running   1          44m
try1-9869bdb88-2wfnr           0/1       Running   0          4m
try1-9869bdb88-6bknl           0/1       Running   0          4m
try1-9869bdb88-786v8           0/1       Running   0          4m
try1-9869bdb88-gmvs4           0/1       Running   0          4m
try1-9869bdb88-lfvlx           0/1       Running   0          4m
try1-9869bdb88-rtchc           1/1       Running   0          4m
```

7. Touch the file in the remaining pods. Consider using a **for** loop, as an easy method to update each pod. Note the >shown in the output represents the secondary prompt, you would not type in that character

```
student@master:˜$ for name in try1-9869bdb88-2wfnr try1-9869bdb88-6bknl \
> try1-9869bdb88-786v8 try1-9869bdb88-gmvs4 try1-9869bdb88-lfvlx
> do
> kubectl exec $name -- touch /tmp/healthy
> done
```

8. It may take a short while for the probes to check for the file and the health checks to succeed.

```
student@master:˜/app1$ kubectl get pods
```

```
NAME                           READY     STATUS    RESTARTS   AGE
nginx-6b58d9cdfd-g7lnk         1/1       Running   1          1h
registry-795c6c8b8f-7vwdn      1/1       Running   1          1h
try1-9869bdb88-2wfnr           1/1       Running   0          22m
try1-9869bdb88-6bknl           1/1       Running   0          22m
try1-9869bdb88-786v8           1/1       Running   0          22m
try1-9869bdb88-gmvs4           1/1       Running   0          22m
try1-9869bdb88-lfvlx           1/1       Running   0          22m
try1-9869bdb88-rtchc           1/1       Running   0          22m
```

9. Now that we know when a pod is healthy, we may want to keep track that it stays healthy, using a `livenessProbe`. You could use one probe to determine when a pod becomes available and a second probe, to a different location, to ensure ongoing health.

   Edit the deployment again. Add in a `livenessProbe` section as seen below. This time we will add a `Sidecar` container to the pod running a `simple` application which will respond to port 8080. Note that the dash (**-**) in front of the name. Also `goproxy` is indented the same number of spaces as the **-** in front of the `image:` line for `simpleapp` earlier in the file. In this example that would be seven spaces

```
student@master:˜/app1$ vim simpleapp.yaml
```

simpleapp.yaml

```
....
         terminationMessagePath: /dev/termination-log
         terminationMessagePolicy: File
       - name: goproxy                #<-- Indented 7 spaces, add lines from here...
```

```yaml
         image: k8s.gcr.io/goproxy:0.1
         ports:
         - containerPort: 8080
         readinessProbe:
           tcpSocket:
             port: 8080
           initialDelaySeconds: 5
           periodSeconds: 10
         livenessProbe:                #<-- This line is 9 spaces indented, fyi
           tcpSocket:
             port: 8080
           initialDelaySeconds: 15
           periodSeconds: 20           #<-- ....to here
       dnsPolicy: ClusterFirst
       restartPolicy: Always
....
```

10. Delete and recreate the deployment.

```
student@master:~$ kubectl delete deployment try1
```

```
deployment.apps "try1" deleted
```

```
student@master:~$ kubectl create -f simpleapp.yaml
```

```
deployment.apps/try1 created
```

11. View the newly created pods. You'll note that there are two containers per pod, and only one is running. The new `simpleapp` containers will not have the `/tmp/healthy` file, so they will not become available until we touch the `/tmp/healthy` file again. We could include a command which creates the file into the container arguments. The output below shows it can take a bit for the old pods to terminate.

```
student@master:~$ kubectl get pods
```

```
NAME                           READY     STATUS              RESTARTS    AGE
nginx-6b58d9cdfd-g7lnk         1/1       Running             1           13h
registry-795c6c8b8f-7vwdn      1/1       Running             1           13h
try1-76cc5ffcc6-4rjvh          1/2       Running             0           3s
try1-76cc5ffcc6-bk5f5          1/2       Running             0           3s
try1-76cc5ffcc6-d8n5q          0/2       ContainerCreating   0           3s
try1-76cc5ffcc6-mm6tw          1/2       Running             0           3s
try1-76cc5ffcc6-r9q5n          1/2       Running             0           3s
try1-76cc5ffcc6-tx4dz          1/2       Running             0           3s
try1-9869bdb88-2wfnr           1/1       Terminating         0           12h
try1-9869bdb88-6bknl           1/1       Terminating         0           12h
try1-9869bdb88-786v8           1/1       Terminating         0           12h
try1-9869bdb88-gmvs4           1/1       Terminating         0           12h
try1-9869bdb88-lfvlx           1/1       Terminating         0           12h
try1-9869bdb88-rtchc           1/1       Terminating         0           12h
```

12. Create the health check file for the `readinessProbe`. You can use a **for** loop again for each action, this setup will leverage labels so you don't have to look up the pod names. As there are now two containers in the pod, you should include the container name for which one will execute the command. If no name is given, it will default to the first container. Depending on how you edited the YAML file `try1` should be the first pod and `goproxy` the second. To ensure the correct container is updated, add **-c simpleapp** to the **kubectl** command. Your pod names will be different. Use the names of the newly started containers from the **kubectl get pods** command output. Note the >character represents the secondary prompt, you would not type in that character.

```
student@master:~$ for name in $(kubectl get pod -l app=try1 -o name)
> do
> kubectl exec $name -c simpleapp -- touch /tmp/healthy
> done
```

13. In the next minute or so the `Sidecar` container in each pod, which was not running, will change status to `Running`. Each should show `2/2` containers running.

```
student@master:~$ kubectl get pods
```

```
NAME                          READY     STATUS            RESTARTS    AGE
nginx-6b58d9cdfd-g7lnk        1/1       Running           1           13h
registry-795c6c8b8f-7vwdn     1/1       Running           1           13h
try1-76cc5ffcc6-4rjvh         2/2       Running           0           3s
try1-76cc5ffcc6-bk5f5         2/2       Running           0           3s
try1-76cc5ffcc6-d8n5q         2/2       Running           0           3s
try1-76cc5ffcc6-mm6tw         2/2       Running           0           3s
try1-76cc5ffcc6-r9q5n         2/2       Running           0           3s
try1-76cc5ffcc6-tx4dz         2/2       Running           0           3s
```

14. View the events for a particular pod. Even though both containers are currently running and the pod is in good shape, note the events section shows the issue.

```
student@master:~/app1$ kubectl describe pod try1-76cc5ffcc6-tx4dz | tail
```

```
  Normal   SuccessfulMountVolume  9m                 kubelet, master-lab-x6dj
MountVolume.SetUp succeeded for volume "default-token-jf69w"
  Normal   Pulling                9m                 kubelet, master-lab-x6dj
pulling image "10.108.143.90:5000/simpleapp"
  Normal   Pulled                 9m                 kubelet, master-lab-x6dj
Successfully pulled image "10.108.143.90:5000/simpleapp"
  Normal   Created                9m                 kubelet, master-lab-x6dj
Created container
  Normal   Started                9m                 kubelet, master-lab-x6dj
Started container
  Normal   Pulling                9m                 kubelet, master-lab-x6dj
pulling image "k8s.gcr.io/goproxy:0.1"
  Normal   Pulled                 9m                 kubelet, master-lab-x6dj
Successfully pulled image "k8s.gcr.io/goproxy:0.1"
  Normal   Created                9m                 kubelet, master-lab-x6dj
Created container
  Normal   Started                9m                 kubelet, master-lab-x6dj
Started container
  Warning  Unhealthy              4m (x60 over 9m)   kubelet, master-lab-x6dj
Readiness probe failed: cat: /tmp/healthy: No such file or directory
```

15. If you look for the status of each container in the pod, they should show that both are `Running` and ready showing `True`.

```
student@master:~/app1$ kubectl describe pod try1-76cc5ffcc6-tx4dz | grep -E 'State|Ready'
```

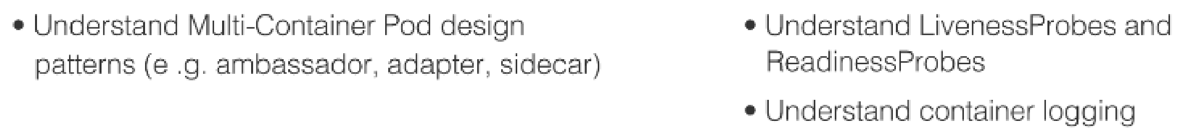
```
    State:          Running
    Ready:          True
    State:          Running
    Ready:          True
  Ready             True
  ContainersReady   True
```

Exercise 3.4: Domain Review

Very Important

The source pages and content in this review could change at any time. **IT IS YOUR RESPONSIBILITY TO CHECK THE CURRENT INFORMATION.**

Revisit the CKAD domain list on Curriculum Overview and locate some of the topics we have covered in this chapter.



- Understand Multi-Container Pod design patterns (e .g. ambassador, adapter, sidecar)
- Understand LivenessProbes and ReadinessProbes
- Understand container logging

Figure 3.1: **Observability Domain**

Focus on ensuing you have all the necessary files and processes understood first. Repeat the review until you are sure you have bookmarks of YAML samples and can complete each step quickly.

1. Using the three URL locations allowed by the exam, find and bookmark working YAML examples for LivenessProbes, ReadinessProbes, and multi-container pods.
2. Deploy a new nginx webserver. Add a LivenessProbe and a ReadinessProbe on port 80. Test that both probes and the webserver work.
3. Use the `build-review1.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.

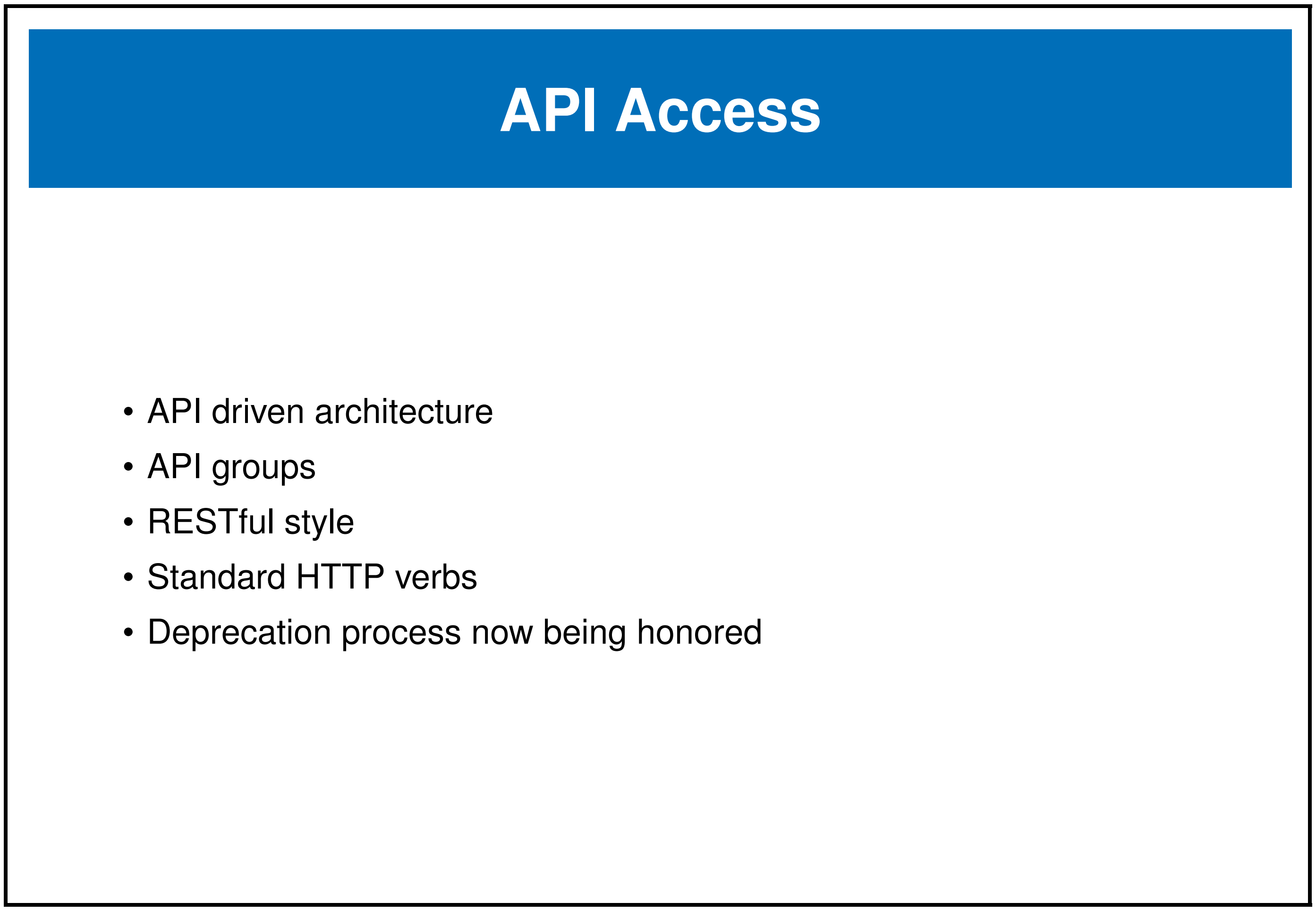
```
192.168.124.0 - - [30/Jan/2020:03:30:31 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.58.0" "-"
```

5. Remove any resources created in this review.

Chapter 4

APIs and Access

4.1 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

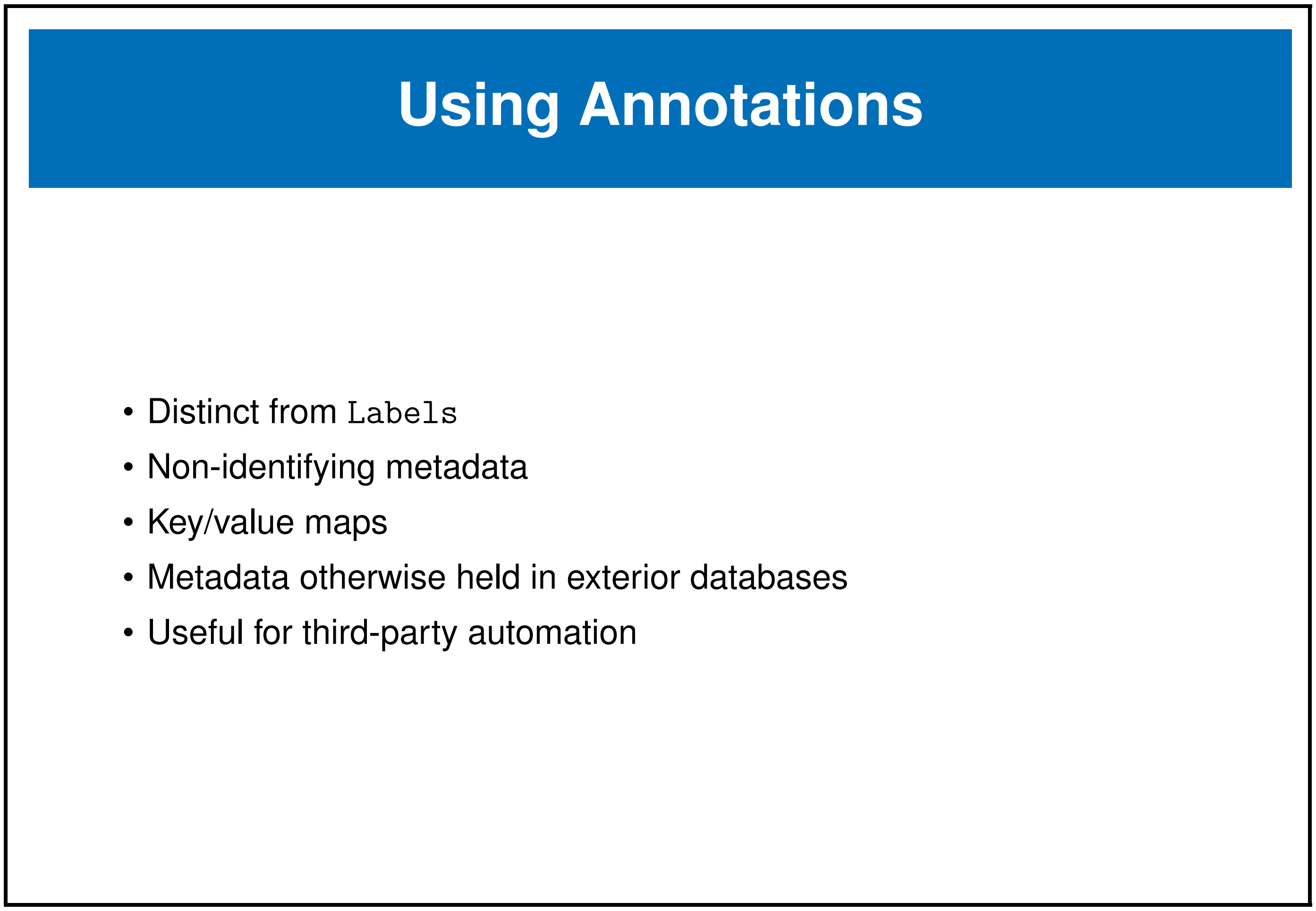
- Currently leverage JSON
- `resourceVersion`
- Clients must handle `409 CONFLICT Errors`

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 `modifedIndex` 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.

4.2 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-
```

4.3 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.

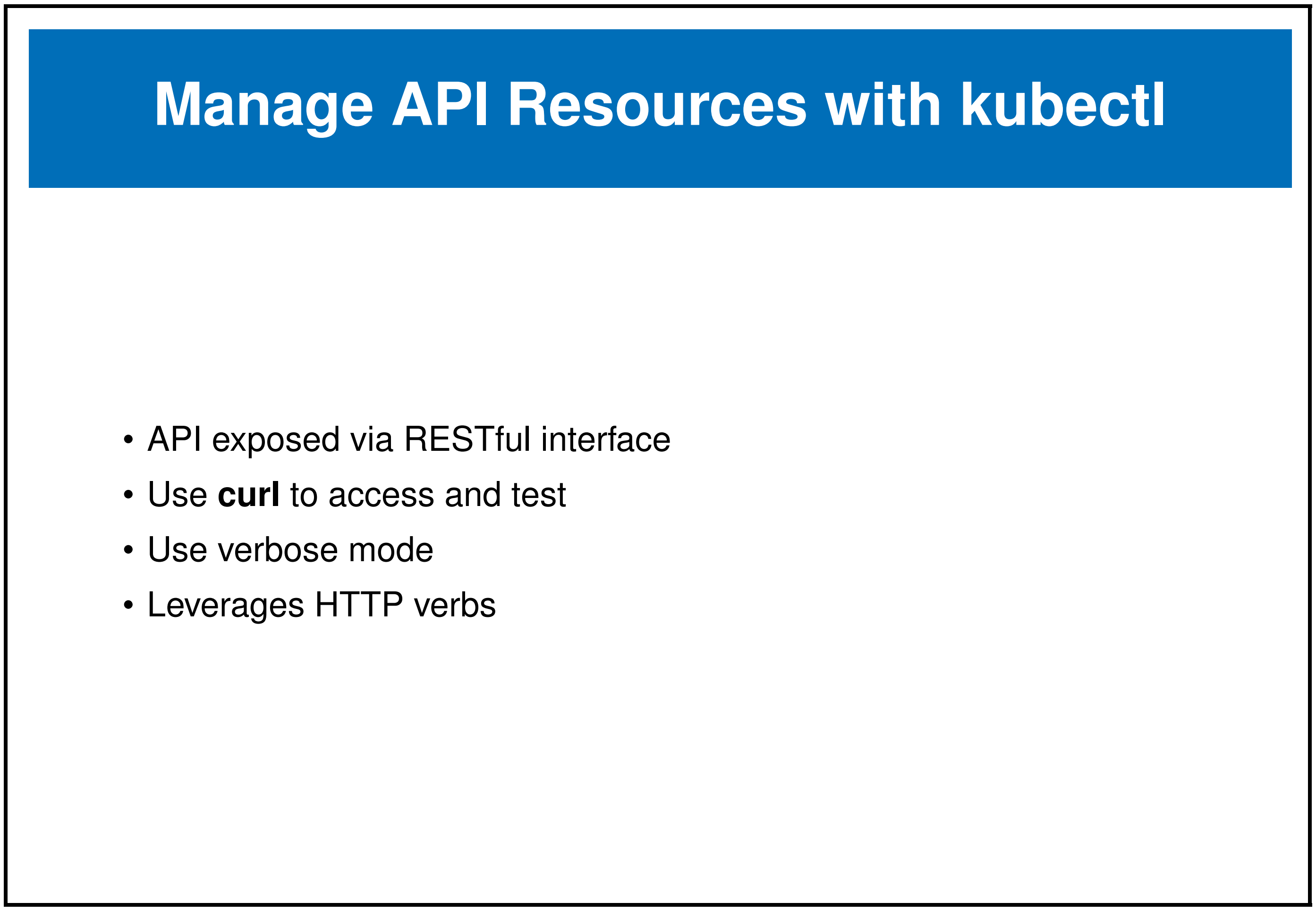
Below is an example of a simple pod manifest in YAML format. You can see the `apiVersion`, the `kind`, the `metadata`, and its `spec`, which define the container that actually runs in this pod:

```yaml
apiVersion: v1
kind: Pod
metadata:
    name: firstpod
spec:
    containers:
    - image: nginx
      name:  stan
```

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
```

4.4 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
....
```

If you delete this pod, you will see that the HTTP method changes from `XGET` to `XDELETE`.

```
$ kubectl --v=10 delete pods firstpod
```

```
....
I1215 17:49:32.166115   30452 round_trippers.go:417]
   curl -k -v -XDELETE  -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 discus later in the course.

Once a namespace has been created you can reference via YAML when creating resource:

```
$ cat redis.yaml
```

redis.yaml

```yaml
apiVersion: V1
kind: Pod
metadata:
    name: redis
    namespace: linuxcon
...
```

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
- replicasets (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
```

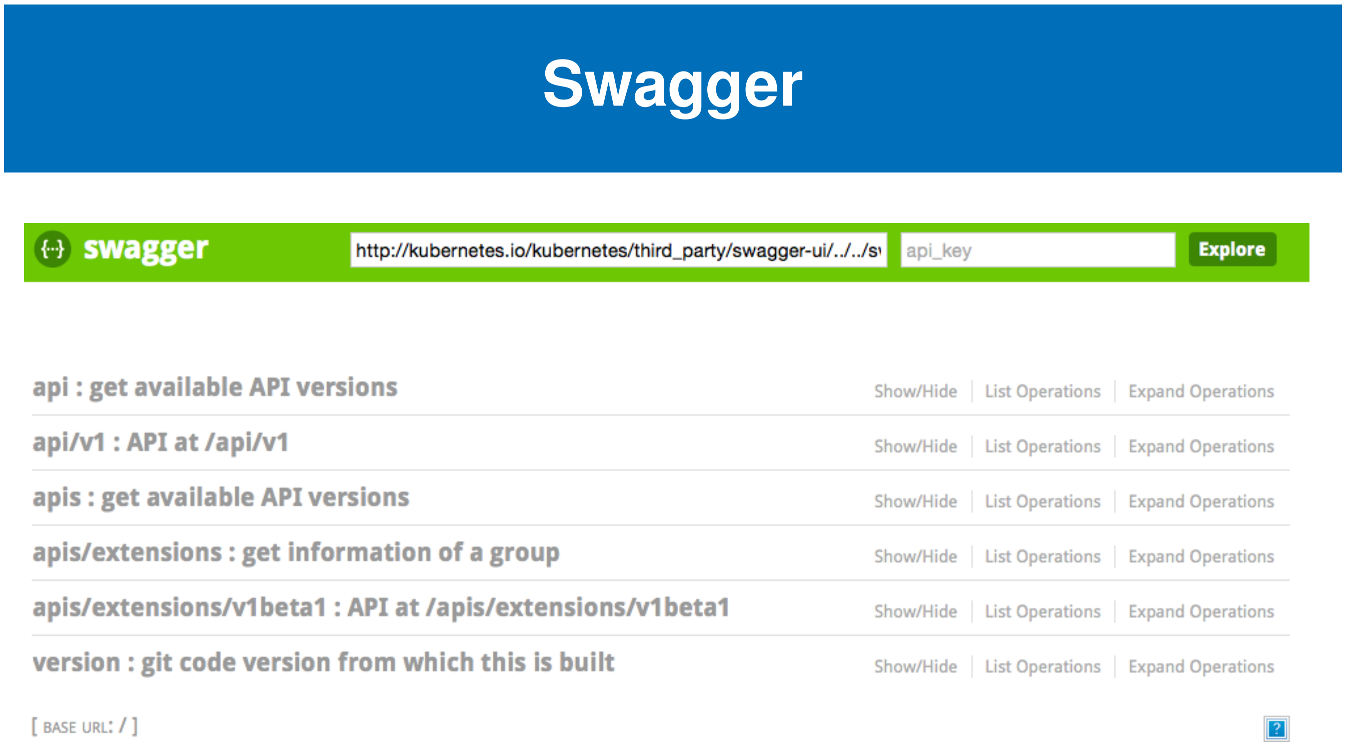
This would be the same as the following. If the container does not have any standard out, there would be no logs.

```
$ 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}
```

4.5 Swagger and OpenAPI


Swagger
swagger
http://kubernetes.io/kubernetes/third_party/swagger-ui/../../sw
api_key
Explore

api : get available API versions — Show/Hide | List Operations | Expand Operations
api/v1 : API at /api/v1 — Show/Hide | List Operations | Expand Operations
apis : get available API versions — Show/Hide | List Operations | Expand Operations
apis/extensions : get information of a group — Show/Hide | List Operations | Expand Operations
apis/extensions/v1beta1 : API at /apis/extensions/v1beta1 — Show/Hide | List Operations | Expand Operations
version : git code version from which this is built — Show/Hide | List Operations | Expand Operations

[BASE URL: /]

Figure 4.1: **Swagger Screenshot**

The entire Kubernetes API uses a **Swagger** specification. This is 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 name, may be buggy and is disabled by default. Features could change or disappear at any time. Only use these features on a test cluster which is often rebuilt.

The `Beta` level, found with `beta` in the name, 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.

4.6 Labs

Exercise 4.1: 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@master:~$ 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@master:~$ export client=$(grep client-cert $HOME/.kube/config |cut -d" " -f 6)

student@master:~$ echo $client
```

```
LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUM4akNDQWRxZ0F3SUJ
BZ0lJRy9wbC9rWEpNdmd3RFFZSktvWklodmNOQVFFTEJRQXdGVEVUTUJFR0
ExVUUKQXhNS2EzVmlaWEp1WlhSbGN6QWVGdzB4TnpFeU1UTXhOelEyTXpKY
UZ3MHhPREV5TVRNeE56UTJNelJhTURReApGekFWQmdOVkJBb1REbk41YzNS
<output_omitted>
```

3. Almost the same command, but this time collect the `client-key-data` as the `key` variable.

```
student@master:~$ export key=$(grep client-key-data $HOME/.kube/config |cut -d " " -f 6)

student@master:~$ echo $key
```

```
<output_omitted>
```

4. Finally set the `auth` variable with the `certificate-authority-data` key.

```
student@master:~$ export auth=$(grep certificate-authority-data $HOME/.kube/config |cut -d " " -f 6)

student@master:~$ echo $auth
```

```
<output_omitted>
```

5. Now encode the keys for use with **curl**.

```
student@master:~$ echo $client | base64 -d - > ./client.pem

student@master:~$ echo $key | base64 -d - > ./client-key.pem

student@master:~$ 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@master:~$ kubectl config view |grep server
```

```
    server: https://k8smaster:6443
```

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@master:~$ curl --cert ./client.pem \
    --key ./client-key.pem \
    --cacert ./ca.pem \
    https://k8smaster: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@master:~$ 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@master:~$ curl --cert ./client.pem \
   --key ./client-key.pem --cacert ./ca.pem \
   https://k8smaster: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@master:~$ kubectl get pods
```

```
NAME       READY     STATUS    RESTARTS    AGE
curlpod    1/1       Running   0           45s
```

Exercise 4.2: 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@master:~$ sudo apt-get install -y strace
```

```
student@master:~$ 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/k8smaster_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@master:~$ strace kubectl get endpoints
```

```
execve("/usr/bin/kubectl", ["kubectl", "get", "endpoints"], [/*....
....
openat(AT_FDCWD, "/home/student/.kube/cache/discovery/k8smaster_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@master:~$ cd /home/student/.kube/cache/discovery/
```

```
student@master:~/.kube/cache/discovery$ ls
```

```
k8smaster_6443
```

```
student@master:~/.kube/cache/discovery$ cd k8smaster_6443/
```

4. View the contents. You will find there are directories with various configuration information for kubernetes.

```
student@master:~/.kube/cache/discovery/k8smaster_6443$ ls
```

```
admissionregistration.k8s.io  certificates.k8s.io          node.k8s.io
apiextensions.k8s.io          coordination.k8s.io          policy
apiregistration.k8s.io        crd.projectcalico.org        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@master:./k8smaster_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
./scheduling.k8s.io
./scheduling.k8s.io/v1beta1
./scheduling.k8s.io/v1beta1/serverresources.json
<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@master:.$ python3 -m json.tool v1/serverresources.json
```

serverresources.json

```
{
    "apiVersion": "v1",
    "groupVersion": "v1",
    "kind": "APIResourceList",
    "resources": [
        {
            "kind": "Binding",
            "name": "bindings",
            "namespaced": true,
            "singularName": "",
            "verbs": [
                "create"
            ]
        },
<output_omitted>
```

7. Some of the objects have `shortNames`, which makes using them on the command line much easier. Locate the `shortName` for endpoints.

```
student@master:.$ python3 -m json.tool v1/serverresources.json | less
```

serverresources.json

```
....
{
"kind": "Endpoints",
"name": "endpoints",
"namespaced": true,
"shortNames": [
      "ep"
],
"singularName": "",
"verbs": [
   "create",
   "delete",
....
```

8. Use the `shortName` to view the endpoints. It should match the output from the previous command.

```
student@master:.$ 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@master:.$ 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@master:$ 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.a

```
student@master:$ kubectl delete po curlpod
```

```
pod "curlpod" deleted
```

12. Take a look around the other files in this directory as time permits.

Chapter 5

API Objects

5.1 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.

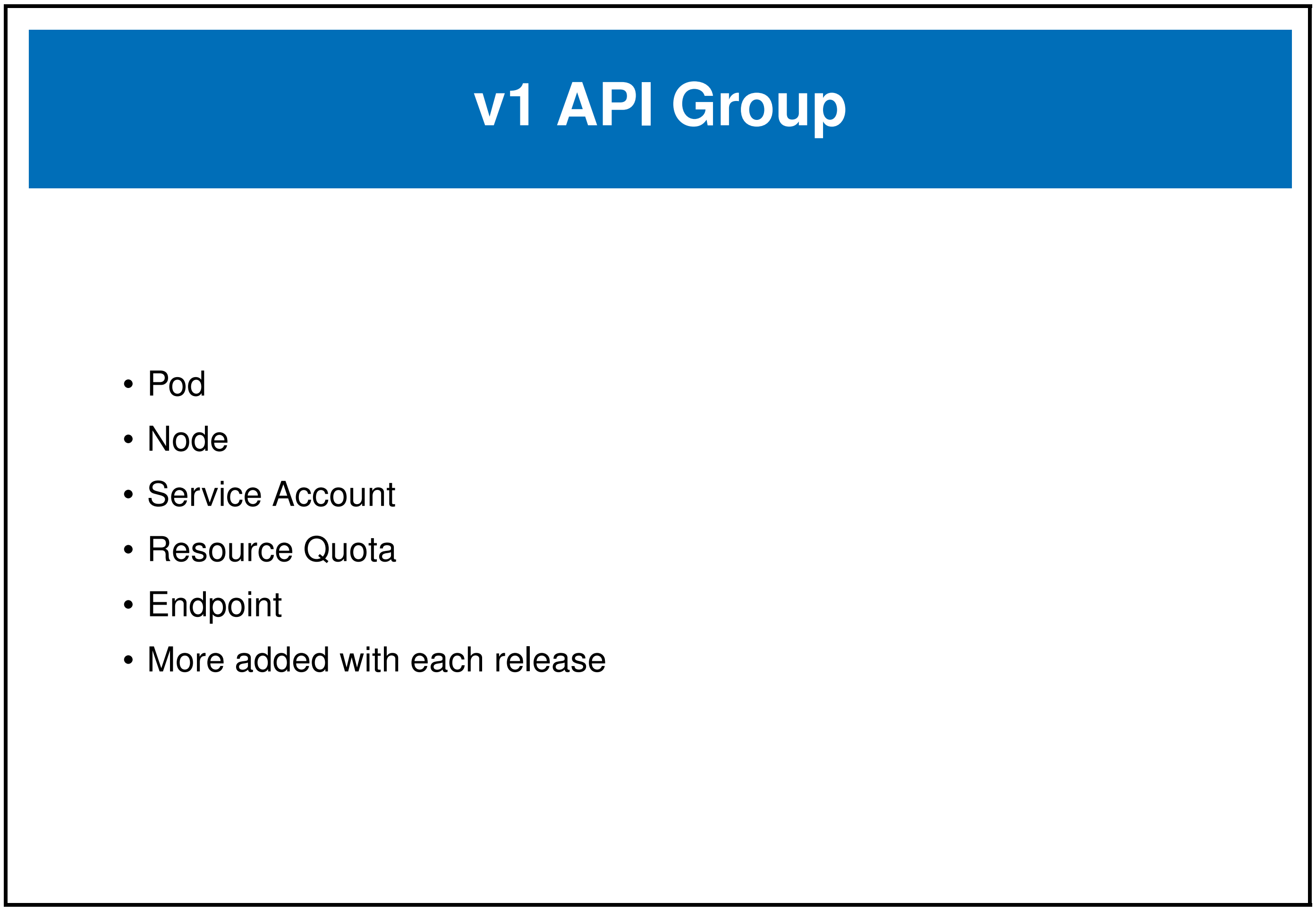
`DaemonSets`, which ensure a Pod on every node, and `StatefulSets`, which stick a container to a node and otherwise act like a deployment, have progressed to `apps/v1` stability.

Role-based Access Control (RBAC), essential to security has made the leap from `v1alpha1` to the stable `v1` status.

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 subdirectories 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.

5.2 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 eight 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

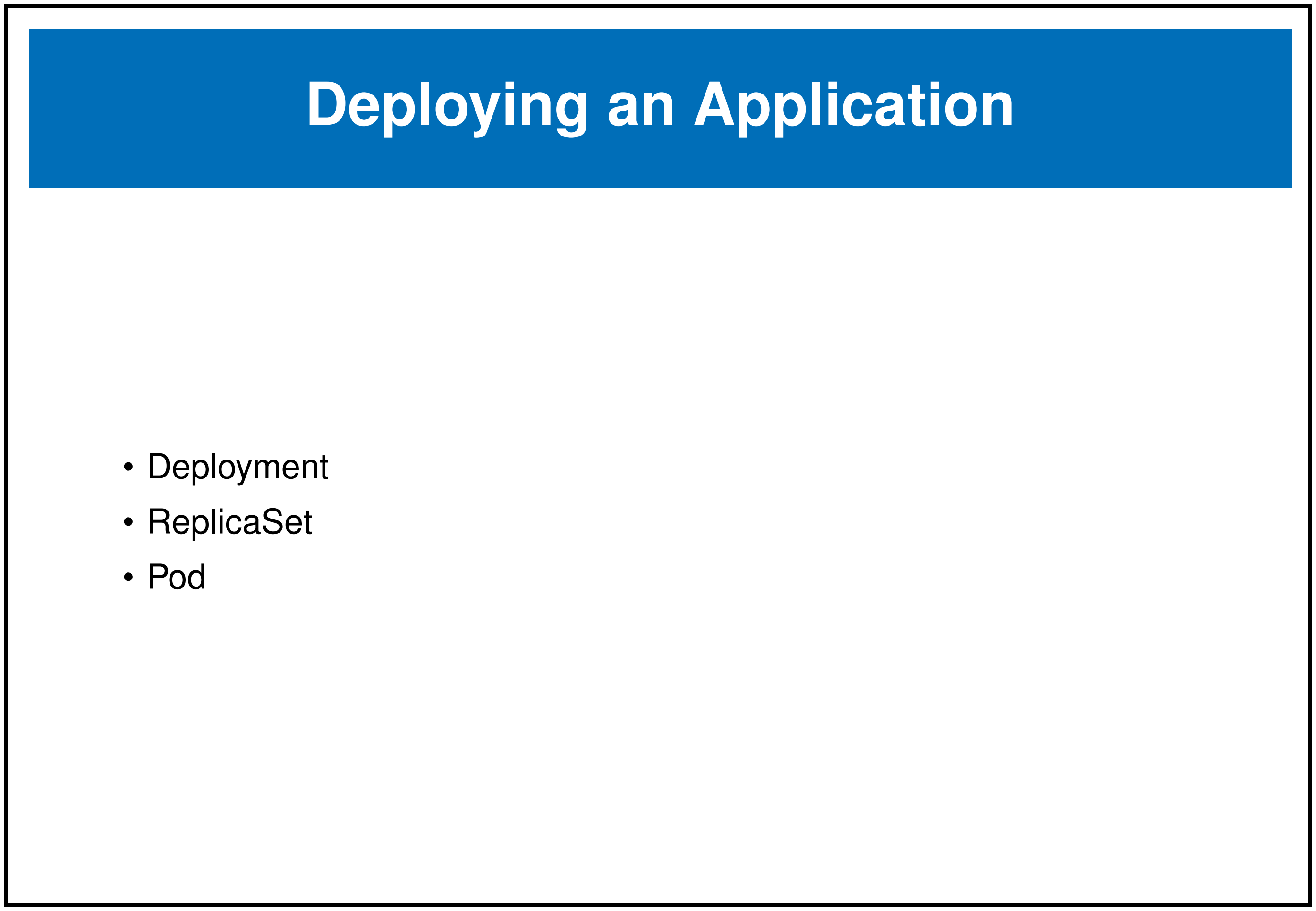
```
$ curl https://localhost:6443/apis \
 --header "Authorization: Bearer $token" -k

{
  "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 URIs and discover additional API objects, their characteristics and associated verbs.

5.3 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
- Stable in v1.9

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 50% CPU usage by default. The usage is checked by `kubelet` every 30 seconds and retrieved by `Metrics Server` API call every minute. HPA checks with `Metrics Server` every 30 seconds. Should a Pod be added or removed HPA waits 180 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 Pod on regular basis
  - Creates a Pod 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.

5.4 RBAC APIs

RBAC

- `rbac.authorization.k8s.io`
- Provide resources
  - `ClusterRole`
  - `ClusterRoleBinding`
  - `RoleBinding`
  - `Role`
- Combinded with quotas for typcial 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.

5.5 Labs

Exercise 5.1: 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.

1. First we need to know the IP and port of a node running a replica of the API server. The master 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@master:~$ kubectl config view
```

```
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: DATA+OMITTED
    server: https://k8smaster:6443
  name: kubernetes
<output_omitted>
```

2. Next we need to find the bearer token. This is part of a default token. Look at a list of tokens, first all on the cluster, then just those in the default namespace. There will be a `secret` for each of the controllers of the cluster.

```
student@master:~$ kubectl get secrets --all-namespaces
```

```
NAMESPACE        NAME                    TYPE                                   ...
default          default-token-jdqp7    kubernetes.io/service-account-token...
kube-node-lease  default-token-j67mt    kubernetes.io/service-account-token...
kube-public      default-token-b2prn    kubernetes.io/service-account-token...
kube-system      attachdetach-controller-token-ckwvh kubernetes.io/servic...
kube-system      bootstrap-signer-token-wpx66 kubernetes.io/service-accou...
<output_omitted>
```

```
student@master:~$ kubectl get secrets
```

```
NAME                  TYPE                                  DATA    AGE
default-token-jdqp7   kubernetes.io/service-account-token   3       23h
```

3. Look at the details of the secret. We will need the `token:` information from the output.

```
student@master:~$ kubectl describe secret default-token-jdqp7
```

```
Name:         default-token-jdqp7
Namespace:    default
Labels:       <none>
<output_omitted>
token:      eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJrdWJlcm5ldGVz
L3NlcnZpY2VhY2NvdW50Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9uYW1lc3Bh
Y2UiOiJkZWZhdWx0Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9zZWNyZXQubm
<output_omitted>
```

4. Using your mouse to cut and paste, or **cut**, or **awk** to save the data, from the first character `eyJh` to the last, to a variable named `token`. Your token data will be different.

```
student@master:˜$ export token=$(kubectl describe \
   secret default-token-jdqp7 |grep ˆtoken |cut -f7 -d ' ')
```

5. 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@master:˜$ curl https://k8smaster: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>
```

6. Try the same command, but look at API v1. Note that the path has changed to `api`.

```
student@master:˜$ curl https://k8smaster:6443/api/v1 \
         --header "Authorization: Bearer $token" -k
```

```
<output_omitted>
```

7. 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@master:˜$ curl \
         https://k8smaster:6443/api/v1/namespaces \
         --header "Authorization: Bearer $token" -k
```

```
<output_omitted>
  "message": "namespaces is forbidden: User \"system:serviceaccount:default...
<output_omitted>
```

8. Pods can also make use of included certificates to use the API. The certificates are automatically made available to a pod under the `/var/run/secrets/kubernetes.io/serviceaccount/`. We will deploy a simple Pod and view the resources. If you view the `token` file you will find it is the same value we put into the `$token` variable. The **-i** will request a **-t** terminal session of the `busybox` container. Once you exit the container will not restart and the pod will show as completed.

```
student@master:˜$ kubectl run -i -t busybox --image=busybox --restart=Never
```

Inside container

```
 # ls /var/run/secrets/kubernetes.io/serviceaccount/
ca.crt namespace token
 # exit
```

9. Clean up by deleting the `busybox` container.

```
student@master:˜$ kubectl delete pod busybox
```

```
pod "busybox" deleted
```

Exercise 5.2: 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@master:~$ 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@master:~$ 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@master:~$ 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@master:~$ curl http://127.0.0.1:8001/api/v1/namespaces

{
  "kind": "NamespaceList",
  "apiVersion": "v1",
  "metadata": {
    "selfLink": "/api/v1/namespaces",
    "resourceVersion": "86902"
<output_omitted>
```

5. 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@master:~$ kill 22500
```

Exercise 5.3: 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

1. Create a job which will run a container which sleeps for three seconds then stops.

```
student@master:~$ vim job.yaml
```

job.yaml

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: sleepy
spec:
  template:
    spec:
      containers:
      - name: resting
        image: busybox
        command: ["/bin/sleep"]
        args: ["3"]
      restartPolicy: Never
```

2. 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@master:~$ kubectl create -f job.yaml
```

```
job.batch/sleepy created
```

```
student@master:~$ kubectl get job
```

```
NAME     COMPLETIONS   DURATION   AGE
sleepy   0/1           3s         3s
```

```
student@master:~$ 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:     Tue, 16 Oct 2018 04:22:50 +0000
Completed At:   Tue, 16 Oct 2018 04:22:55 +0000
Duration:       5s
Pods Statuses:  0 Running / 1 Succeeded / 0 Failed
<output_omitted>
```

```
student@master:~$ kubectl get job
```

```
NAME     COMPLETIONS   DURATION   AGE
sleepy   1/1           5s         17s
```

3. 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@master:~$ 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>
```

4. As the job continues to `AGE` in a completion state, delete the job.

```
student@master:~$ kubectl delete jobs.batch sleepy
```

```
job.batch "sleepy" deleted
```

5. Edit the YAML and add the `completions`: parameter and set it to **5**.

```
student@master:~$ vim job.yaml
```

job.yaml

```yaml
<output_omitted>
metadata:
  name: sleepy
spec:
  completions: 5   #<--Add this line
  template:
    spec:
      containers:
<output_omitted>
```

6. Create the job again. As you view the job note that `COMPLETIONS` begins as zero of **5**.

```
student@master:~$ kubectl create -f job.yaml
```

```
job.batch/sleepy created
```

```
student@master:~$ 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@master:~$ 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@master:~$ kubectl get jobs
```

```
NAME     COMPLETIONS   DURATION   AGE
sleepy   5/5           26s        10m
```

```
student@master:˜$ 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@master:˜$ vim job.yaml
```

job.yaml

```yaml
<output_omitted>
  name: sleepy
spec:
  completions: 5
  parallelism: 2   #<-- Add this line
  template:
    spec:
<output_omitted>
```

10. Create the `job` again. You should see the pods deployed two at a time until all five have completed.

```
student@master:˜$ kubectl create -f job.yaml
```

```
job.batch/sleepy created
```

```
student@master:˜$ 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@master:˜$ 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@master:˜$ vim job.yaml
```

```yaml
<output_omitted>
  completions: 5
  parallelism: 2
  activeDeadlineSeconds: 15   #<-- Add this line
  template:
    spec:
      containers:
      - name: resting
        image: busybox
        command: ["/bin/sleep"]
        args: ["5"]            #<-- Edit this line
<output_omitted>
```

12. Delete and recreate the job again. It should run for 15 seconds, usually 3/5, then continue to age without further completions.

```
student@master:~$ kubectl delete jobs.batch sleepy
```

```
job.batch "sleepy" deleted
```

```
student@master:~$ kubectl create -f job.yaml
```

```
job.batch/sleepy created
```

```
student@master:~$ kubectl get jobs
```

```
NAME     COMPLETIONS   DURATION   AGE
sleepy   1/5           6s         6s
```

```
student@master:~$ 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@master:~$ kubectl get job sleepy -o yaml
```

```
<output_omitted>
status:
  conditions:
  - lastProbeTime: 2018-10-16T05:45:14Z
    lastTransitionTime: 2018-10-16T05:45:14Z
    message: Job was active longer than specified deadline
    reason: DeadlineExceeded
    status: "True"
    type: Failed
  failed: 2
  startTime: 2018-10-16T05:44:59Z
  succeeded: 3
```

14. Delete the job.

```
student@master:~$ 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 `Job` file to a new file.

```
student@master:~$ cp job.yaml cronjob.yaml
```

2. Edit the file to look like the annotated file shown below. Edit the lines mentioned below. The three parameters we added will need to be removed. Other lines will need to be further indented.

```
student@master:~$ vim cronjob.yaml
```

```yaml
apiVersion: batch/v1beta1     #<-- Add beta1 to be v1beta1
kind: CronJob                 #<-- Update this line to CronJob
metadata:
  name: sleepy
spec:
  schedule: "*/2 * * * *"     #<-- 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
            image: busybox
            command: ["/bin/sleep"]
            args: ["5"]
          restartPolicy: Never
```

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@master:~$ kubectl create -f cronjob.yaml
```

```
cronjob.batch/sleepy created
```

```
student@master:~$ kubectl get cronjobs.batch
```

```
NAME     SCHEDULE      SUSPEND   ACTIVE   LAST SCHEDULE   AGE
sleepy   */2 * * * *   False     0        <none>          8s
```

```
student@master:~$ kubectl get jobs.batch
```

```
No resources found.
```

4. After two minutes you should see jobs start to run.

```
student@master:~$ kubectl get cronjobs.batch
```

```
NAME     SCHEDULE      SUSPEND   ACTIVE   LAST SCHEDULE   AGE
sleepy   */2 * * * *   False     0        21s             2m1s
```

```
student@master:~$ kubectl get jobs.batch
```

```
NAME                COMPLETIONS   DURATION   AGE
sleepy-1539722040   1/1           5s         18s
```

```
student@master:~$ 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@master:~$ vim cronjob.yaml
```

```yaml
....
  jobTemplate:
    spec:
      template:
        spec:
          activeDeadlineSeconds: 10  #<-- Add this line
          containers:
          - name: resting
....
        command: ["/bin/sleep"]
        args: ["30"]                 #<-- Edit this line
      restartPolicy: Never
....
```

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@master:~$ kubectl delete cronjobs.batch sleepy
```

```
cronjob.batch "sleepy" deleted
```

```
student@master:~$ kubectl create -f cronjob.yaml
```

```
cronjob.batch/sleepy created
```

```
student@master:~$ kubectl get jobs
```

```
NAME                COMPLETIONS   DURATION   AGE
sleepy-1539723240   0/1           61s        61s
```

```
student@master:~$ kubectl get cronjobs.batch
```

```
NAME     SCHEDULE      SUSPEND   ACTIVE   LAST SCHEDULE   AGE
sleepy   */2 * * * *   False     1        72s             94s
```

```
student@master:~$ kubectl get jobs
```

```
NAME                COMPLETIONS   DURATION   AGE
sleepy-1539723240   0/1           75s        75s
```

```
student@master:~$ kubectl get jobs
```

```
NAME                COMPLETIONS   DURATION   AGE
sleepy-1539723240   0/1           2m19s      2m19s
sleepy-1539723360   0/1           19s        19s
```

```
student@master:~$ 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@master:~$  kubectl delete cronjobs.batch sleepy
```

```
cronjob.batch "sleepy" deleted
```

Chapter 6

Design

In this chapter, we are going to talk about resource requirements for our applications, and the ability to set limits for CPU, memory and storage in the root filesystem of the node. We're also going to talk about containers that can assist our primary application, such as a sidecar, which is typically something that enhances the primary application. For example, if our application does not include logging, we can add a second container to the pod, to handle logging on our behalf. An adapter container might conform the traffic to match the other applications or needs of our cluster. This allows for full flexibility for the primary application, to remain generating the data that it does, and not have to be modified to fit that particular cluster. There is also an ambassador container, which is our representation to the outside world. It can act as a proxy then, so that perhaps I might want to split inbound traffic: if it's a read, it goes to one container, and for write, it goes to a different container, allowing me to optimize my environment and my application. We'll also consider other application design concepts, that can be helpful in the planning stage for the deployment of your application. We'll then finish by talking about jobs, whether it's a batch-like job that will run a container once and be done, or cron jobs, which will run a container on a regular basis on our behalf.

6.1 Traditional Applications: Considerations

- One of the larger hurdles towards implementing Kubernetes in a production environment is the suitability of application design. Optimal Kubernetes deployment design changes are more than just the simple containerization of an application. Traditional applications were built and deployed with the expectation of long-term processes and strong interdependence.

- For example, an Apache web server allows for incredible customization. Often, the server would be configured and tweaked without interruption. As demand grows, the application may be migrated to larger and larger servers. The build and maintenance of the application assumes the instance would run without reset and have persistent and tightly coupled connections to other resources, such as networks and storage.
- In early usage of containers, applications were containerized without redevelopment. This lead to issues after resource failure, or upon upgrade, or configuration. The cost and hassle of redesign and re-implementation should be taken into account.

6.2 Decoupled Resources

- The use of decoupled resources is integral to Kubernetes. Instead of an application using a dedicated port and socket, for the life of the instance, the goal is for each component to be decoupled from other resources. The expectation and software development toward separation allows for each component to be removed, replaced, or rebuilt.
- Instead of hard-coding a resource in an application, an intermediary, such as a Service, enables connection and reconnection to other resources, providing flexibility. A single machine is no longer required to meet the application and user needs; any number of systems could be brought together to meet the needs when, and, as long as, necessary.
- As Kubernetes grows, even more resources are being divided out, which allows for an easier deployment of resources. Also, Kubernetes developers can optimize a particular function with fewer considerations of others objects.

6.3 Transience

- Equally important is the expectation of transience. Each object should be developed with the expectation that other components will die and be rebuilt. With any and all resources planned for transient relationships to others, we can update versions or scale usage in an easy manner.
- An upgrade is perhaps not quite the correct term, as the existing application does not survive. Instead, a controller terminates the container and deploys a new one to replace it, using a different version of the application or setting. Typically, traditional applications were not written this way, opting toward long-term relationships for efficiency and ease of use.

6.4 Flexible Framework

- Like a school of fish, or pod of whales, multiple independent resources working together, but decoupled from each other and without expectation of individual permanent relationship, gain flexibility, higher availability and easy scalability. Instead of a monolithic Apache server, we can deploy a flexible number of **nginx** servers, each handling a small part of the workload. The goal is the same, but the framework of the solution is distinct.
- A decoupled, flexible and transient application and framework is not the most efficient. In order for the Kubernetes orchestration to work, we need a series of agents, otherwise known as controllers or watch-loops, to constantly monitor the current cluster state and make changes until that state matches the declared configuration.
- The commoditization of hardware has enabled the use of many smaller servers to handle a larger workload, instead of single, huge systems.

6.5 Managing Resource Usage

- As with any application, an understanding of resource usage can be helpful for a successful deployment. Kubernetes allows us to easily scale clusters, larger or smaller, to meet demand. An understanding of how the Kubernetes clusters view the resources is an important consideration. The **kube-scheduler**, or a custom scheduler, uses the **PodSpec** to determine the best node for deployment.

- In addition to administrative tasks to grow or shrink the cluster or the number of Pods, there are autoscalers which can add or remove nodes or pods, with plans for one which uses cgroup settings to limit CPU and memory usage by individual containers.
- By default, Pods use as much CPU and memory as the workload requires, behaving and coexisting with other Linux processes. Through the use of resource requests, the scheduler will only schedule a Pod to a node if resources exist to meet all requests on that node. The scheduler takes these and several other factors into account when selecting a node for use.
- Monitoring the resource usage cluster-wide is not an included feature of Kubernetes. Other projects, like Prometheus, are used instead. In order to view resource consumption issues locally, use the `kubectl describe pod` command. You may only know of issues after the pod has been terminated.

CPU

- CPU requests are made in CPU units, each unit being a millicore, using *mille* – the Latin word for thousand. Some documentation uses millicore, others use millicpu, but both have the same meaning. Thus, a request for .7 of a CPU would be 700 millicore. Should a container use more resources than allowed, it won't be killed. The exact amount of overuse is not definite. Note the notation often found in documentation. Each dot would represent a new line and indent if converted to YAML.
  - `spec.containers[].resources.limits.cpu`
  - `spec.containers[].resources.requests.cpu`
- The value of CPUs is not relative. It does not matter how many exist, or if other Pods have requirements. One CPU, in Kubernetes, is equivalent to:
  - 1 AWS vCPU
  - 1 GCP Core
  - 1 Azure vCore
  - 1 Hyperthread on a bare-metal Intel processor with Hyperthreading.

Memory (RAM)

- With Docker engine, the `limits.memory` value is converted to an integer value and becomes the value to the `docker run --memory <value> <image>` command. The handling of a container which exceeds its memory limit is not definite. It may be restarted, or, if it asks for more than the memory request setting, the entire Pod may be evicted from the node.
  - `spec.containers[].resources.limits.memory`
  - `spec.containers[].resources.requests.memory`

Ephemeral Storage (Beta Feature in 1.14)

- Container files, logs, and EmptyDir storage, as well as Kubernetes cluster data, reside on the root filesystem of the host node. As storage is a limited resource, you may need to manage it as other resources. The scheduler will only choose a node with enough space to meet the sum of all the container requests. Should a particular container, or the sum of the containers in a Pod, use more then the limit, the Pod will be evicted.
  - `spec.containers[].resources.limits.ephemeral-storage`
  - `spec.containers[].resources.requests.ephemeral-storage`

6.6 Using Label Selectors

- Labels allow for objects to be selected which may not share other characteristics. For example if a developer were to label their pods using their name they could affect all of their pods, regardless of the application or deployment the pods were using.

- Labels are how operators, also known as watch-loops, track and manage objects. As a result if you were to hard-code the same label for two objects they may be managed by different operators and cause issues. For example one deployment may call for ten pods, while another with the same label calls for five. All the pods would be in a constant state of restarting as each operator tries to start or stop pods until the status matches the spec.

- Consider the possible ways you may want to group your pods and other objects in production. Perhaps you may use `development` and `production` labels to differentiate the state of the application. Perhaps add labels for the department, team, and primary application developer.

- Selectors are namespace scoped. Use the **–all-namespaces** argument to select matching objects in all namespaces.

- The labels, annotations, name, and metadata of an object can be found near the top of **kubectl get ¡obj-name¿ -o yaml**. For example:

```
ckad1$ kubectl get pod examplepod-1234-vtlzd -o yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  annotations:
    cni.projectcalico.org/podIP: 192.168.113.220/32
  creationTimestamp: "2020-01-31T15:13:08Z"
  generateName: examplepod-1234-
  labels:
    app: examplepod
    pod-template-hash: 1234
  name: examplepod-1234-vtlzd
....
```

- Using this output one way we could select the pod would be to use the `app` pod: label. In the command line the colon(:) would be replaced with an equals(=) sign and the space removed. The use of **-l** or **–selector** options can be used with **kubectl**.

```
ckad1$ kubectl -n test2 get --selector app=examplepod pod
```

```
NAME                   READY   STATUS    RESTARTS   AGE
examplepod-1234-vtlzd  1/1     Running   0          25m
```

- There are several built-in object labels. For example nodes have labels such as the `arch`, `hostname`, and `os`, which could be used for assigning pods to a particular node, or type of node.

```
ckad1$ kubectl get node worker
```

```
....
  creationTimestamp: "2020-05-12T14:23:04Z"
  labels:
    beta.kubernetes.io/arch: amd64
    beta.kubernetes.io/os: linux
    kubernetes.io/arch: amd64
    kubernetes.io/hostname: worker
    kubernetes.io/os: linux
  managedFields:
....
```

- The `nodeSelector:` entry in the podspec could use this label to cause a pod to be deployed on a particular node with an entry such as:

```
spec:
  nodeSelector:
    kubernetes.io/hostname: worker
  containers:
```

6.7 Multi-Container Pods

- The idea of multiple containers in a Pod goes against the architectural idea of decoupling as much as possible. One could run an entire operating system inside a container, but would lose much of the granular scalability Kubernetes is capable of. But there are certain needs in which a second or third co-located container makes sense. By adding a second container, each container can still be optimized and developed independently, and both can scale and be repurposed to best meet the needs of the workload.

6.8 Sidecar Container

- The idea for a sidecar container is to add some functionality not present in the main container. Rather than bloating code, which may not be necessary in other deployments, adding a container to handle a function such as logging solves the issue, while remaining decoupled and scalable. Prometheus monitoring and Fluentd logging leverage sidecar containers to collect data.

6.9 Adapter Container

- The basic purpose of an adapter container is to modify data, either on ingress or egress, to match some other need. Perhaps, an existing enterprise-wide monitoring tools has particular data format needs. An adapter would be an efficient way to standardize the output of the main container to be ingested by the monitoring tool, without having to modify the monitor or the containerized application. An adapter container transforms multiple applications to singular view.

6.10 Ambassador

- An ambassador allows for access to the outside world without having to implement a service or another entry in an ingress controller:
  - Proxy local connection
  - Reverse proxy
  - Limits HTTP requests
  - Re-route from the main container to the outside world.
- **Ambassador** (https://www.getambassador.io/ is an *"open source, Kubernetes-native API gateway for microservices built on Enjoy"*.

6.11 Points to Ponder

- Is my application as **decoupled** as it could possibly be?

  Is there anything that I could take out, or make its own container?

  These questions, while essential, often require an examination of the application within the container. Optimal use of Kubernetes is not typically found by containerization of a legacy application and deployment. Rather most applications are rebuilt entirely, with a focus on `decoupled micro-services`.

  When every container can survive any other containers regular termination, without the end-user noticing, you have probably decoupled the application properly.

- Is each container **transient**, does it expect and have code to properly react when other containers are transient?

  Could I run **Chaos Monkey** to kill **ANY** and multiple pods, and my end user would not notice?

  Most Kubernetes deployments are not as transient as they should be, especially among legacy applications. The developer hold on to a previous approach and does not create the containerized application such that every connection and communication is transient and will be terminated. With code waiting for the `service` to connect to a replacement Pod and the containers within.

  Some will note that this is not efficient as it could be. This is correct. We are not optimizing and working against the orchestration tool. Instead we are taking advantage of the decoupled and transient environment to scale and use only the particular micro-service the end user needs.

- Can I scale any particular component to meet workload demand?

  The minimization of an application, breaking it down into the smallest part possible often goes against what developers have spent a career doing. Each division requires more code to handle errors and communication, and is less efficient that a tightly coupled application.

  Machine code is very efficient, for example, but not portable. Instead code is written in a higher level language, which may not be interpreted to run as efficiently, but will run in varied environments. Perhaps approach code meant for Kubernetes in the sense that you are making the highest, and most non-specific way possible. If you have broken down each component then you can **scale** only the most necessary component. In this way the actual workload is more efficient, as the only software consuming CPU cycles is that which the end-user requires. There would be minimal application overhead and waste.

  Ensure you understand how to leverage a **service** to handle networking. Every Pod is given a single IP address, shared by all containers in a pod. The watch-loop for services will manipulate firewall rules to get traffic to pods with matching **labels**, and interact with the network plugin, such as `Calico`.

- Have I used the most open standard stable enough to meet my needs?

  This item can take a while to investigate, but usually a good investment. With tight production schedules some may use what they know best, or what is easiest at the moment. Due to the fast changing nature, or CI/CD, of a production environment this may lead to more work in the long run.

6.12 Jobs

- Just as we may need to redesign our applications to be decoupled we may also consider that microservices may not need to run all the time. The use of `Jobs` and `CronJobs` can further assist with implementing decoupled and transient microservices.
- 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 run 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 a 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.

6.13 Labs

Exercise 6.1: Planning the Deployment

Overview

In this exercise we will investigate common network plugins. Each **kubelet** agent uses one plugin at a time. Due to complexity, the entire cluster uses one plugin which is configured prior to application deployment. Some plugins don't honor security configurations such as network policies. Should you design a deployment which and use a network policy there wouldn't be an error; the policy would have no effect. While developers typically wouldn't care much about the mechanics of it can affect the availability of features and troubleshooting of newly decoupled microservices.

While still new, the community is moving towards the **Container Network Interface** (**CNI**) specification (https://github.com/containernetworking/cni). This provides the most flexibility and features in the fast changing space of container networking.

A common alternative is **kubenet**, a basic plugin which relies on the cloud provider to handle routing and cross-node networking. In a previous lab exercise we configured **Project Calico**. Classic and external modes are also possible. Several software defined network projects intended for Kubernetes have been created recently, with new features added regularly. Be aware that **Calico** is a dynamic project with ongoing and frequent changes.

Evaluate Network Plugins

While developers don't need to configure cluster networking, they may need to understand the plugin in use, and it's particular features and quirks. This section to to ensure you have made a quick review of the most commonly used plugins, and where to find more information as necessary.

1. Verify your nodes are using a CNI plugin. Read through the startup process of CNI. Each message begins with a time stamp, type of message and what is reporting the message.

```
student@master:~$ less /var/log/calico/cni/cni.log
```

```
2020-10-05 16:50:00.960 [INFO][21091] ipam.go 936: Attempting to assign 1 addresses
from block block=192.168.219.64/26 handle="k8s-pod-network.d303bf71efef750a3f420486f
e7b8b9e945abe841ef192673a4435af1c20428c" host="master"
```

2. There are many CNI providers possible. The following list represents some of the more common choices, but it is not exhaustive. With many new plugins being developed there may be another which better serves your needs. Use these websites to answer questions which follow. While we strive to keep the answers accurate, please be aware that this area has a lot of attention and development and changes often.

   - **Project Calico**
     https://docs.projectcalico.org/v3.0/introduction/
   - **Calico with Canal**
     https://docs.projectcalico.org/v3.0/getting-started/kubernetes/installation/hosted/canal
   - **Weave Works**
     https://www.weave.works/docs/net/latest/kubernetes/kube-addon
   - **Flannel**
     https://github.com/coreos/flannel
   - **Romana**
     http://romana.io/how/romana_basics/
   - **Kube Router**
     https://www.kube-router.io
   - **Kopeio**
     https://github.com/kopeio/networking

3. Which of the plugins allow vxlans?

4. Which are layer 2 plugins?

5. Which are layer 3?
6. Which allow network policies?
7. Which can encrypt all TCP and UDP traffic?

Multi-container Pod Considerations

Using the information learned from this chapter, consider the following questions:

1. Which deployment method would allow the most flexibility, multiple applications per pod or one per pod?
2. Which deployment method allows for the most granular scalability?
3. Which have the best performance?
4. How many IP addresses are assigned per pod?
5. What are some ways containers can communicate within the same pod?
6. What are some reasons you should have multiple containers per pod?

Do you really know?

When and why would you use a multi-container pod?

Have you found a YAML example online?

Go back and review multi-container pod types and content on decoupling if you can't easily answer these questions. We touched on adding a second logging and a readiness container in a previous chapter and will work more with logging a future exercise.

Solution 6.1

Plugin Answers

1. Which of the plugins allow vxlans?

   Canal, Project Calico, Flannel, Kopeio-networking, Weave Net

2. Which are layer 2 plugins?

   Canal, Flannel, Kopeio-networking, Weave Net

3. Which are layer 3?

   Project Calico, Romana, Kube Router

4. Which allow network policies?

   Project Calico, Canal, Kube Router, Romana Weave Net

5. Which can encrypt all TCP and UDP traffic?

   Project Calico, Kopeio, Weave Net

Multi Pod Answers

1. Which deployment method would allow the most flexibility, multiple applications per pod or one per Pod?

   One per pod

2. Which deployment method allows for the most granular scalability?

   One per pod

3. Which have the best inter-container performance?

   Multiple per pod.

4. How many IP addresses are assigned per pod?

   One

5. What are some ways containers can communicate within the same pod?

   IPC, loopback or shared filesystem access.

6. What are some reasons you should have multiple containers per pod?

   Lean containers may not have functionality like logging. Able to maintain lean execution but add functionality as necessary, like Ambassadors and Sidecar containers.

Exercise 6.2: Designing Applications With Duration: Create a Job

While most applications 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`

1. Create a job which will run a container which sleeps for three seconds then stops.

```
student@master:~$ vim job.yaml
```

job.yaml

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: sleepy
spec:
  template:
    spec:
      containers:
      - name: resting
        image: busybox
        command: ["/bin/sleep"]
        args: ["3"]
      restartPolicy: Never
```

2. 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@master:~$ kubectl create -f job.yaml
```

```
job.batch/sleepy created
```

```
student@master:~$ kubectl get job
```

```
NAME     COMPLETIONS   DURATION   AGE
sleepy   0/1           3s         3s
```

```
student@master:˜$ 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:     Sun, 03 Nov 2019  04:22:50 +0000
Completed At:   Sun, 03 Nov 2019  04:22:55 +0000
Duration:       5s
Pods Statuses:  0 Running / 1 Succeeded / 0 Failed
<output_omitted>
```

```
student@master:˜$ kubectl get job
```

```
NAME     COMPLETIONS   DURATION   AGE
sleepy   1/1           5s         17s
```

3. 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@master:˜$ 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>
```

4. As the job continues to `AGE` in a completion state, delete the job.

```
student@master:˜$ kubectl delete jobs.batch sleepy
```

```
job.batch "sleepy" deleted
```

5. Edit the YAML and add the `completions:` parameter and set it to `5`.

```
student@master:˜$ vim job.yaml
```

YAML job.yaml

```yaml
<output_omitted>
metadata:
  name: sleepy
spec:
  completions: 5   #<--Add this line
  template:
    spec:
      containers:
<output_omitted>
```

6. Create the job again. As you view the job note that `COMPLETIONS` begins as zero of 5.

```
student@master:~$ kubectl create -f job.yaml
```

```
job.batch/sleepy created
```

```
student@master:~$ 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@master:~$ kubectl get pods
```

```
NAME                          READY   STATUS             RESTARTS   AGE
nginx-67f8fb575f-g4468        1/1     Running            2          2d
registry-56cffc98d6-xlhhf     1/1     Running            1          2d
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@master:~$ kubectl get jobs
```

```
NAME     COMPLETIONS   DURATION   AGE
sleepy   5/5           26s        10m
```

```
student@master:~$ 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@master:~$ vim job.yaml
```

job.yaml

```yaml
<output_omitted>
  name: sleepy
spec:
  completions: 5
  parallelism: 2   #<-- Add this line
  template:
    spec:
<output_omitted>
```

10. Create the `job` again. You should see the pods deployed two at a time until all five have completed.

```
student@master:~$ kubectl create -f job.yaml
```

```
student@master:~$ kubectl get pods
```

```
NAME                          READY   STATUS    RESTARTS   AGE
nginx-67f8fb575f-g4468        1/1     Running   2          2d
registry-56cffc98d6-xlhhf     1/1     Running   1          2d
sleepy-8xwpc                  1/1     Running   0          5s
sleepy-xjqnf                  1/1     Running   0          5s
try1-c9cb54f5d-b45gl          2/2     Running   0          8h
<output_omitted>
```

```
student@master:~$ 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.

```
student@master:~$ vim job.yaml
```

job.yaml

```
<output_omitted>
  completions: 5
  parallelism: 2
  activeDeadlineSeconds: 15   #<-- Add this line
  template:
    spec:
      containers:
      - name: resting
        image: busybox
        command: ["/bin/sleep"]
        args: ["3"]
<output_omitted>
```

12. Delete and recreate the job again. It should run for four times then continue to age without further completions.

```
student@master:~$ kubectl delete jobs.batch sleepy
```

```
job.batch "sleepy" deleted
```

```
student@master:~$ kubectl create -f job.yaml
```

```
job.batch/sleepy created
```

```
student@master:~$ kubectl get jobs
```

```
NAME     COMPLETIONS   DURATION   AGE
sleepy   2/5           6s         6s
```

```
student@master:~$ kubectl get jobs
```

```
NAME     COMPLETIONS   DURATION   AGE
sleepy   4/5           16s        16s
```

13. View the `message:` entry in the `Status` section of the object YAML output. You may see less `status` if the job has yet to run. Wait and try again, if so.

```
student@master:~$ kubectl get job sleepy -o yaml
```

```
<output_omitted>
status:
  conditions:
  - lastProbeTime: "2019-11-03T16:06:10Z"
    lastTransitionTime: "2019-11-03T16:06:10Z"
    message: Job was active longer than specified deadline
    reason: DeadlineExceeded
    status: "True"
    type: Failed
  failed: 1
  startTime: "2019-11-03T16:05:55Z"
  succeeded: 4
```

14. Delete the job.

```
student@master:~$ kubectl delete jobs.batch sleepy
```

```
job.batch "sleepy" deleted
```

Exercise 6.3: Designing Applications With Duration: 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 `Job` file to a new file.

```
student@master:~$ cp job.yaml cronjob.yaml
```

2. Edit the file to look like the annotated file shown below.

```
student@master:~$ vim cronjob.yaml
```

cronjob.yaml

```yaml
apiVersion: batch/v1beta1    #<-- Add beta1 to be v1beta1
kind: CronJob   #<-- Change this line
metadata:
  name: sleepy
spec:                        #<-- Remove completions:, parallelism:, and activeDeadlineSeconds:
  schedule: "*/2 * * * *"    #<-- Add Linux style cronjob syntax
  jobTemplate:               #<-- New jobTemplate and spec
    spec:
      template:   #<-- This and following lines space four to right
        spec:
          containers:
          - name: resting
            image: busybox
            command: ["/bin/sleep"]
            args: ["3"]
          restartPolicy: Never
```

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@master:~$ kubectl create -f cronjob.yaml
```

```
cronjob.batch/sleepy created
```

```
student@master:~$ kubectl get cronjobs.batch
```

```
NAME     SCHEDULE      SUSPEND   ACTIVE   LAST SCHEDULE   AGE
sleepy   */2 * * * *   False     0        <none>          8s
```

```
student@master:~$ kubectl get job
```

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

4. After two minutes you should see jobs start to run.

```
student@master:~$ kubectl get cronjobs.batch
```

```
NAME     SCHEDULE      SUSPEND   ACTIVE   LAST SCHEDULE   AGE
sleepy   */2 * * * *   False     0        21s             2m1s
```

```
student@master:~$ kubectl get jobs.batch
```

```
NAME                COMPLETIONS   DURATION   AGE
sleepy-1539722040   1/1           5s         18s
```

```
student@master:~$ 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@master:~$ vim cronjob.yaml
```

YAML cronjob.yaml

```yaml
....
  jobTemplate:
    spec:
      template:
        spec:
          activeDeadlineSeconds: 10  #<-- Add this line
          containers:
          - name: resting
....
            command: ["/bin/sleep"]
            args: ["30"]             #<-- Edit this line
          restartPolicy: Never
```

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@master:~$ kubectl delete cronjobs.batch sleepy
```

```
cronjob.batch "sleepy" deleted
```

```
student@master:~$ kubectl create -f cronjob.yaml
```

```
cronjob.batch/sleepy created
```

```
student@master:~$ sleep 120 ; kubectl get jobs
```

```
NAME                COMPLETIONS   DURATION   AGE
sleepy-1539723240   0/1           61s        61s
```

```
student@master:~$ kubectl get cronjobs.batch
```

```
NAME     SCHEDULE      SUSPEND   ACTIVE   LAST SCHEDULE   AGE
sleepy   */2 * * * *   False     1        72s             94s
```

```
student@master:~$ kubectl get jobs
```

```
NAME                COMPLETIONS   DURATION   AGE
sleepy-1539723240   0/1           75s        75s
```

```
student@master:~$ kubectl get jobs
```

```
NAME                COMPLETIONS   DURATION   AGE
sleepy-1539723240   0/1           2m19s      2m19s
sleepy-1539723360   0/1           19s        19s
```

```
student@master:~$ 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@master:~$  kubectl delete cronjobs.batch sleepy
```

```
cronjob.batch "sleepy" deleted
```

Exercise 6.4: Using Labels

Create and work with labels. We will understand how the deployment, replicaSet, and pod labels interact.

1. Create a new deployment called `design2`

```
student@master:~$ kubectl create deployment design2 --image=nginx
```

```
deployment.apps/design2 created
```

2. View the wide **kubectl get** output for the `design2` deployment and make note of the `SELECTOR`

```
student@master:~$ kubectl get deployments.apps design2 -o wide
```

```
NAME      READY   UP-TO-DATE   AVAILABLE   AGE     CONTAINERS   IMAGES   SELECTOR
design2   1/1     1            1           2m13s   nginx        nginx    app=design2
```

3. Use the **-l** option to use the selector to list the pods running inside the deployment. There should be only one pod running.

```
student@master:~$ kubectl get -l app=design2 pod
```

```
NAME                        READY   STATUS    RESTARTS   AGE
design2-766d48574f-5w274    1/1     Running   0          3m1s
```

4. View the pod details in YAML format using the deployment selector. This time use the **–selector** option. Find the pod label in the output. It should match that of the deployment.

```
student@master:~$ kubectl get --selector app=design2 pod  -o yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  annotations:
    cni.projectcalico.org/podIP: 192.168.113.222/32
  creationTimestamp: "2020-01-31T16:29:37Z"
  generateName: design2-766d48574f-
  labels:
    app: design2
    pod-template-hash: 766d48574f
....
```

5. Edit the pod label to be your favorite color.

```
student@master:~$ kubectl edit pod design2-766d48574f-5w274
```

```
....
  labels:
    app: orange                      #<<-- Edit this line
    pod-template-hash: 766d48574f
  name: design2-766d48574f-5w274
....
```

6. Now view how many pods are in the deployment. Then how many have `design2` in their name. Note the `AGE` of the pods.

```
student@master:~$ kubectl get deployments.apps design2 -o wide
```

```
NAME     READY  UP-TO-DATE  AVAILABLE  AGE  CONTAINERS  IMAGES  SELECTOR
design2  1/1    1           1          56s  nginx       nginx   app=design2
```

```
student@master:~$ kubectl get pods | grep design2
```

```
design2-766d48574f-5w274   1/1     Running             0          82s
design2-766d48574f-xttgg   1/1     Running             0          2m12s
```

7. Delete the `design2` deployment.

```
student@master:~$ kubectl delete deploy design2
```

```
deployment.apps "design2" deleted
```

8. Check again for pods with `design2` in their names. You should find one pod, with an `AGE` of when you first created the deployment. Once the label was edited the deployment created a new pod in order that the status matches the spec and there be a replica running with the intended label.

```
student@master:~$ kubectl get pods | grep design2
```

```
design2-766d48574f-5w274   1/1     Running             0          38m
```

9. Delete the pod using the **-l** and the label you edited to be your favorite color in a previous step. The command details have been omitted. Use previous steps to figure out these commands.

Exercise 6.5: Setting Pod Resource Limits and Requirements

1. Create a new pod running the `vish/stress` image. A YAML `stress.yaml` file has been included in the course tarball.

2. Run the **top** command on the master and worker nodes. You should find a `stress` command consuming the majority of the CPU on one node, the `worker`. Use **ctrl-c** to exit from top. Delete the deployment.

3. Edit the `stress.yaml` file add in the following limits and requests.

```
student@master:~$
```

```yaml
....
        name: stressmeout
        resources:                 #<<-- Add this and following six lines
          limits:
            cpu: "1"
            memory: "1Gi"
          requests:
            cpu: "0.5"
            memory: "500Mi"
        args:
        - -cpus
....
```

4. Create the deployment again. Check the status of the pod. You should see that it shows an `OOMKilled` status and a growing number of restarts. You may see a status of `Running` if you catch the pod in early in a restart. If you wait long enough you may see `CrashLoopBackOff`.

```
student@master:~$ kubectl get pod stressmeout-7fbbbcc887-v9kvb
```

```
NAME                           READY   STATUS      RESTARTS   AGE
stressmeout-7fbbbcc887-v9kvb   0/1     OOMKilled   2          32s
```

5. Delete then edit the deployment. Change the `limit:` parameters such that pod is able to run, but not too much extra resources. Try setting the memory limit to exactly what the stress command requests.

   As we allow the pod to run on the master node, this could cause issues, such as the `kube-apiserver` restarting due to lack of resources. We will also add a `nodeSelector` to use the built in label of `kubernetes.io/hostname`.

```
student@master:~$ kubectl delete -f stress.yaml
```

```
student@master:~$ vim stress.yaml
```

```yaml
....
    spec:
      nodeSelector:                    #<-- Uncomment and edit
        kubernetes.io/hostname: worker  #<-- to by YOUR worker hostname
      containers:

....
        resources:
          limits:
            cpu: "2"
            memory: "2Gi"
          requests:
....
```

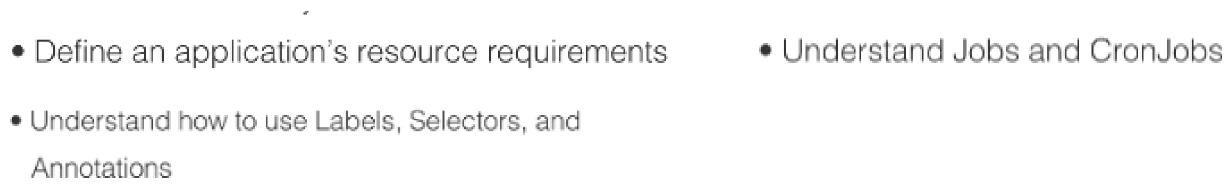
6. Create the deployment and ensure the pod runs without error. Use **top** to verify the stress command is running on one of the nodes and view the pod details to ensure the CPU and memory limits are in use. Also use the **kubectl describe node** command to view the resources your master and worker node are using. The command details have been omitted. Use previous steps to figure out the commands.

Exercise 6.6: Domain Review

Very Important

The source pages and content in this review could change at any time. **IT IS YOUR RESPONSIBILITY TO CHECK THE CURRENT INFORMATION**.

Revisit the CKAD domain list on Curriculum Overview and locate some of the topics we have covered in this chapter. They may be in multiple sections. The graphic below shows the topics covered in this chapter.



- Define an application's resource requirements
- Understand how to use Labels, Selectors, and Annotations
- Understand Jobs and CronJobs

Figure 6.1: **Multiple Domain**

Focus on ensuing you have all the necessary files and processes understood first. Repeat the review until you are sure you have bookmarks of necessary YAML samples and can complete each step quickly, and ensure each object is running properly.

1. Find and use the `design-review1.yaml` file to create a pod.
2. Determine the CPU and memory resource requirements of `design-pod1`.
3. Edit the pod resource requirements such that the CPU limit is exactly twice the amount requested by the container. (Hint: subtract .22)
4. 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.
5. Use the `design-review2.yaml` file to create several pods with various labels.
6. 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.
7. 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.
8. 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.

Chapter 7

Deployment Configuration

In this chapter, we are going to cover some of the typical tasks necessary for full application deployment inside of Kubernetes. We will begin by talking about attaching storage to our containers. By default, storage would only last as long as the container using it. Through the use of Persistent Volumes and Persistent Volume Claims, we can attach storage such that it lives longer than the container and/or the pod the container runs inside of. We'll cover adding dynamic storage, which might be made available from a cloud provider, and we will talk about ConfigMaps and secrets. These are ways of attaching configuration information into our containers in a flexible manner. They are ingested in almost the exact same way. The difference between them, currently, is a secret is encoded. There are plans for it to be encrypted in the future. We will also talk about how do we update our applications. We can do a rolling update, where there's always some containers available for response to client requests, or update all of them at once. We'll also cover how to roll back an application to a previous version, looking at our entire history of updates, and choosing a particular one.

7.1 Volumes Overview

- 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 at least the Pod lifetime, not the containers within. Should a container terminate, the data would continue to be available to the new container. A volume can persist longer than a pod, and can be accessed by multiple pods, using `PersistenVolumeClaims` This allows for **state persistence**.

- A volume is a directory, possibly pre-populated, made available to containers in a Pod. The creation of the directory, the backend storage of the data and the contents depend on the volume type. As of v1.14, there are 28 different volume types ranging from `rbd` to for Ceph, to `NFS`, to dynamic volumes from a cloud provider like Google's `gcePersistentDisk`. Each has particular configuration options and dependencies.
- Adoption of **Container Storage Interface** (**CSI**) enables the goal of an industry standard interface for container orchestration to 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 can be passed with a `ConfigMap`. These can be used to pass important data like SSH keys, passwords, or even a configuration file like `/etc/hosts`.

7.2 Introducing 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.
- Keeping acquired data or ingesting it into other containers is a common task, typically requiring the use of a **Persistent Volume Claim (PVC)**.
- 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 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.
- A particular `access mode` is part of a Pod request. 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 `RWO` (ReadWriteOnce), which allows read-write by a single node, `ROX` (ReadOnlyMany), which allows read-only by multiple nodes, and `RWX` (ReadWriteMany), which allows read-write by many nodes.
- We had seen an example of containers within a pod while learning about networking. Using the same example we see that the same volume can be used by all containers. In this case both are using the /data/ directory. `MainApp` reads and writes to the volume, and `Logger` uses the volume read-only. It then writes to a different volume /log/.

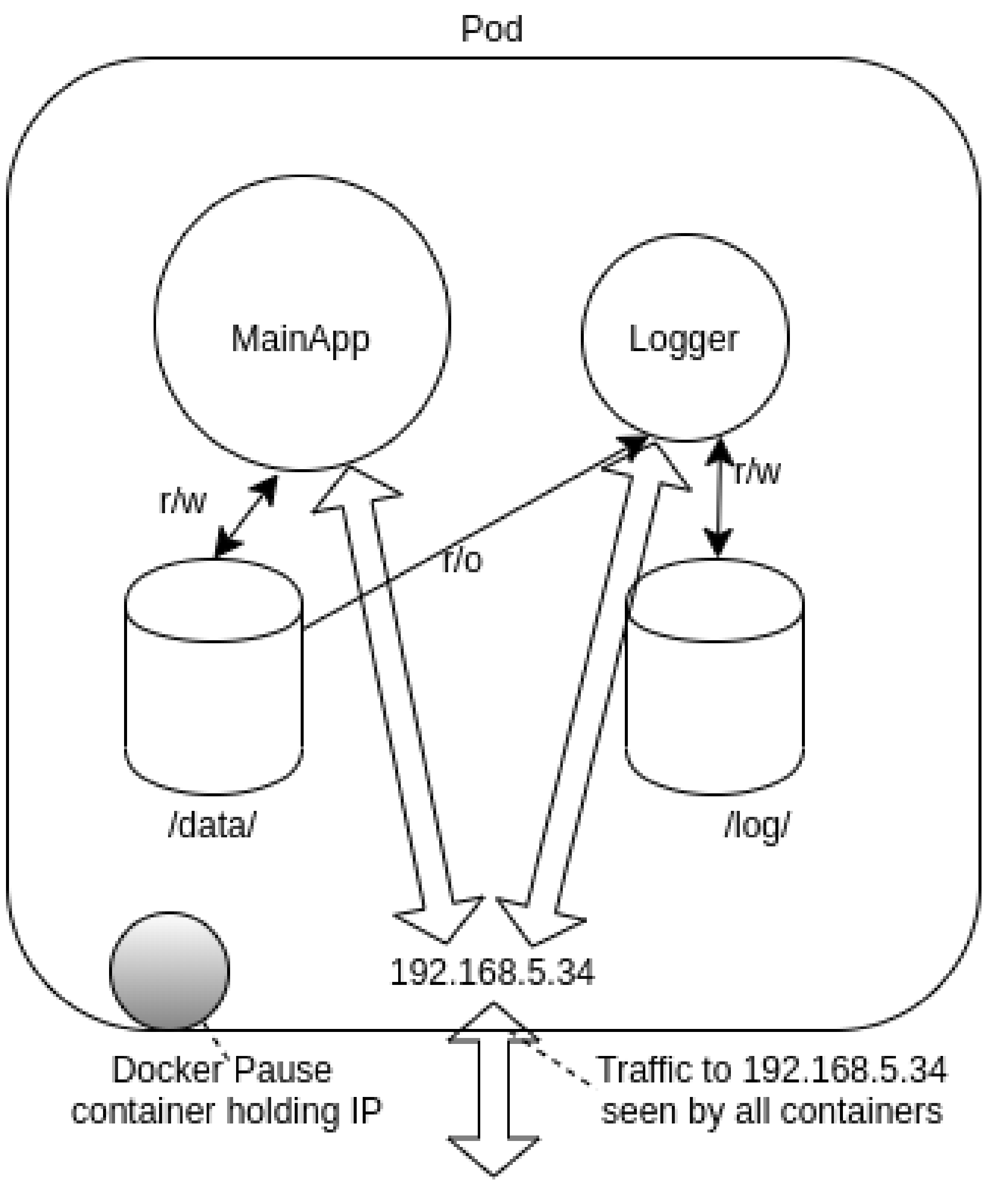


Figure 7.1: **Multiple volumes in a pod**

- 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 backend 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 ones will be used.

7.3 Volume Spec

- 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.

```yaml
apiVersion: v1
kind: Pod
metadata:
    name: busybox
    namespace: default
spec:
    containers:
    - image: busybox
      name: busy
      command:
        - sleep
        - "3600"
      volumeMounts:
      - mountPath: /scratch
        name: scratch-volume
    volumes:
    - name: scratch-volume
            emptyDir: {}
```

- The YAML file 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.

7.4 Volume Types

- There are several types that you can use to define volumes, each with their pros and cons. Some are local, and 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.
- `rbd` for block storage or `CephFS` and `GlusterFS`, if available in your Kubernetes cluster, can be a good choice for multiple writer needs.
- Besides the volume types we just mentioned, there are many other possible, with more being added: `azureDisk`, `azureFile`, `csi`, `downwardAPI`, `fc` (fibre channel), `flocker`, `gitRepo`, `local`, `projected`, `portworxVolume`, `quobyte`, `scaleIO`, `secret`, `storageos`, `vsphereVolume`, `persistentVolumeClaim`, etc.

7.5 Shared Volume Example

- The following YAML file, for pod exampleA creates a pod with two containers, one called `alphacont`, the other called `betacont` both with access to a shared volume called `sharevol`:

```yaml
....
   containers:
   - name: alphacont
     image: busybox
```

```yaml
    volumeMounts:
    - mountPath: /alphadir
      name: sharevol
  - name: betacont
    image: busybox
    volumeMounts:
    - mountPath: /betadir
      name: sharevol
  volumes:
  - name: sharevol
    emptyDir: {}
```

```
$ 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
```

- You could use `emptyDir` or `hostPath` easily, since those types do not require any additional setup, and will work in your Kubernetes cluster.
- Note that one container wrote, and the other container 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 application to avoid corruption.

7.6 Persistent Volumes and Claims

- 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` (`pvc`) with various parameters for size and possibly the type of backend 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 backend 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 master 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 administrator to handle the storage and data.
    - `Delete` tells the volume plugin to delete the API object, as well as the storage behind it.
    - The `Recycle` option runs an `rm -rf /mountpoint` and then makes it available to a new claim. With the stability of dynamic provisioning, the `Recycle` option is planned to be deprecated.

```
$ kubectl get pv
$ kubectl get pvc
```

7.7 Persistent Volume

- The following example shows a basic declaration of a `PersistentVolume` using the `hostPath` type.

```yaml
kind: PersistentVolume
apiVersion: v1
metadata:
    name: 10Gpv01
    labels:
             type: local
spec:
    capacity:
        storage: 10Gi
    accessModes:
        - ReadWriteOnce
    hostPath:
        path: "/somepath/data01"
```

- Each type will have its own configuration settings. For example, an already created Ceph or GCE Persistent Disk would not need to be configured, but could be claimed from the provider.
- Persistent volumes are cluster-scoped, but persistent volume claims are namespace-scoped. An alpha feature since v1.11 this allows for static provisioning of Raw Block Volumes, which currently support the Fibre Channel plugin. There is a lot of development and change in this area, with plugins adding dynamic provisioning.

7.8 Persistent Volume Claim

- 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`.

```yaml
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
```

- The Pod configuration could also be as complex as this:

```yaml
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"
```

Point to Ponder

If you had one application ingesting data but also want to archive, ingest into a data lake , and forward the data, how would you use persistent volume claims?

7.9 Dynamic Provisioning

- While handling volumes with a persistent volume definition and abstracting the storage provider using a claim is powerful, a cluster administrator still needs to create those volumes in the first place. Starting with Kubernetes v1.4, **Dynamic Provisioning** allowed for the cluster to request storage from an exterior, pre-configured source. API calls made by the appropriate plugin 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. Single, default class is possible via annotation.
- Here is an example of a `StorageClass` using GCE:

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast                            # Could be any name
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-ssd
```

- There are also providers you can create inside of you cluster, which use `Custom Resource Definitions` and local operators to manage storage.
  - **Rook** - From https://rook.io is a CNCF project in `incubation` status which allows easy management of several back-end storage types, including an easy way to deploy **Ceph**
  - **Longhorn** - From https://longhorn.io is a CNCF project in `sandbox` status, developed by **Rancher** which also allows management of local storage.

7.10 Secrets

- 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. Someone reading through a YAML file may read a password and remember it. Using the Secret API resource, the same password could be encoded. A casual reading would not give away the password. You can create, get, or delete secrets:

```
$ kubectl get secrets
```

- Secrets can be manually encoded with kubectl create secret:

```
$ kubectl create secret generic --help

$ kubectl create secret generic mysql --from-literal=password=root
```

- A secret is not encrypted by default, only base64-encoded. You can see the encoded string inside the secret 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.

  In order to encrypt secrets one must create a `EncryptionConfiguration` object 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.

- A secret can be made manually as well, then inserted into a YAML file:

```
$ echo LFTr@1n | base64
```

```
TEZUckAxbgo=
```

```
$ vim secret.yaml
```

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: LF-secret
data:
  password: TEZUckAxbgo=
```

- Prior to Kubernetes v1.18 secrets (and configMaps) were automatically updated. This could lead to issues if a configuration were updated and a pod restarted it may be configured differently than other replicas. In newer versions these objects can be made immutable.

7.11 Using Secrets via Environment Variables

- A secret can be used as an environmental variable in a Pod. You can see one being configured in the following example:

```yaml
...
spec:
    containers:
    - image: mysql:5.5
      env:
      - name: MYSQL_ROOT_PASSWORD
        valueFrom:
            secretKeyRef:
                name: mysql
                key: password
            name: mysql
```

- There is no 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 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.

7.12 Mounting Secrets as Volumes

- 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.

```yaml
...
spec:
    containers:
    - image: busybox
      command:
        - sleep
        - "3600"
      volumeMounts:
      - mountPath: /mysqlpassword
        name: mysql
      name: busy
    volumes:
    - name: mysql
        secret:
            secretName: mysql
```

- 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
```

7.13 Portable Data with ConfigMaps

- A similar API resource to Secrets is the ConfigMap, except the data is not encoded. In keeping with the concept of decoupling in Kubernetes, using a ConfigMap decouples a 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 Pod 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` that contains this file. The `configmap` object will have a `data` section containing the content of the file:

```yaml
$ kubectl get configmap foobar -o yaml
kind: ConfigMap
apiVersion: v1
metadata:
    name: foobar
data:
    config.js: |
        {
...
```

- `ConfigMaps` can be consumed in various ways:
  - Pod environmental variables from single or multiple `ConfigMaps`
  - 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.

7.14 Using ConfigMaps

- 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. For instance:

```yaml
env:
- name: SPECIAL_LEVEL_KEY
  valueFrom:
    configMapKeyRef:
      name: special-config
      key: special.how
```

- With volumes, you define a volume with the `configMap` type in your pod and mount it where it needs to be used.

```yaml
volumes:
      - name: config-volume
      configMap:
        name: special-config
```

7.15 Deployment Configuration Status

- The `Status` output is generated when the information is requested:

```yaml
status:
  availableReplicas: 2
  conditions:
  - lastTransitionTime: 2017-12-21T13:57:07Z
    lastUpdateTime: 2017-12-21T13:57:07Z
    message: Deployment has minimum availability.
    reason: MinimumReplicasAvailable
    status: "True"
    type: Available
  observedGeneration: 2
  readyReplicas: 2
  replicas: 2
  updatedReplicas: 2
```

- 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` and to 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 rollout and rollback situation of the deployment.

7.16 Scaling and Rolling Updates

- 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     DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE
dev-web  4        4        4           1          12m
```

- Non-immutable values can be edited via a text editor, as well. Use **edit** to trigger an update. For example, to change the deployed version of the **nginx** web server to an older version:

```
$ kubectl edit deployment nginx
```

```yaml
....
      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.

7.17 Deployment Rollbacks

- With all the `ReplicaSets` of a Deployment being kept, you can also roll back to a previous revision by scaling up and down the ReplicaSets the other way. Next, we will have a closer look at rollbacks, using the `--record` option of the `kubectl` command, which allows annotation in the resource definition. The `create` generator does not have a record function.

```
$ kubectl set image deployment ghost --image=ghost:0.9 --record
$ kubectl get deployments ghost -o yaml
```

```
metadata:
    annotations:
            deployment.kubernetes.io/revision: "1"
            kubernetes.io/change-cause: kubectl set image deployment ghost --image=ghost0.9 --record
```

- Should an update fail, due to 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:0.9 --all
$ kubectl rollout history deployment/ghost deployments "ghost":
```

```
REVISION     CHANGE-CAUSE
1            <none>
2            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
```

- 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. One can also pause a Deployment, and then resume.

```
$ kubectl rollout pause deployment/ghost
$ kubectl rollout resume deployment/ghost
```

- Please note that you can still do a rolling update on `ReplicationControllers` 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.

7.18 Labs

Exercise 7.1: Configure the Deployment: Secrets and ConfigMap

Very Important

Save a copy of your `$HOME/app1/simpleapp.yaml` file, in case you would like to repeat portions of the labs, or you find your file difficult to use due to typos and whitespace issues.

```
student@master:˜$ cp $HOME/app1/simpleapp.yaml $HOME/beforeLab5.yaml
```

Overview

In this lab we will add resources to our deployment with further configuration you may need for production.

There are three different ways a **ConfigMap** can ingest data, from a literal value, from a file, or from a directory of files.

1. Create a **ConfigMap** containing primary colors. We will create a series of files to ingest into the **ConfigMap**. First create a directory `primary` and populate it with four files. Then we create a file in our home directory with our favorite color.

```
student@master:~/app1$ cd

student@master:~$ mkdir primary
student@master:~$ echo c > primary/cyan
student@master:~$ echo m > primary/magenta
student@master:~$ echo y > primary/yellow
student@master:~$ echo k > primary/black
student@master:~$ echo "known as key" >> primary/black
student@master:~$ echo blue > favorite
```

2. Generate a **configMap** using each of the three methods.

```
student@master:~$ kubectl create configmap colors \
  --from-literal=text=black \
  --from-file=./favorite \
  --from-file=./primary/
```

```
configmap/colors created
```

3. View the newly created **configMap**. Note the way the ingested data is presented.

```
student@master:~$ kubectl get configmap colors
```

```
NAME      DATA      AGE
colors    6         11s
```

```
student@master:~$ 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
metadata:
<output_omitted>
```

4. Update the YAML file of the application to make use of the **configMap** as an environmental parameter. Add the six lines from the `env:` line to `key:favorite`.

```
student@master:~$ vim $HOME/app1/simpleapp.yaml
```

simpleapp.yaml

```yaml
....
    spec:
      containers:
      - image: 10.105.119.236:5000/simpleapp
        env:                                 #<-- Add from here
        - name: ilike
          valueFrom:
            configMapKeyRef:
              name: colors
              key: favorite                  #<-- to here
        imagePullPolicy: Always
....
```

5. Delete and re-create the deployment with the new parameters.

```
student@master-lab-7xtx:~$ kubectl delete deployment try1
```

```
deployment.apps "try1" deleted
```

```
student@master-lab-7xtx:~$ kubectl create -f $HOME/app1/simpleapp.yaml
```

```
deployment.apps/try1 created
```

6. Even though the `try1` pod is not in a fully ready state, it is running and useful. Use **kubectl exec** to view a variable's value. View the pod state then verify you can see the `ilike` value within the `simpleapp` container. Note that the use of double dash (- -) tells the shell to pass the following as standard in.

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

```
<output_omitted>
```

```
student@master:~$ kubectl exec -c simpleapp -it try1-5db9bc6f85-whxbf \
    -- /bin/bash -c 'echo $ilike'
```

```
blue
```

7. Edit the YAML file again, this time adding the another method of using a **configMap**. Edit the file to add three lines. `envFrom` should be indented the same amount as `env` earlier in the file, and `configMapRef` should be indented the same as `configMapKeyRef`.

```
student@master:~$ vim $HOME/app1/simpleapp.yaml
```

simpleapp.yaml

```yaml
....
              configMapKeyRef:
                name: colors
                key: favorite
        envFrom:                #<-- Add this and the following two lines
        - configMapRef:
            name: colors
        imagePullPolicy: Always
....
```

8. Again delete and recreate the deployment. Check the pods restart.

```
student@master:~$ kubectl delete deployment try1
```

```
deployment.apps "try1" deleted
```

```
student@master:~$ kubectl create -f $HOME/app1/simpleapp.yaml
```

```
deployment.apps/try1 created
```

```
student@master:~$ kubectl get pods
```

```
NAME                        READY   STATUS          RESTARTS    AGE
nginx-6b58d9cdfd-9fnl4      1/1     Running         1           23h
registry-795c6c8b8f-hl5w    1/1     Running         2           23h
try1-d4fbf76fd-46pkb        1/2     Running         0           40s
try1-d4fbf76fd-9kw24        1/2     Running         0           39s
try1-d4fbf76fd-bx9j9        1/2     Running         0           39s
try1-d4fbf76fd-jw8g7        1/2     Running         0           40s
try1-d4fbf76fd-lppl5        1/2     Running         0           39s
try1-d4fbf76fd-xtfd4        1/2     Running         0           40s
```

9. View the settings inside the `try1` container of a pod. The following output is truncated in a few places. Omit the container name to observe the behavior. Also execute a command to see all environmental variables instead of logging into the container first.

```
student@master:~$ kubectl exec -it try1-d4fbf76fd-46pkb -- /bin/bash -c 'env'
```

```
Defaulting container name to simpleapp.
Use 'kubectl describe pod/try1-d4fbf76fd-46pkb -n default' to see all of the containers in this pod.
REGISTRY_PORT_5000_TCP_ADDR=10.105.119.236
HOSTNAME=try1-d4fbf76fd-46pkb
TERM=xterm
yellow=y
<output_omitted>
REGISTRY_SERVICE_HOST=10.105.119.236
KUBERNETES_SERVICE_PORT=443
REGISTRY_PORT_5000_TCP=tcp://10.105.119.236:5000
KUBERNETES_SERVICE_HOST=10.96.0.1
text=black
REGISTRY_SERVICE_PORT_5000=5000
<output_omitted>
black=k
known as key

<output_omitted>
ilike=blue
<output_omitted>
magenta=m

cyan=c
<output_omitted>
```

10. For greater flexibility and scalability **ConfigMaps** can be created from a YAML file, then deployed and redeployed as necessary. Once ingested into the cluster the data can be retrieved in the same manner as any other object. Create another **configMap**, this time from a YAML file.

```
student@master:~$ vim car-map.yaml
```

car-map.yaml

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: fast-car
  namespace: default
data:
  car.make: Ford
  car.model: Mustang
  car.trim: Shelby
```

```
student@master:˜$ kubectl create -f car-map.yaml
```

```
configmap/fast-car created
```

11. View the ingested data, note that the output is just as in file created.

```
student@master:˜$ kubectl get configmap fast-car -o yaml
```

```
apiVersion: v1
data:
  car.make: Ford
  car.model: Mustang
  car.trim: Shelby
kind: ConfigMap
metadata:
<output_omitted>
```

12. Add the **configMap** settings to the `simpleapp.yaml` file as a volume. Both containers in the try1 deployment can access to the same volume, using `volumeMounts` statements. Remember that the volume stanza is of equal depth to the containers stanza, and should come after the containers have been declared, the example below has the volume added just before the `status`: output..

```
student@master:˜$ vim $HOME/app1/simpleapp.yaml
```

simpleapp.yaml

```yaml
....
    spec:
      containers:
      - image: 10.105.119.236:5000/simpleapp
        volumeMounts:                 #<-- Add this and following two lines
        - mountPath: /etc/cars
          name: car-vol
        env:
        - name: ilike
....
      securityContext: {}
      terminationGracePeriodSeconds: 30
      volumes:                        #<-- Add this and following four lines
      - name: car-vol
        configMap:
          defaultMode: 420
          name: fast-car
status:
....
```

13. Delete and recreate the deployment.

```
student@master:˜$ kubectl delete deployment try1
```

```
deployment.apps "try1" deleted
```

```
student@master:˜$ kubectl create -f $HOME/app1/simpleapp.yaml
```

```
deployment.apps/try1 created
```

14. Verify the deployment is running. Note that we still have not automated the creation of the `/tmp/healthy` file inside the container, as a result the `AVAILABLE` count remains zero until we use the **for** loop to create the file. We will remedy this in the next step.

```
student@master:˜$ kubectl get deployment
```

```
NAME        READY   UP-TO-DATE   AVAILABLE   AGE
nginx       1/1     1            1           1d
registry    1/1     1            1           1d
try1        0/6     6            0           39s
```

15. Our health check was the successful execution of a command. We will edit the command of the existing `readinessProbe` to check for the existence of the mounted configMap file and re-create the deployment. After a minute both containers should become available for each pod in the deployment.

```
student@master:˜$ kubectl delete deployment try1
```

```
deployment.apps "try1" deleted
```

```
student@master:˜$ vim $HOME/app1/simpleapp.yaml
```

simpleapp.yaml

```yaml
....
        readinessProbe:
          exec:
            command:
            - ls                        #<-- Add/Edit this and following line.
            - /etc/cars
          periodSeconds: 5
....
```

```
student@master:˜$ kubectl create -f $HOME/app1/simpleapp.yaml
```

```
deployment.apps/try1 created
```

16. Wait about a minute and view the deployment and pods. All six replicas should be running and report that 2/2 containers are in a ready state within.

```
student@master:˜$ kubectl get deployment
```

```
NAME        READY   UP-TO-DATE   AVAILABLE   AGE
nginx       1/1     1            1           1d
registry    1/1     1            1           1d
try1        6/6     6            6           1m
```

```
student@master:˜$ kubectl get pods
```

```
NAME                         READY     STATUS     RESTARTS    AGE
nginx-6b58d9cdfd-9fn14       1/1       Running    1           1d
registry-795c6c8b8f-hl5wf    1/1       Running    2           1d
try1-7865dcb948-2dzc8        2/2       Running    0           1m
try1-7865dcb948-7fkh7        2/2       Running    0           1m
try1-7865dcb948-d85bc        2/2       Running    0           1m
try1-7865dcb948-djrcj        2/2       Running    0           1m
try1-7865dcb948-kwlv8        2/2       Running    0           1m
try1-7865dcb948-stb2n        2/2       Running    0           1m
```

17. View a file within the new volume mounted in a container. It should match the data we created inside the configMap. Because the file did not have a carriage-return it will appear prior to the following prompt.

```
student@master:~$ kubectl exec -c simpleapp -it try1-7865dcb948-stb2n \
    -- /bin/bash -c 'cat /etc/cars/car.trim'
```

```
Shelby student@master:~$
```

Exercise 7.2: Configure the Deployment: Attaching Storage

There are several types of storage which can be accessed with Kubernetes, with flexibility of storage being essential to scalability. In this exercise we will configure an NFS server. With the NFS server we will create a new **persistent volume (pv)** and a **persistent volume claim (pvc)** to use it.

1. Search for `pv` and `pvc` YAML example files on http://kubernetes.io/docs and http://kubernetes.io/blog.
2. Use the `CreateNFS.sh` script from the tarball to set up NFS on your master node. This script will configure the server, export `/opt/sfw` and create a file `/opt/sfw/hello.txt`. Use the **find** command to locate the file if you don't remember where you extracted the tar file. This example narrows the search to your `$HOME` directory. Change for your environment. directory. You may find the same file in more than one sub-directory of the tarfile.

```
student@master:~$ find $HOME -name CreateNFS.sh
```

```
/home/student/LFD259/SOLUTIONS/s_05/CreateNFS.sh
```

```
student@master:~$ cp /home/student/LFD259/SOLUTIONS/s_05/CreateNFS.sh $HOME
```

```
student@master:~$ bash $HOME/CreateNFS.sh
```

```
Hit:1 http://us-central1.gce.archive.ubuntu.com/ubuntu xenial InRelease
Get:2 http://us-central1.gce.archive.ubuntu.com/ubuntu xenial-updates InRelease [102 kB]

<output_omitted>

Should be ready. Test here and second node

Export list for localhost:
/opt/sfw *
```

3. Test by mounting the resource from your **second node**. Begin by installing the client software.

```
student@worker:~$ sudo apt-get -y install nfs-common nfs-kernel-server
```

```
<output_omitted>
```

4. Test you can see the exported directory using **showmount** from you second node.

```
student@worker:~$ showmount -e master #<-- Edit to be first node's name or IP
```

```
Export list for master:
/opt/sfw *
```

5. Mount the directory. Be aware that unless you edit `/etc/fstab` this is not a persistent mount. Change out the node name for that of your master node.

```
student@worker:~$ sudo mount master:/opt/sfw /mnt
```

6. Verify the `hello.txt` file created by the script can be viewed.

```
student@worker:~$ ls -l /mnt
```

```
total 4
-rw-r--r-- 1 root root 9 Sep 28 17:55 hello.txt
```

7. Return to the master node and create a YAML file for an object with kind **PersistentVolume**. The included example file needs an edit to the `server:` parameter. Use the hostname of the master 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 incorrect resource will not start. Note that the `accessModes` do not currently affect actual access and are typically used as labels instead.

```
student@master:~$ find $HOME -name PVol.yaml
```

```
/home/student/LFD259/SOLUTIONS/s_05/PVol.yaml
```

```
student@master:~$ cp /home/student/LFD259/SOLUTIONS/s_05/PVol.yaml $HOME
```

```
student@master:~$ vim PVol.yaml
```

PVol.yaml

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pvvol-1
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteMany
  persistentVolumeReclaimPolicy: Retain
  nfs:
    path: /opt/sfw
    server: master                  #<-- Edit to match master node name or IP
    readOnly: false
```

8. Create and verify you have a new 1Gi volume named **pvvol-1**. Note the status shows as `Available`. Remember we made two persistent volumes for the image registry earlier.

```
student@master:~$ kubectl create -f PVol.yaml
```

```
persistentvolume/pvvol-1 created
```

```
student@master:~$ kubectl get pv
```

```
NAME             CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS      CLAIM    STORAGECLASS   REASON   AGE
pvvol-1          1Gi        RWX            Retain           Available                                    4s
registryvm       200Mi      RWO            Retain           Bound       default/nginx-claim0                4d
task-pv-volume   200Mi      RWO            Retain           Bound       default/registry-claim0             4d
```

9. Now that we have a new volume we will use a **persistent volume claim (pvc)** to use it in a Pod. We should have two existing claims from our local registry.

```
student@master:˜/$ kubectl get pvc
```

```
NAME              STATUS    VOLUME           CAPACITY   ACCESS MODES   STORAGECLASS   AGE
nginx-claim0      Bound     registryvm       200Mi      RWO                           4d
registry-claim0   Bound     task-pv-volume   200Mi      RWO                           4d
```

10. Create or copy a yaml file with the kind **PersistentVolumeClaim**.

```
student@master:˜$ vim pvc.yaml
```

pvc.yaml

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-one
spec:
  accessModes:
  - ReadWriteMany
  resources:
     requests:
       storage: 200Mi
```

11. Create and verify the new pvc status is `bound`. Note the size is `1Gi`, even though `200Mi` was suggested. Only a volume of at least that size could be used, the first volume with found with at least that much space was chosen.

```
student@master:˜$ kubectl create -f pvc.yaml
```

```
persistentvolumeclaim/pvc-one created
```

```
student@master:˜$ kubectl get pvc
```

```
NAME           STATUS    VOLUME        CAPACITY    ACCESS MODES    STORAGECLASS    AGE
nginx-claim0      Bound     registryvm       200Mi      RWO                           4d
pvc-one           Bound     pvvol-1          1Gi        RWX                           4s
registry-claim0   Bound     task-pv-volume   200Mi      RWO                           4d
```

12. Now look at the status of the physical volume. It should also show as bound.

```
student@master:˜$ kubectl get pv
```

```
NAME           CAPACITY ACCESS MODES RECLAIM POLICY STATUS
 CLAIM      STORAGECLASS   REASON    AGE
pvvol-1        1Gi      RWX          Retain         Bound
 default/pvc-one                     14m
registryvm     200Mi    RWO          Retain         Bound
 default/nginx-claim0                4d
task-pv-volume 200Mi    RWO          Retain         Bound
 default/registry-claim0             4d
```

13. Edit the `simpleapp.yaml` file to include two new sections. One section for the container while will use the volume mount point, you should have an existing entry for `car-vol`. The other section adds a volume to the deployment in general, which you can put after the configMap volume section.

```
student@master:˜$ vim $HOME/app1/simpleapp.yaml
```

simpleapp.yaml

```
....
        volumeMounts:
        - name: car-vol
          mountPath: /etc/cars
        - name: nfs-vol                           #<-- Add this and following line
          mountPath: /opt
....
     volumes:
      - name: car-vol
        configMap:
          defaultMode: 420
          name: fast-car
      - name: nfs-vol                             #<-- Add this and following two lines
        persistentVolumeClaim:
          claimName: pvc-one
status:
....
```

14. Delete and re-create the deployment.

```
student@master:~$ kubectl delete deployment try1 ; kubectl create -f $HOME/app1/simpleapp.yaml
```

```
deployment.apps "try1" deleted
deployment.apps/try1 created
```

15. View the details any of the pods in the deployment, you should see `nfs-vol` mounted under `/opt`. The use to command line completion with the **tab** key can be helpful for using a pod name.

```
student@master:~$ kubectl describe pod try1-594fbb5fc7-5k7sj
```

```
<output_omitted>
    Mounts:
      /etc/cars from car-vol (rw)
      /opt from nfs-vol (rw)
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-j7cqd (ro)
<output_omitted>
```

Exercise 7.3: Using ConfigMaps Configure Ambassador Containers

In an earlier lab we added a second `Ambassador` container to handle logging. Now that we have learned about using `ConfigMaps` and attaching storage we will use configure our `basic` pod.

1. Review the YAML for our earlier simple pod. Recall that we added an Ambassador style logging container to the pod but had not fully configured the logging.

```
student@master:~$ cat basic.yaml
```

```
<output_omitted>
  containers:
  - name: webcont
    image: nginx
    ports:
    - containerPort: 80
  - name: fdlogger
    image: fluent/fluentd
```

2. Let us begin by adding shared storage to each container. We will use the `hostPath` storage class to provide the `PV` and `PVC`. First we create the directory.

```
student@master:˜$ sudo mkdir /tmp/weblog
```

3. Now we create a new `PV` to use that directory for the `hostPath` storage class. We will use the `storageClassName` of manual so that only `PVC`s which use that name will bind the resource.

```
student@master:˜$ vim weblog-pv.yaml
```

weblog-pv.yaml

```yaml
kind: PersistentVolume
apiVersion: v1
metadata:
  name: weblog-pv-volume
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 100Mi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/tmp/weblog"
```

4. Create and verify the new `PV` exists and shows an `Available` status.

```
student@master:˜$ kubectl create -f weblog-pv.yaml
```

```
persistentvolume/weblog-pv-volume created
```

```
student@master:˜$ kubectl get pv weblog-pv-volume
```

```
NAME             CAPACITY  ACCESS MODES   RECLAIM POLICY
  STATUS         CLAIM    STORAGECLASS    REASON    AGE

weblog-pv-volume 100Mi     RWO            Retain
  Available               manual                    21s
```

5. Next we will create a `PVC` to use the `PV` we just created.

```
student@master:˜$ vim weblog-pvc.yaml
```

weblog-pvc.yaml

```yaml
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: weblog-pv-claim
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 100Mi
```

6. Create the PVC and verify it shows as Bound to the the PV we previously created.

```
student@master:~$ kubectl create -f weblog-pvc.yaml
```

```
persistentvolumeclaim/weblog-pv-claim created
```

```
student@master:~$ kubectl get pvc weblog-pv-claim
```

```
NAME             STATUS   VOLUME            CAPACITY ACCESS MODES
   STORAGECLASS   AGE
weblog-pv-claim  Bound    weblog-pv-volume 100Mi    RWO
   manual          79s
```

7. We are ready to add the storage to our pod. We will edit three sections. The first will declare the storage to the pod in general, then two more sections which tell each container where to make the volume available.

```
student@master:~$ vim basic.yaml
```

basic.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: basicpod
  labels:
    type: webserver
spec:
  volumes:                               #<-- Add three lines, same depth as containers
    - name: weblog-pv-storage
      persistentVolumeClaim:
        claimName: weblog-pv-claim
  containers:
  - name: webcont
    image: nginx
    ports:
    - containerPort: 80
    volumeMounts:                        #<-- Add three lines, same depth as ports
      - mountPath: "/var/log/nginx/"
        name: weblog-pv-storage          # Must match volume name above
  - name: fdlogger
    image: fluent/fluentd
    volumeMounts:                        #<-- Add three lines, same depth as image:
      - mountPath: "/var/log"
        name: weblog-pv-storage          # Must match volume name above
```

8. At this point we can create the pod again. When we create a shell we will find that the access.log for **nginx** is no longer a symbolic link pointing to stdout it is a writable, zero length file. Leave a **tailf** of the log file running.

```
student@master:~$ kubectl create -f basic.yaml
```

```
pod/basicpod created
```

```
student@master:~$ kubectl exec -c webcont -it basicpod -- /bin/bash
```

On Container

```
root@basicpod:/#  ls -l /var/log/nginx/access.log
```

```
-rw-r--r-- 1 root root 0 Oct 18 16:12 /var/log/nginx/access.log
```

```
root@basicpod:/# tail -f /var/log/nginx/access.log
```

9. Open a second connection to your master node. We will use the pod IP as we have not yet configured a `service` to expose the pod.

```
student@master:~$ kubectl get pods -o wide
```

```
NAME     READY STATUS    RESTARTS  AGE    IP                NODE
   NOMINATED NODE
basicpod 2/2   Running   0         3m26s  192.168.213.181   master
   <none>
```

10. Use **curl** to view the welcome page of the webserver. When the command completes you should see a new entry added to the log. Right after the `GET` we see a `200` response indicating success. You can use **ctrl-c** and **exit** to return to the host shell prompt.

```
student@master:~$ curl http://192.168.213.181
```

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

On Container

192.168.32.128 - - [18/Oct/2018:16:16:21 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.47.0" "-"

11. Now that we know the `webcont` container is writing to the `PV` we will configure the logger to use that directory as a source. For greater flexibility we will configure **fluentd** using a `configMap`.

    Fluentd has many options for input and output of data. We will read from a file of the `webcont` container and write to standard out of the `fdlogger` container. The details of the data settings can be found in **fluentd** documentation here: https://docs.fluentd.org/v1.0/categories/config-file

```
student@master:~$ vim weblog-configmap.yaml
```

weblog-configmap.yaml

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: fluentd-config
  namespace: default
data:
  fluentd.conf: |
    <source>
      @type tail
      format none
      path /var/log/access.log
      tag count.format1
    </source>

    <match *.**>
    @type stdout
    id stdout_output
    </match>
```

12. Create the new configMap.

```
student@master:~$ kubectl create -f weblog-configmap.yaml
```

```
configmap/fluentd-config created
```

13. View the logs for both containers in the `basicpod`. You should see some startup information, but not the HTTP traffic.

```
student@master:~$ kubectl logs basicpod webcont
```

```
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perform configuration
/docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
/docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-default.sh
10-listen-on-ipv6-by-default.sh: Getting the checksum of /etc/nginx/conf.d/default.conf
10-listen-on-ipv6-by-default.sh: Enabled listen on IPv6 in /etc/nginx/conf.d/default.conf
/docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-on-templates.sh
/docker-entrypoint.sh: Configuration complete; ready for start up
```

```
student@master:~$  kubectl logs basicpod fdlogger
```

```
2020-09-02 19:32:59 +0000 [info]: reading config file path="/etc/fluentd-config/fluentd.conf"
2020-09-02 19:32:59 +0000 [info]: starting fluentd-0.12.29
2020-09-02 19:32:59 +0000 [info]: gem 'fluent-mixin-config-placeholders' version '0.4.0'
2020-09-02 19:32:59 +0000 [info]: gem 'fluent-mixin-plaintextformatter' version '0.2.6'

<output_omitted>

  <source>
    @type tail
    format none
    path /var/log/access.log
<output_omitted>
```

14. Now we will edit the pod yaml file so that the **fluentd** container will mount the configmap as a volume and reference the variables inside the config file. You will add three areas, the volume declaration to the pod, the `env` parameter and the mounting of the volume to the fluentd container

```
student@master:~$ vim basic.yaml
```

basic.yaml

```yaml
....
  volumes:
    - name: weblog-pv-storage
      persistentVolumeClaim:
        claimName: weblog-pv-claim
    - name: log-config                    #<-- This and two lines following
      configMap:
        name: fluentd-config              # Must match existing configMap
....
    image: fluent/fluentd
    env:                                  #<-- This and two lines following
    - name: FLUENTD_OPT
      value: -c /etc/fluentd-config/fluentd.conf
....
```

```yaml
    volumeMounts:
      - mountPath: "/var/log"
        name: weblog-pv-storage
      - name: log-config                  #<-- This and next line
        mountPath: "/etc/fluentd-config"
```

15. At this point we can delete and re-create the pod, which would cause the configmap to be used by the new pod, among other changes.

```
student@master:~$ kubectl delete pod basicpod
```

```
pod "basicpod" deleted
```

```
student@master:~$ kubectl create -f basic.yaml
```

```
pod/basicpod created
```

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

```
NAME       READY   STATUS    RESTARTS   AGE   IP                NODE     NOMINATED....
basicpod   2/2     Running   0          8s    192.168.171.122   worker   <none>    ....
```

16. Use **curl** a few times to look at the default page served by `basicpod`

```
student@master:~$ curl http://192.168.171.122
```

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

17. Look at the logs for both containers. In addition to the standard startup information, you should also see the HTTP requests from the curl commands you just used at the end of the `fdlogger` output.

```
student@master:~$ kubectl logs basicpod webcont
```

```
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perform configuration
/docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
/docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-default.sh
10-listen-on-ipv6-by-default.sh: Getting the checksum of /etc/nginx/conf.d/default.conf
10-listen-on-ipv6-by-default.sh: Enabled listen on IPv6 in /etc/nginx/conf.d/default.conf
/docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-on-templates.sh
/docker-entrypoint.sh: Configuration complete; ready for start up
```

```
student@master:~$  kubectl logs basicpod fdlogger
```

```
2020-09-02 19:32:59 +0000 [info]: reading config file path="/etc/fluentd-config/fluentd.conf"
2020-09-02 19:32:59 +0000 [info]: starting fluentd-0.12.29
2020-09-02 19:32:59 +0000 [info]: gem 'fluent-mixin-config-placeholders' version '0.4.0'
2020-09-02 19:32:59 +0000 [info]: gem 'fluent-mixin-plaintextformatter' version '0.2.6'

<output_omitted>

  <source>
    @type tail
    format none
    path /var/log/access.log

<output_omitted>

2020-09-02 19:47:38 +0000 count.format1: {"message":"192.168.219.64 - - [02/Sep/2020:19:47:38 +0000] \"GET / HTTP
2020-09-02 19:47:41 +0000 count.format1: {"message":"192.168.219.64 - - [02/Sep/2020:19:47:41 +0000] \"GET / HTTP
2020-09-02 19:47:47 +0000 count.format1: {"message":"192.168.219.64 - - [02/Sep/2020:19:47:47 +0000] \"GET / HTTP
```

Exercise 7.4: Rolling Updates and Rollbacks

When we started working with `simpleapp` we used a **Docker** tag called `latest`. While this is the default tag when pulling an image, and commonly used, it remains just a string, it may not be the actual latest version of the image.

1. Make a slight change to our source and create a new image. We will use updates and rollbacks with our application. Adding a comment to the last line should be enough for a new image to be generated.

```
student@master:~$ cd ~/app1

student@master:~/app1$ vim simple.py


<output_omitted>
## Sleep for five seconds then continue the loop

  time.sleep(5)

## Adding a new comment so image is different.
```

2. Build the image again. A new container and image will be created. Verify when successful. There should be a different image ID and a recent creation time.

```
student@master:~/app1$ sudo docker build -t simpleapp .
```

```
Sending build context to Docker daemon 7.168 kB
Step 1/3 : FROM python:2
 ---> 2863c80c418c
Step 2/3 : ADD simple.py /
 ---> cde8ecf8492b
Removing intermediate container 3e908b76b5b4
Step 3/3 : CMD python ./simple.py
 ---> Running in 354620c97bf5
 ---> cc6bba0ea213
Removing intermediate container 354620c97bf5
Successfully built cc6bba0ea213
```

```
student@master:~/app1$ sudo docker images
```

```
REPOSITORY                              TAG
 IMAGE ID            CREATED       SIZE
simpleapp                               latest
 cc6bba0ea213    8 seconds ago     886 MB
10.105.119.236:5000/simpleapp           latest
15b5ad19d313     4 days ago        886 MB
<output_omitted>
```

3. Tag and push the updated image to your locally hosted registry. A reminder your IP address will be different than the example below. Use the tag `v2` this time instead of `latest`.

```
student@master:~/app1$ sudo docker tag simpleapp \
   10.105.119.236:5000/simpleapp:v2

student@master:~/app1$ sudo docker push 10.105.119.236:5000/simpleapp:v2
```

```
The push refers to a repository [10.105.119.236:5000/simpleapp]
d6153c8cc7c3: Pushed
ca82a2274c57: Layer already exists
de2fbb43bd2a: Layer already exists
4e32c2de91a6: Layer already exists
6e1b48dc2ccc: Layer already exists
ff57bdb79ac8: Layer already exists
6e5e20cbf4a7: Layer already exists
86985c679800: Layer already exists
8fad67424c4e: Layer already exists
v2: digest: sha256:6cf74051d09463d89f1531fceb9c44cbf99006f8d9b407
dd91d8f07baeee7e9c size: 2218
```

4. Connect to a terminal running on your second node. Pull the `latest` image, then pull `v2`. Note the latest did not pull the new version of the image. Again, remember to use the IP for your locally hosted registry. You'll note the digest is different.

```
student@worker:~$ sudo docker pull 10.105.119.236:5000/simpleapp
```

```
Using default tag: latest
latest: Pulling from simpleapp
Digest: sha256:cefa3305c36101d32399baf0919d3482ae8a53c926688be33
86f9bbc04e490a5
Status: Image is up to date for 10.105.119.236:5000/simpleapp:latest
```

```
student@worker:~$ sudo docker pull 10.105.119.236:5000/simpleapp:v2
```

```
v2: Pulling from simpleapp
f65523718fc5: Already exists
1d2dd88bf649: Already exists
c09558828658: Already exists
0e1d7c9e6c06: Already exists
c6b6fe164861: Already exists
45097146116f: Already exists
f21f8abae4c4: Already exists
1c39556edcd0: Already exists
fa67749bf47d: Pull complete
Digest: sha256:6cf74051d09463d89f1531fceb9c44cbf99006f8d9b407dd91d8
f07baeee7e9c
Status: Downloaded newer image for 10.105.119.236:5000/simpleapp:v2
```

5. Use **kubectl edit** to update the image for the `try1` deployment to use `v2`. As we are only changing one parameter we could also use the **kubectl set** command. Note that the configuration file has not been updated, so a delete or a replace command would not include the new version. It can take the pods up to a minute to delete and to recreate each pod in sequence.

```
student@master:~/app1$ kubectl edit deployment try1

....
      containers:
      - image: 10.105.119.236:5000/simpleapp:v2   #<-- Edit tag
        imagePullPolicy: Always
....
```

6. Verify each of the pods has been recreated and is using the new version of the image. Note some messages will show the scaling down of the old **replicaset**, others should show the scaling up using the new image.

```
student@master:~/app1$ kubectl get events
```

```
42m         Normal    ScalingReplicaSet   Deployment   Scaled up replica set try1-7fdbb5d557 to 6
32s         Normal    ScalingReplicaSet   Deployment   Scaled up replica set try1-7fd7459fc6 to 2
32s         Normal    ScalingReplicaSet   Deployment   Scaled down replica set try1-7fdbb5d557 to 5
32s         Normal    ScalingReplicaSet   Deployment   Scaled up replica set try1-7fd7459fc6 to 3
23s         Normal    ScalingReplicaSet   Deployment   Scaled down replica set try1-7fdbb5d557 to 4
23s         Normal    ScalingReplicaSet   Deployment   Scaled up replica set try1-7fd7459fc6 to 4
22s         Normal    ScalingReplicaSet   Deployment   Scaled down replica set try1-7fdbb5d557 to 3
22s         Normal    ScalingReplicaSet   Deployment   Scaled up replica set try1-7fd7459fc6 to 5
18s         Normal    ScalingReplicaSet   Deployment   Scaled down replica set try1-7fdbb5d557 to 2
18s         Normal    ScalingReplicaSet   Deployment   Scaled up replica set try1-7fd7459fc6 to 6
8s          Normal    ScalingReplicaSet   Deployment   (combined from similar events):
Scaled down replica set try1-7fdbb5d557 to 0

```

7. View the images of a Pod in the deployment. Narrow the output to just view the images. The `goproxy` remains unchanged, but the `simpleapp` should now be `v2`.

```
student@master:~/app1$ kubectl describe pod try1-895fccfb-ttqdn |grep Image
```

```
    Image:          10.105.119.236:5000/simpleapp:v2
    Image ID:\
       docker-pullable://10.105.119.236:5000/simpleapp@sha256:6cf74051d09
463d89f1531fceb9c44cbf99006f8d9b407dd91d8f07baeee7e9c
    Image:          k8s.gcr.io/goproxy:0.1
    Image ID:\
       docker-pullable://k8s.gcr.io/goproxy@sha256:5334c7ad43048e3538775c
b09aaf184f5e8acf4b0ea60e3bc8f1d93c209865a5
```

8. View the update history of the deployment.

```
student@master:~/app1$ kubectl rollout history deployment try1
```

```
deployments "try1"
REVISION  CHANGE-CAUSE
1         <none>
2         <none>
```

9. Compare the output of the **rollout history** for the two revisions. Images and labels should be different, with the image `v2` being the change we made.

```
student@master:~/app1$ kubectl rollout history deployment try1 --revision=1 > one.out

student@master:~/app1$ kubectl rollout history deployment try1 --revision=2 > two.out

student@ckad-/app11:~$ diff one.out two.out
```

```
1c1
< deployments "try1" with revision #1
---
> deployments "try1" with revision #2
3c3
<   Labels:        pod-template-hash=1509661973
---
>   Labels:        pod-template-hash=45197796
7c7
<     Image:        10.105.119.236:5000/simpleapp
---
>     Image:        10.105.119.236:5000/simpleapp:v2
```

10. View what would be undone using the **–dry-run** option while undoing the rollout. This allows us to see the new template prior to using it.

```
student@master:~/app1$ kubectl rollout undo --dry-run=client deployment/try1
```

```
deployment.apps/try1
Pod Template:
  Labels:        pod-template-hash=1509661973
        run=try1
  Containers:
   try1:
    Image:        10.105.119.236:5000/simpleapp:latest
    Port:        <none>
<output_omitted>
```

11. View the pods. Depending on how fast you type the `try1` pods should be about 2 minutes old.

```
student@master:~/app1$ kubectl get pods
```

```
NAME                         READY     STATUS    RESTARTS    AGE
nginx-6b58d9cdfd-9fnl4       1/1       Running   1           5d
registry-795c6c8b8f-hl5wf    1/1       Running   2           5d
try1-594fbb5fc7-7dl7c        2/2       Running   0           2m
try1-594fbb5fc7-8mxlb        2/2       Running   0           2m
try1-594fbb5fc7-jr7h7        2/2       Running   0           2m
try1-594fbb5fc7-s24wt        2/2       Running   0           2m
try1-594fbb5fc7-xfffg        2/2       Running   0           2m
try1-594fbb5fc7-zfmz8        2/2       Running   0           2m
```

12. In our case there are only two revisions, which is also the default number kept. Were there more we could choose a particular version. The following command would have the same effect as the previous, without the **–dry-run** option.

```
student@master:~/app1$ kubectl rollout undo deployment try1 --to-revision=1
```

```
deployment.apps/try1 rolled back
```

13. Again, it can take a bit for the pods to be terminated and re-created. Keep checking back until they are all running again.

```
student@master:~/app1$ kubectl get pods
```

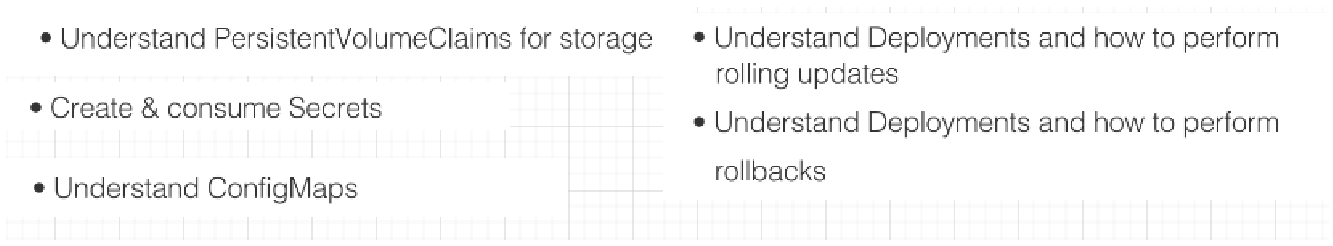
```
NAME                         READY     STATUS        RESTARTS    AGE
nginx-6b58d9cdfd-9fnl4       1/1       Running       1           5d
registry-795c6c8b8f-hl5wf    1/1       Running       2           5d
try1-594fbb5fc7-7dl7c        2/2       Terminating   0           3m
try1-594fbb5fc7-8mxlb        0/2       Terminating   0           2m
try1-594fbb5fc7-jr7h7        2/2       Terminating   0           3m
try1-594fbb5fc7-s24wt        2/2       Terminating   0           2m
try1-594fbb5fc7-xfffg        2/2       Terminating   0           3m
try1-594fbb5fc7-zfmz8        1/2       Terminating   0           2m
try1-895fccfb-8dn4b          2/2       Running       0           22s
try1-895fccfb-kz72j          2/2       Running       0           10s
try1-895fccfb-rxxtw          2/2       Running       0           24s
try1-895fccfb-srwq4          1/2       Running       0           11s
try1-895fccfb-vkvmb          2/2       Running       0           31s
try1-895fccfb-z46qr          2/2       Running       0           31s
```

Exercise 7.5: Domain Review

Very Important

The source pages and content in this review could change at any time. **IT IS YOUR RESPONSIBILITY TO CHECK THE CURRENT INFORMATION.**

Revisit the CKAD domain list on Curriculum Overview and locate some of the topics we have covered in this chapter. The graphic shows bullet points covered in this chapter.



- Understand PersistentVolumeClaims for storage
- Create & consume Secrets
- Understand ConfigMaps
- Understand Deployments and how to perform rolling updates
- Understand Deployments and how to perform rollbacks

Figure 7.2: **Deployment Related Domain Topics**

Focus on ensuing you have all the necessary files and processes understood first. Repeat the review until you are sure you have bookmarks of necessary YAML samples and can complete each step quickly, and ensure each object is running properly.

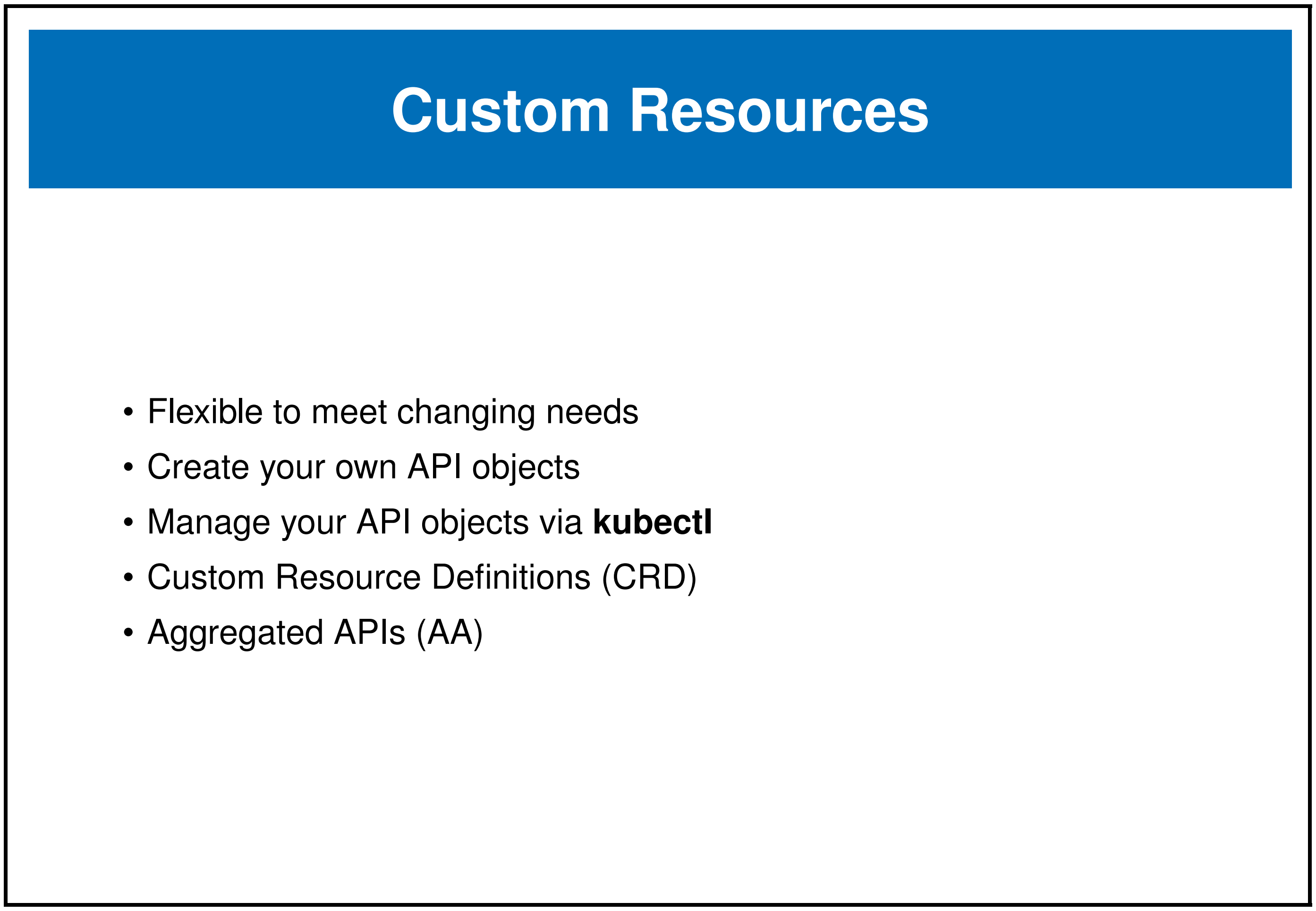
Using only the allowed browser, URLs, and subdomains search for and bookmark a YAML example to create and configure the resources called for in this review.

1. Create a new secret called `specialofday` using the key `entree` and the value `meatloaf`.
2. Create a new deployment called `foodie` running the `nginx` image.
3. Add the `specialofday` secret to pod mounted as a volume under the `/food/` directory.
4. Execute a bash shell inside a `foodie` pod and verify the secret has been properly mounted.
5. Update the deployment to use the `nginx:1.12.1-alpine` image and verify the new image is in use.
6. Roll back the deployment and verify the typical, current stable version of nginx is in use again.
7. Create a new 200M NFS volume called `reviewvol` using the NFS server configured earlier in the lab.
8. Create a new PVC called `reviewpvc` which will uses the `reviewvol` volume.
9. Edit the deployment to use the PVC and mount the volume under `/newvol`
10. Execute a bash shell into the nginx container and verify the volume has been mounted.
11. Delete any resources created during this review.

Chapter 8

Custom Resource Definition

8.1 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 mange one or more instances of a specific stateful application. We have worked with built-in controllers such for `Deployments`, `DaemonSets` and other resources.

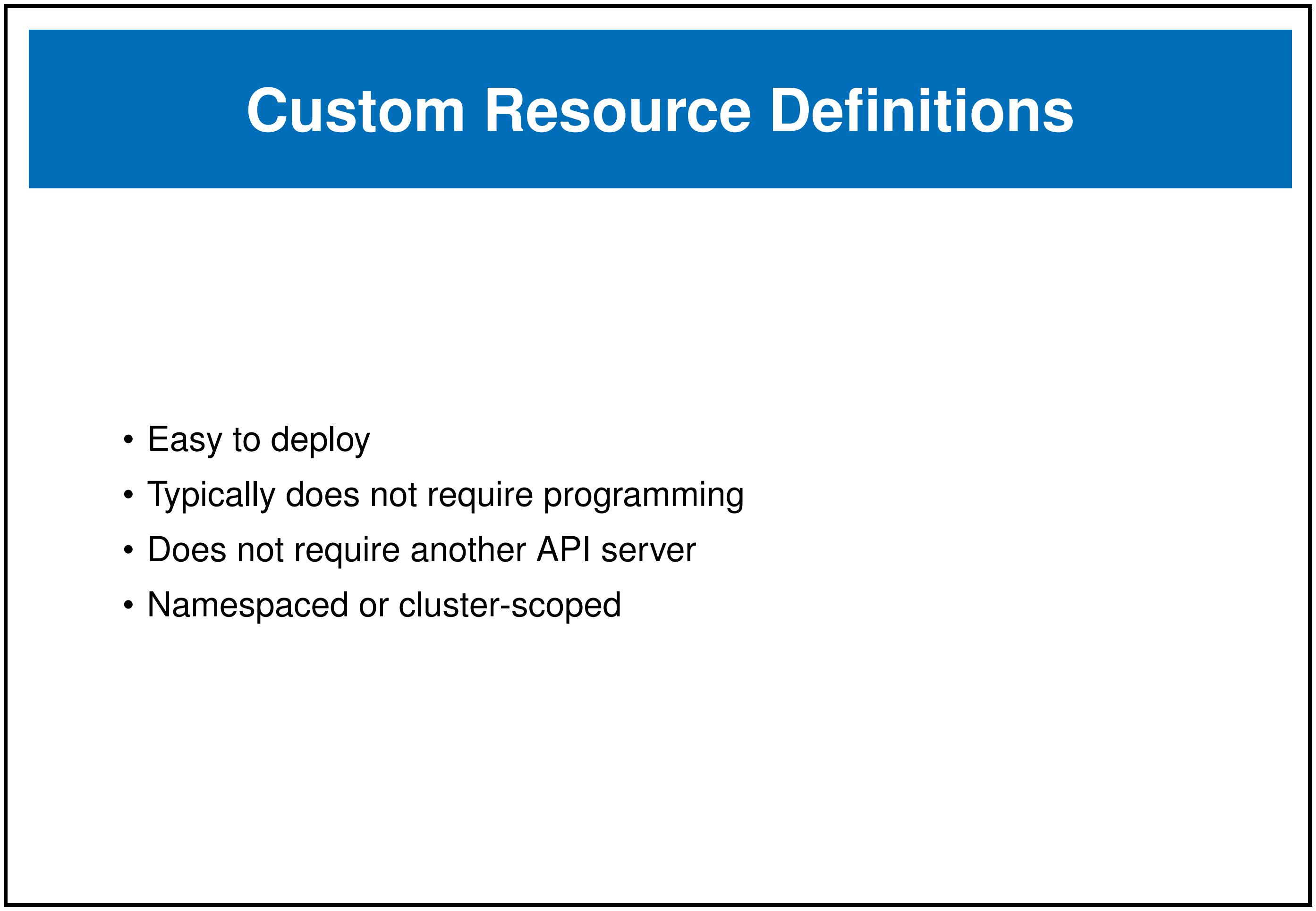
The functions encoded into a custom operator should be all the tasks a human would need to perform if deploying the application outside of Kubernetes. The details of building a custom controller are outside the scope of this course and not included.

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.

8.2 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
  version: 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 version 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 and 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
```

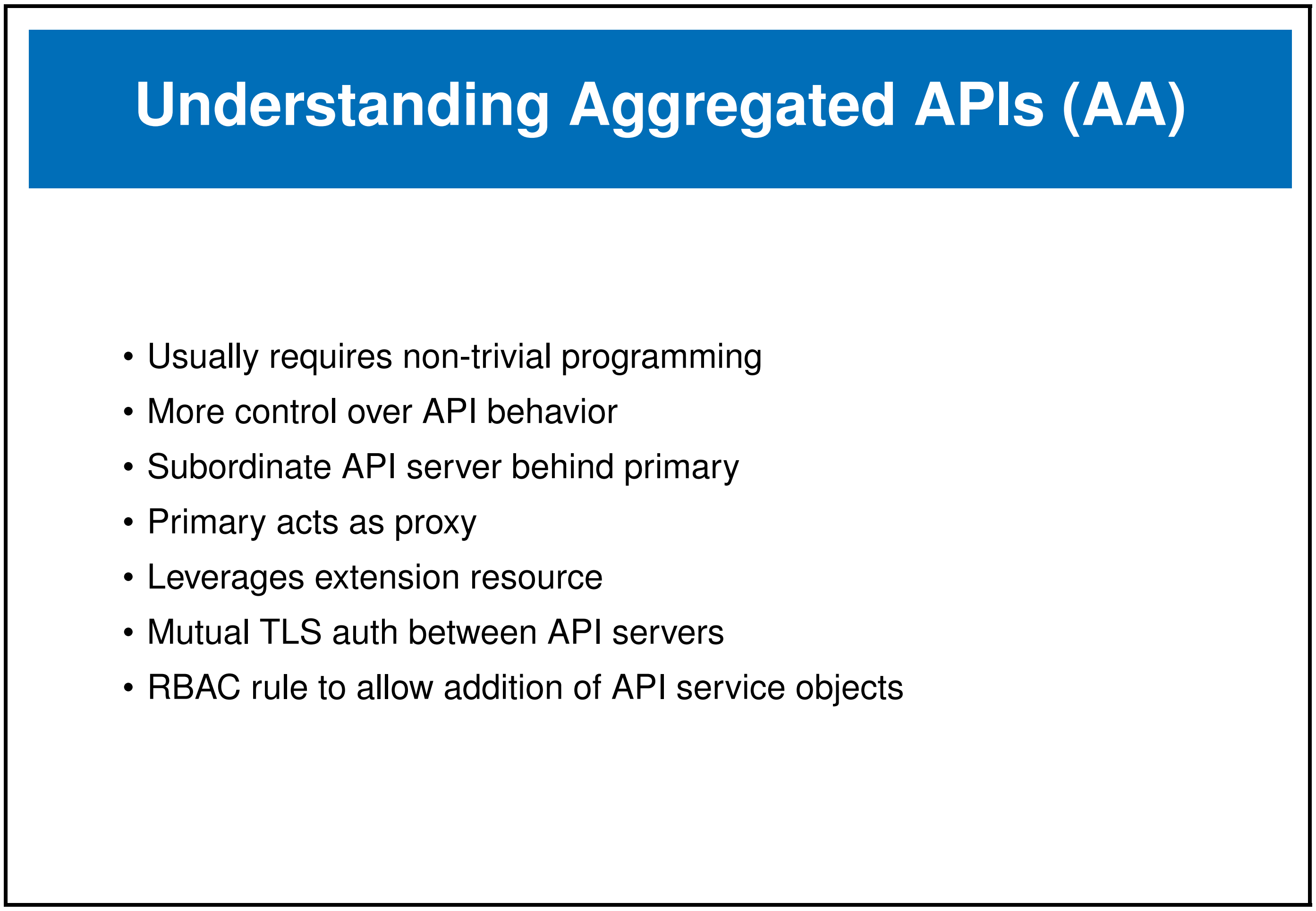
- Validation

```
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.

8.3 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-stype 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

8.4 Labs

Exercise 8.1: 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 controller can be found here: https://coreos.com/blog/introducing-operators.html.

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@master:~$ kubectl get crd --all-namespaces
```

```
NAME                                          CREATED AT
bgpconfigurations.crd.projectcalico.org       2020-04-19T17:29:02Z
bgppeers.crd.projectcalico.org                2020-04-19T17:29:02Z
blockaffinities.crd.projectcalico.org         2020-04-19T17:29:02Z
<output_omitted>
```

2. We can see from the names that these CRDs are all working on `Calico`, out network plugin. View the `calico.yaml` file we used when we initialized the cluster to see how these objects were created, and some CRD templates to review.

```
student@master:~$ less calico.yaml
```

```
<output_omitted>
---
# Source: calico/templates/kdd-crds.yaml

apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: bgpconfigurations.crd.projectcalico.org
<output_omitted>
```

3. Now that we have seen some examples, we will create a new YAML file.

```
student@master:~$ vim crd.yaml
```

crd.yaml

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: crontabs.training.lfs458.com
         # This name must match names below.
         # <plural>.<group> syntax
spec:
  scope: Cluster    #Could also be Namespaced
  group: training.lfs458.com
  version: v1
  names:
    kind: CronTab      #Typically CamelCased for resource manifest
    plural: crontabs   #Shown in URL
    singular: crontab  #Short name for CLI alias
    shortNames:
    - ct               #CLI short name
```

4. Add the new resource to the cluster.

```
student@master:~$ kubectl create -f crd.yaml
```

```
customresourcedefinition.apiextensions.k8s.io/crontabs.training.lfs458.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@master:~$ kubectl get crd
```

```
NAME                             CREATED AT
<output_omitted>
crontabs.training.lfs458.com   2018-08-03T05:25:20Z
<output_omitted>
```

```
student@master:~$ kubectl describe crd crontab<Tab>
```

```
Name:          crontabs.training.lfs458.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@master:~$ vim new-crontab.yaml
```

new-crontab.yaml

```
apiVersion: "training.lfs458.com/v1"
    # This is from the group and version of new CRD
kind: CronTab
    # The kind from the new CRD
metadata:
  name: new-cron-object
spec:
  cronSpec: "*/5 * * * *"
  image: some-cron-image
    #Does not exist
```

7. Create the new object and view the resource using short and long name.

```
student@master:~$ kubectl create -f new-crontab.yaml
```

```
crontab.training.lfs458.com/new-cron-object created
```

```
student@master:~$ kubectl get CronTab
```

```
NAME              AGE
new-cron-object   22s
```

```
student@master:~$ kubectl get ct
```

```
NAME              AGE
new-cron-object   29s
```

```
student@master:~$ kubectl describe ct
```

```
Name:         new-cron-object
Namespace:
Labels:       <none>

<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@master:~$ kubectl delete -f crd.yaml
```

```
customresourcedefinition.apiextensions.k8s.io
"crontabs.training.lfs458.com" deleted
```

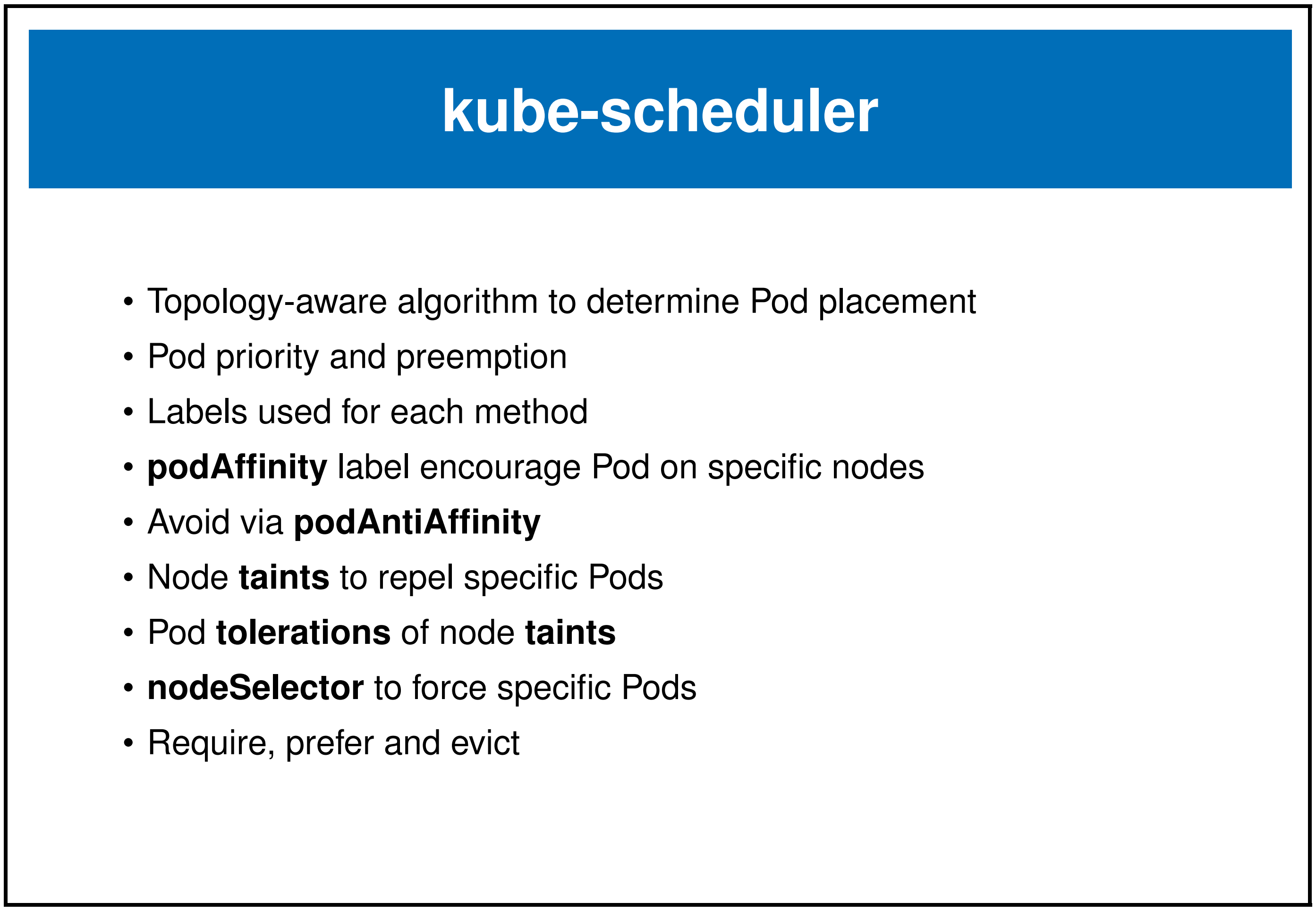
```
student@master:~$ kubectl get ct
```

```
Error from server (NotFound): Unable to list "training.lfs458.com/v1, Resource=crontabs":
the server could not find the requested resource (get crontabs.training.lfs458.com)
```

Chapter 9

Scheduling

9.1 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 them based on a set of `predicates`, then uses `priority functions` 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.

9.2 Scheduler Settings

Predicates

- **Predicates** check used to filter out inappropriate nodes
- Particular order from `predicates.go`

```
predicatesOrdering = []string{CheckNodeConditionPred,
GeneralPred, HostNamePred, PodFitsHostPortsPred,
MatchNodeSelectorPred, PodFitsResourcesPred, NoDiskConflictPred,
PodToleratesNodeTaintsPred, PodToleratesNodeNoExecuteTaintsPred,
CheckNodeLabelPresencePred, checkServiceAffinityPred,
MaxEBSVolumeCountPred, MaxGCEPDVolumeCountPred,
MaxAzureDiskVolumeCountPred, CheckVolumeBindingPred,
NoVolumeZoneConflictPred, CheckNodeMemoryPressurePred,
CheckNodeDiskPressurePred, MatchInterPodAffinityPred}
```

The scheduler goes through a set of filters, or `predicates` to find available nodes, then ranks each node using `priority functions`. The node with the highest rank is selected to run the Pod.

The **predicates**, such as `PodFitsHost` or `NoDiskConflict` are evaluated in a particular and configurable order. In this way a node has the least amount of checks for new Pod deployment, which can be useful to exclude a node from unnecessary checks if node is not in the proper condition.

For example, there is a filter called `HostNamePred`, which is also known as `HostName`, which filters out nodes that do not match the node name specified in the pod specification. Another predicate for `PodFitsResources` to make sure that the available CPU and memory can fit the resources required by the Pod.

The scheduler can be updated by passing a configuration of `kind: Policy` which can order predicates, give special weights to priorities and even `hardPodAffinitySymmetricWeight` which deploys Pods such that if we set Pod A to run with Pod B, then Pod B should automatically be run with Pod A.

Priorities

- Set of **priority functions** used to weight resources
- Iteration of current status each adding or removing 1 to priority
- Highest value node chosen by scheduler
- `SelectorSpreadPriority` assigned to node with least pods
- View list at `master/pkg/scheduler/algorithm/priorities`
- **PriorityClasses** can be used to allocate and preempt other pods
- Use a `Pod Disruption Budget` to limit disruption by preemption

Unless Pod and node affinity has been configured the `SelectorSpreadPriority` setting, which ranks nodes based on the number of existing running pods, will select the node with the least amount of Pods. This is a basic way to spreading Pods across the cluster.

Other priorities can be used for particular cluster needs. The `ImageLocalityPriorityMap` favors nodes which already have downloaded container images. The total sum of image size is compared with the largest having highest priority, but does not check the image about to be used.

There currently are more than ten included priorities which range from checking the existence of a label to choosing a node with the most requested CPU and memory usage.

A feature stable as of v1.14 allows the setting of a `PriorityClass` and assigning pods via the use of `PriorityClassName` settings. This allows user to preempt, or evict, lower priority pods so that their higher priority pods can be scheduled. The **kube-scheduler** determines a node where the pending pod could run if one or more existing pods were evicted. If a node is found the low priority pod(s) are evicted and the higher priority pod scheduled. The use of a `Pod Disruption Budget (PDB)` is a way to limit the number of pods preemption evicts to ensure enough pods remain running. The scheduler will remove pods even if the PDB is violated if no other options are available.

9.3 Policies

Scheduling Policies

```
"kind" : "Policy",
"apiVersion" : "v1",
"predicates" : [
        {"name" : "MatchNodeSelector", "order": 6},
        {"name" : "PodFitsHostPorts", "order": 2},
        {"name" : "PodFitsResources", "order": 3},
        {"name" : "NoDiskConflict", "order": 4},
        {"name" : "PodToleratesNodeTaints", "order": 5},
        {"name" : "PodFitsHost", "order": 1}
        ],
"priorities" : [
        {"name" : "LeastRequestedPriority", "weight" : 1},
        {"name" : "BalancedResourceAllocation", "weight" : 1},
        {"name" : "ServiceSpreadingPriority", "weight" : 2},
        {"name" : "EqualPriority", "weight" : 1}
        ],
"hardPodAffinitySymmetricWeight" : 10
}
```

The default scheduler contains a number of predicates and priorities; however, these can be changed via a scheduler policy file.

Typically, you will configure a scheduler with this policy using the `--policy-config-file` parameter and define a name for this scheduler using the `--scheduler-name` parameter. You will then have two schedulers running and will be able to specify which scheduler to use in the Pod specification.

With multiple schedulers there could be conflict in Pod allocation. Each Pod should declare which scheduler should be used. But if separate schedulers determine a node is eligible because of available resources and both attempt to deploy, causing the resource to no longer be available, a conflict would occur. The current solution is for the local `kubelet` to return the Pods to the scheduler for reassignment. Eventually one Pod will succeed and the other be scheduled elsewhere.

Pod Specification

- Pod Specification fields
  - `nodeName`
  - `nodeSelector`
  - `affinity`
  - `tolerations`
  - `schedulerName`

Most scheduling decisions can be made as part of the the Pod spec. 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 `master` 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`.

9.4 Affinity Rules

Pod Affinity Rules

- Uses `In`, `NotIn`, `Exists`, and `DoesNotExist` operators
- requiredDuringSchedulingIgnoredDuringExecution
- preferredDuringSchedulingIgnoredDuringExecution
- affinity
  - podAffinity
  - podAntiAffinity
- topologyKey

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.

The `topologyKey` allows a general grouping of Pod deployment. Affinity (or the inverse anti-affinity) will try to run on nodes with the declared topology key and running Pods with a particular label.

The `topologyKey` could be any legal key, but there some important considerations:

- If using `requiredDuringScheduling` and the admission controller `LimitPodHardAntiAffinityTopology` setting the `topologyKey` must be set to `kubernetes.io/hostname`.
- If using `PreferredDuringScheduling` an empty `topologyKey` is assumed to be all, or the combination of `kubernetes.io/hostname`, `topology.kubernetes.io/zone` and `topology.kubernetes.io/region`.

podAffinity Example

```
spec:
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchExpressions:
          - key: security
            operator: In
            values:
            - S1
        topologyKey: topology.kubernetes.io/zone
```

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.

Inside the declared topology zone 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
      topologyKey: kubernetes.io/hostname
```

With `podAntiAffinity` we can prefer to avoid nodes with a particular label. In this case the scheduler will prefer to avoid a node with a key set to `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:
requiredDuringSchedulingIgnoredDuringExecution:
nodeSelectorTerms:
- matchExpressions:
- key: kubernetes.io/colo-tx-name
operator: In
values:
- tx-aus
- tx-dal
preferredDuringSchedulingIgnoredDuringExecution:
- weight: 1
preference:
matchExpressions:
- key: disk-speed
operator: In
values:
- fast
- quick
```

The first `nodeAffinity` rule requires a node with a key of `kubernetes.io/colo-tx-name` which has one of two possible values `tx-aus` or `tx-dal`.

The second rule gives extra weight to nodes with a key of `disk-speed` with a value of `fast` or `quick`. The Pod will be scheduled on some node in any case this just prefers a particular label.

9.5 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 **effect**s 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.

9.6 Labs

Exercise 9.1: 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@master:˜$ kubectl get nodes
```

```
NAME                STATUS    ROLES     AGE       VERSION
master    Ready     master    44h        v1.19.0
worker       Ready      <none>     43h         v1.19.0
```

2. View the current labels and taints for the nodes.

```
student@master:˜$ 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=c-jg5n
                    kubernetes.io/os=linux
                    node-role.kubernetes.io/master=
--
Labels:             beta.kubernetes.io/arch=amd64
                    beta.kubernetes.io/os=linux
                    kubernetes.io/arch=amd64
                    kubernetes.io/hostname=c-v2t3
                    kubernetes.io/os=linux
--
```

```
student@master:˜$ kubectl describe nodes |grep -i taint
```

```
Taints:                        <none>
Taints:                        <none>
```

3. Get a count of how many containers are running on both the master and worker nodes. There are about 24 containers running on the master 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. If you are using **cri-o** you can view containers using **crictl ps**.

```
student@master:˜$ kubectl get deployments --all-namespaces
```

```
NAMESPACE        NAME                       READY   UP-TO-DATE  AVAILABLE   AGE
default          secondapp                  1       1           1           37m
default          thirdpage                  1       1           1           24m
kube-system      calico-kube-controllers    0       0           0           44h
kube-system      coredns                    2/2     2           2           44h
```

```
student@master:˜$ sudo docker ps |wc -l
```

```
21
```

```
student@worker:~$ sudo docker ps |wc -l
```

```
11
```

4. For the purpose of the exercise we will assign the master node to be VIP hardware and the secondary node to be for others.

```
student@master:~$ kubectl label nodes master status=vip
```

```
node/master labeled
```

```
student@master:~$ 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@master:~$ kubectl get nodes --show-labels
```

```
NAME              STATUS    ROLES     AGE    VERSION   LABELS
master   Ready     master    44h    v1.19.0   beta.kubernetes.io/arch=
amd64,beta.kubernetes.io/os=linux,kubernetes.io/hostname=master,
node-role.kubernetes.io/master=,status=vip
worker     Ready     <none>    44h    v1.19.0   beta.kubernetes.io/arch=
amd64,beta.kubernetes.io/os=linux,kubernetes.io/hostname=worker,status=other
```

6. Create `vip.yaml` to spawn four `busybox` containers which sleep the whole time. Include the `nodeSelector` entry.

```
student@master:~$ vim vip.yaml
```

vip.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: vip
spec:
  containers:
  - name: vip1
    image: busybox
    args:
    - sleep
    - "1000000"
  - name: vip2
    image: busybox
    args:
    - sleep
    - "1000000"
  - name: vip3
    image: busybox
    args:
    - sleep
    - "1000000"
  - name: vip4
    image: busybox
```

```yaml
      args:
      - sleep
      - "1000000"
    nodeSelector:
      status: vip
```

7. Deploy the new pod. Verify the containers have been created on the master node. It may take a few seconds for all the containers to spawn. Check both the master and the secondary nodes.

```
student@master:~$ kubectl create -f vip.yaml
```

```
pod/vip created
```

```
student@master:~$ sudo docker ps |wc -l
```

```
26
```

```
student@worker:~$ sudo docker ps |wc -l
```

```
11
```

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@master:~$ kubectl delete pod vip
```

```
pod "vip" deleted
```

```
student@master:~$ vim vip.yaml
```

```
....
#  nodeSelector:
 #    status: vip
```

9. Create the pod again. Containers should now be spawning on either node. You may see pods for the `daemonsets` as well.

```
student@master:~$ kubectl get pods
```

```
NAME                          READY   STATUS    RESTARTS   AGE
secondapp-85765cd95c-2q9sx    1/1     Running   0          43m
thirdpage-7c9b56bfdd-2q5pr    1/1     Running   0          30m
```

```
student@master:~$ 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.

```
student@master:~$ sudo docker ps |wc -l
```

```
21
```

```
student@worker:~$ sudo docker ps |wc -l
```

```
16
```

11. Create another file for other users. Change the names from vip to others, and uncomment the nodeSelector lines.

```
student@master:~$ cp vip.yaml other.yaml

student@master:~$ sed -i s/vip/other/g other.yaml

student@master:~$ vim other.yaml
```

other.yaml

```yaml
....
  nodeSelector:
    status: other
```

12. Create the other containers. Determine where they deploy.

```
student@master:~$ kubectl create -f other.yaml
```

```
pod/other created
```

```
student@master:~$ sudo docker ps |wc -l
```

```
21
```

```
student@worker:~$ sudo docker ps |wc -l
```

```
21
```

13. Shut down both pods and verify they terminated. Only our previous pods should be found.

```
student@master:~$ kubectl delete pods vip other
```

```
pod "vip" deleted
pod "other" deleted
```

```
student@master:~$ kubectl get pods
```

```
NAME                         READY   STATUS    RESTARTS   AGE
secondapp-85765cd95c-2q9sx   1/1     Running   0          51m
thirdpage-7c9b56bfdd-2q5pr   1/1     Running   0          40m
```

Exercise 9.2: 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 master node begins with a NoSchedule taint. We will work with three taints to limit or remove running pods.

1. Verify that the master and secondary node have the minimal number of containers running.

```
student@master:~$ kubectl delete deployment secondapp \
     thirdpage
```

```
deployment.apps "secondapp" deleted
deployment.apps "thirdpage" deleted
```

2. Create a deployment which will deploy eight **nginx** containers. Begin by creating a YAML file.

```
student@master:~$ vim taint.yaml
```

taint.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: taint-deployment
spec:
  replicas: 8
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name:  nginx
        image: nginx:1.16.1
        ports:
        - containerPort: 80
```

3. Apply the file to create the deployment.

```
student@master:~$ kubectl apply -f taint.yaml
```

```
deployment.apps/taint-deployment created
```

4. Determine where the containers are running. In the following example three have been deployed on the master node and five on the secondary node. Remember there will be other housekeeping containers created as well. Your numbers may be slightly different.

```
student@master:~$ sudo docker ps |grep nginx
```

```
00c1be5df1e7        nginx@sha256:e3456c851a152494c3e.....
<output_omitted>
```

```
student@master:~$ sudo docker ps |wc -l
```

```
27
```

```
student@worker:~$ sudo docker ps |wc -l
```

```
17
```

5. Delete the deployment. Verify the containers are gone.

```
student@master:~$ kubectl delete deployment taint-deployment
```

```
deployment.apps "taint-deployment" deleted
```

```
student@master:~$ sudo docker ps |wc -l
```

```
21
```

6. 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@master:~$ kubectl taint nodes worker \
          bubba=value:PreferNoSchedule
```

```
node/worker tainted
```

```
student@master:~$ kubectl describe node |grep Taint
```

```
Taints:                       bubba=value:PreferNoSchedule
Taints:                       <none>
```

```
student@master:~$ kubectl apply -f taint.yaml
```

```
deployment.apps/taint-deployment created
```

7. Locate where the containers are running. We can see that more containers are on the master, but there still were some created on the secondary. Delete the deployment when you have gathered the numbers.

```
student@master:~$ sudo docker ps |wc -l
```

```
21
```

```
student@worker:~$ sudo docker ps |wc -l
```

```
23
```

```
student@master:~$ kubectl delete deployment taint-deployment
```

```
deployment.apps "taint-deployment" deleted
```

8. Remove the taint, verify it has been removed. Note that the key is used with a minus sign appended to the end.

```
student@master:~$ kubectl taint nodes worker bubba-
```

```
node/worker untainted
```

```
student@master:~$ kubectl describe node |grep Taint
```

```
Taints:                       <none>
Taints:                       <none>
```

9. 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@master:~$ kubectl taint nodes worker \
         bubba=value:NoSchedule
```

```
node/worker tainted
```

```
student@master:~$ kubectl apply -f taint.yaml
```

```
deployment.apps/taint-deployment created
```

```
student@master:~$ sudo docker ps |wc -l
```

```
21
```

```
student@worker:~$ sudo docker ps |wc -l
```

```
23
```

10. 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@master:~$ kubectl delete deployment taint-deployment
```
```
deployment.apps "taint-deployment" deleted
```

```
student@master:~$ kubectl taint nodes worker bubba-
```
```
node/worker untainted
```

```
student@master:~$ kubectl apply -f taint.yaml
```
```
deployment.apps/taint-deployment created
```

```
student@master:~$ sudo docker ps |wc -l
```
```
27
```

```
student@worker:~$ sudo docker ps |wc -l
```
```
17
```

11. 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@master:~$ kubectl taint nodes worker \
        bubba=value:NoExecute
```
```
node "worker" tainted
```

```
student@master:~$ sudo docker ps |wc -l
```
```
37
```

```
student@worker:~$ sudo docker ps |wc -l
```
```
5
```

12. Remove the taint. Wait a minute. Note that all of the containers did not return to their previous placement.

```
student@master:~$ kubectl taint nodes worker bubba-
```
```
node/worker untainted
```

```
student@master:~$ sudo docker ps |wc -l
```
```
32
```

```
student@worker:~$ sudo docker ps |wc -l
```
```
6
```

13. Remove the deployment a final time to free up resources.

```
student@master:~$ kubectl delete deployment taint-deployment
```
```
deployment.apps "taint-deployment" deleted
```

Chapter 10

Security

In this chapter, we are going to talk about how an API call is ingested into the cluster. We will go through the details of the three different phases each call goes through. We'll look at the different types of authentication that are available to us, and work with RBAC, which handles the authorization on our behalf. We'll also configure pod policies. The use of a pod policy allows us to configure the details of what a container can do, and how processes run within that container.

We'll finish by understanding a network policy. If the network plugin honors a network policy, this allows you to isolate a pod from other pods in the environment. The default Kubernetes architecture says that all pods should be able to see all pods. So, this is a change from that historic approach. But, as we use it in a production environment, you may want to limit ingress

10.1 Security Overview

- 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.
- Then, we are going to focus on the authentication aspect of the API server and we will dive into authorization, looking at things like ABAC and 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 on 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 not to turn on network policies, which let any traffic flow through all of 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.

10.2 Accessing the API

- To perform any action in a Kubernetes cluster, you need to access the API and go through three main steps:
  - Authentication
  - Authorization (ABAC or RBAC)
  - Admission Control.
- These steps are described in more detail in the official documentation at https://kubernetes.io/docs/reference/access-authn-authz/controlling-access/ about controlling access to the API and illustrated by the picture below, from https://kubernetes.io/docs/reference/command-line-tools-reference/kube-apiserver/:

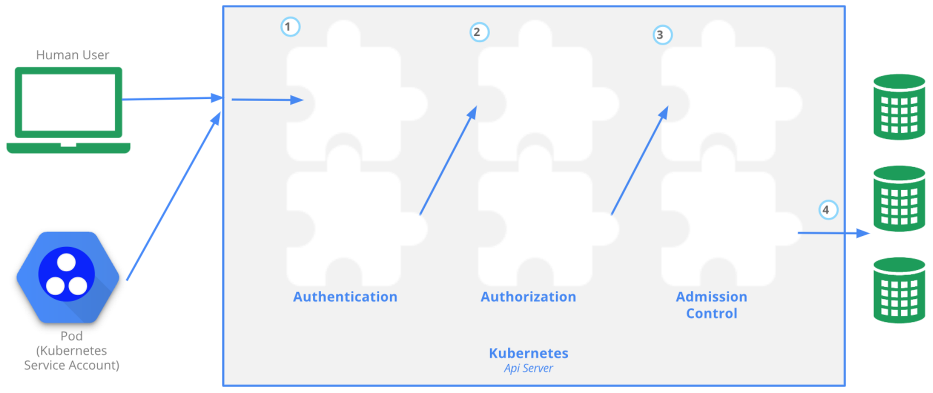


Figure 10.1: **Accessing the API**

- 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 Kelsey Hightower's guide, ***Kubernetes The Hard Way*** at https://github.com/kelseyhightower/kubernetes-the-hard-way, or the API server configuration options at https://kubernetes.io/docs/reference/command-line-tools-reference/kube-apiserver/.

10.3 Authentication

- There are three main points to remember with authentication in Kubernetes:

  1. In its straightforward form, authentication is done with certificates, tokens or basic authentication (i.e. username and password).
  2. Users are not created by the API, but should be managed by an external system.
  3. 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:

  – `--basic-auth-file`
  – `--oidc-issuer-url`
  – `--token-auth-file`
  – `--authorization-webhook-config-file`

- One or more Authenticator Modules are used: x509 Client Certs; static token, bearer or bootstrap token; static password file; service account and OpenID connect tokens. Each is tried until successful, and the order is not guaranteed. Anonymous access can also be enabled, otherwise you will get a 401 response. Users are not created by the API, and should be managed by an external system.

- To learn more about authentication, see the official Kubernetes documentation at https://kubernetes.io/docs/reference/access-authn-authz/authentication/.

10.4 Authorization

- 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. The three main modes are:

  1. ABAC
  2. RBAC
  3. WebHook

- They can be configured as kube-apiserver startup options:

```
--authorization-mode=ABAC 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).

10.5 ABAC

- **ABAC** stands for **Attribute Based Access Control**. It was the first authorization model in Kubernetes that allowed administrators to implement the right policies. Today, **RBAC** is becoming the default authorization mode.
- Policies are defined in a JSON file and referenced to by a **kube-apiserver** startup option:

```
--authorization-policy-file=my_policy.json
```

- For example, the policy file shown below authorizes user `Bob` to read pods in the namespace `foobar`:

```
{ "apiVersion":
  "abac.authorization.kubernetes.io/v1beta1", "kind":
  "Policy", "spec": { "user": "bob", "namespace":
    "foobar", "resource": "pods", "readonly": true } }
```

You can check other policy examples in the Kubernetes documentation.

10.6 RBAC

- **RBAC** (https://kubernetes.io/docs/reference/access-authn-authz/rbac/) stands for **Role Based Access Control**.
- 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 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.

10.7 RBAC Process Overview

- While RBAC can be complex, the basic flow is to create a certificate for a user. As a user is not an API object of Kubernetes, we are requiring outside authentication, such as OpenSSL certificates. After generating 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 in the cluster.
- Here is a summary of the RBAC process:
  - 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.

10.8 Admission Controller

- 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 matured. Starting with 1.13.1 release of the **kube-apiserver** the admission controllers are now compiled into the binary, instead of a list passed during execution. To enable or disable you can pass the following options, changing out the plugins you want to enable or disable.

  ```
  --enable-admission-plugins=NamespaceLifecycle,LimitRanger
  --disable-admission-plugins=PodNodeSelector
  ```

- The first controller is `Initializers` which will allow the 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.

10.9 Security Contexts

- 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 filesystem 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 below:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  securityContext:
    runAsNonRoot: true
  containers:
  - image: nginx
    name: nginx
```

- 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
```

- You can read more in the Kubernetes documentation about configuring security contexts to give proper constraints to your pods or containers at https://kubernetes.io/docs/tasks/configure-pod-container/security-context/.

10.10 Pod Security Policies

- To automate the enforcement of security contexts, you can define Pod Security Policies (**PSP**). (See https://kubernetes.io/docs/concepts/policy/pod-security-policy/. A PSP is defined via a standard Kubernetes manifest following the PSP API schema. An example is presented on the next page.
- These policies are cluster-level rules that govern what a pod can do, what they can access, what user they run as, etc.
- For instance, if you do not want any of the containers in your cluster to run as the root user, you can define a PSP to that effect. You can also prevent containers from being privileged or use the host network namespace, or the host PID namespace.
- While **PSP** has been helpful there are other methods gaining popularity. The **Open Policy Agent (OPA)**, often pronounced as ”oh-pa”, provides a unified set of tools and policy framework. This allows a single point of configuration for all of your cloud deployments.
- **OPA** can be deployed as an admission controller inside of Kubernetes, which allows OPA to enforce or mutate requests as they are received. Using the **OPA Gatekeeper** it can be deployed using Custom Resource Definitions. More can be found here: https://www.openpolicyagent.org/
- You can see an example of a PSP below:

```yaml
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: restricted
spec:
  seLinux:
    rule: RunAsAny
  supplementalGroups:
    rule: RunAsAny
  runAsUser:
    rule: MustRunAsNonRoot
  fsGroup:
    rule: RunAsAny
```

- For Pod Security Policies to be enabled, you need to configure the admission controller of the `controller-manager` to contain `PodSecurityPolicy`. These policies make even more sense when coupled with the **RBAC** configuration in your cluster. This will allow you to finely tune what your users are allowed to run and what capabilities and low level privileges their containers will have.
- See the PSP RBAC example at
  https://github.com/kubernetes/examples/blob/master/staging/podsecuritypolicy/rbac/README.md
  on **GitHub** for more details.

10.11 Network Security Policies

- 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 configuring 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 the 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 traffic 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.
- On the next page, you can find an example of a `NetworkPolicy` recipe. More network policy recipes can be found on GitHub at https://github.com/ahmetb/kubernetes-network-policy-recipes.

10.12 Network Security Policy Example

- The use of policies has become stable, noted with the `v1 apiVersion`. The example below narrows down the policy to affect the default namespace.
- Only Pods with the label of `role: db` will be affected by this policy, and the policy 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.

```yaml
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
  - 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 the `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 this is not suggested. Use another dedicated `NetworkPolicy` instead.
- Note that 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"]}
```

10.13 Default Policy Example

- The empty braces will match all Pods not selected by other `NetworkPolicy` and will not allow ingress traffic. Egress traffic would be unaffected by this policy.

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny
spec:
  podSelector: {}
  policyTypes:
  - Ingress
```

- 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:

```yaml
kind: Namespace
apiVersion: v1
metadata:
  name: myns
  annotations:
    net.beta.kubernetes.io/network-policy: |
     {
        "ingress": {
          "isolation": "DefaultDeny"
        }
     }
```

10.14 Labs

Exercise 10.1: Set SecurityContext for a Pod and Container

Working with Security: Overview

In this lab we will implement security features for new applications, as the simpleapp YAML file is getting long and more difficult to read. Kubernetes architecture favors smaller, decoupled, and transient applications working together. We'll continue to emulate that in our exercises.

In this exercise we will create two new applications. One will be limited in its access to the host node, but have access to encoded data. The second will use a `network security policy` to move from the default all-access Kubernetes policies to a mostly closed network. First we will set `security contexts` for pods and containers, then create and consume secrets, then finish with configuring a network security policy.

1. Begin by making a new directory for our second application. Change into that directory.

```
student@master:~$ mkdir $HOME/app2
```

```
student@master:~$ cd $HOME/app2/
```

2. Create a YAML file for the second application. In the example below we are using a simple image, busybox, which allows access to a shell, but not much more. We will add a `runAsUser` to both the pod as well as the container.

```
student@master:~/app2$ vim second.yaml
```

second.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: secondapp
spec:
  securityContext:
    runAsUser: 1000
  containers:
  - name: busy
    image: busybox
    command:
      - sleep
      - "3600"
    securityContext:
      runAsUser: 2000
      allowPrivilegeEscalation: false
```

3. Create the secondapp pod and verify it's running. Unlike the previous deployment this application is running as a pod. Look at the YAML output, to compare and contrast with what a deployment looks like. The status section probably has the largest contrast.

```
student@master:~/app2$ kubectl create -f second.yaml
```

```
pod/secondapp created
```

```
student@master:~/app2$ kubectl get  pod secondapp
```

```
NAME        READY     STATUS    RESTARTS    AGE
secondapp   1/1       Running   0           21s
```

```
student@master:~/app2$ kubectl get pod secondapp -o yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  annotations:
    cni.projectcalico.org/podIP: 192.168.158.97/32
  creationTimestamp: "2019-11-03T21:23:12Z"
  name: secondapp
<output_omitted>
```

4. Execute a Bourne shell within the Pod. Check the user ID of the shell and other processes. It should show the container setting, not the pod. This allows for multiple containers within a pod to customize their UID if desired. As there is only one container in the pod we do not need to use the **-c busy** option.

```
student@master:~/app2$ kubectl exec -it secondapp -- sh
```

On Container

```
/ $ ps aux
```

```
PID   USER     TIME  COMMAND
    1   2000        0:00   sleep 3600
    8   2000        0:00   sh
   12   2000       0:00   ps aux
```

5. While here check the capabilities of the kernel. In upcoming steps we will modify these values.

On Container

```
/ $ grep Cap /proc/1/status
```

```
CapInh:        00000000a80425fb
CapPrm:        0000000000000000
CapEff:        0000000000000000
CapBnd:        00000000a80425fb
CapAmb:        0000000000000000
```

```
/ $ exit
```

6. Use the capability shell wrapper tool, the **capsh** command, to decode the output. We will view and compare the output in a few steps. Note that there are 14 comma separated capabilities listed.

```
student@master:~/app2$ capsh --decode=00000000a80425fb
```

```
0x00000000a80425fb=cap_chown,cap_dac_override,cap_fowner,cap_fsetid,
cap_kill,cap_setgid,cap_setuid,cap_setpcap,cap_net_bind_service,
cap_net_raw,cap_sys_chroot,cap_mknod,cap_audit_write,cap_setfcap
```

7. Edit the YAML file to include new capabilities for the container. A capability allows granting of specific, elevated privileges without granting full root access. We will be setting **NET_ADMIN** to allow interface, routing, and other network configuration. We'll also set **SYS_TIME**, which allows system clock configuration. More on kernel capabilities can be read here: https://github.com/torvalds/linux/blob/master/include/uapi/linux/capability.h

   It can take up to a minute for the pod to fully terminate, allowing the future pod to be created.

```
student@master:~/app2$ kubectl delete pod secondapp
```

```
pod "secondapp" deleted
```

```
student@master:~/app2$ vim second.yaml
```

second.yaml

```
<output_omitted>
      - sleep
      - "3600"
```

```yaml
    securityContext:
      runAsUser: 2000
      allowPrivilegeEscalation: false
      capabilities:                     #<-- Add this and following line
        add: ["NET_ADMIN", "SYS_TIME"]
```

8. Create the pod again. Execute a shell within the container and review the Cap settings under `/proc/1/status`. They should be different from the previous instance.

```
student@master:~/app2$ kubectl create -f second.yaml
```

```
pod/secondapp created
```

```
student@master:~/app2$ kubectl exec -it secondapp -- sh
```

On Container

```
/ $ grep Cap /proc/1/status
```

```
CapInh:         00000000aa0435fb
CapPrm:         0000000000000000
CapEff:         0000000000000000
CapBnd:         00000000aa0435fb
CapAmb:         0000000000000000
```

```
/ $ exit
```

9. Decode the output again. Note that the instance now has 16 comma delimited capabilities listed. **cap_net_admin** is listed as well as **cap_sys_time**.

```
student@master:~/app2$ capsh --decode=00000000aa0435fb
```

```
0x00000000aa0435fb=cap_chown,cap_dac_override,cap_fowner,
cap_fsetid,cap_kill,cap_setgid,cap_setuid,cap_setpcap,
cap_net_bind_service,cap_net_admin,cap_net_raw,cap_sys_chroot,
cap_sys_time,cap_mknod,cap_audit_write,cap_setfcap
```

Exercise 10.2: Create and consume Secrets

Secrets are consumed in a manner similar to `ConfigMaps`, covered in an earlier lab. While at-rest encryption is just now enabled, historically a secret was just base64 encoded. There are three types of encryption which can be configured.

1. Begin by generating an encoded password.

```
student@master:~/app2$ echo LFTr@1n | base64
```

```
TEZUckAxbgo=
```

2. Create a YAML file for the object with an API object kind set to Secret. Use the encoded key as a password parameter.

```
student@master:~/app2$ vim secret.yaml
```

secret.yaml

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: lfsecret
data:
  password: TEZUckAxbgo=
```

3. Ingest the new object into the cluster.

```
student@master:~/app2$ kubectl create -f secret.yaml
```

```
secret/lfsecret created
```

4. Edit secondapp YAML file to use the secret as a volume mounted under /mysqlpassword. volumeMounts: lines up with the container name: and volumes: lines up with containers: Note the pod will restart when the sleep command finishes every 3600 seconds, or every hour.

```
student@master:~/app2$ vim second.yaml
```

second.yaml

```yaml
....
        runAsUser: 2000
        allowPrivilegeEscalation: false
        capabilities:
          add: ["NET_ADMIN", "SYS_TIME"]
      volumeMounts:                         #<-- Add this and six following lines
      - name: mysql
        mountPath: /mysqlpassword
    volumes:
    - name: mysql
      secret:
        secretName: lfsecret
```

```
student@master:~/app2$ kubectl delete pod secondapp
```

```
pod "secondapp" deleted
```

```
student@master:~/app2$ kubectl create -f second.yaml
```

```
pod/secondapp created
```

5. Verify the pod is running, then check if the password is mounted where expected. We will find that the password is available in its clear-text, decoded state.

```
student@master:~/app2$ kubectl get pod secondapp
```

```
NAME        READY     STATUS    RESTARTS   AGE
secondapp   1/1       Running   0          34s
```

```
student@master:~/app2$ kubectl exec -ti secondapp -- /bin/sh
```

On Container

```
/ $ cat /mysqlpassword/password
```

```
LFTr@1n
```

6. View the location of the directory. Note it is a symbolic link to `..data` which is also a symbolic link to another directory. After taking a look at the filesystem within the container, exit back to the node.

On Container

```
/ $ cd /mysqlpassword/

/mysqlpassword $ ls
```

```
password
```

```
/mysqlpassword $ ls -al
```

```
total 4
drwxrwxrwt    3 root     root         100 Apr 11 07:24 .
drwxr-xr-x   21 root     root        4096 Apr 11 22:30 ..
drwxr-xr-x    2 root     root          60 Apr 11 07:24 ..4984_11_04_07_24_47.831222818
lrwxrwxrwx    1 root     root          31 Apr 11 07:24 ..data -> ..4984_11_04_07_24_47.831222818
lrwxrwxrwx    1 root     root          15 Apr 11 07:24 password -> ..data/password
```

```
/mysqlpassword $ exit
```

Exercise 10.3: Working with ServiceAccounts

We can use `ServiceAccounts` to assign cluster roles, or the ability to use particular HTTP verbs. In this section we will create a new `ServiceAccount` and grant it access to view secrets.

1. Begin by viewing secrets, both in the default namespace as well as all.

```
student@master:~/app2$ cd

student@master:~$ kubectl get secrets
```

```
NAME                  TYPE                                  DATA   AGE
default-token-c4rdg   kubernetes.io/service-account-token   3      4d16h
lfsecret              Opaque                                1      6m5s
```

```
student@master:~$ kubectl get secrets --all-namespaces
```

```
NAMESPACE     NAME
 TYPE                                  DATA   AGE
default       default-token-c4rdg
 kubernetes.io/service-account-token   3      4d16h
kube-public   default-token-zqzbg
 kubernetes.io/service-account-token   3      4d16h
kube-system   attachdetach-controller-token-wxzvc
 kubernetes.io/service-account-token   3      4d16h
<output_omitted>
```

2. We can see that each agent uses a secret in order to interact with the API server. We will create a new `ServiceAccount` which will have access.

```
student@master:˜$ vim serviceaccount.yaml
```

serviceaccount.yaml

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
 name: secret-access-sa
```

```
student@master:˜$ kubectl create -f serviceaccount.yaml
```

```
serviceaccount/secret-access-sa created
```

```
student@master:˜$ kubectl get serviceaccounts
```

```
NAME                SECRETS   AGE
default             1         1d17h
secret-access-sa    1         34s
```

3. Now we will create a `ClusterRole` which will list the actual actions allowed cluster-wide. We will look at an existing role to see the syntax.

```
student@master:˜$ kubectl get clusterroles
```

```
NAME                                                          AGE
admin                                                         1d17h
calico-cni-plugin                                             1d17h
calico-kube-controllers                                       1d17h
cluster-admin                                                 1d17h
<output_omitted>
```

4. View the details for the `admin` and compare it to the `cluster-admin`. The `admin` has particular actions allowed, but `cluster-admin` has the meta-character '*' allowing all actions.

```
student@master:˜$ kubectl get clusterroles admin -o yaml
```

```
<output_omitted>
```

```
student@master:˜$ kubectl get clusterroles cluster-admin -o yaml
```

```
<output_omitted>
```

5. Using some of the output above, we will create our own file.

```
student@master:˜$ vim clusterrole.yaml
```

clusterrole.yaml

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
 name: secret-access-cr
rules:
- apiGroups:
   - ""
  resources:
  - secrets
  verbs:
  - get
  - list
```

6. Create and verify the new `ClusterRole`.

```
student@master:~$ kubectl create -f clusterrole.yaml
```

```
clusterrole.rbac.authorization.k8s.io/secret-access-cr created
```

```
student@master:~$ kubectl get clusterrole secret-access-cr -o yaml
```

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  creationTimestamp: 2018-10-18T19:27:24Z
  name: secret-access-cr
<output_omitted>
```

7. Now we bind the role to the account. Create another YAML file which uses `roleRef`::

```
student@master:~$ vim rolebinding.yaml
```

rolebinding.yaml

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
 name: secret-rb
subjects:
- kind: ServiceAccount
  name: secret-access-sa
roleRef:
 kind: ClusterRole
 name: secret-access-cr
 apiGroup: rbac.authorization.k8s.io
```

8. Create the new `RoleBinding` and verify.

```
student@master:~$ kubectl create -f rolebinding.yaml
```

```
rolebinding.rbac.authorization.k8s.io/secret-rb created
```

```
student@master:~$ kubectl get rolebindings
```

```
NAME        AGE
secret-rb   17s
```

9. View the `secondapp` pod and **grep** for secret settings. Note that it uses the default settings.

```
student@master:~$ kubectl describe pod secondapp |grep -i secret
```

```
   /var/run/secrets/kubernetes.io/serviceaccount from
default-token-c4rdg (ro)
   Type:         Secret (a volume populated by a Secret)
   SecretName:   lfsecret
   Type:         Secret (a volume populated by a Secret)
   SecretName:   default-token-c4rdg
```

10. Edit the second.yaml file and add the use of the serviceAccount.

```
student@master:~$ vim $HOME/app2/second.yaml
```

second.yaml

```yaml
....
  name: secondapp
spec:
  serviceAccountName: secret-access-sa  #<-- Add this line
  securityContext:
    runAsUser: 1000
....
```

11. We will delete the secondapp pod if still running, then create it again. View what the secret is by default.

```
student@master:~$ kubectl delete pod secondapp ; kubectl create -f $HOME/app2/second.yaml
```

```
pod "secondapp" deleted
pod/secondapp created
```

```
student@master:~$ kubectl describe pod secondapp | grep -i secret
```

```
    /var/run/secrets/kubernetes.io/serviceaccount from
 secret-access-sa-token-wd7vm (ro)
  secret-access-sa-token-wd7vm:
    Type:         Secret (a volume populated by a Secret)
    SecretName:   secret-access-sa-token-wd7vm

```

Exercise 10.4: Implement a NetworkPolicy

An early architecture decision with Kubernetes was non-isolation, that all pods were able to connect to all other pods and nodes by design. In more recent releases the use of a NetworkPolicy allows for pod isolation. The policy only has effect when the network plugin, like **Project Calico**, are capable of honoring them. If used with a plugin like **flannel** they will have no effect. The use of matchLabels allows for more granular selection within the namespace which can be selected using a namespaceSelector. Using multiple labels can allow for complex application of rules. More information can be found here: https://kubernetes.io/docs/concepts/services-networking/network-policies

1. Begin by creating a default policy which denies all traffic. Once ingested into the cluster this will affect every pod not selected by another policy, creating a mostly-closed environment. If you want to only deny ingress or egress traffic you can remove the other policyType.

```
student@master:~$ cd $HOME/app2/
```

```
student@master:~/app2$ vim allclosed.yaml
```

allclosed.yaml

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-default
spec:
  podSelector: {}
  policyTypes:
  - Ingress
  - Egress
```

2. Before we can test the new network policy we need to make sure network access works without it applied. Update **secondapp** to include a new container running **nginx**, then test access. Begin by adding two lines for the **nginx** image and name `webserver`, as found below. It takes a bit for the pod to terminate, so we'll delete then edit the file.

```
student@master:~/app2$ kubectl delete pod secondapp
```

```
pod "secondapp" deleted
```

```
student@master:~/app2$ vim second.yaml
```

second.yaml

```yaml
  runAsUser: 1000
containers:
- name: webserver                    #<-- Add this and following line
  image: nginx
- name: busy
  image: busybox
  command:
```

3. Create the new pod. Be aware the pod will move from `ContainerCreating` to Error to `CrashLoopBackOff`, as only one of the containers will start. We will troubleshoot the error in following steps.

```
student@master:~/app2$ kubectl create -f second.yaml
```

```
pod/secondapp created
```

```
student@master:~/app2$ kubectl get pods
```

```
NAME                        READY   STATUS             RESTARTS    AGE
nginx-6b58d9cdfd-9fnl4      1/1     Running            1           2d
Registry-795c6c8b8f-hl5wf   1/1     Running            2           2d
secondapp                   1/2     CrashLoopBackOff   1           13s
<output_omitted>
```

4. Take a closer look at the events leading up to the failure. The images were pulled and the container was started. It was the full execution of the container which failed.

```
student@master:~/app2$ kubectl get event
```

```
<output_omitted>
25s         Normal    Scheduled   Pod    Successfully assigned default/secondapp to master
4s          Normal    Pulling     Pod    pulling image "nginx"
2s          Normal    Pulled      Pod    Successfully pulled image "nginx"
2s          Normal    Created     Pod    Created container
2s          Normal    Started     Pod    Started container
```

```
23s         Normal    Pulling     Pod    pulling image "busybox"
21s         Normal    Pulled      Pod    Successfully pulled image "busybox"
21s         Normal    Created     Pod    Created container
21s         Normal    Started     Pod    Started container
1s          Warning   BackOff     Pod    Back-off restarting failed container
```

5. View the logs of the **webserver** container mentioned in the previous output. Note there are errors about the user directive and not having permission to make directories.

```
student@master:~/app2$ kubectl logs secondapp webserver
```

```
2018/04/13 19:51:13 [warn] 1#1: the "user" directive makes sense
only if the master process runs with super-user privileges,
 ignored in /etc/nginx/nginx.conf:2
nginx: [warn] the "user" directive makes sense only if the master
process runs with super-user privileges,
 ignored in /etc/nginx/nginx.conf:2
2018/04/13 19:51:13 [emerg] 1#1: mkdir() "/var/cache/nginx/client_temp"
failed (13: Permission denied)
nginx: [emerg] mkdir() "/var/cache/nginx/client_temp" failed
(13: Permission denied)
```

6. Delete the pods. Edit the YAML file to comment out the setting of a UID for the entire pod.

```
student@master:~/app2$ kubectl delete -f second.yaml
```

```
pod "secondapp" deleted
```

```
student@master:~/app2$ vim second.yaml
```

second.yaml

```
spec:
  serviceAccountName: secret-access-sa
#  securityContext:                       #<-- Comment this and following line
#    runAsUser: 1000
  containers:
  - name: webserver
```

7. Create the pod again. This time both containers should run. You may have to wait for the previous pod to fully terminate, depending on how fast you type.

```
student@master:~/app2$ kubectl create -f second.yaml
```

```
pod/secondapp created
```

```
student@master:~/app2$ kubectl get pods
```

```
NAME                        READY     STATUS      RESTARTS    AGE
<output_omitted>
secondapp                   2/2       Running     0           5s
```

8. Expose the **webserver** using a NodePort service. Expect an error due to lack of labels.

```
student@master:~/app2$ kubectl expose pod secondapp --type=NodePort --port=80
```

```
error: couldn't retrieve selectors via --selector flag or
introspection: the pod has no labels and cannot be exposed
See 'kubectl expose -h' for help and examples.
```

9. Edit the YAML file to add a label in the metadata, adding the `example: second` label right after the pod name. Note you can delete several resources at once by passing the YAML file to the delete command. Delete and recreate the pod. It may take up to a minute for the pod to shut down.

```
student@master:~/app2$ kubectl delete -f second.yaml
```

```
pod "secondapp" deleted
```

```
student@master:~/app2$ vim second.yaml
```

second.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: secondapp
  labels:                    #<-- This and following line
    example: second
spec:
#  securityContext:
#    runAsUser: 1000
<output_omitted>
```

```
student@master:~/app2$ kubectl create -f second.yaml
```

```
pod/secondapp created
```

```
student@master:~/app2$ kubectl get pods
```

```
NAME                        READY     STATUS      RESTARTS    AGE
<output_omitted>
secondapp                   2/2       Running     0           15s
```

10. This time we will expose a `NodePort` again, and create the service separately, then add a label to illustrate how labels are essential for tying resources together inside of kubernetes.

```
student@master:~/app2$ kubectl create service nodeport secondapp --tcp=80
```

```
service/secondapp created
```

11. Look at the details of the service. Note the selector is set to `app: secondapp`. Also take note of the nodePort, which is 31655 in the example below, yours may be different.

```
student@master:~/app2$ kubectl get svc secondapp -o yaml
```

```
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: "2020-04-16T04:40:07Z"
  labels:
    example: second
  managedFields:

....

spec:
  clusterIP: 10.97.96.75
  externalTrafficPolicy: Cluster
  ports:
  - nodePort: 31665
```

```
    port: 80
    protocol: TCP
    targetPort: 80
  selector:
    example: second
  sessionAffinity: None
  type: NodePort
status:
  loadBalancer: {}
```

12. Test access to the service using **curl** and the `ClusterIP` shown in the previous output. As the label does not match any other resources, the **curl** command should fail. If it hangs **control-c** to exit back to the shell.

```
student@master:~/app2$ curl http://10.97.96.75
```

13. Edit the service. We will change the label to match **secondapp**, and set the `nodePort` to a new port, one that may have been specifically opened by our firewall team, port `32000`.

```
student@master:~/app2$ kubectl edit svc secondapp
```

YAML **secondapp service**

```
 <output_omitted>
  ports:
  - name: "80"
    nodePort: 32000       #<-- Edit this line
    port: 80
    protocol: TCP
    targetPort: 80
  selector:
    example: second       #<-- Edit this line
  sessionAffinity: None
<output_omitted>
```

14. Verify the updated port number is showing properly, and take note of the ClusterIP. The example below shows a ClusterIP of `10.97.96.75` and a port of `32000` as expected.

```
student@master:~/app2$ kubectl get svc
```

```
NAME         TYPE      CLUSTER-IP     EXTERNAL-IP  PORT(S)        AGE
<output_omitted>
secondapp    NodePort  10.97.96.75    <none>       80:32000/TCP   5m
```

15. Test access to the high port. You should get the default nginx welcome page both if you test from the node to the `ClusterIP:<low-port-number>` and from the exterior `hostIP:<high-port-number>`. As the high port is randomly generated make sure it's available. Both of your nodes should be exposing the web server on port 32000. The example shows the use of the **curl** command, you could also use a web browser.

```
student@master:~/app2$ curl http://10.97.96.75
```

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

```
[user@laptop ~]$ curl http://35.184.219.5:32000
```

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

16. Now test egress from a container to the outside world. We'll use the **netcat** command to verify access to a running web server on port 80. First test local access to nginx, then a remote server.

```
student@master:~/app2$ kubectl exec -it -c busy secondapp -- sh
```

On Container

```
/ $ nc -vz 127.0.0.1 80
```

```
127.0.0.1 (127.0.0.1:80) open
```

```
/ $ nc -vz www.linux.com 80
```

```
www.linux.com (151.101.185.5:80) open
```

```
/ $ exit
```

Exercise 10.5: Testing the Policy

1. Now that we have tested both ingress and egress we can implement the network policy.

```
student@master:~/app2$ kubectl create -f $HOME/app2/allclosed.yaml
```

```
networkpolicy.networking.k8s.io/deny-default created
```

2. Use the ingress and egress tests again. Three of the four should eventually timeout. Start by testing from outside the cluster, and interrupt if you get tired of waiting.

```
[user@laptop ~]$ curl http://35.184.219.5:32000
```

```
curl: (7) Failed to connect to 35.184.219.5 port
32000: Connection timed out
```

3. Then test from the host to the container.

```
student@master:~/app2$ curl http://10.97.96.75:80
```

```
curl: (7) Failed to connect to 10.97.96.75 port 80: Connection timed out
```

4. Now test egress. From container to container should work, as the filter is outside of the pod. Then test egress to an external web page. It should eventually timeout.

```
student@master:~/app2$ kubectl exec -it -c busy secondapp -- sh
```

On Container

```
/ $ nc -vz 127.0.0.1 80
```

```
127.0.0.1 (127.0.0.1:80) open
```

```
/ $ nc -vz www.linux.com 80
```

```
nc: bad address 'www.linux.com'
```

```
/ $ exit
```

5. Update the NetworkPolicy and comment out the Egress line. Then replace the policy.

```
student@master:~/app2$ vim $HOME/app2/allclosed.yaml
```

allclosed.yaml

```yaml
....
spec:
  podSelector: {}
  policyTypes:
  - Ingress
#  - Egress            #<-- Comment out this line
```

```
student@master:~/app2$ kubectl replace -f $HOME/app2/allclosed.yaml
```

```
networkpolicy.networking.k8s.io/deny-default replaced
```

6. Test egress access to an outside site. Get the IP address of the **eth0** inside the container while logged in. The IP is `192.168.55.91` in the example below, yours may be different.

```
student@master:~/app2$ kubectl exec -it -c busy secondapp -- sh
```

On Container

```
/ $ nc -vz www.linux.com 80
```

```
www.linux.com (151.101.185.5:80) open
```

```
/ $ ip a
```

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: tunl0@NONE: <NOARP> mtu 1480 qdisc noop qlen 1000
    link/ipip 0.0.0.0 brd 0.0.0.0
4: eth0@if59: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
    link/ether 1e:c8:7d:6a:96:c3 brd ff:ff:ff:ff:ff:ff
    inet 192.168.55.91/32 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::1cc8:7dff:fe6a:96c3/64 scope link
       valid_lft forever preferred_lft forever
```

```
/ $ exit
```

7. Now add a selector to allow ingress to only the nginx container. Use the IP from the **eth0** range.

```
student@master:~/app2$ vim $HOME/app2/allclosed.yaml
```

allclosed.yaml

```
<output_omitted>
 policyTypes:
  - Ingress
  ingress:                          #<-- Add this and following three lines
  - from:
    - ipBlock:
        cidr: 192.168.0.0/16
# - Egress
```

8. Recreate the policy, and verify its configuration.

```
student@master:~/app2$ kubectl replace -f $HOME/app2/allclosed.yaml
```

```
networkpolicy.networking.k8s.io/deny-default replaced
```

```
student@master:~/app2$ kubectl get networkpolicy
```

```
NAME           POD-SELECTOR     AGE
deny-default   <none>           3m2s
```

```
student@master:~/app2$ kubectl get networkpolicy -o yaml
```

```
apiVersion: v1
items:
- apiVersion: networking.k8s.io/v1
  kind: NetworkPolicy
  metadata:
<output_omitted>
```

9. Test access to the container both using **curl** as well as **ping**, the IP address to use was found from the IP inside the container. You may need to install **iputils-ping** or other software to use **ping**.

```
student@master:~/app2$ curl http://192.168.55.91
```

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

```
student@master:~/app2$ ping -c5 192.168.55.91
```

```
PING 192.168.55.91 (192.168.55.91) 56(84) bytes of data.
64 bytes from 192.168.55.91: icmp_seq=1 ttl=63 time=1.11 ms
64 bytes from 192.168.55.91: icmp_seq=2 ttl=63 time=0.352 ms
64 bytes from 192.168.55.91: icmp_seq=3 ttl=63 time=0.350 ms
64 bytes from 192.168.55.91: icmp_seq=4 ttl=63 time=0.359 ms
64 bytes from 192.168.55.91: icmp_seq=5 ttl=63 time=0.295 ms

--- 192.168.55.91 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4054ms
rtt min/avg/max/mdev = 0.295/0.495/1.119/0.312 ms
```

10. Update the policy to only allow ingress for TCP traffic on port 80, then test with **curl**, which should work. The ports entry should line up with the from entry a few lines above.

```
student@master:~/app2$ vim $HOME/app2/allclosed.yaml
```

allclosed.yaml

```
<output_omitted>
  - Ingress
  ingress:
  - from:
    - ipBlock:
        cidr: 192.168.0.0/16
    ports:                        #<-- Add this and two following lines
    - port: 80
      protocol: TCP
#   - Egress
```

```
student@master:~/app2$ kubectl replace -f $HOME/app2/allclosed.yaml
```

```
networkpolicy.networking.k8s.io/deny-default replaced
```

```
student@master:~/app2$ curl http://192.168.55.91
```

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

11. All five pings should fail, with zero received.

```
student@master:~/app2$ ping -c5 192.168.55.91
```

```
PING 192.168.55.91 (192.168.55.91) 56(84) bytes of data.

--- 192.168.55.91 ping statistics ---
5 packets transmitted, 0 received, 100% packet loss, time 4098ms
```

12. You may want to remove the `default-deny` policy, in case you want to get to your registry or other pods.

```
student@master:~/app2$ kubectl delete networkpolicies deny-default
```

```
networkpolicy.networking.k8s.io "deny-default" deleted
```

Exercise 10.6: Domain Review

Very Important

The source pages and content in this review could change at any time. **IT IS YOUR RESPONSIBILITY TO CHECK THE CURRENT INFORMATION.**

Revisit the CKAD domain list on Curriculum Overview and locate some of the topics we have covered in this chapter.

• Understand SecurityContexts
• Understand ServiceAccounts
• Demonstrate basic understanding of NetworkPolicies

Figure 10.2: **Domain**

Focus on ensuing you have all the necessary files and processes understood first. Repeat the review until you are sure you have bookmarks of necessary YAML samples and can complete each step quickly, and ensure each object is running properly.

1. Create a new deployment which uses the `nginx` image.
2. Create a new `LoadBalancer` service to expose the newly created deployment. Test that it works.
3. Create a new NetworkPolicy called netblock which blocks all traffic to pods in this deployment only. Test that all traffic is blocked to deployment.
4. Update the `netblock` policy to allow traffic to the pod on port 80 only. Test that you can access the default nginx web page.
5. Find and use the `security-review1.yaml` file to create a pod.

```
student@master:~$ kubectl create -f security-review1.yaml
```

6. View the status of the pod.
7. Use the following commands to figure out why the pod has issues.

```
student@master:~$ kubectl get pod securityreview
```

```
student@master:~$ kubectl describe pod securityreview
```

```
student@master:~$ kubectl logs securityreview
```

8. After finding the errors, log into the container and find the proper id of the nginx user.
9. Edit the yaml and re-create the pod such that the pod runs without error.
10. Create a new `serviceAccount` called securityaccount.
11. Create a ClusterRole named secrole which only allows create, delete, and list of pods in all apiGroups.
12. Bind the clusterRole to the serviceAccount.
13. 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.
14. Remove any resources you have added during this review

Chapter 11

Exposing Applications

In this chapter, we are going to talk about exposing our applications using services. We'll start by talking about a ClusterIP service. This allows us to expose our application inside of the cluster only. A NodePort will create a ClusterIP and then also, get a high number port from the node. This allows us then to expose our applications internally on a low number port, and externally, but only on a high number port. The use of a LoadBalancer first creates a NodePort on our behalf. The difference is it also sends an asynchronous request to a cloud provider to make use of a LoadBalancer. If we are not using a cloud provider, or if a load balancer is not available, it continues to run as a NodePort. The fourth service that we are going to discuss is called ExternalName. This is useful for handling requests to an outside DNS resource, instead of one inside of the cluster. This is helpful if you were in the process of moving your application from outside of the cluster inside. We'll also discuss the configuration of an Ingress Controller. The two reasons that you might want to use an Ingress Controller is, for example, if you have thousands of services, having them independent can become difficult to manage, and wasteful of resources. You can then consolidate it to a single Ingress Controller. Or multiple Ingress Controllers if you want more flexibility. The second reason is an Ingress Controller allows you to expose a low numbered port to your application. Without that, you could get there, but you'd have to use iptables, which could become difficult and not terribly flexible to manage.

11.1 Service Types

- Services can be of the following 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. The 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. As **CoreDNS** has become more stable this service is not used as much.
- The `kubectl proxy` command creates a local service to access a ClusterIP. This can be useful for troubleshooting or development work.

11.2 Services Diagram

- The **kube-proxy** running on cluster nodes watches the API server service resources. It presents a type of virtual IP address for services other than `ExternalName`. The mode for this process has changed over versions of Kubernetes.
- In v1.0, services ran in `userspace` mode as TCP/UDP over IP or Layer 4. In the v1.1 release, the `iptables` proxy was added and became the default mode starting with v1.2.

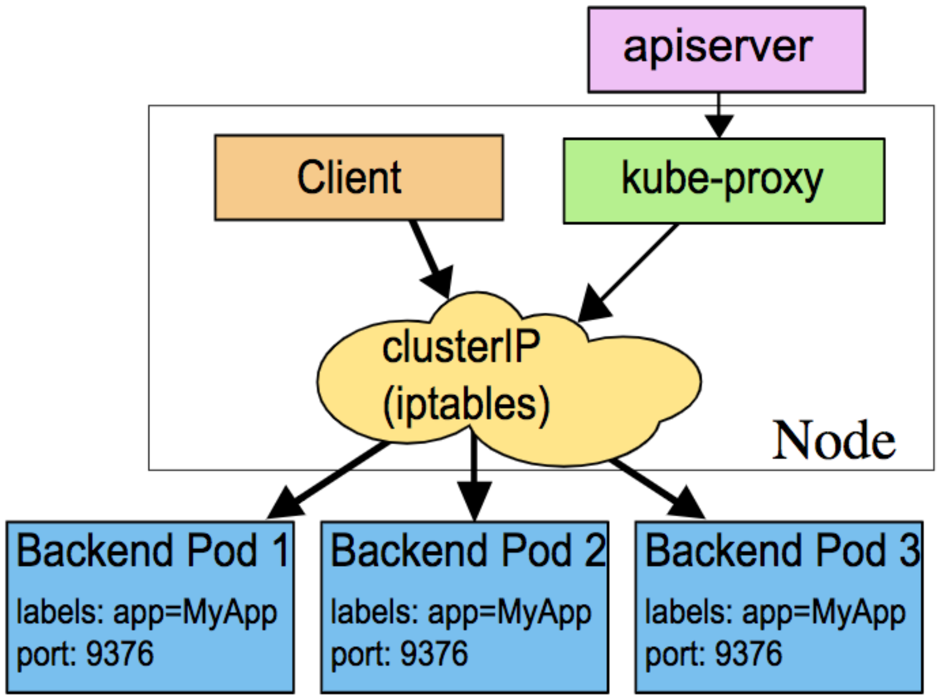


Figure 11.1: **Traffic from Cluster IP to Pod**

- In the `iptables` proxy mode, **kube-proxy** continues to monitor the API server for changes in Service and Endpoint objects, and updates rules for each object when created or removed. One limitation to the new mode is an inability to connect to a Pod should the original request fail, so it uses a `Readiness Probe` to ensure all containers are functional prior to connection. This mode allows for up to approximately 5000 nodes. Assuming multiple Services and Pods per node, this leads to a bottleneck in the kernel.
- Another mode introduced in v1.9 and GA in v1.11 is `ipvs`. This mode works in the kernel space for greater speed, and allows for a configurable load-balancing algorithm, such as round-robin, shortest expected delay, least connection and several others. This can be helpful for large clusters, much past the previous 5000 node limitation. This mode assumes IPVS kernel modules are installed and running prior to **kube-proxy**. Clusters built with **kubeadm** do not enable `ipvs` by default, but this can be passed during cluster initialization.
- The **kube-proxy** mode is configured via a flag sent during initialization, such as `mode=iptables` and could also be `IPVS` or `userspace`.

11.3 Service Update Pattern

- Labels are used to determine which Pods should receive traffic from a service. `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 replication controller 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.

11.4 Accessing an Application with a Service

- The basic step to access a new service is to use **kubectl**:

```
$ kubectl expose deployment/nginx --port=80 --type=NodePort
$ kubectl get svc
```

```
NAME         CLUSTER-IP    EXTERNAL-IP   PORT(S)        AGE
kubernetes   10.0.0.1      <none>        443/TCP        18h
nginx        10.0.0.112    <nodes>       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 `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 backend 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.

11.5 Service without a Selector

- Typically, a service creates a new endpoint for connectivity. Should you want to create the service, but later add the endpoint, such as connecting to a remote database, you can use a service without selectors. This can also be used to direct the service to another service, in a different namespace or cluster.

11.6 ClusterIP

- For inter-cluster communication, frontends talking to back-ends can use `ClusterIPs`. These addresses and endpoints only work within the cluster.
- `ClusterIP` is the default type of service created.

```
spec: clusterIP: 10.108.95.67 ports: - name: "443" port:
443 protocol: TCP targetPort: 443
```

11.7 NodePort

- `NodePort` is a simple connection from a high-port routed to a `ClusterIP` using `iptables`, or `ipvs` in newer versions. The creation of a `NodePort` generates a `ClusterIP` by default. Traffic is routed from the `NodePort` to the `ClusterIP`. Only high ports can be used, as declared in the source code. The `NodePort` is accessible via calls to `<NodeIP>:<NodePort>`.

```
spec:
  clusterIP: 10.97.191.46
  externalTrafficPolicy: Cluster
  ports:
  - nodePort: 31070
    port: 80
    protocol: TCP
    targetPort: 800a0
  selector:
    io.kompose.service: nginx
  sessionAffinity: None
  type: NodePort
```

11.8 LoadBalancer

- Creating a `LoadBalancer` service generates a `NodePort`, which then creates a `ClusterIP`. It also sends an asynchronous call to an external load balancer, typically supplied by a cloud provider. The `External-IP` value will remain in a `<Pending>` state until the load balancer returns. Should it not return, the `NodePort` created acts as it would otherwise.

```
Type: LoadBalancer
loadBalancerIP: 12.45.105.12
clusterIP: 10.5.31.33
ports:
- protocol: TCP
  Port: 80
```

- The routing of traffic to a particular backend pod depends on the cloud provider in use.

11.9 ExternalName

- The use of an `ExternalName` service, which is a special type of service without selectors, is to point to an external DNS server. Use of the service returns a `CNAME` record. Working with the `ExternalName` service is handy when using a resource external to the cluster, perhaps prior to full integration.

```
spec:
  Type: ExternalName
  externalName: ext.db.example.com
```

- DNS services are now handled by the `kube-dns` service which uses the **CoreDNS** image on kubeadm created clusters. The **CoreDNS** service has many features in addition to being a DNS server, and is extensible using plugins. More can be found here: https://coredns.io/manual/toc/

11.10 Ingress Resource

- An ingress resource is an API object containing a list of rules matched against all incoming requests. Only HTTP rules are currently supported. In order for the controller to direct traffic to the backend, the HTTP request must match both the host and the path declared in the ingress.

11.11 Ingress Controller

- Handling a few services can be easily done. However, managing thousands or tens of thousands of services can create inefficiencies. The use of an **Ingress Controller** manages ingress rules to route traffic to existing services. Ingress can be used for fan out to services, name-based hosting, TLS, or load balancing. Another feature is the ability to expose low-numbered ports. `Services` have been hard-coded not to expose ports lower than 1024.

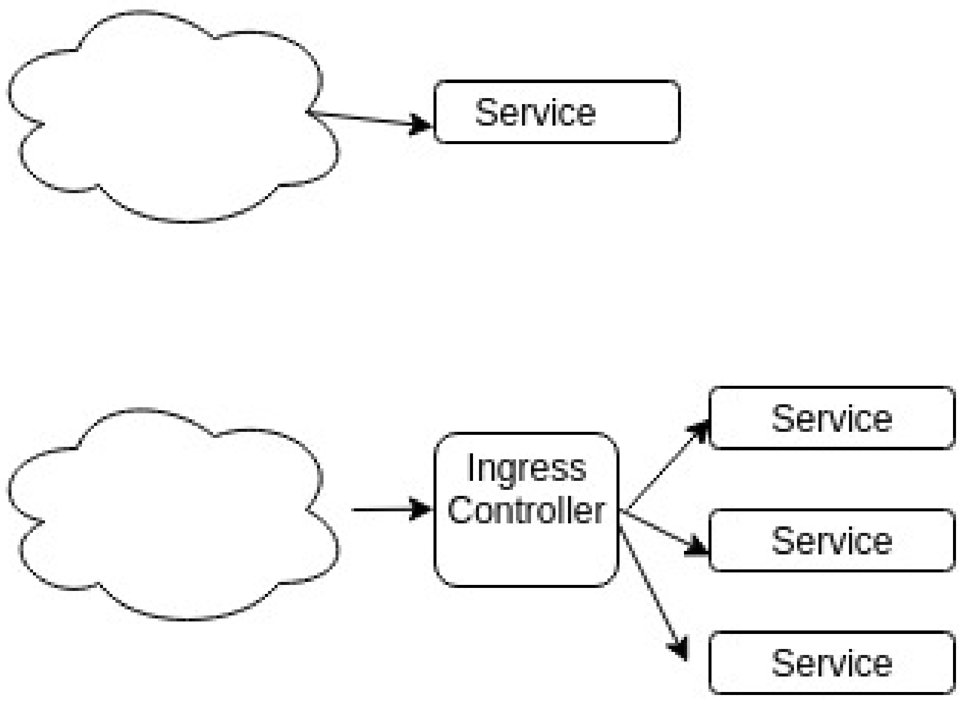


Figure 11.2: **Ingress Controller**

- There are a few Ingress Controllers with **nginx** and **GCE** that are "officially supported" by the community. **Traefik** (pronounced "traffic") https://traefik.io/ and **HAProxy** https://haproxy.org are in common use, as well. More controllers are planned, as is support for more HTTPS/TLS modes, combining L4 and L7 ingress and requesting name or IP via `claims`.

11.12 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 handing 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 unmonitized. Often used as a data plane under other tools of a service mesh. https://www.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. https://istio.io
  - **linkerd** - Another `service mesh` purpose built to be easy to deploy, fast, and ultralight. https://linkerd.io/

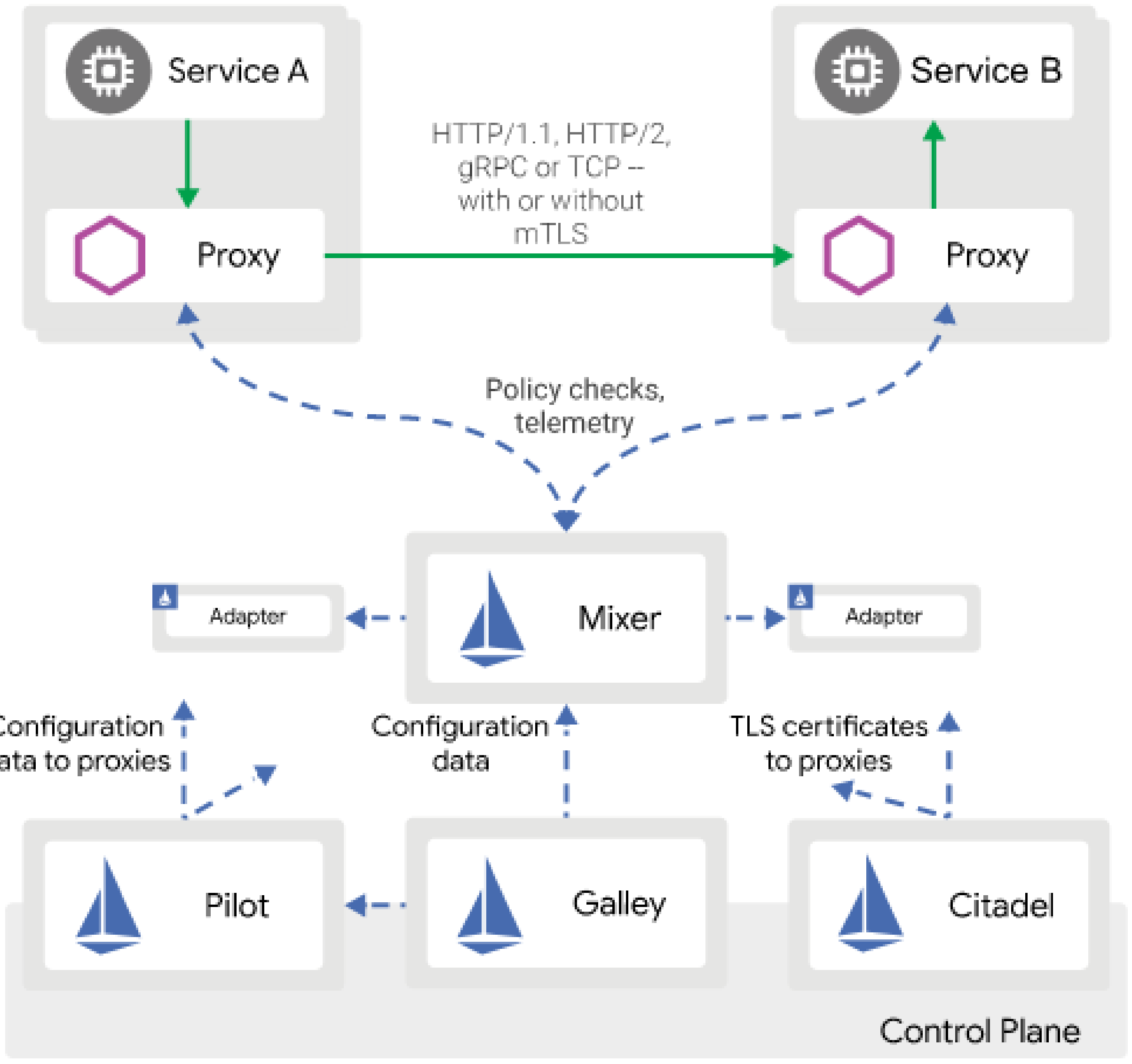


Figure 11.3: **Istio Service Mesh**

[1]

11.13 Labs

Exercise 11.1: Exposing Applications: Expose a Service

Overview

In this lab we will explore various ways to expose an application to other pods and outside the cluster. We will add to the NodePort used in previous labs other service options.

1. We will begin by using the default service type ClusterIP. This is a cluster internal IP, only reachable from within the cluster. Begin by viewing the existing services.

[1]Image downloaded from https://istio.io/docs/concepts/what-is-istio/ 23-sep-2019

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

```
NAME          TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)        AGE
kubernetes    ClusterIP   10.96.0.1        <none>        443/TCP        8d
nginx         ClusterIP   10.108.95.67     <none>        443/TCP        8d
registry      ClusterIP   10.105.119.236   <none>        5000/TCP       8d
secondapp     NodePort    10.111.26.8      <none>        80:32000/TCP   7h
```

2. Delete the existing service for secondapp.

```
student@master:~$ cd ~/app2
```

```
student@master:~/app2$ kubectl delete svc secondapp
```

```
service "secondapp" deleted
```

3. Create a YAML file for a replacement service, which would be persistent. Use the label to select the secondapp. Expose the same port and protocol of the previous service.

```
student@master:~/app2$ vim service.yaml
```

service.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: secondapp
  labels:
    run: my-nginx
spec:
  ports:
  - port: 80
    protocol: TCP
  type: NodePort
  selector:
    example: second
```

4. Create the service, find the new IP and port. Note there is no high number port as this is internal access only.

```
student@master:~/app2$ kubectl create -f service.yaml
```

```
service/secondapp created
```

```
student@master:~/app2$ kubectl get svc
```

```
NAME          TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)      AGE
kubernetes    ClusterIP   10.96.0.1        <none>        443/TCP      8d
nginx         ClusterIP   10.108.95.67     <none>        443/TCP      8d
registry      ClusterIP   10.105.119.236   <none>        5000/TCP     8d
secondapp     NodePort    10.98.148.52     <none>        80:32212/TCP 14s
```

5. Test access. You should see the default welcome page again.

```
student@master:~/app2$ curl http://10.98.148.52
```

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

6. To expose a port to outside the cluster we will create a NodePort. We had done this in a previous step from the command line. When we create a NodePort it will create a new ClusterIP automatically. Edit the YAML file again. Add type: NodePort. Also add the high-port to match an open port in the firewall as mentioned in the previous chapter. You'll have to delete and re-create as the existing IP is immutable, but not able to be reused. The NodePort will try to create a new ClusterIP instead.

```
student@master:~/app2$ vim service.yaml
```

service.yaml

```
....
    protocol: TCP
    nodePort: 32000          #<-- Add this and following line
  type: NodePort
  selector:
    example: second
```

```
student@master:~/app2$ kubectl delete svc secondapp ; kubectl create  -f service.yaml
```

```
service "secondapp" deleted
service/secondapp created
```

7. Find the new ClusterIP and ports for the service.

```
student@master:~/app2$ kubectl get svc
```

```
NAME         TYPE         CLUSTER-IP     EXTERNAL-IP   PORT(S)        AGE
kubernetes   ClusterIP    10.96.0.1      <none>        443/TCP        8d
nginx        ClusterIP    10.108.95.67   <none>        443/TCP        8d
registry     ClusterIP    10.105.119.236 <none>        5000/TCP       8d
secondapp    NodePort     10.109.134.221 <none>        80:32000/TCP   4s
```

8. Test the low port number using the new ClusterIP for the secondapp service.

```
student@master:~/app2$ curl 10.109.134.221
```

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

9. Test access from an external node to the host IP and the high container port. Your IP and port will be different. It should work, even with the network policy in place, as the traffic is arriving via a 192.168.0.0 port.

```
user@laptop:~/Desktop$ curl http://35.184.219.5:32000
```

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

10. The use of a LoadBalancer makes an asynchronous request to an external provider for a load balancer if one is available. It then creates a NodePort and waits for a response including the external IP. The local NodePort will work even before the load balancer replies. Edit the YAML file and change the type to be LoadBalancer.

```
student@master:~/app2$ vim service.yaml
```

service.yaml

```yaml
....
  - port: 80
    protocol: TCP
    nodePort: 32000
  type: LoadBalancer    #<-- Edit this line
  selector:
    example: second
```

```
student@master:~/app2$ kubectl delete svc secondapp ; kubectl create -f service.yaml
```

```
service "secondapp" deleted
service/secondapp created
```

11. As mentioned the cloud provider is not configured to provide a load balancer; the External-IP will remain in pending state. Some issues have been found using this with VirtualBox.

```
student@master:~/app2$ kubectl get svc
```

```
NAME          TYPE           CLUSTER-IP       EXTERNAL-IP    PORT(S)         AGE
kubernetes    ClusterIP      10.96.0.1        <none>         443/TCP         8d
nginx         ClusterIP      10.108.95.67     <none>         443/TCP         8d
registry      ClusterIP      10.105.119.236   <none>         5000/TCP        8d
secondapp     LoadBalancer   10.109.26.21     <pending>      80:32000/TCP    4s
```

12. Test again local and from a remote node. The IP addresses and ports will be different on your node.

```
serewic@laptop:~/Desktop$ curl http://35.184.219.5:32000
```

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

13. You can also use DNS names provided by **CoreDNS** which dynamically are added when the service is created. Start by logging into the `busy` container of secondapp.

```
student@master:~/app2$ kubectl exec -it secondapp -c busy -- sh
```

On Container

(a) Use the **nslookup** command to find the `secondapp` service. Then find the `registry` service we configured to provide container images.

```
/ $ nslookup secondapp
```

```
Server:         10.96.0.10
Address:        10.96.0.10:53

Name:       secondapp.default.svc.cluster.local
Address: 10.96.214.133

*** Can't find secondapp.svc.cluster.local: No answer
*** Can't find secondapp.cluster.local: No answer
*** Can't find secondapp.c.endless-station-188822.internal: No answer
<output_omitted>
```

```
/ $ nslookup registry
```

```
Server:         10.96.0.10
Address:        10.96.0.10:53

Name:       registry.default.svc.cluster.local
Address: 10.110.95.21
<output_omitted>
```

(b) Lookup the FQDN associated with the DNS server IP displayed by the commands. Your IP may be different.

```
/ $ nslookup 10.96.0.10
```

```
Server:         10.96.0.10
Address:        10.96.0.10:53

10.0.96.10.in-addr.arpa        name = kube-dns.kube-system.svc.cluster.local
```

(c) Attempt to resolve the service name, which should not bring back any records. Then try with the FQDN. Read through the errors. You'll note that only the default namespaces is checked. You may have to check the FQDN a few times as it doesn't always reply with an answer.

```
/ $ nslookup kube-dns
```

```
Server:         10.96.0.10
Address:        10.96.0.10:53

** server can't find kube-dns.default.svc.cluster.local: NXDOMAIN

*** Can't find kube-dns.svc.cluster.local: No answer
*** Can't find kube-dns.cluster.local: No answer
*** Can't find kube-dns.c.endless-station-188822.internal: No answer
```

```
/ $ nslookup kube-dns.kube-system.svc.cluster.local
```

```
Server:         10.96.0.10
Address:        10.96.0.10:53

Name:       kube-dns.kube-system.svc.cluster.local
Address: 10.96.0.10

*** Can't find kube-dns.kube-system.svc.cluster.local: No answer
```

(d) Exit out of the container

```
/ $ exit
```

14. Create a new namespace named `multitenant` and a new deployment named `mainapp`. Expose the deployment port 80 using the name `shopping`

```
student@master:~/app2$ kubectl create ns multitenant
student@master:~/app2$ kubectl -n multitenant create deployment mainapp --image=nginx
student@master:~/app2$ kubectl -n multitenant expose deployment mainapp --name=shopping \
                          --type=NodePort --port=80
```

15. Log back into the secondapp busy container and test access to mainapp.

```
student@master:~/app2$ kubectl  exec -it secondapp -c busy -- sh
```

On Container

(a) Use **nslookup** to determine the address of the new service. Start with using just the service name. Then add the service name and the namespaces. As we have not configured coreDNS at all it may not know about all the possible domains to serve.

```
/ $ nslookup shopping
```

```
Server:         10.96.0.10
Address:        10.96.0.10:53

** server can't find shopping.default.svc.cluster.local: NXDOMAIN

*** Can't find shopping.svc.cluster.local: No answer
<output_omitted>
```

```
/ $ nslookup shopping.multitenant
```

```
Server:         10.96.0.10
Address:        10.96.0.10:53

** server can't find shopping.multitenant: NXDOMAIN

*** Can't find shopping.multitenant: No answer
```

```
/ $ nslookup shopping.multitenant.svc.cluster.local
```

```
Server:         10.96.0.10
Address:        10.96.0.10:53

*** Can't find shopping.multitenant.svc.cluster.local: No answer

*** Can't find shopping.multitenant.svc.cluster.local: No answer
```

(b) Now try to use the service name and then the name with namespace, to see if it works. The DNS using the namespace should work, even if you don't have access to the default page. RBAC could be used to grant access.

```
/ $wget shopping
```

```
wget: bad address 'shopping'
```

```
/ $wget shopping.multitenant
```

```
Connecting to shopping.multitenant (10.101.4.142:80)
wget: can't open 'index.html': Permission denied
```

Exercise 11.2: Ingress Controller

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 officially supported by Kubernetes.io, the Traefik Ingress Controller is easier to install. At the moment.

1. As we have RBAC configured we need to make sure the controller will run and be able to work with all necessary ports, endpoints and resources. Create a YAML file to declare a clusterrole and a clusterrolebinding

```
student@master:~/app2$ vim ingress.rbac.yaml
```

ingress.rbac.yaml

```yaml
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: traefik-ingress-controller
rules:
  - apiGroups:
      - ""
    resources:
      - services
      - endpoints
      - secrets
    verbs:
      - get
      - list
      - watch
  - apiGroups:
      - extensions
    resources:
      - ingresses
    verbs:
      - get
      - list
      - watch
---
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: traefik-ingress-controller
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: traefik-ingress-controller
subjects:
- kind: ServiceAccount
  name: traefik-ingress-controller
  namespace: kube-system
```

2. Create the new role and binding.

```
student@master:~/app2$ kubectl create -f ingress.rbac.yaml
```

```
clusterrole.rbac.authorization.k8s.io/traefik-ingress-controller created
clusterrolebinding.rbac.authorization.k8s.io/traefik-ingress-controller created
```

3. Create the Traefik controller. The source web page changes on a regular basis. You can find a recent release by going here https://github.com/containous/traefik/releases, The recent 2.X release has many changes and some "undocumented features" being worked on. Find a copy of the file in the course tarball using the **find** command.

```
student@master:~/app2$ find $HOME -name traefik-ds.yaml
```

4. The output below represents the changes in a **diff** output, from a downloaded version to the edited file in the tarball. You do not have to do this step, it is so you understand the kind of changes one may need for the dashboard to work. One line was added, six lines removed. Also with version 2.0 the dashboard does not appear to work, so we are declaring the use of version `1.7.13`.

```
student@master:~/app2$ diff download.yaml traefik-ds.yaml
```

traefik-ds.yaml

```
23a24             ##  Add the following line 24
>       hostNetwork: true
34,39d34            ##  Remove these lines around line 34
<         securityContext:
<           capabilities:
<             drop:
<             - ALL
<             add:
<             - NET_BIND_SERVICE
```

The included file looks like this:

traefik-ds.rule.yaml

```
....
      terminationGracePeriodSeconds: 60
      hostNetwork: True
      containers:
      - image: traefik
        name: traefik-ingress-lb
        ports:
        - name: http
          containerPort: 80
          hostPort: 80
        - name: admin
          containerPort: 8080
          hostPort: 8080
        args:
        - --api
....
```

5. Create the objects using the edited file.

```
student@master:~/app2$ kubectl apply -f traefik-ds.yaml
```

```
serviceaccount/traefik-ingress-controller created
daemonset.apps/traefik-ingress-controller created
service/traefik-ingress-service created
```

6. Now that there is a new controller we need to pass some rules, so it knows how to handle requests. Note that the host mentioned is www.example.com, which is probably not your node name. We will pass a false header when testing. Also the service name needs to match the secondapp we've been working with.

```
student@master:~/app2$ vim ingress.rule.yaml
```

ingress.rule.yaml

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: ingress-test
  annotations:
```

```yaml
    kubernetes.io/ingress.class: traefik
spec:
  rules:
  - host: www.example.com
    http:
      paths:
      - backend:
          serviceName: secondapp
          servicePort: 80
        path: /
```

7. Now ingest the rule into the cluster.

```
student@master:~/app2$ kubectl create -f ingress.rule.yaml
```

```
ingress.networking.k8s.io/ingress-test2 created
```

8. We should be able to test the internal and external IP addresses, and see the nginx welcome page. The loadbalancer would present the traffic, a curl request in this case, to the externally facing interface. Use **ip** a to find the IP address of the interface which would face the load balancer. In this example the interface would be `ens4`, and the IP would be `10.128.0.7`.

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

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: ens4: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1460 qdisc mq state UP group default qlen 1000
    link/ether 42:01:0a:80:00:03 brd ff:ff:ff:ff:ff:ff
    inet 10.128.0.7/32 brd 10.128.0.3 scope global ens4
       valid_lft forever preferred_lft forever
<output_omitted>
```

```
student@master:~/app2$ curl -H "Host: www.example.com" http://10.128.0.7/
```

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

```
user@laptop:~$ curl -H "Host: www.example.com" http://35.193.3.179
```

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

9. At this point we would keep adding more and more web servers. We'll configure one more, which would then be a process continued as many times as desired. Begin by deploying another **nginx** server. Give it a label and expose port 80.

```
student@master:~/app2$ kubectl create deployment thirdpage --image=nginx
```

```
deployment.apps "thirdpage" created
```

10. Assign a label for the ingress controller to match against. Your pod name is unique, you can use the **Tab** key to complete the name.

```
student@master:~/app2$ kubectl label pod thirdpage-<tab> example=third
```

11. Expose the new server as a NodePort.

```
student@master:~/app2$ kubectl expose deployment thirdpage --port=80 --type=NodePort
```

```
service/thirdpage exposed
```

12. Now we will customize the installation. Run a bash shell inside the new 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 `Third Page`. Much of the command output is not shown below.

```
student@master:~/app2$ kubectl exec -it thirdpage-<Tab> -- /bin/bash
```

On Container

```
root@thirdpage-:/# apt-get update

root@thirdpage-:/# apt-get install vim -y

root@thirdpage-:/# vim /usr/share/nginx/html/index.html
```

```
<!DOCTYPE html>
<html>
<head>
<title>Third Page</title>     #<-- Edit this line
<style>
<output_omitted>
```

```
root@thirdpage-:/$ exit
```

Edit the ingress rules to point the thirdpage service.

13. `student@master:~/app2$ kubectl edit ingress ingress-test`

ingress test

```yaml
....
spec:
  rules:
  - host: www.example.com
    http:
      paths:
      - backend:
          serviceName: secondapp
          servicePort: 80
        path: /
        pathType: ImplementationSpecific
  - host: thirdpage.org                   #<<-- Add this and the following 7 lines
    http:
      paths:
      - backend:
          serviceName: thirdpage
          servicePort: 80
        path: /
        pathType: ImplementationSpecific
status:
....
```

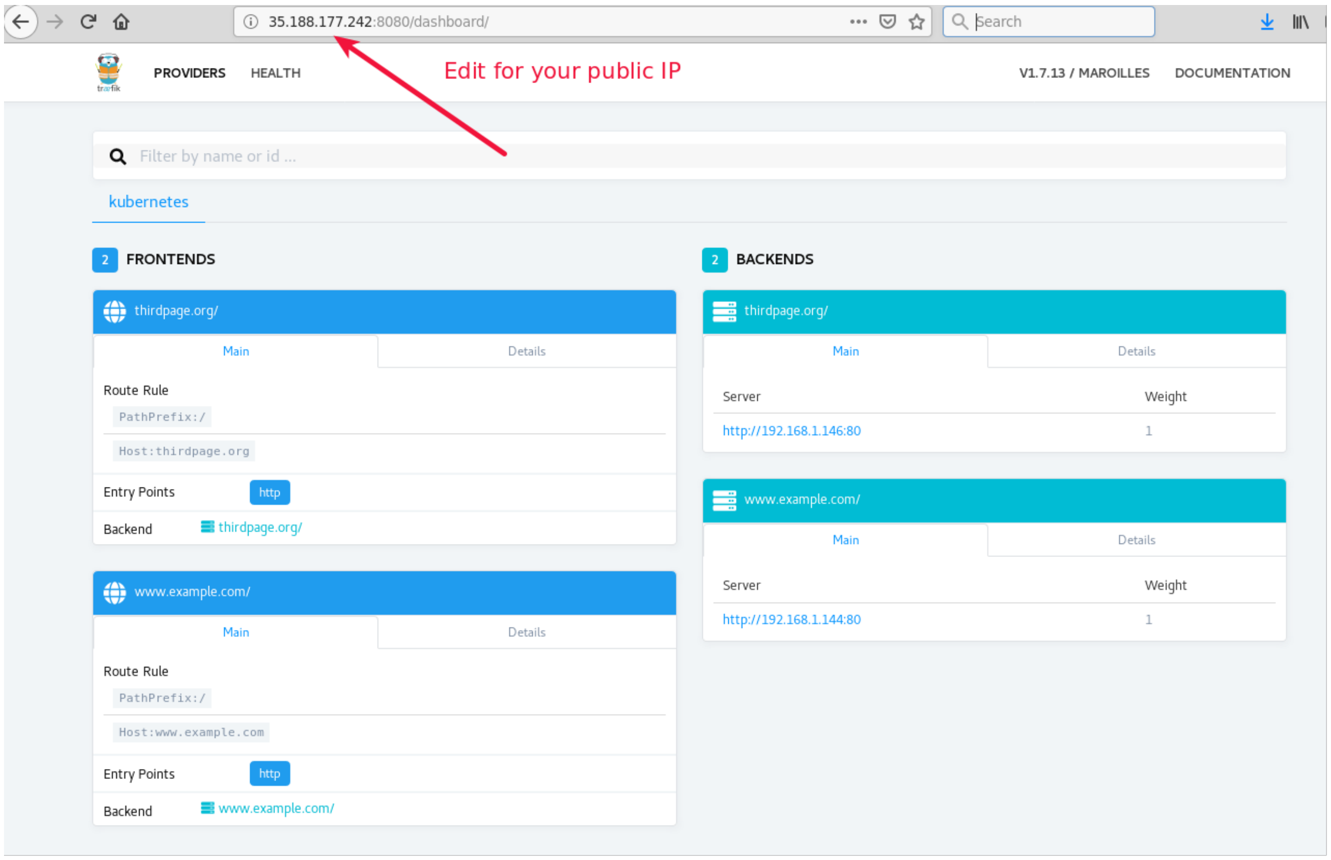
14. Test the second `Host:` setting using **curl** locally as well as from a remote system, be sure the `<title>` shows the non-default page. Use the main IP of either node.

```
student@master:~/app2$ curl -H "Host: thirdpage.org" http://10.128.0.7/
```

```
<!DOCTYPE html>
<html>
<head>
<title>Third Page</title>
<style>
<output_omitted>
```

15. The **Traefik.io** ingress controller also presents a dashboard which allows you to monitor basic traffic. From your local system open a browser and navigate to the public IP of your master node with a like this ⟨YOURPUBLICIP⟩:8080/dashboard/. The trailing slash makes a difference.

    Follow the `HEALTH` and `PROVIDERS` links at the top, as well as the the node IP links and you can view traffic when you reference the pages, from inside or outside the node. Typo the domain names inside the **curl** command and you can also see `404 error traffic`. Explore as time permits.



2 FRONTENDS

thirdpage.org/ — Main | Details

Route Rule: PathPrefix:/ Host:thirdpage.org
Entry Points: http
Backend: thirdpage.org/

www.example.com/ — Main | Details

Route Rule: PathPrefix:/ Host:www.example.com
Entry Points: http
Backend: www.example.com/

2 BACKENDS

thirdpage.org/ — Main | Details

Server	Weight
http://192.168.1.146:80	1

www.example.com/ — Main | Details

Server	Weight
http://192.168.1.144:80	1

Figure 11.4: **Accessing the API**

```
student@master:˜/app2$ curl -H "Host: thirdpage.org" http://10.128.0.7/

student@master:˜/app2$ curl -H "Host: nopage.net" http://10.128.0.7/
```

Exercise 11.3: Domain Review

Very Important

The source pages and content in this review could change at any time. **IT IS YOUR RESPONSIBILITY TO CHECK THE CURRENT INFORMATION**.

Revisit the CKAD domain list on Curriculum Overview and locate some of the topics we have covered in this chapter.

- Understand Services

Figure 11.5: **Service Domain Topic**

Focus on ensuing you have all the necessary files and processes understood first. Repeat the review until you are sure you have bookmarks of necessary YAML samples and can complete each step quickly, and ensure each object is running properly.

Using the browser and the three URL locations allowed by the exam, find and bookmark working YAML examples to do the following:

1. Create a new pod called `webone`, running the `nginx` service. Expose port 80.
2. Create a new service named `webone-svc`. The service should be accessible from outside the cluster.
3. Update both the pod and the service with selectors so that traffic for to the service IP shows the web server content.
4. 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.
5. 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.
6. 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.
7. Remove any resources created in this review.

Chapter 12

Troubleshooting

Troubleshooting, which we'll cover in this chapter, can be difficult in a distributed and decoupled environment like Kubernetes. We will cover the basic flow that can be used to diagnose and identify issues you may have. We do not have cluster-wide logging built-in into Kubernetes. So, we will talk about other projects you may want to implement, like Prometheus and Fluentd, that offer monitoring and logging in a cluster-wide sense. We will talk about agent logs locally, and node logs locally, as well. With conformance testing, we can ensure that the deployment of Kubernetes that we have chosen conforms to standards used in the community. We'll also talk about cluster certification, which is a way of knowing that the Kubernetes distribution you are using is up to the standards of the community, in general.

12.1 Troubleshooting Overview

- 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, disk usage, and network bandwidth on your nodes. It can also be important to monitor well as key metrics in your applications. These features have not been ingested into Kubernetes, so exterior tools will be necessary. Exterior tools such as **Prometheus**. It combines logging, monitoring, and alerting. You can learn more from the Prometheus website, https://www.prometheus.io/. It provides a time-series database, as well as integration with **Grafana** for visualization and dashboards.

- 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 issues, and provides the ability to search all logs. It is a good place to start when local network troubleshooting does not expose the root cause. It can be downloaded from the Fluentd website, https://www.fluentd.org/.
- 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.

12.2 Basic Troubleshooting Steps

- The troubleshooting flow 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 deployment troubleshoot --image=nginx
$ kubectl exec -ti troubleshoot-<tab> -- /bin/sh
```

- If the Pod is running, use `kubectl logs pod-name` to view the 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.
- The issues found with a decoupled system like Kubernetes are similar to those of a traditional datacenter, plus the added layers of Kubernetes controllers:
  - Errors from the command line
  - Pod logs and state of Pods
  - Use shell to troubleshoot Pod DNS and network
  - Check node logs for errors, make sure there are enough resources allocated
  - RBAC, SELinux or AppArmor for security settings
  - API calls to and from controllers to kube-apiserver
  - Inter-node network issues, DNS and firewall
  - Master server controllers (control Pods in pending or error state, errors in log files, sufficient resources, etc).

12.3 Ongoing (Constant) Change

- Unlike typical single vendor programs, there is no one organization responsible for updates to Kubernetes. As a high-velocity open source software (OSS) project, there are thousands of people working all the time. As each one of them works to solve problems or add features, it becomes possible for new problems to be caused, or features to be duplicated. Project teams and regular community meetings try to minimize issues, but they do happen.
- With single vendor products, slow to change by OSS standards, one can get a list of known issues and troubleshoot to discover the particular problem they encounter. With open access, issues are typically fixed quickly, meaning that a traditional approach may not be optimal. Instead, troubleshooting is an ongoing process, where even code or a feature which worked yesterday may have an issue today. Regular revisiting of assumptions and applying an iterative flow to discover and fix issues becomes essential.
- Be aware that the growth of agents follows project-based needs. When people working on cloud interaction needed to wait and integrate each change with overall agent management, a new agent was created, as well as a new project to develop it, called cloud controller manager (CCM). As a result, finding known bug information may require regular review of various websites as the project changes.

12.4 Basic Troubleshooting Flow: Pods

- Should the error not be seen while starting the Pod, investigate from within the Pod. A flow working from the application running inside a container to the larger cluster as a whole may be a good idea. Some applications will have a shell available to run commands, for example:

```
$ kubectl create deployment tester --image=nginx
$ kubectl exec -ti tester-<tab> -- /bin/sh
```

- A feature which is still in `alpha` state, and may change in anyway or disappear is the ability to attach an image to a running process. This is very helpful for testing the existing pod rather than a new pod, with different configuration. Begin by reading through some of the included examples, then try to attach to an existing pod. You will get an error as `--feature-gates=EphemeralContainers=true` is not set for the kube-apiserver and kube-scheduler by default.

```
$ kubectl alpha debug -h
```

```
<output_omitted>
```

```
$ kubectl alpha debug registry-6b5bb79c4-jd8fj --image=busybox
```

```
error: ephemeral containers are disabled for this cluster
(error from server: "the server could not find the requested resource").
```

- Now test a different feature. Create a container to understand the view a container has of the node it is running on. Change out the node name, `master` for your master node. Explore the commands you have available.

```
$ kubectl alpha debug node master -it --image=busybox
```

On Container

```
/ # ps
```

```
....
31585 root      2:00 /coredns -conf /etc/coredns/Corefile
31591 root      0:21 containerd-shim -namespace moby -workdir /var/lib/containe
31633 root      0:12 /usr/bin/kube-controllers
31670 root      0:05 bird -R -s /var/run/calico/bird.ctl -d -c /etc/calico/conf
31671 root      0:05 bird6 -R -s /var/run/calico/bird6.ctl -d -c /etc/calico/co
31676 root      0:01 containerd-shim -namespace moby -workdir /var/lib/containe
31704 root      0:07 /usr/local/bin/kube-proxy --config=/var/lib/kube-proxy/con
```

```
/ # df
```

```
<output_omitted>
```

- If the Pod is running, use `kubectl logs pod-name` to view the 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.

- For pods without the ability to log on their own, you may have to attach a sidecar container. These can be configured to either stream application logs to their own standard out, or run a logging agent.

- Troubleshooting an application begins with the application itself. Is the application behaving as expected? Transient issues are difficult to troubleshoot; difficulties in troubleshooting are also encountered if the issue is intermittent or if it concerns occasional slow performance.

- Assuming the application is not the issue, begin by viewing the pods with the kubectl get command. Ensure the pods report a status of Running. A status of Pending typically means a resource is not available from the cluster, such as a properly tainted node, expected storage, or enough resources. Other error codes typically require looking at the logs and events of the containers for further troubleshooting. Also, look for an unusual number of restarts. A container is expected to restart due to several reasons, such as a command argument finishing. If the restarts are not due to that, it may indicate that the deployed application is having an issue and failing due to panic or probe failure.
- View the details of the pod and the status of the containers using the `kubectl describe pod command`. This will report overall pod status, container configurations and container events. Work through each section looking for a reported error.
- Should the reported information not indicate the issue, the next step would be to view any logs of the container, in case there is a misconfiguration or missing resource unknown to the cluster, but required by the application. These logs can be seen with the `kubectl logs <pod-name> <container-name>` command.

12.5 Basic Troubleshooting Flow: Node and Security

- 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.
- Disabling security for testing would be a common response to an issue. Each tool has its own logs and indications of rule violation. There could be multiple issues to troubleshoot, so be sure to re-enable security and test the workload again.
- Internode networking can also be an issue. Changes to switches, routes, or other network settings can adversely affect Kubernetes. Historically, the primary causes were DNS and firewalls. As Kubernetes integrations have become more common and DNS integration is maturing, this has become less of an issue. Still, check connectivity and for recent infrastructure changes as part of your troubleshooting process. Every so often, an update which was said shouldn't cause an issue may, in fact, be causing an issue.

12.6 Basic Troubleshooting Flow: Agents

- The issues found with a decoupled system like Kubernetes are similar to those of a traditional datacenter, plus the added layers of Kubernetes controllers:
  - Control pods in pending or error state
  - Look for errors in log files
  - Are there enough resources?
  - etc.

12.7 Monitoring

- Monitoring is about collecting metrics from the infrastructure, as well as applications.
- **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.
- Collecting the metrics is the first step, making use of the data the next. As open source software we have the ability to expose lots of data points. Graphing the data with a tool like **Grafana** can allow for visual understanding of the cluster and application status. Some facilities fill walls with large TVs offering the entire company a real time view of demand and utilization.

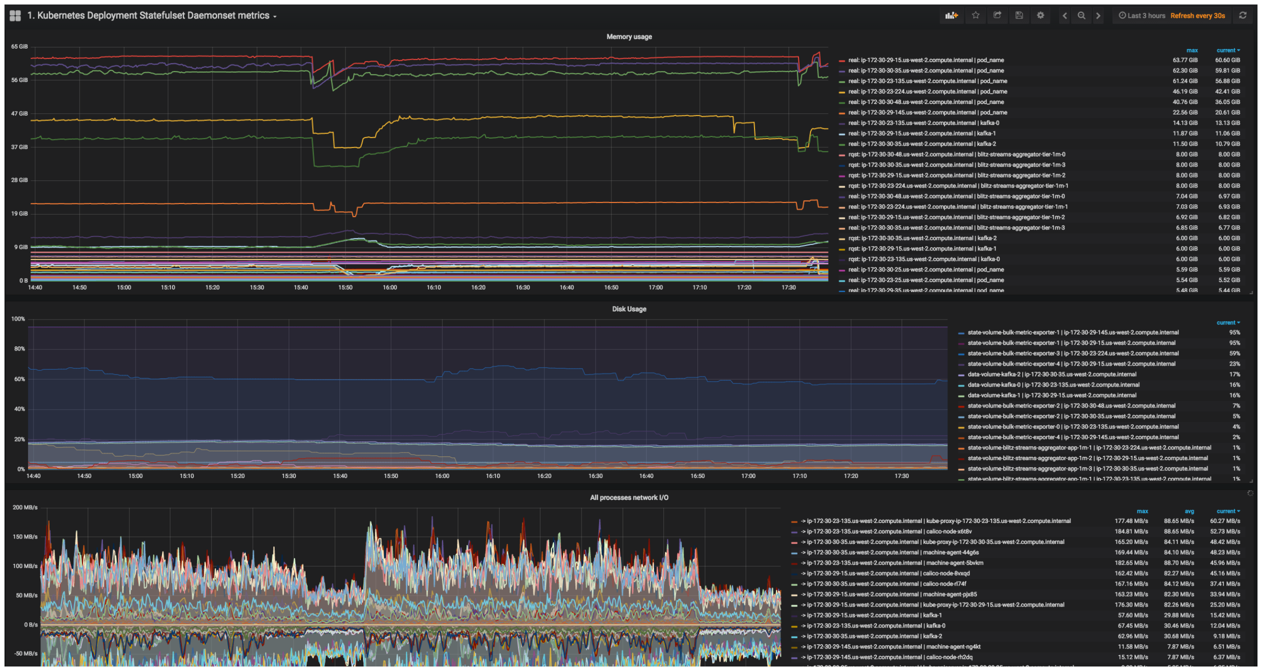

1. Kubernetes Deployment Statefulset Daemonset metrics
Memory usage
Disk Usage
All processes network I/O

Figure 12.1: **Grafana Dashboard Example**

1

12.8 Logging Tools

- 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**) at https://www.elastic.co/videos/introduction-to-the-elk-stack has become quite common.
- In Kubernetes, the **kubelet** writes container logs to local files (via the Docker logging driver). The `kubectl logs` command allows you to retrieve these logs.
- Cluster-wide, you can use **Fluentd** (https://www.fluentd.org/ to aggregate logs. Check the cluster administration logging concepts at https://www.fluentd.org/ for a detailed description.
- Fluentd is part of the Cloud Native Computing Foundation and, together with Prometheus, they make a nice combination for monitoring and logging. You can find a detailed walk-through of running Fluentd on Kubernetes in the Kubernetes documentation at
https://kubernetes.io/docs/tasks/debug-application-cluster/logging-elasticsearch-kibana/.
- Setting up Fluentd for Kubernetes logging is a good exercise in understanding `DaemonSets`. Fluentd agents run on each node via a `DaemonSet`, they aggregate the logs, and feed them to an Elasticsearch instance prior to visualization in a Kibana dashboard.

12.9 Monitoring Applications

- As a distributed system, Kubernetes lacks monitoring and tracing tools which are cluster-aware. Other CNCF projects have started to handle various areas. As they are independent projects, you may find they have some overlap in capability:

[1] Image downloaded from https://grafana.com/grafana/dashboards/8588 17-August-2019

- **Prometheus**
  Focused on metrics and alerting. Provides a time-series database, queries and alerts to gather information and alerts for the entire cluster. With integration with Grafana, as well as simple graphs and console templates, you can easily visualize the cluster condition graphically
- **Fluentd**
  A unified logging layer which can collect logs from over 500 plugins and deliver to a multitude of outputs. A single, enterprise-wide tool to filter, buffer, and route messages just about anywhere you may want.
- **OpenTracing**
  While we just learned about logging and metrics, neither of the previously mentioned projects allow for a granular understanding of a transaction across a distributed architecture. This project seems to provide worthwhile instrumentation to propagate tracing among all services, code and packages, so the trace can be complete. Its goal is a "single, standard mechanism" for everything.
- **Jaeger**
  A tracing system developed by Uber, focused on distributed context propagation, transaction monitoring, and root cause analysis, among other features. This has become a common tracing implementation of OpenTracing.

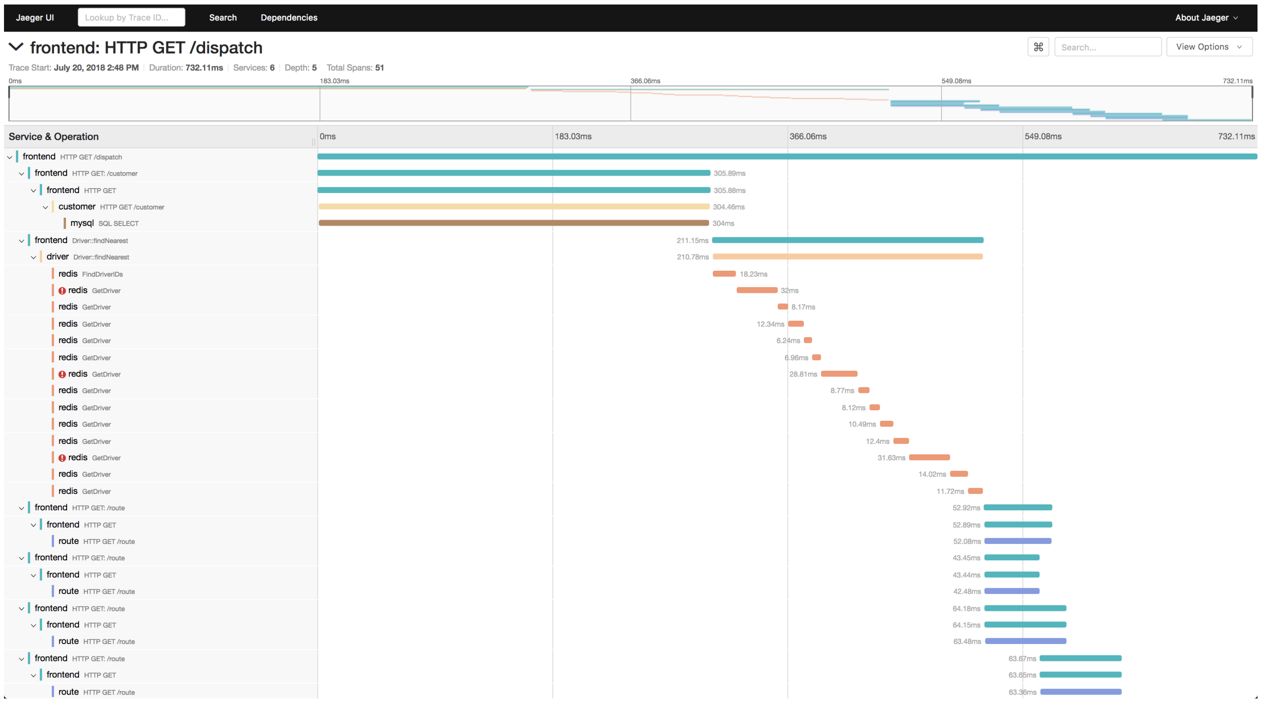

Jaeger UI
Lookup by Trace ID...
Search
Dependencies
About Jaeger
frontend: HTTP GET /dispatch
Search...
View Options
Trace Start: July 20, 2018 2:48 PM | Duration: 732.11ms | Services: 6 | Depth: 5 | Total Spans: 51
0ms
183.03ms
366.06ms
549.08ms
732.11ms
Service & Operation
frontend HTTP GET /dispatch
frontend HTTP GET: /customer
frontend HTTP GET
customer HTTP GET /customer
mysql SQL SELECT
frontend Driver::findNearest
driver Driver::findNearest
redis FindDriverIDs
redis GetDriver
frontend HTTP GET: /route
route HTTP GET /route

Figure 12.2: **Jaeger Tracing Example**

[2]

12.10 System and Agent Logs

- Where system and agent files are found depends on the existence of **systemd**. Those with systemd will log to **journalctl**, which can be viewed with `journalctl -a`. Unless the `/var/log/journal` directory exists, the journal is volatile. As Kubernetes can generate a lot of logs, it would be advisable to configure log rotation, should this directory be created.
- Without **systemd**, the logs will be created under `/var/log/<agent>.log`, in which case it would also be advisable to configure log rotation.
- Container components:

[2]Image downloaded from https://www.jaegertracing.io/img/trace-detail-ss.png 17-August-2019

    - kube-scheduler
    - kube-proxy

- Non-container components:
    - kubelet
    - Docker
    - Many others.

12.11 Conformance Testing

- The flexibility of Kubernetes can lead to the development of a non-conforming cluster.
    - Meet the demands of your environment
    - Several vendor-provided tools for conformance testing
    - For ease of use, Sonobuoy by Heptio can be used to understand the state of the cluster.

Certified Kubernetes Conformant Program

- With more than 60 known distributions, there is a challenge to consistency and portability. In late 2017, a new program was started by CNCF to certify distributions that meet essential requirements and adhere to complete API functionality:
    - Confidence that workloads from one distribution work on others
    - Guarantees complete API functions
    - Testing and Architecture Special Interest Groups.
- You can read more about submitting conformance results on GitHub at https://github.com/cncf/k8s-conformance/blob/master/instructions.md.

12.12 More Resource

- There are several things that you can do to quickly diagnose potential issues with your application and/or cluster. The official documentation offers additional materials to help you get familiar with troubleshooting:
    - General guidelines and instructions (Troubleshooting)
      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 website for issues and bug tracking
      https://github.com/kubernetes/kubernetes/issues
    - Kubernetes Slack channel
      https://kubernetes.slack.com/

12.13 Labs

Exercise 12.1: Troubleshooting: Monitor Applications

Overview

Troubleshooting can be difficult in a multi-node, decoupled and transient environment. Add in the rapid pace of change and it becomes more difficult. Instead of focusing and remembering a particular error and the fix it may be more useful to learn a flow of troubleshooting and revisit assumptions until the pace of change slows and various areas further mature.

1. View the `secondapp` pod, it should show as `Running`. This may not mean the application within is working properly, but that the pod is running. The restarts are due to the command we have written to run. The pod exists when done, and the controller restarts another container inside. The count depends on how long the labs have been running.

```
student@master/app2:~$ cd
student@master:~$ kubectl get pods secondapp
```

```
NAME                            READY     STATUS      RESTARTS    AGE
secondapp                       2/2       Running     49          2d
```

2. Look closer at the pod. Working slowly through the output check each line. If you have issues, are other pods having issues on the same node or volume? Check the state of each container. Both `busy` and `webserver` should report as `Running`. Note `webserver` has a restart count of zero while `busy` has a restart count of 49. We expect this as, in our case, the pod has been running for 49 hours.

```
student@master:~$ kubectl describe pod secondapp
```

```
Name:         secondapp
Namespace:    default
Node:         worker-wdrq/10.128.0.2
Start Time:   Fri, 13 Apr 2018 20:34:56 +0000
Labels:       example=second
Annotations:  <none>
Status:       Running
IP:           192.168.55.91
Containers:
  webserver:
<output_omitted>
    State:          Running
      Started:      Fri, 13 Apr 2018 20:34:58 +0000
    Ready:          True
    Restart Count:  0
<output_omitted>

  busy:
<output_omitted>

    State:          Running
      Started:      Sun, 15 Apr 2018 21:36:20 +0000
    Last State:     Terminated
      Reason:       Completed
      Exit Code:    0
      Started:      Sun, 15 Apr 2018 20:36:18 +0000
      Finished:     Sun, 15 Apr 2018 21:36:18 +0000
    Ready:          True
    Restart Count:  49
    Environment:    <none>
```

3. There are three values for conditions. Check that the pod reports Initialized, Ready and scheduled.

```
<output_omitted>
Conditions:
  Type            Status
  Initialized     True
  Ready           True
  PodScheduled    True
<output_omitted>
```

4. Check if there are any events with errors or warnings which may indicate what is causing any problems.

```
Events:
  Type    Reason   Age                 From                   Message
  ----    ------   ----                ----                   -------
  Normal  Pulling  34m (x50 over 2d)  kubelet, worker-wdrq  pulling
image "busybox"
  Normal  Pulled   34m (x50 over 2d)  kubelet, worker-wdrq  Successfully
pulled image "busybox"
  Normal  Created  34m (x50 over 2d)  kubelet, worker-wdrq  Created
container
  Normal  Started  34m (x50 over 2d)  kubelet, worker-wdrq  Started
container
```

5. View each container log. You may have to sift errors from expected output. Some containers may have no output at all, as is found with `busy`.

```
student@master:~$ kubectl logs secondapp webserver
```

```
192.168.55.0 - - [13/Apr/2018:21:18:13 +0000] "GET / HTTP/1.1" 200
612 "-" "curl/7.47.0" "-"
192.168.55.0 - - [13/Apr/2018:21:20:35 +0000] "GET / HTTP/1.1" 200
612 "-" "curl/7.53.1" "-"
127.0.0.1 - - [13/Apr/2018:21:25:29 +0000] "GET" 400 174 "-" "-" "-"
127.0.0.1 - - [13/Apr/2018:21:26:19 +0000] "GET index.html" 400 174
"-" "-" "-"
<output_omitted>
```

```
student@master:~$ kubectl logs secondapp busy
```

```
student@master:~$
```

6. Check to make sure the container is able to use DNS and communicate with the outside world. Remember we still have limited the UID for `secondapp` to be UID **2000**, which may prevent some commands from running. It can also prevent an application from completing expected tasks, and other errors.

```
student@master:~$ kubectl exec  -it  secondapp -c busy -- sh
```

On Container

```
/ $ nslookup www.linuxfoundation.org
```

```
/ $ nslookup www.linuxfoundation.org
Server:                 10.96.0.10
Address:        10.96.0.10:53

Non-authoritative answer:
Name:         www.linuxfoundation.org
Address: 23.185.0.2

*** Can't find www.linuxfoundation.org: No answer
```

```
/ $ 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
```

Test access to a remote node using **nc (NetCat)**. There are several options to **nc** which can help troubleshoot if the problem is the local node, something between nodes or in the target. In the example below the connect never completes and a **control-c** was used to interrupt.

7. `/ $ nc www.linux.com 25`

```
^Cpunt!
```

8. Test using an IP address in order to narrow the issue to name resolution. In this case the IP in use is a well known IP for Google's DNS servers. The following example shows that Internet name resolution is working, but our UID issue prevents access to the `index.html` file.

```
/ $ wget http://www.linux.com/
```

```
Connecting to www.linux.com (151.101.45.5:80)
Connecting to www.linux.com (151.101.45.5:443)
wget: can't open 'index.html': Permission denied

/ $ exit
```

9. Make sure traffic is being sent to the correct Pod. Check the details of both the service and endpoint. Pay close attention to ports in use as a simple typo can prevent traffic from reaching the proper pod. Make sure labels and selectors don't have any typos as well.

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

```
NAME         TYPE          CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
kubernetes   ClusterIP     10.96.0.1       <none>        443/TCP        10d
nginx        ClusterIP     10.108.95.67    <none>        443/TCP        10d
registry     ClusterIP     10.105.119.236  <none>        5000/TCP       10d
secondapp    LoadBalancer  10.109.26.21    <pending>     80:32000/TCP   1d
thirdpage    NodePort      10.109.250.78   <none>        80:31230/TCP   1h
```

```
student@master:~$ kubectl get svc secondapp -o yaml
```

```
<output_omitted>
  clusterIP: 10.109.26.21
  externalTrafficPolicy: Cluster
  ports:
  - nodePort: 32000
    port: 80
    protocol: TCP
    targetPort: 80
  selector:
    example: second
<output_omitted>
```

10. Verify an endpoint for the service exists and has expected values, including namespaces, ports and protocols.

```
student@master:~$ kubectl get ep
```

```
NAME         ENDPOINTS           AGE
kubernetes   10.128.0.3:6443     10d
nginx        192.168.55.68:443   10d
registry     192.168.55.69:5000  10d
secondapp    192.168.55.91:80    1d
thirdpage    192.168.241.57:80   1h
```

```
student@master:~$ kubectl get ep secondapp -o yaml
```

```
apiVersion: v1
kind: Endpoints
metadata:
  creationTimestamp: 2018-04-14T05:37:32Z
<output_omitted>
```

11. If the containers, services and endpoints are working the issue may be with an infrastructure service like **kube-proxy**. Ensure it's running, then look for errors in the logs. As we have two nodes we will have two proxies to look at. As we built our cluster with **kubeadm** the proxy runs as a container. On other systems you may need to use **journalctl** or look under `/var/log/kube-proxy.log`.

```
student@master:~$ ps -elf |grep kube-proxy
```

```
4 S root      2864  2847  0  80   0 - 14178 -      15:45 ?
 00:00:56 /usr/local/bin/kube-proxy --config=/var/lib/kube-proxy/config.conf
0 S student  23513 18282  0  80   0 -  3236 pipe_w 22:49 pts/0
 00:00:00 grep --color=auto kube-proxy
```

```
student@master:~$ journalctl -a | grep proxy
```

```
Apr 15 15:44:43 worker-nzjr audit[742]: AVC apparmor="STATUS"
 operation="profile_load" profile="unconfined" \
  name="/usr/lib/lxd/lxd-bridge-proxy" pid=742 comm="apparmor_parser"
Apr 15 15:44:43 worker-nzjr kernel: audit: type=1400
 audit(1523807083.011:11): apparmor="STATUS" \
  operation="profile_load" profile="unconfined" \
   name="/usr/lib/lxd/lxd-bridge-proxy" pid=742 comm="apparmor_parser"
Apr 15 15:45:17 worker-nzjr kubelet[1248]: I0415 15:45:17.153670
 1248 reconciler.go:217] operationExecutor.VerifyControllerAttachedVolume\
  started for volume "xtables-lock" \
   (UniqueName: "kubernetes.io/host-path/e701fc01-38f3-11e8-a142-\
   42010a800003-xtables-lock") \
   pod "kube-proxy-t8k4w" (UID: "e701fc01-38f3-11e8-a142-42010a800003")
```

12. Look at both of the proxy logs. Lines which begin with the character **I** are `info`, **E** are `errors`. In this example the last message says access to listing an endpoint was denied by RBAC. It was because a default installation via Helm wasn't RBAC aware. If not using command line completion, view the possible pod names first.

```
student@master:~$ kubectl -n kube-system get pod
```

```
student@master:~$ kubectl -n kube-system logs kube-proxy-fsdfr
```

```
I0405 17:28:37.091224       1 feature_gate.go:190] feature gates: map[]
W0405 17:28:37.100565       1 server_others.go:289] Flag proxy-mode=""
unknown, assuming iptables proxy
I0405 17:28:37.101846       1 server_others.go:138] Using iptables Proxier.
I0405 17:28:37.121601       1 server_others.go:171] Tearing down
inactive rules.
<output_omitted>
E0415 15:45:17.086081       1 reflector.go:205] \
  k8s.io/kubernetes/pkg/client/informers/informers_generated/
  internalversion/factory.go:85: \
  Failed to list *core.Endpoints: endpoints is forbidden: \
    User "system:serviceaccount:kube-system:kube-proxy" cannot \
   list endpoints at the cluster scope:\
 [clusterrole.rbac.authorization.k8s.io "system:node-proxier" not found, \
   clusterrole.rbac.authorization.k8s.io "system:basic-user" not found,
 clusterrole.rbac.authorization.k8s.io \
 "system:discovery" not found]
```

13. Check that the proxy is creating the expected rules for the problem service. Find the destination port being used for the service, **32000** in this case.

```
student@master:˜$ sudo iptables-save |grep secondapp
```

```
-A KUBE-NODEPORTS -p tcp -m comment --comment "default/secondapp:" \
-m tcp --dport 32000 -j KUBE-MARK-MASQ
-A KUBE-NODEPORTS -p tcp -m comment --comment "default/secondapp:" \
-m tcp --dport 32000 -j KUBE-SVC-DAASHM5XQZF5XI3E
-A KUBE-SERVICES ! -s 192.168.0.0/16 -d 10.109.26.21/32 -p tcp \
-m comment --comment "default/secondapp: \
      cluster IP" -m tcp --dport 80 -j KUBE-MARK-MASQ
-A KUBE-SERVICES -d 10.109.26.21/32 -p tcp -m comment --comment \
"default/secondapp: cluster IP" -m tcp \
      --dport 80 -j KUBE-SVC-DAASHM5XQZF5XI3E
<output_omitted>
```

14. Ensure the proxy is working by checking the port targeted by **iptables**. If it fails open a second terminal and view the proxy logs when making a request as it happens.

```
student@master:˜$ curl localhost:32000
```

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

Exercise 12.2: OPTIONAL LAB: Conformance Testing

The **cncf.io** group is in the process of formalizing what is considered to be a conforming Kubernetes cluster. While that project matures there is an existing tool provided by **Heptio** which can be useful. We will need to make sure a newer version of **Golang** is installed for it to work. You can download the code from github and look around with git or with go, depending on which tool you are most familiar. **Things change quickly these steps may not work....today**

1. Download a compiled binary. A shorter URL is shown first, then the longer, just in case the link changes and you need to navigate. They should download the same file.

```
student@master:˜$ curl -sLO https://tinyurl.com/yyu5bs28

student@master:˜$ mv yyu5bs28 sonobuoy.tar.gz

student@master:˜$ tar -xvf sonobuoy.tar.gz
```

```
LICENSE
sonobuoy
```

```
student@master:˜$ curl -sLO \
https://github.com/heptio/sonobuoy/releases/download/v0.15.4/sonobuoy_0.15.4_linux_amd64.tar.gz
```

2. Run the test. We will not use the `--wait` option, which will capture the screen until the test finishes. This could take a while to finish. You should get some output indicating testing objects being created.

```
student@master:~$ sudo mv sonobuoy /usr/local/bin/

student@master:~$ sonobuoy run
```

```
WARN[0000] The maximum supported Kubernetes version is 1.15.99, but
the server version is v1.16.1. Sonobuoy will continue but unexpected results may occur.
INFO[0000] created object                     name=sonobuoy namespace= resource=namespaces
INFO[0000] created object                     name=sonobuoy-serviceaccount namespace=sonobuoy ....
INFO[0000] created object                     name=sonobuoy-serviceaccount-sonobuoy namespace=....
INFO[0000] created object                     name=sonobuoy-serviceaccount namespace= resource....
INFO[0000] created object                     name=sonobuoy-config-cm namespace=sonobuoy resou....
INFO[0000] created object                     name=sonobuoy-plugins-cm namespace=sonobuoy reso....
INFO[0000] created object                     name=sonobuoy namespace=sonobuoy resource=pods
INFO[0000] created object                     name=sonobuoy-master namespace=sonobuoy resource....
```

3. View the results inside the `sonobuoy` pod.

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

```
<output_omitted>
sonobuoy      sonobuoy                                                  1/1
     Running   0          90s
sonobuoy      sonobuoy-e2e-job-b3bcb52b4fd54367                         2/2
     Running   0          85s
sonobuoy      sonobuoy-systemd-logs-daemon-set-f7ca2bb9a7174908-h47kb   2/2     Running   0          85s
sonobuoy      sonobuoy-systemd-logs-daemon-set-f7ca2bb9a7174908-s22d6   2/2     Running   0          85s
```

```
student@master:~$ kubectl -n sonobuoy exec -it sonobuoy -- /bin/bash
```

On Container

4. View the files inside the container.

```
root@sonobuoy:/# ls
```

```
bin    home   mnt         root                       sbin      tmp
boot   lib    opt         run                        sonobuoy  usr
dev    lib64  plugins.d   run_master.sh              srv       var
etc    media  proc        run_single_node_worker.sh  sys
```

5. View the `run_master.sh` script. Note that it mentions both the **sonobuoy** command and where to find the results.

```
root@sonobuoy:/# cat run_master.sh
```

```
#!/bin/bash
##########################################################################
# Copyright 2017 Heptio Inc.
#

<output_omitted>
RESULTS_DIR="${RESULTS_DIR:-/tmp/sonobuoy}"
# It's ok for these env vars to be unbound
RESULTS_DIR="${RESULTS_DIR}" SONOBUOY_CONFIG="${SONOBUOY_CONFIG}"
SONOBUOY_ADVERTISE_IP="${SONOBUOY_ADVERTISE_IP}" /sonobuoy master -v 3 --logtostderr

echo -n "${RESULTS_DIR}/$(ls -t "${RESULTS_DIR}" | grep -v done | head -n 1)" > "${RESULTS_DIR}"/done
```

6. View the contents of the `/tmp/sonobuoy` directory. Note the subdirectory is a generated number, yours will be different. The **Tab** key can be used to complete the path.

```
root@sonobuoy:/# ls /tmp/sonobuoy/
```

```
d39f2629-fa3c-4a0b-9b33-53080e78b57b
```

```
root@sonobuoy:/# cd /tmp/sonobuoy/d39f2629-fa3c-4a0b-9b33-53080e78b57b ; ls
```

```
meta  plugins
```

```
root@sonobuoy:...57b# find .
```

```
.
./plugins
./plugins/systemd-logs
./plugins/systemd-logs/results
./plugins/systemd-logs/results/e-6clr
./plugins/systemd-logs/results/e-6clr/systemd_logs
./plugins/systemd-logs/results/e-5c7t
./plugins/systemd-logs/results/e-5c7t/systemd_logs
./meta
./meta/run.log
./meta/config.json
```

7. The **sonobuoy** command has several options. We will use two to explore the test output.

```
root@sonobuoy:...57b# cd /

root@sonobuoy:/# ./sonobuoy status
```

```
         PLUGIN     STATUS   RESULT   COUNT
            e2e    running                1
   systemd-logs   complete                2

Sonobuoy is still running. Runs can take up to 60 minutes.
```

```
root@sonobuoy:/# ./sonobuoy logs
```

```
<output_omitted>
```

8. Continue to look through tests and results as time permits. Connect to the other pods in the `sonobuoy` namespace and look for log and result files.

   There is also an online, graphical scanner. In testing, inside GCE, the results were blocked and never returned. You may have different outcome in other environments.

Exercise 12.3: Domain Review

Very Important

The source pages and content in this review could change at any time. **IT IS YOUR RESPONSIBILITY TO CHECK THE CURRENT INFORMATION.**

Revisit the CKAD domain list on Curriculum Overview and locate some of the topics we have covered in this chapter.

• Understand debugging in Kubernetes

Figure 12.3: **Troubleshooting Domain Topic**

Focus on ensuing you have all the necessary files and processes understood first. Repeat the review until you are sure you have bookmarks of necessary YAML samples and can complete each step quickly, and ensure each object is running properly.

1. Find and use the `troubleshoot-review1.yaml` file to create a deployment. The **create** command will fail. Edit the file to fix issues such that a single pod runs for at least a minute without issue. There are several things to fix.

   ```
   student@master:~$ kubectl create -f troubleshoot-review1.yaml
   ```

   ```
   <Fix any errors found here>
   ```

   When fixed it should look like this:

   ```
   student@master:~$ kubectl get deploy igottrouble
   ```

   ```
   NAME          READY   UP-TO-DATE   AVAILABLE   AGE
   igottrouble   1/1     1            1           5m13s
   ```

2. Remove any resources created during this review.

Chapter 13

Logging and Troubleshooting - II

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 been ingested into Kubernetes with the **Metric Server**, which 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 master 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
- Master 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
```

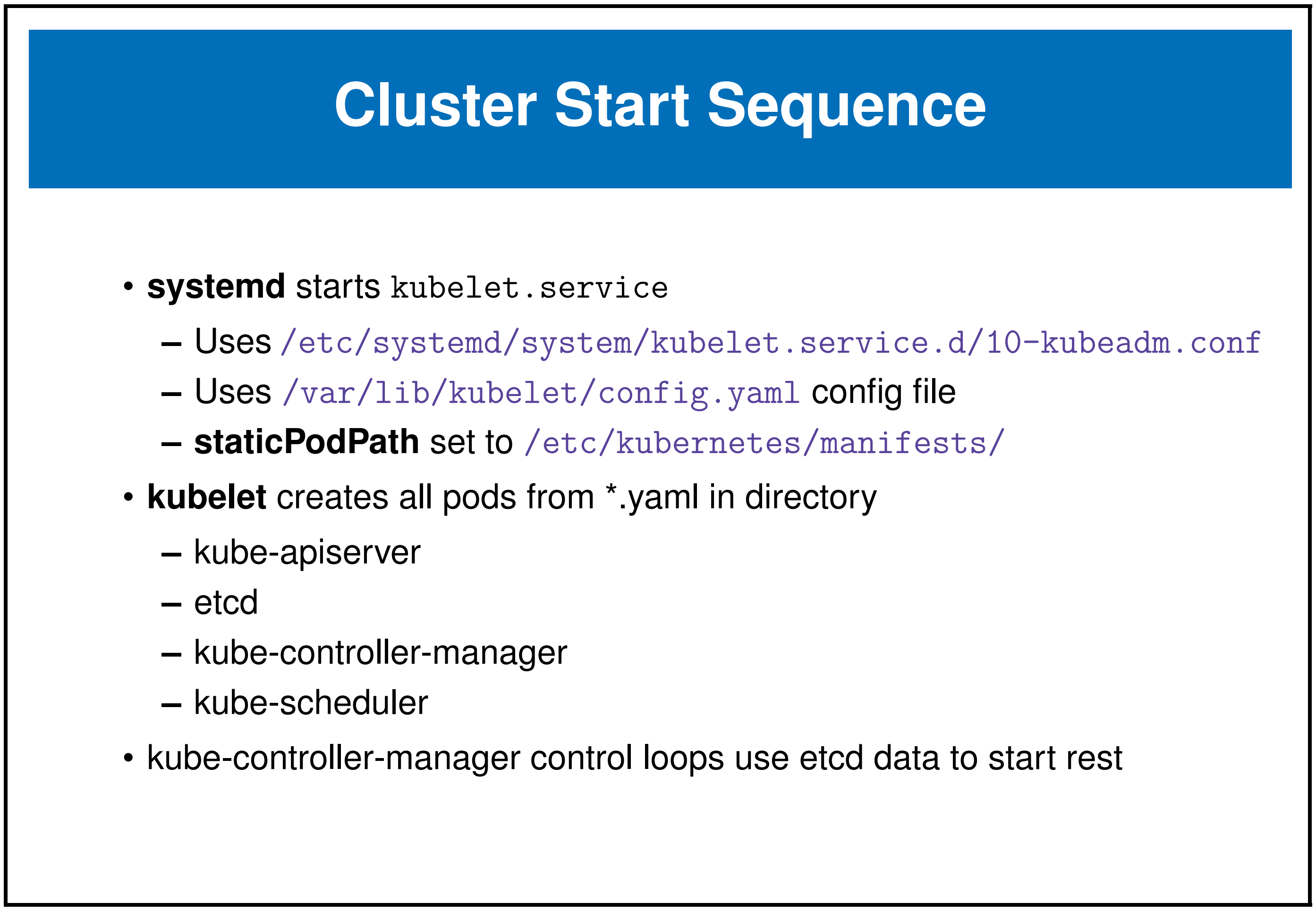
A feature new to the 1.16 version is the ability to add a container to a running pod. This would allow a feature-filled container to be added to an existing pod without having to terminate and re-create. Intermittent and difficult to determine problems may take a while to reproduce, or not exist with 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

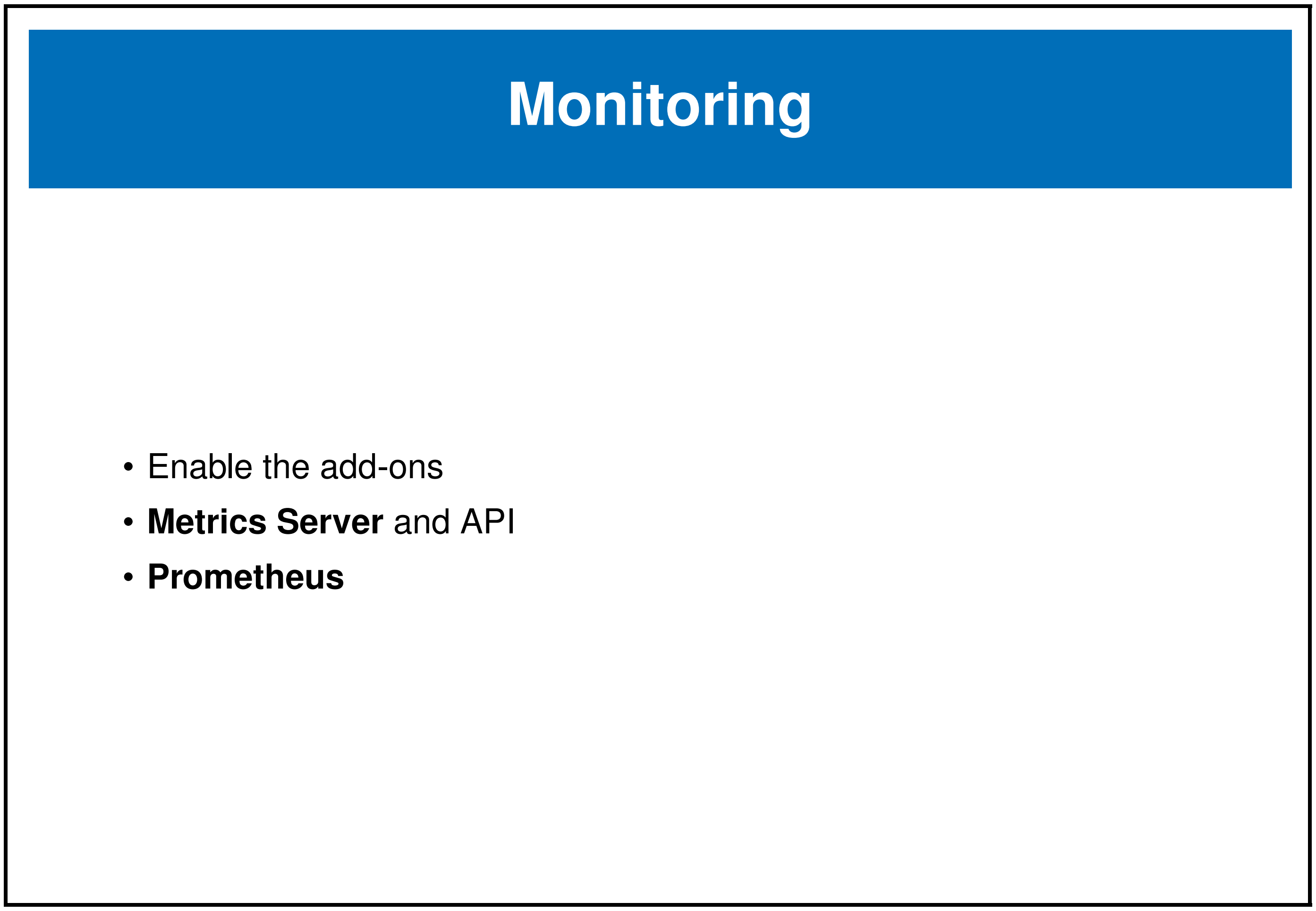
- **systemd** starts `kubelet.service`
  - Uses `/etc/systemd/system/kubelet.service.d/10-kubeadm.conf`
  - Uses `/var/lib/kubelet/config.yaml` config file
  - **staticPodPath** set to `/etc/kubernetes/manifests/`
- **kubelet** creates all pods from *.yaml in directory
  - kube-apiserver
  - etcd
  - 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**

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.

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.

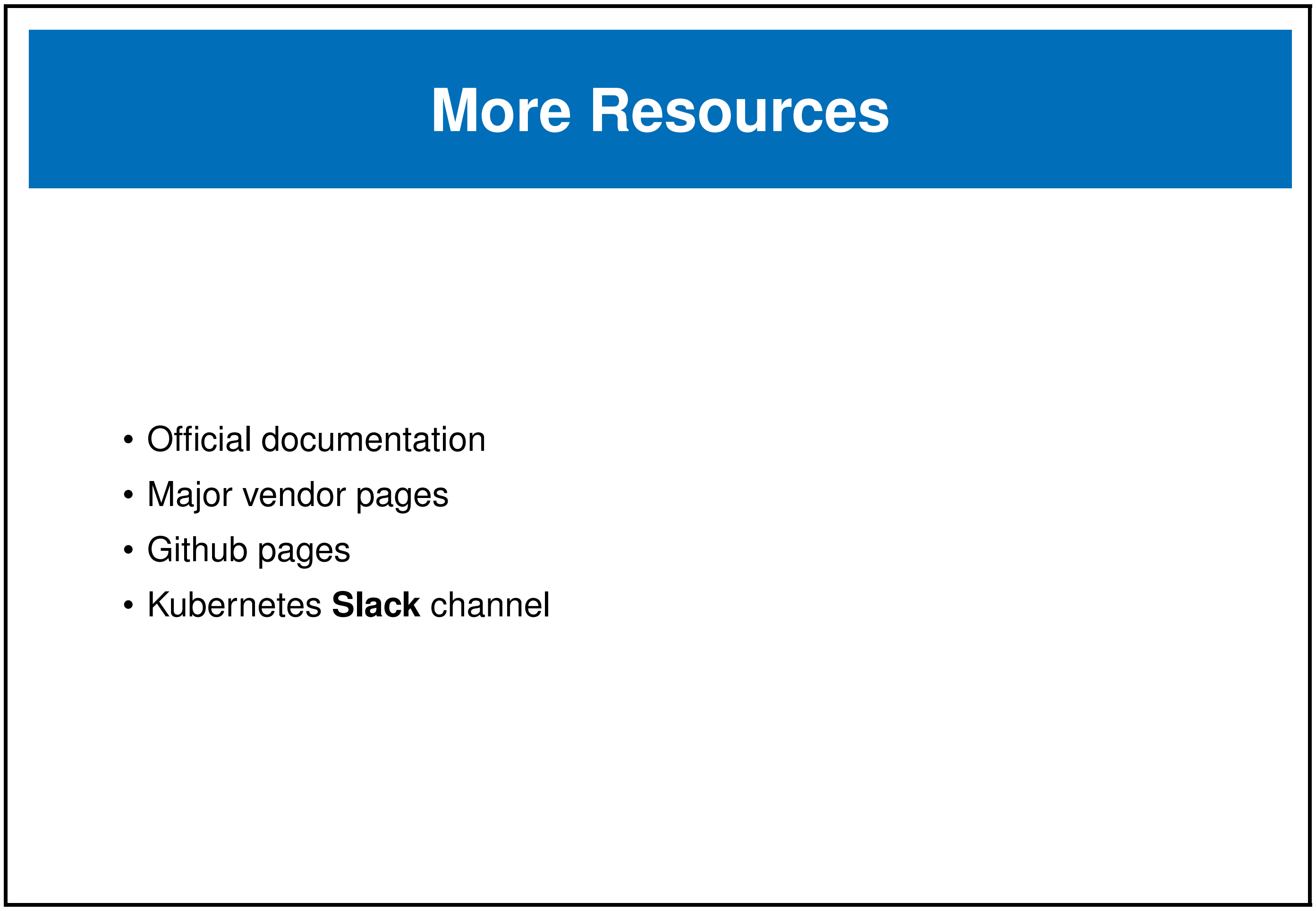
In Kubernetes, the **kubelet** writes container logs to local files (via the **Docker** logging driver). The command `kubectl logs` allows you to retrieve these logs.

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@master:˜$ 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@master:˜$ sudo find / -name "*apiserver*log"
```

```
/var/log/containers/kube-apiserver-master_kube-system_kube-apiserver-423
d25701998f68b503e64d41dd786e657fc09504f13278044934d79a4019e3c.log
```

3. Take a look at the log file.

```
student@master:˜$ sudo less /var/log/containers/kube-apiserver-master_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 master 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@master:~$ kubectl get po --all-namespaces
```

```
NAMESPACE     NAME                                       READY     STATUS    RESTARTS   AGE
kube-system   calico-kube-controllers-7b9dcdcc5-qg6zd   1/1      Running   0          13m
kube-system   calico-node-dr279                          1/1      Running   0          6d1h

....
kube-system   etcd-master                       1/1       Running   2          44h
kube-system   kube-apiserver-master             1/1       Running   2          44h
kube-system   kube-controller-manager-master    1/1       Running   2          44h
kube-system   kube-scheduler-master             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@master:~$ kubectl -n kube-system logs <Tab><Tab>
```

```
calico-kube-controllers-7b9dcdcc5-qg6zd
calico-node-dr279
calico-node-xtvfd
coredns-5644d7b6d9-k7kts
coredns-5644d7b6d9-rnr2v
etcd-master
kube-apiserver-master
kube-controller-manager-master
kube-proxy-qhc4f
kube-proxy-s56hl
kube-scheduler-f-master
traefik-ingress-controller-hw5tv
traefik-ingress-controller-mcn47
```

```
student@master:~$  kubectl -n kube-system logs \
          kube-apiserver-master
```

```
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: v1.19.0
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>
```

3. 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

Very Important

The `metrics-server` is written to interact with Docker. If you chose to use crio the logs will show errors and inability to collect metrics.

1. Begin by cloning the software. The **git** command should be installed already. Install it if not found.

```
student@master:~$ git clone \
      https://github.com/kubernetes-incubator/metrics-server.git
```

```
<output_omitted>
```

2. As the software may have changed it is a good idea to read the **README.md** file for updated information.

```
student@master:~$ cd metrics-server/ ; less README.md
```

```
<output_omitted>
```

3. 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@master:~$ kubectl create -f \
https://github.com/kubernetes-sigs/metrics-server/releases/download/v0.3.7/components.yaml
```

```
clusterrole.rbac.authorization.k8s.io/system:aggregated-metrics-reader created
clusterrolebinding.rbac.authorization.k8s.io/metrics-server:system:auth-delegator created
rolebinding.rbac.authorization.k8s.io/metrics-server-auth-reader created
apiservice.apiregistration.k8s.io/v1beta1.metrics.k8s.io created
serviceaccount/metrics-server created
deployment.apps/metrics-server created
service/metrics-server created
clusterrole.rbac.authorization.k8s.io/system:metrics-server created
clusterrolebinding.rbac.authorization.k8s.io/system:metrics-server created
```

4. View the current objects, which are created in the `kube-system` namespace. All should show a `Running` status.

```
student@master:~$ kubectl -n kube-system get pods
```

```
<output_omitted>
kube-proxy-ld2hb                    1/1     Running   0          2d21h
kube-scheduler-u16-1-13-1-2f8c      1/1     Running   0          2d21h
metrics-server-fc6d4999b-b9rjj      1/1     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@master:~$ kubectl -n kube-system edit deployment metrics-server
```

```yaml
....
    spec:
      containers:
      - args:
        - --cert-dir=/tmp
        - --secure-port=4443
        - --kubelet-insecure-tls                    #<-- Add this line
        - --kubelet-preferred-address-types=InternalIP,ExternalIP,Hostname #<--May be needed
        image: k8s.gcr.io/metrics-server/metrics-server:v0.3.7
....
```

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@master:~$ 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@master:~$ sleep 120 ; kubectl top pod --all-namespaces
```

```
NAMESPACE     NAME                                       CPU(cores)   MEMORY(bytes)
kube-system   calico-kube-controllers-7b9dcdcc5-qg6zd   2m           6Mi
kube-system   calico-node-dr279                          23m          22Mi
kube-system   calico-node-xtvfd                          21m          22Mi
kube-system   coredns-5644d7b6d9-k7kts                   2m           6Mi
kube-system   coredns-5644d7b6d9-rnr2v                   3m           6Mi
<output_omitted>
```

```
student@master:~$ kubectl top nodes
```

```
NAME              CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
master   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@master:~$ curl --cert ./client.pem \
    --key ./client-key.pem --cacert ./ca.pem \
    https://k8smaster: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": "2019-01-10T20:27:00Z"
      },
      "timestamp": "2019-01-10T20:26:18Z",
      "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 wanted functionality.

Compatibility With Metric Server

The dashboard has not been updated to work with the **Metrics Server** now that **Heapster** has been deprecated. While there is some interest in getting the metrics to show in the dashboard there has been difficulty finding developers to work on the issue. https://github.com/kubernetes/dashboard/issues/2986

1. Create the dashboard. The short URL in the step below, which has an capital "oh", not number zero, is for this longer URL: https://raw.githubusercontent.com/kubernetes/dashboard/v2.0.0-beta6/aio/deploy/recommended.yaml.

```
student@master:˜$ kubectl create -f https://bit.ly/2OFQRMy
```

```
secret/kubernetes-dashboard-certs created
serviceaccount/kubernetes-dashboard created
role.rbac.authorization.k8s.io/kubernetes-dashboard-minimal created
rolebinding.rbac.authorization.k8s.io/kubernetes-dashboard-minimal created
deployment.apps/kubernetes-dashboard created
service/kubernetes-dashboard created
```

2. View the current services in all namespaces. Note that the `kubernetes-dashboard` is a `ClusterIP` and part of the `kube-system` namespace.

```
student@master:˜$ kubectl get svc --all-namespaces
```

```
<output_omitted>
default                thirdpage                   NodePort    10.101.190.109   <none>        80:30588/TCP
kube-system            coredns                     ClusterIP   10.96.0.10       <none>        53/UDP,53/TCP,9153/TCP
kube-system            metrics-server              ClusterIP   10.109.37.204    <none>        443/TCP
kube-system            traefik-ingress-service     ClusterIP   10.103.146.169   <none>        80/TCP,8080/TCP
kubernetes-dashboard   dashboard-metrics-scraper   ClusterIP   10.111.17.237    <none>        8000/TCP
kubernetes-dashboard   kubernetes-dashboard        ClusterIP   10.105.169.5     <none>        443/TCP

```

3. Edit the `kubernetes-dashboard` and change the type to a `NodePort`.

```
student@master:˜$ kubectl -n kubernetes-dashboard edit svc kubernetes-dashboard
```

```yaml
....
  selector:
    k8s-app: kubernetes-dashboard
  sessionAffinity: None
  type: NodePort                    #<-- Edit this line
status:
  loadBalancer: {}
```

4. Check the `kubernetes-dashboard` service again. The Type should show as `NodePort`. Take note of the high-numbered port, which is `30968` in the example below. Yours will be different.

```
student@master:~$ kubectl -n kubernetes-dashboard get svc kubernetes-dashboard
```

```
NAME                   TYPE       CLUSTER-IP      EXTERNAL-IP   PORT(S)         AGE
kubernetes-dashboard   NodePort   10.107.224.246  <none>        443:30968/TCP   6m39s
```

5. There has been some issues with RBAC and the dashboard permissions to see objects. In order to ensure access to view various resources give the dashboard admin access.

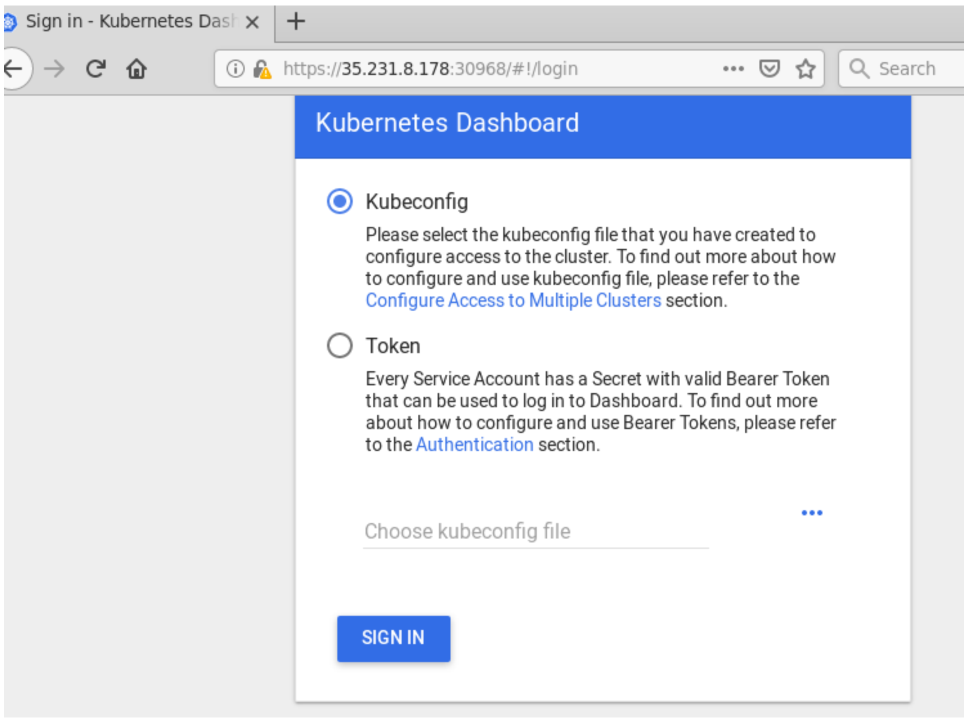
```
student@master:~$ kubectl create clusterrolebinding dashaccess \
        --clusterrole=cluster-admin \
        --serviceaccount=kubernetes-dashboard:kubernetes-dashboard
```

```
clusterrolebinding.rbac.authorization.k8s.io/dashaccess created
```

6. On your local node 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**. The page should then show the `Kubernetes Dashboard`. You may be able to find the public IP address using **curl**.

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

```
35.231.8.178
```


Sign in - Kubernetes Dash
https://35.231.8.178:30968/#!/login
Search
Kubernetes Dashboard

Kubeconfig

Please select the kubeconfig file that you have created to configure access to the cluster. To find out more about how to configure and use kubeconfig file, please refer to the Configure Access to Multiple Clusters section.

Token

Every Service Account has a Secret with valid Bearer Token that can be used to log in to Dashboard. To find out more about how to configure and use Bearer Tokens, please refer to the Authentication section.

Figure 13.1: **External Access via Browser**

7. 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@master:˜$ kubectl -n kubernetes-dashboard describe secrets kubernetes-dashboard-token-<TAB>
```

```
....
Data
====
ca.crt:      1025 bytes
namespace:   11 bytes
token:       eyJhbGciOiJSUzI1NiIsImtpZCI6IiJ9.eyJpc3MiOiJrdWJlcm5ldGVzL3NlcnZpY2VhY2NvdW50Iiwia3ViZX
JuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9uYW1lc3BhY2UiOiJrdWJlLXN5c3RlbSIsImt1YmVybmV0ZXMuaW8vc2VydmljZWFjY
291bnQvc2VjcmV0Lm5hbWUiOiJrdWJlcm5ldGVzLWRhc2hib2FyZC10b2tlbi1wbW04NCIsImt1YmVybmV0ZXMuaW8vc2Vydmlj
ZWFjY291bnQvc2VydmljZS1hY2NvdW50Lm5hbWUiOiJrdWJlcm5ldGVzLWRhc2hib2FyZCIsImt1YmVybmV0ZXMuaW8vc2Vydml
jZWFjY291bnQvc2VydmljZS1hY2NvdW50LnVpZCI6IjE5MDY4ZDIzLTE1MTctMTFlOS1hZmMyLTQyMDEwYThlMDAwMyIsInN1Yi
I6InN5c3RlbTpzZXJ2aWNlYWNjb3VudDprdWJlLXN5c3RlbTprdWJlcm5ldGVzLWRhc2hib2FyZCJ9.aYTUMWr290pjt5i32rb8
qXpq4onn3hLhvz6yLSYexgRd6NYsygVUyqnkRsFE1trg9i1ftNXKJdzkY5kQzN3AcpUTvyj_BvJgzNh3JM9p7QMjI8LHTz4TrRZ
rvwJVWitrEn4VnTQuFVcADFD_rKB9FyI_gvT_QiW5fQm24ygTIgfOYd44263oakG8sL64q7UfQNW2wt5SOorMUtybOmX4CXNUYM8
G44ejEtv9GW5OsVjEmLIGaoEMX7fctwUN_XCyPdzcCg2WOxRHahBJmbCuLz2SSWL52q4nXQmhTq_L8VDDpt6LjEqXW6LtDJZGjVC
s2MnBLerQz-ZAgsVaubbQ
```

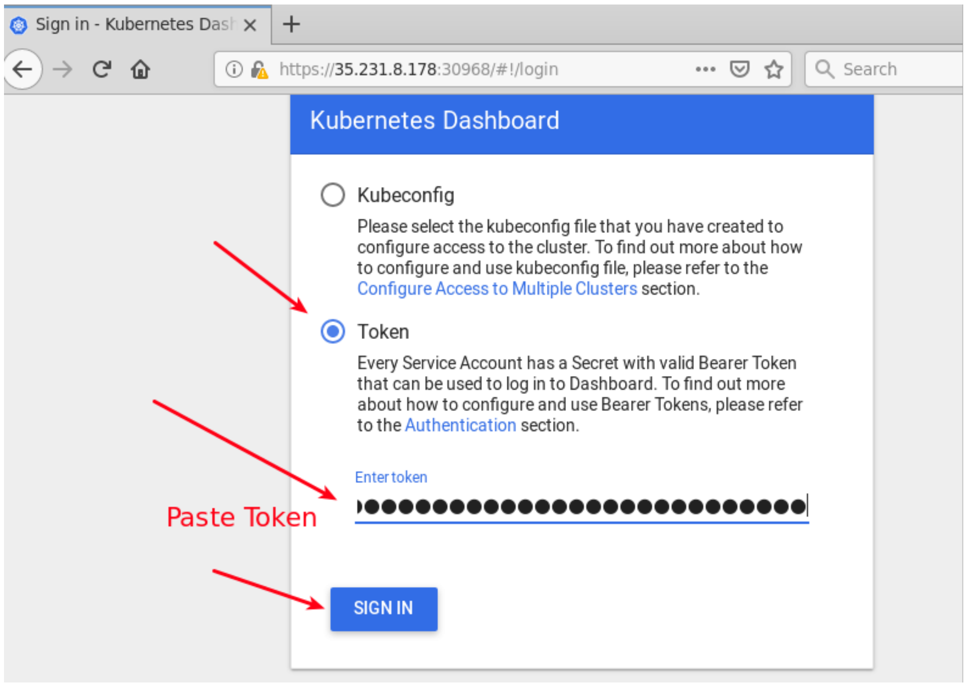
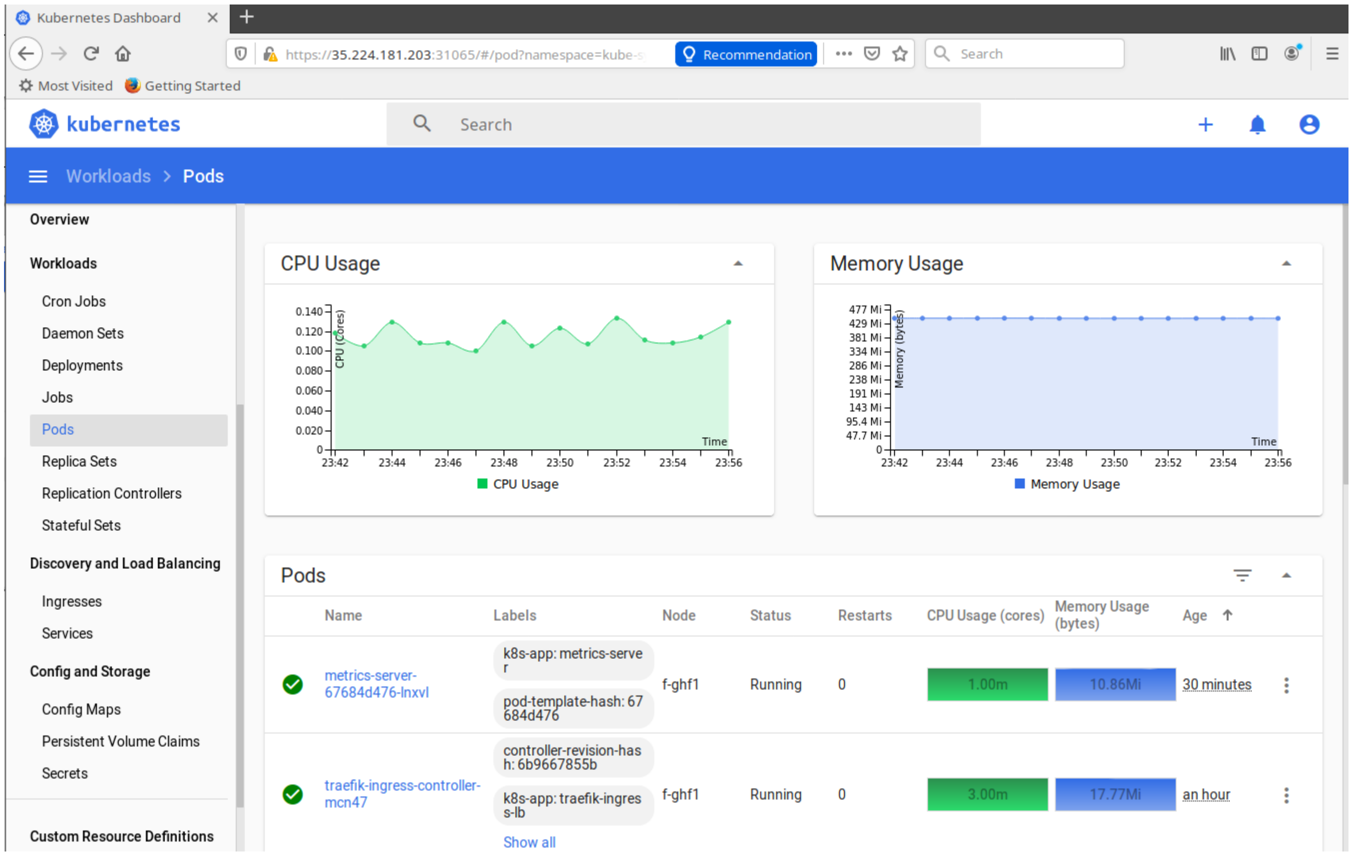

Sign in - Kubernetes Dash
https://35.231.8.178:30968/#!/login
Search
Kubernetes Dashboard
Kubeconfig
Please select the kubeconfig file that you have created to configure access to the cluster. To find out more about how to configure and use kubeconfig file, please refer to the Configure Access to Multiple Clusters section.
Token
Every Service Account has a Secret with valid Bearer Token that can be used to log in to Dashboard. To find out more about how to configure and use Bearer Tokens, please refer to the Authentication section.
Enter token
Paste Token
SIGN IN

Figure 13.2: **External Access via Browser**

8. 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.



	Name	Labels	Node	Status	Restarts	CPU Usage (cores)	Memory Usage (bytes)	Age
✓	metrics-server-67684d476-lnxvl	k8s-app: metrics-server pod-template-hash: 67684d476	f-ghf1	Running	0	1.00m	10.86Mi	30 minutes
✓	traefik-ingress-controller-mcn47	controller-revision-hash: 6b9667855b k8s-app: traefik-ingress-lb	f-ghf1	Running	0	3.00m	17.77Mi	an hour

Figure 13.3: **External Access via Browser**

Chapter 14

Helm

14.1 Overview

Deploying Complex Applications

- Package your Kubernetes application via **chart** template
- **Helm** client to request install of **chart**
- **Tiller** creates cluster resources according to **chart**
- New version 3 changes quite a bit

We have used Kubernetes tools to deploy simple **Docker** applications. Starting with the v1.4 release, the goal 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.

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.

The server runs in your Kubernetes cluster, and your client can run anywhere, even a local laptop. With your client, 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.

14.2 Helm

Helm v2 and Tiller

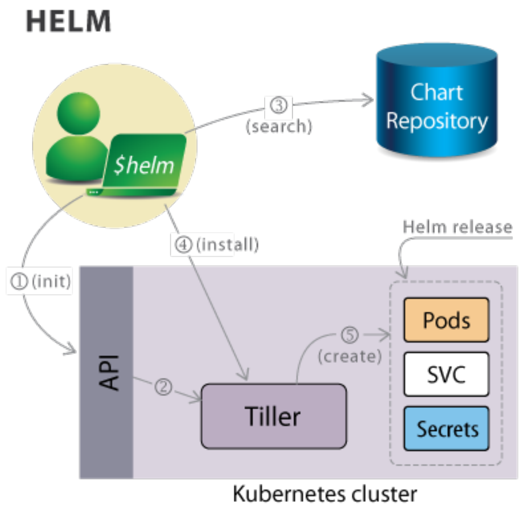


Figure 14.1: **Basic Helm and Tiller Flow**

The **helm** version 2 uses a `Tiller` pod to deploy objects in the cluster. This has led to a lot of issues with security and cluster permissions. The new version 3 does not deploy a pod.

The Helm tool packages a Kubernetes application using a series of YAML files into a `chart`, or package. This allows for simple sharing between users, tuning using a templating scheme, as well as provenance tracking, among other things. **Helm v2** is made of two components:

- A server called **Tiller**, which runs inside your Kubernetes cluster
- A client called **Helm**, which runs on your local machine.

With the `Helm` client you can browse package repositories containing published `harts`, and deploy those `charts` on your Kubernetes cluster. The **helm** will download the `chart` and pass a request to **Tiller** to create a release, otherwise known as an instance of a Chart. The release will be made of various resources running in the Kubernetes cluster.

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 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`.

14.3 Using Helm

Initializing Helm v2

```
    $ helm init

...
Tiller (the helm server side component) has been installed into your \
                                        Kubernetes Cluster.
Happy Helming!

$ kubectl get deployments --namespace=kube-system

NAMESPACE     NAME            READY  UP-TO-DATE  AVAILABLE AGE
kube-system   tiller-deploy  1/1    1           1          15s
```

Helm v3 does not need to be initialized.

As always, you can build **Helm** from source or download a tarball. We expect to see **Linux** packages for the stable release soon. The current RBAC security requirements to deploy **helm** require the creation of a new `serviceaccount` and assigning of permissions and roles. There are several optional settings which can be passed to the **helm init** command, typically for particular security concerns, storage options and also a dry-run option.

The **helm v2** initialization should have created a new `tiller-deploy` pod in your cluster. Please note that this will create a deployment in the `kube-system` namespace.

The client will be able to communicate with the **tiller** pod using port forwarding. Hence, you will not see any service exposing **tiller**.

Chart Repositories

- Search for charts
- Add a new repository
- Simple HTTP servers with index file and tarball of charts
- **helm repo**

A default repository is included when initializing `helm`, but it's common to add other repositories. Repositories are currently simple HTTP servers that contain an index file and a tarball of all the Charts present.

You can interact with a repository using the `helm repo` commands.

```
$ helm repo add testing \
http://storage.googleapis.com/kubernetes-charts-testing

$ helm repo list
```

```
NAME     URL
stable   http://storage.googleapis.com/kubernetes-charts
local    http://localhost:8879/charts
testing  http://storage.googleapis.com/kubernetes-charts...
```

Once you have a repository available, you can search for Charts based on keywords. Below, we search for a redis Chart:

```
$ helm search redis
```

```
WARNING: Deprecated index file format. Try 'helm repo update'
NAME                        VERSION   DESCRIPTION
testing/redis-cluster       0.0.5     Highly available Redis cluster with ...
testing/redis-standalone    0.0.1     Standalone Redis Master testing/...
```

Once you can find the chart within a repository you can deploy it on your cluster.

Deploying a Chart

```
$ helm install testing/redis-standalone

Fetched testing/redis-standalone to redis-standalone-0.0.1.tgz
amber-eel
Last Deployed: Fri Oct 21 12:24:01 2016
Namespace: default
Status: DEPLOYED

Resources:
==> v1/ReplicationController
NAME               DESIRED   CURRENT   READY   AGE
redis-standalone   1         1         0       1s
==> v1/Service
NAME    CLUSTER-IP   EXTERNAL-IP   PORT(S)    AGE
redis   10.0.81.67   <none>        6379/TCP   0s
```

To deploy a Chart, you can just 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

You will be able to list the release, delete it, even upgrade it and roll back.

```
$ helm list
```

```
NAME       REVISION  UPDATED                   STATUS    CHART
amber-eel  1         Fri Oct 21 12:24:01 2016  DEPLOYED  redis-stan...
```

A unique, colorful name will be created for each helm instance deployed. You can also use **kubectl** to view new resources **Helm** created in your cluster.

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.

14.4 Labs

Exercise 14.1: 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

1. On the `master` node use **wget** to download the compressed tar file. Various versions can be found here: https://github.com/helm/helm/releases/

```
student@master:˜$ wget https://get.helm.sh/helm-v3.0.0-linux-amd64.tar.gz
```

```
<output_omitted>
helm-v3.0.0-linux-a 100%[===================>]  11.52M  --.-KB/s    in 0.1s

2019-11-25 06:28:39 (112 MB/s) - ‘helm-v3.0.0-linux-amd64.tar.gz’ saved [12082866/12082866]
```

2. Uncompress and expand the file.

```
student@master:˜$ tar -xvf helm-v3.0.0-linux-amd64.tar.gz
```

```
linux-amd64/
linux-amd64/helm
linux-amd64/README.md
linux-amd64/LICENSE
```

3. Copy the **helm** binary to the `/usr/local/bin/` directory, so it is usable via the shell search path.

```
student@master:˜$ sudo cp linux-amd64/helm /usr/local/bin/helm3
```

4. 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@master:˜$ helm3 search hub database
```

```
URL                                             CHART VERSION       APP VERSION
             DESCRIPTION
https://hub.helm.sh/charts/choerodon/postgresql 3.18.4              10.7.0
             Chart for PostgreSQL, an object-relational data...
https://hub.helm.sh/charts/inspur/redis-cluster 0.0.1               5.0.6
             Highly available Kubernetes implementation of R...
https://hub.helm.sh/charts/lohmag/clickhouse    0.2.0               19.17
             ClickHouse is an open source column-oriented da...
https://hub.helm.sh/charts/halkeye/turtl        0.1.7               0.7
             The secure, collaborative notebook - Totally pr...
<output_omitted>
```

5. We will begin by adding a common repository and call it `stable`.

```
student@master:~$ helm3 repo add stable https://kubernetes-charts.storage.googleapis.com
```

```
"stable" has been added to your repositories
```

```
student@master:~$ helm3 repo update
```

```
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "stable" chart repository
Update Complete.  Happy Helming!
```

6. We will install the **mariadb**. The **- -debug** option will create a lot of output. The output will typically suggest ways to access the software. As well we will indicate that we do not want persistent storage, which would require us to create an available `PV`.

```
student@master:~$ helm3 --debug install firstdb stable/mariadb \
   --set master.persistence.enabled=false \
   --set slave.persistence.enabled=false
```

```
install.go:148: [debug] Original chart version: ""
install.go:165: [debug] CHART PATH: /home/student/.cache/helm/repository/mariadb-7.0.1.tgz

client.go:87: [debug] creating 8 resource(s)
NAME: firstdb
LAST DEPLOYED: Mon Nov 25 08:03:30 2019
<output_omitted>
```

7. Using some of the information at the end of the previous command output we will deploy another container and access the database. We begin by getting the root password for `illmannered-salamander`. Be aware the output lacks a carriage return, so the next prompt will appear on the same line. We will need the password to access the running `MariaDB` database.

```
student@master:~$ kubectl get secret --namespace default firstdb-mariadb \
   -o jsonpath="{.data.mariadb-root-password}" | base64 --decode
```

```
TQ93fCLP13
```

8. Now we will install another container to act as a client for the database. We will use **apt-get** to install client software. Using copy and paste from the installation output may be helpful.

```
student@master:~$ kubectl run firstdb-mariadb-client \
         --rm --tty -i --restart='Never' \
         --image  docker.io/bitnami/mariadb:10.3.22-debian-10-r27 \
         --namespace default --command -- bash
```

9. Use the client software to access the database. The following command uses the server name and the root password we found in a previous step. Both of yours will be different.

Inside container

```
I have no name!@firstdb-mariadb-client:/$ mysql -h firstdb-mariadb.default.svc.cluster.local \
    -uroot -p my_database
Enter password: TQ93fCLP13
```

```
Welcome to the MariaDB monitor.  Commands end with ; or \  g.
Your MariaDB connection id is 153
Server version: 10.1.38-MariaDB Source distribution

```

```
Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\  h' for help. Type '\  c' to clear the current input statement.

MariaDB [(none)]> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| my_database        |
| mysql              |
| performance_schema |
| test               |
+--------------------+
5 rows in set (0.00 sec)

MariaDB [(none)]>
MariaDB [(none)]> quit
Bye
I have no name!@firstdb-mariadb-client:/$ exit

```

10. View the Chart history on the system. The use of the **-a** option will show all Charts including deleted and failed attempts.

```
student@master:~$ helm3 list
```

```
NAME            NAMESPACE        REVISION        UPDATED
STATUS           CHART                APP VERSION
firstdb         default          1               2019-11-25 08:03:30.693410615
+0000 UTCdeployed        mariadb-7.0.1        10.3.20
```

11. Delete the **mariadb** Chart. No releases of mariadb should be found.

```
student@master:~$ helm3 uninstall firstdb
```

```
release "firstdb" uninstalled
```

```
student@master:~$ helm3 list
```

```
NAME        NAMESPACE        REVISION        UPDATED        STATUS        CHART        APP VERSION
```

12. Find the downloaded chart. It should be a compressed tarball under the user's home directory. Your **mariadb** version may be slightly different.

```
student@master:~$ find $HOME -name *mariadb*
```

```
/home/student/.cache/helm/repository/mariadb-7.3.14.tgz

```

13. Move to the archive directory and extract the tarball. Take a look at the files within.

```
student@master:~$ cd $HOME/.cache/helm/repository ; tar -xvf mariadb-*
```

```
mariadb/Chart.yaml
mariadb/values.yaml
mariadb/templates/NOTES.txt
mariadb/templates/_helpers.tpl
mariadb/templates/initialization-configmap.yaml
<output_omitted>
```

14. Copy and rename the `values.yaml` file back to the home directory and return there as well.

```
student@master:~/.cache/helm/repository$ cp mariadb/values.yaml $HOME/custom.yaml ; cd
```

15. Review the `custom.yaml` file, note there are many possible configurations. While most are commented out, take a moment to work slowly through the file.

```
student@master:~$ less custom.yaml
```

```
<output_omitted>
```

16. Edit the file. We will change or add two items, the `rootUser password` and storage `persistence` parameters.

```
student@master:~$ vim custom.yaml
```

```
....
rootUser:
  ## MariaDB admin password
  ## ref: https://github.com/bitnami/bitnami-docker-mariadb#setting-the-root-password-on-first-run
  ##
  password: LFTr@1n                             #<-- Add a password, such as LFTr@1n
  ##
....
  persistence:
    ## If true, use a Persistent Volume Claim, If false, use emptyDir
    ##
    enabled: false                              #<--- Change this to false
    # Enable persistence using an existing PVC
    # existingClaim:
....
```

17. We will now deploy another **MariaDB** instance using our custom settings. Note we no longer have to pass statements to avoid using persistent storage. Near the end of the output will be commands on using the database.

```
student@master:~$ helm3 install -f custom.yaml seconddb stable/mariadb
```

```
NAME: seconddb
LAST DEPLOYED: Mon Nov 25 08:54:08 2019
NAMESPACE: default
STATUS: deployed


<output_omitted>

To connect to your database:

  1. Run a pod that you can use as a client:

      kubectl run seconddb-mariadb-client --rm --tty -i --restart='Never' --image
docker.io/bitnami/mariadb:10.3.20-debian-9-r0 --namespace default --command -- bash

  2. To connect to master service (read/write):

<output_omited>
```

18. Copy and paste the command from the output to run a client pod. If you are re-typing, don't include the backlashes. The command could be on typed on a single line. Once in the container log into the database with the password we set and check the default databases.

```
student@master:~$ kubectl run seconddb-mariadb-client \
         --rm --tty -i --restart='Never' \
         --image  docker.io/bitnami/mariadb:10.3.22-debian-10-r27 \
         --namespace default --command -- bash
```

On Container

```
I have no name!@iron-pika-mariadb-client:/$ mysql -h seconddb-mariadb.default.svc.cluster.local \
   -uroot -p my_database
        Enter password: LFTr@1n   #<-- The password we set in the config file
```

```
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 25
Server version: 10.3.17-MariaDB-log Source distribution

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
```

```
MariaDB [my_database]> SHOW DATABASES;
```

```
+--------------------+
| Database           |
+--------------------+
| information_schema |
| my_database        |
| mysql              |
| performance_schema |
| test               |
+--------------------+
5 rows in set (0.001 sec)
```

```
MariaDB [my_database]> quit
```

```
Bye
```

```
I have no name!@seconddb-mariadb-client:/$ exit
```

19. Remove anything you have installed using **helm**. Reference earlier in the chapter if you don't remember the command.

Chapter 15

Closing and Evaluation Survey

15.1 Evaluation Survey

Thank you for taking this **Linux Foundation** course.

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.

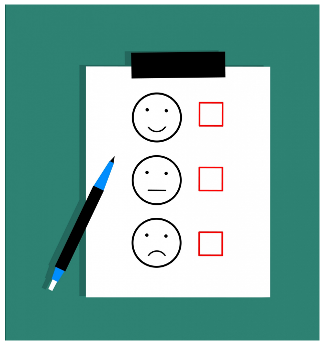


Figure 15.1: **Course Survey**