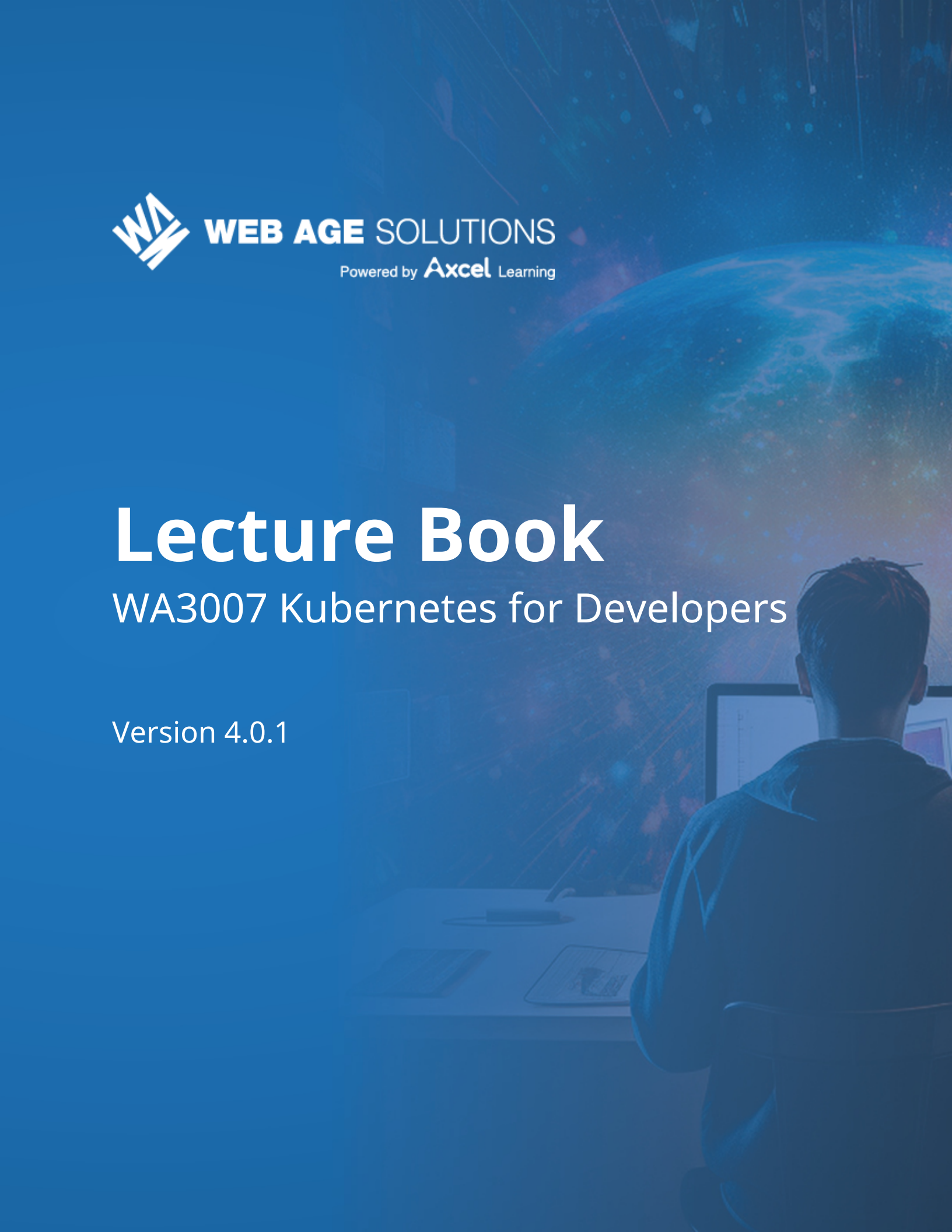


Lecture Book

WA3007 Kubernetes for Developers

Version 4.0.1

Revision 4.0.1 published on 2025-01-07.

Web Age Solutions, LLC
1 California Street Suite 2900
San Francisco, CA 94111
https://www.webagesolutions.com

USA: 1-877-517-6540, email: getinfousa@webagesolutions.com
Canada: 1-877-812-8887 toll free, email: getinfo@webagesolutions.com

Table of Contents

Chapter 1. Kubernetes Core Concepts

Objectives

Key objectives of this chapter

- What is Kubernetes
- Container Orchestration
- Architecture
- Nodes
- Master
- Pods,
- Labels,
- Controllers,
- Services,
- Storage Volumes,
- Secrets,
- Quotas
- Security
- Container Registry

1.1. Kubernetes Basics

- Kubernetes is Greek for "helmsman" or "pilot"
- Originally founded by Joe Beda, Brendan Burns and Craig McLuckie
- Kubernetes is commonly referred to as **K8s**
- An open-source system for automating deployment, scaling, and management of containerized applications
- Most often associated with Docker but supports other container types as well

1.2. What is Kubernetes?

Kubernetes:

- Is an open-source system for automating deployment, scaling, and management of containerized applications
- Groups application containers into logical units for easy management and discovery.
- Is designed to easily scale applications
- Can be implemented on-premises or in hybrid, or public cloud infrastructures
- Can be deployed on a bare-metal cluster (real machines) or on a cluster of virtual machines.

1.3. Container Orchestration

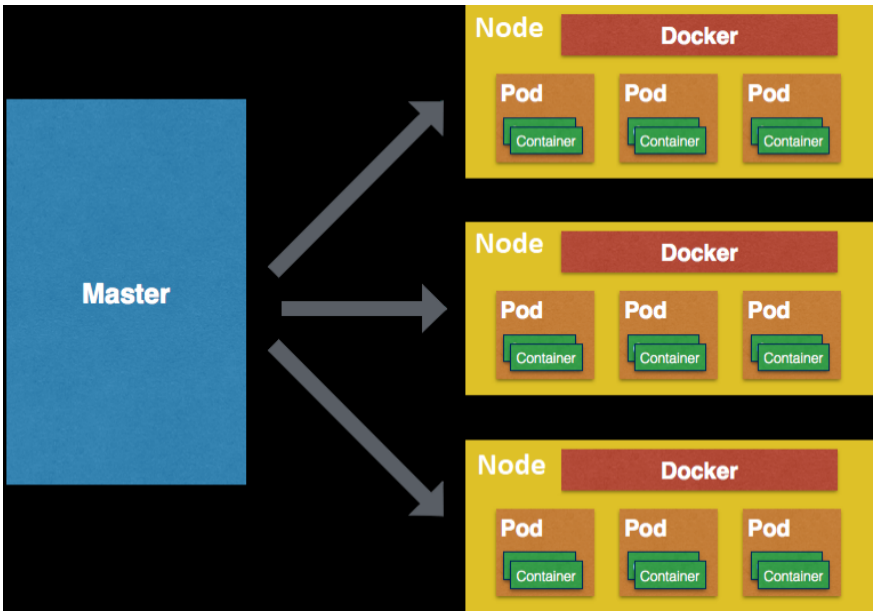
- Container orchestration is the primary responsibility of Kubernetes

- Containers involved in workflows are scheduled to run on physical or virtual machines
- The health of running containers is monitored so that dead, unresponsive, or otherwise healthy containers can be replaced.
- Supports application and container clustering
- Is particularly useful in implementing microservices

1.4. Kubernetes Architecture

Kubernetes is based on three basic concepts:

- Pods
- Master
- Nodes



1.5. Kubernetes Concepts

- Cluster and Namespace
- Node
- Master
- Pod
- Label
- Annotation
- Label Selector
- Replication Controller and replica set
- Services
- Volume
- Secret

1.6. Cluster and Namespace

- Cluster
    - A collection of physical resources, such as hosts storage and networking resources
    - The entire system may consist of multiple clusters
- Namespace
    - It is a virtual cluster
    - A single physical cluster can contain multiple virtual clusters segregated by namespaces

- Virtual clusters can communicate through public interfaces
- Pods can live in a namespace, but nodes can not.
- Kubernetes can schedule pods from different namespaces to run on the same node

1.7. Nodes

Nodes:

- Were formerly referred to as a 'minions'
- Are single Host computers (physical or virtual machines)
- Are used to run **pods**
- Are managed by a **Kubernetes Master**
- Are where work is executed

Each node runs several Kubernetes components, such as a kubelet and a kube proxy. kubelet is a service which reads container manifests as YAML files that describes a pod.

1.8. Master

- The master is the control plane of Kubernetes
- It consists of components, such as
  - API server
  - a scheduler

    - a controller manager
- The master is responsible for the global, cluster-level scheduling of pods and handling events.
- Often, all the master components are set up on a single host
- For implementing high-availability scenarios or very large clusters, you will want to have master redundancy.

1.9. Pod

- A pod is the unit of work on Kubernetes
- Pods provide a way to group one or more containers
- Pods provide a solution for managing containers that depend on each other and need to cooperate on the same host
- Pods are considered throwaway entities that can be discarded and replaced as needed
- Each pod gets a unique ID (UID)
- Pods are always scheduled together and always run on the same machine
- All the containers in a pod have the same IP address and port space
- The containers within a pod can communicate using localhost or standard inter-process communication
- Containers within a pod have access to the shared local storage on the node hosting the pod

1.10. Using Pods to Group Containers

- The benefits of grouping related containers within a pod, as opposed to having one container with multiple applications, are:
    - **Transparency** – making the containers within the pod visible to the infrastructure enables the infrastructure to provide services to those containers, such as process management and resource monitoring
    - **Decoupling** software dependencies – the individual containers maybe be versioned, rebuilt, and redeployed independently
    - **Ease of use** – users don’t need to run their own process managers
    - **Efficiency** – because the infrastructure takes on more responsibility, containers can be more lightweight

1.11. Label

- Labels are key-value pairs that are used to group together sets of objects, often pods.
- Labels are important for several other concepts, such as replication controller, replica sets, and services that need to identify the members of the group
- Each pod can have multiple labels, and each label may be assigned to different pods.
- Each label on a pod must have a unique key

1.12. Label Syntax

The label key must adhere to a strict syntax

- Label has two parts: prefix and name
- Prefix is optional. If it exists then it is separated from the name by a forward slash (/) and it must be a valid DNS sub-domain. The prefix must be 253 characters long at most
- Name is mandatory and must be 63 characters long at most. Name must begin with an alphanumeric character and contain only alphanumeric characters, dots, dashes, and underscores. You can create another object with the same name as the deleted object, but the UIDs must be unique across the lifetime of the cluster. UIDs are generated by Kubernetes
- Values follow the same restrictions as names

1.13. Annotation

- Unlike labels, annotation can be used to associate arbitrary metadata with Kubernetes objects.
- Kubernetes stores the annotations and makes their metadata available
- Unlike labels, annotations don’t have strict restrictions about allowed characters and size limits

1.14. Label Selector

- Label selectors are used to group objects based on their labels
- A value can be assigned to a key name using equality-based selectors, (=, ==, !=).
    - e.g.
        - role = webserver
        - role = dbserver, application != sales
- **in** operator can be used as a set-based selector
    - .e.g
        - role in (dbserver, backend, webserver)

1.15. Replication Controller and Replica Set

- They both manage a group of pods identified by a label selector
- They ensure that a certain number of pods are always up and running
- Whenever the number drops due to a problem with the hosting node or the pod itself, Kubernetes fires up new instances
- If you manually start pods and exceed the specified number, the replication controller kills some extra pods
- Replication controllers test for membership by name equality, whereas replica sets can use set-based selection
- Replica sets are newer and considered as next-gen replication controllers

1.16. Service

- Services are used to expose some functionality to users or other services
- They usually involve a group of pods, usually identified by a label
- Kubernetes services are exposed through endpoints (TCP/UDP)
- Services are published or discovered via DNS, or environment variables
- Services can be load-balanced by Kubernetes

1.17. Storage Volume

- When a pod is destroyed, the data used by the pod is also destroyed.
- If you want the data to outlive the pod or share data between pods, volume concept can be utilized.

1.18. Secret

- Secrets are small objects that contain sensitive info, such as credentials
- They are stored as plain-text in an etcd distributed key store
- They can be mounted as files into pods
- The same secret can be mounted into multiple pods
- Internally, Kubernetes creates secrets for its components, and you

can create your own secrets

1.19. Resource Quota

- Kubernetes allows management of different types of quota
- Compute resource quota
  - Compute resources are CPU and memory
  - You can specify a limit or request a certain amount
  - Uses fields, such as requests.cpu, requests. memory
- Storage resource quota
  - You can specify the amount of storage and the number of persistent volumes
  - Uses fields, such as requests.storage, persistentvolumeclaims
- Object count quota
  - You can control API objects, such as replication controllers, pods, services, and secrets
  - You can not limit API objects, such as replica sets and namespaces.

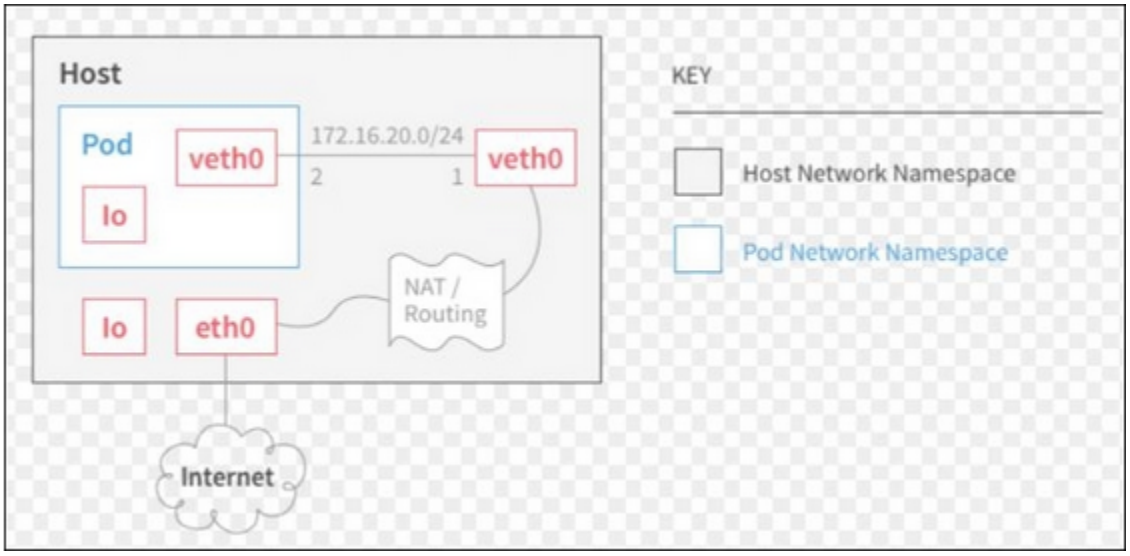
1.20. Authentication and Authorization

- Permission rules can be added to the Kubernetes system for more advanced management
- Applying authentication and authorization is a secure solution to prevent your data being accessed by others.

- Authentication is currently supported in the form of tokens, passwords, and certificates.
- Authorization supports three modes:
  - RBAC (Role-Based Access Control)
  - ABAC (Attribute-Based Access Control) – lets a user define privileges via attributes in a file
  - Webhook – allows for integration with third-party authorization via REST web service calls.

1.21. Routing

- Routing connects separate networks
- Routing is based on routing tables
- Routing table instructs network devices how to forward packets to their destination
- Routing is done through various network devices, such as routers, bridges, gateways, switches, and firewalls



1.22. Docker Registry

- Docker Registry is a stateless, highly scalable server-side application that stores and lets you distribute Docker images.
- Docker Registry is open-source
- Docker Registry gives you following benefits
    - tight control where your images are being stored
    - fully own your images distribution pipeline
    - integrate image storage and distribution tightly into your in-house development workflow

1.23. Summary

In this chapter we covered:

- What is Kubernetes
- Container Orchestration
- Architecture
- Nodes
- Master
- Pods,
- Labels,
- Controllers,
- Services,
- Storage Volumes,

- Secrets,
- Quotas
- Security
- Container Registry

Chapter 2. Kubernetes Architecture

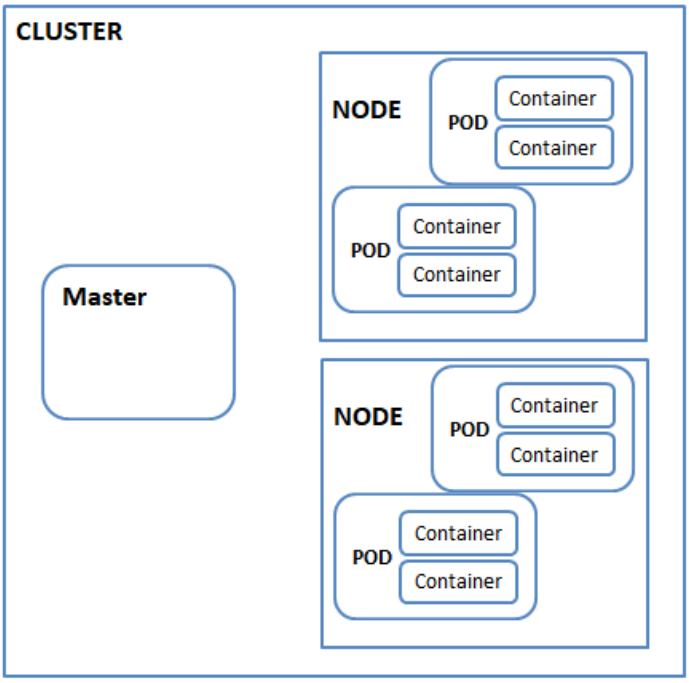
Objectives

Key objectives of this chapter

- Architecture Diagram
- Components
- Cluster
- Master
- Node
- Pod
- Container
- Interaction through API

2.1. Architecture Diagram

This chapter will review various parts of the following architecture diagram:



2.2. Components

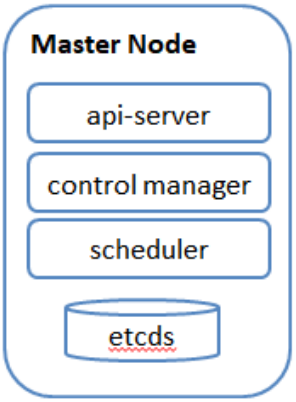
- **Cluster** - Includes one or more master and worker nodes
- **Master** - Manages nodes and pods
- (worker) **Node** - a physical, virtual or cloud machine
- **Pod** - A group of one or more containers, created and managed by kubernetes
- **Container** - Are most commonly Docker containers where application processes are run
- **Volume** - A directory of data accessible to containers in a pod. It shares lifetime with the pod it works with.
- **Namespace** - A virtual cluster. Allows for multiple virtual clusters within a physical one.

2.3. Kubernetes Cluster

- A Kubernetes cluster is a set of machines(nodes) used to run containerized applications.
- To do work a clusters need to have at least a one master node and one worker node.
- The Master node determines where and what is run on the cluster.
- Worker nodes contain pods which contain containers. Containers hold execution environments where work can be done.
- A cluster is configured via the kubectl command line interface or by the Kubernetes API

2.4. Master Node

- The Master node manages worker nodes.
- The master node includes several components:
  - Kube-APIServer - traffic enters the cluster here
  - Kube-Controller-Manager - runs the cluster's controllers
  - Etcd - Maintains cluster state, provides key-value persistence
  - Kube Scheduler - schedules activities to worker nodes
- Clusters can have **more than one** master node
- Clusters can have **only one active** master node



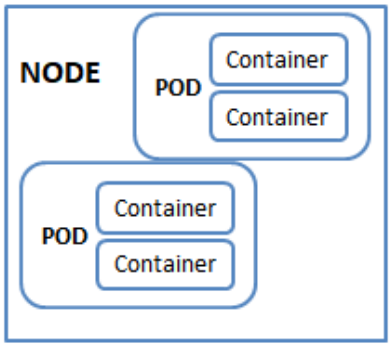
2.5. Kube-Control-Manager

- The Kube-Control-Manager (part of the Master Node) manages the following controllers:
  - Node controller
  - Replication controller
  - Endpoints controller
  - Service account controller
  - Token controller

- All these controller operations are compiled into a single application.
- The controllers are responsible for the configuration and health of the cluster's components

2.6. Nodes

- A node consists of a physical, virtual or cloud machine where Kubernetes can run Pods that house containers.
- Clusters have one or more nodes
- Nodes can be configured manually through kubectl
- Nodes can also self-configure by sending their information to the Master when they start up
- Information about running nodes can be viewed with kubectl



2.6.1. Notes

Other components found on the worker node include:

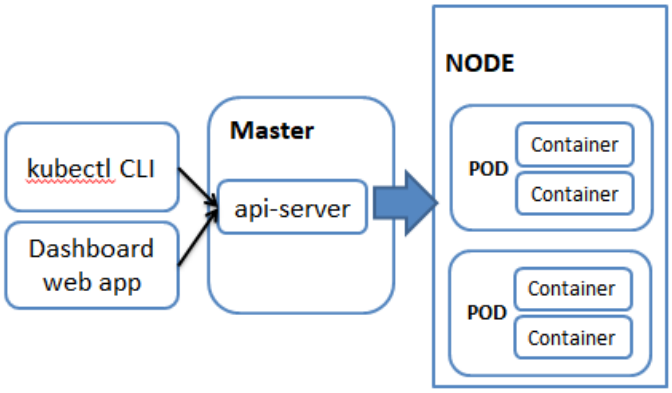
- kubelet - interacts with master node, manages containers and pods on the node
- kube-proxy - responsible for network configuration

- container runtime - responsible for running containers in the pods (typically Docker)

2.7. Other Components

- Pods - Logical container for runtime containers
- Containers - Pods typically contain Docker runtime containers holding OS images and applications. Work is run in containers.

2.8. Interacting with Kubernetes



- All user interaction goes through the master node's api-server
- kubectl provides a command line interface to the API
- Control of Kubernetes can also be done through the Kubernetes Dashboard (web UI)

2.9. Summary

In this chapter we covered:

- Architecture Diagram
- Components
- Cluster
- Master
- Node
- Pod
- Container
- Interaction through API

Chapter 3. Build

Objectives

Key objectives of this chapter

- Containerizing an Application
- Creating the Dockerfile
- Hosting a Local Registry
- Creating a Deployment
- Running Commands in a Container
- Multi-Container Pod

3.1. What is Docker

- Docker is an open-source (and 100% free) project for IT automation
- You can view Docker as a system or a platform for creating virtual environments which are extremely lightweight virtual machines
- Docker allows developers and system administrators to quickly assemble, test, and deploy applications and their dependencies inside Linux containers supporting the multi-tenancy deployment model on a single host

- Docker's lightweight containers lend themselves to rapid scaling up and down
    - NOTE: A container is a group of controlled processes associated with a separate tenant executed in isolation from other tenants
- Written in the Go programming language

> The Go programming language (also referred to as *golang*) was developed at Google in 2007 and release in 2009. It is a compiled language – it does not require a VM to run it (like in C# or Java) – with automated garbage collection. Go offers a balance between type safety and dynamic type capabilities; it supports imperative and concurrent programming paradigms.

3.2. Where Can I Run Docker?

- Docker runs on any modern-kernel 64-bit Linux distributions
- The minimum supported kernel version is 3.10
    - Kernels older than 3.10 lack some of the features required by Docker containers
- You can install Docker on VirtualBox and run it on OS X or Windows
- Docker can be installed natively on Windows using Docker Machine but requires Hyper-V/WSL2
- Docker can be booted from the small footprint Linux distribution *boot2docker*

3.3. Docker and Containerization on Linux

- Docker leverages resource isolation features of the modern Linux kernel offered by cgroups and kernel namespaces
    - The cgroups and kernel namespaces features allow the creation of strongly isolated containers acting as very lightweight virtual machines running on a single Linux host
- Docker helps abstract operating-system-level virtualization on Linux using abstracted virtualization interfaces based on *libvirt*, *LXC*(**L**inu**X** **C**ontainers) and systemd-nspawn
    - As of version 0.9, Docker can directly use virtualization facilities provided by the Linux kernel via its *libcontainer* library

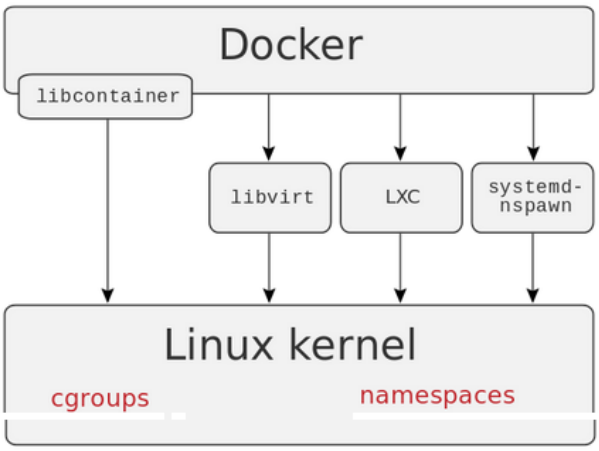
3.4. Linux Kernel Features: cgroups and namespaces

- The control group kernel feature (cgroup) is used by the Linux kernel to allocate system resources such as CPU, I/O, memory, and network subject to limits, quotas, prioritization, and other control arrangements
- The kernel provides access to multiple subsystems through the *cgroup* interface
    - Examples of subsystems (controllers):
        - The memory controller for limiting memory use
        - The *cpuacct* controller for keeping track of CPU usage
- The cgroups facility was merged into the Linux kernel version

2.6.24

- Systems that use cgroups: Docker, Linux Containers (LXC), Hadoop, etc.
- The namespaces feature is related to cgroups facility that enables different applications to act as separate tenants with completely isolated views of the operating environment, including users, process trees, network, and mounted file systems

3.5. The Docker-Linux Kernel Interfaces



Source: Adapted from http://en.wikipedia.org/wiki/Docker_(software)

3.6. Containerizing an Application

- To containerize an application, you are required to build a Docker image.
- A Docker image contains your application and all the dependencies required by the application.

    - For example, jar/war/dll files along with JRE/dotnet core.
- A manual process to build a docker image would involve the following steps:
    - Download a base image
    - Create a container based on the downloaded image.
    - Customize the container, e.g. install runtimes and copy the deployable artifacts.
    - Stop the container
    - Commit the container as a custom Docker image.
- Instead of performing the above steps manually, the best practice is to create a Dockerfile script that should do all these things.

3.7. Building a Docker Images using Dockerfile

- Docker can build images automatically by reading the instructions from a Dockerfile.
- A Dockerfile is a text document that contains all the commands a user could call on the command-line to assemble an image.
    - Since it's a simple text file, you can commit it to your source control repository, such as Git.
- Using docker build users can create an automated build that executes several command-line instructions in succession.
- Dockerfile has the following format:

```
# Comment
```

```
INSTRUCTION arguments
```

3.8. Sample Dockerfile

```
FROM openjdk:8
RUN mkdir -p /opt/my/service
ADD target/myservice-0.0.1-SNAPSHOT.jar /opt/my/service/
EXPOSE 8080
CMD ["java", "-jar", "/opt/my/service/myservice-0.0.1-SNAPSHOT.jar"]
```

The above Dockerfile does following

- Uses openjdk:8 image from the Docker hub
- Creates a directory for storing custom service files
- Copies myservice*.jar file into the container
- Makes port 8080 so it can be accessed from outside the container
- Executes the custom service by using java command-line tool.

3.9. Environment Variables

- Environment variables are declared with the ENV statement.
- Environment variables are notated in the Dockerfile either with $variable_name or ${variable_name}.
- The ${variable_name} syntax also supports a few of the standard bash modifiers as specified below:
    - ${variable:-word} indicates that if the variable is set then the

result will be that value. If the variable is not set then word will be the result.

- ${variable:+word} indicates that if the variable is set then word will be the result, otherwise the result is the empty string.

- Environment variables are supported by the following list of instructions in the Dockerfile:
  - ADD, COPY, ENV, EXPOSE, FROM, LABEL, USER, USER, VOLUME, WORKDIR

3.10. Environment Variables - Example

```
FROM busybox
ENV foo /bar
WORKDIR ${foo}      # WORKDIR /bar
ADD . $foo          # ADD . /bar
COPY \$foo /quux    # COPY $foo /quux
```

3.11. Arguments

- The ARG instruction defines a variable that users can pass at build-time to the builder with the docker build command using the --build-arg <varname>=<value> flag.
- ARG is useful when you need customization at build-time.
- ENV is useful when you need run-time customization, i.e. you want to run the same image with different settings.
- If a user specifies a build argument that was not defined in the

Dockerfile, the build outputs a warning.

- A Dockerfile may include one or more ARG instructions.
- For example:

```
FROM <image>:<tag>
ARG user_name
ARG volume_location=/tmp # this argument has a default value
```

- You build the above Dockerfile by calling:
- `docker build --build-arg user_name=bob .`

3.12. Multi-stage Builds

- A multi-stage build involves multiple FROM instructions.
- The output of each stage can be used by the subsequent stages.
- It makes the image size small.
- Here's a sample multi-stage Dockerfile:

```
FROM gradle:jdk10 _AS_ builder
COPY --chown=gradle:gradle . /app
WORKDIR /app
RUN gradle bootJar

FROM openjdk:8-jdk-alpine
EXPOSE 8080
VOLUME /tmp
ARG LIBS=app/build/libs
COPY --from=builder ${LIBS}/ /app/lib
ENTRYPOINT ["java","-jar","./app/lib/spring-boot-jpa-0.0.1-
```

```
SNAPSHOT.jar"]
```

3.13. Multi-stage Builds (Contd.)

- By default, the stages are not named and you refer to them by their integer number, starting with 0 for the first FROM instruction.
- For example:

```
COPY --from=0 /app/compiled-code .
```

- You can name your stages, by adding an AS <NAME> to the FROM instruction.
- For example:

```
FROM <image>:<tag> AS build-env
...
COPY --from=build-env /app/compiled-code .
```

- Naming the stages is better since even if the instructions in your Dockerfile are re-ordered later, the COPY doesn't break.

3.14. Stop at a Specific Build Stage

- When you build your image, you don't necessarily need to build the entire Dockerfile including every stage.
- You can specify a target build stage.

- The following command assumes you are using the previous Dockerfile but stops at the stage named builder:

  ```
  $ docker build --target builder -t myimage:latest .
  ```

- A few scenarios where this might be very powerful are:
  - Debugging a specific build stage
  - Using a debug stage with all debugging symbols or tools enabled, and a lean production stage
  - Using a testing stage in which your app gets populated with test data, but building for production using a different stage which uses real data

3.15. RUN

- The RUN command has 2 forms:
  - RUN <command>
  - RUN [“executable”, “param1”, “param2”]
- The RUN instruction will execute any commands in a new layer on top of the current image and commit the results.
- The resulting committed image will be used for the next step in the Dockerfile.
- Use the RUN instruction to update your package repository, install, and configure the software.

3.16. EXPOSE

- The EXPOSE instruction informs Docker that the container listens on the specified network ports at runtime.
- You can specify whether the port listens on TCP or UDP, and the default is TCP if the protocol is not specified.
- The EXPOSE instruction does not publish the port.
    - By default, it is used as a documentation mechanism.
    - Using -P (capital) will automatically create port mapping rules for you.
- It functions as a type of documentation between the person who builds the image and the person who runs the container, about which ports are intended to be published.
- E.g.

```
EXPOSE 80 # is the same as EXPOSE 80/tcp
EXPOSE 80/tcp # is the same as EXPOSE 80
EXPOSE 80/udp # use udp protocol
```

- You can also use -p or --ports argument to explicitly map a single port or range of ports.

  `docker run -p 80:80 my_image service nginx start`

- All published (-p or -P) ports are exposed, but not all exposed (EXPOSE) ports are exposed.

3.17. EXPOSE (Contd.)

- To publish the port when running the container, use the -p flag on docker run to publish and map one or more ports
- Regardless of the EXPOSE settings, you can override them at runtime by using the -p flag. For example

  `docker run -p 80:80/tcp -p 80:80/udp`

3.18. COPY

- The COPY instruction has two forms:
  - COPY [--chown=<user>:<group>] <src>... <dest>
  - COPY [--chown=<user>:<group>] ["<src>",... "<dest>"] # this form is required for paths containing whitespace
- The COPY instruction copies new files or directories from <src> and adds them to the filesystem of the container at the path <dest>
- For example:

```
# adds all files starting with "hom"
ADD hom* /mydir/

# ? is replaced with any single character,
e.g.,"home.txt"
ADD hom?.txt /mydir/
```

3.19. ADD

- The ADD instruction has two forms:
  - ADD [--chown=<user>:<group>] <src>... <dest>
  - ADD [--chown=<user>:<group>] [“<src>”,... “<dest>”] # this form is required for paths containing whitespace
- The ADD instruction copies new files, directories, or remote file URLs from <src> and adds them to the filesystem of the image at the path <dest>
- For example:

```
# adds all files starting with "hom"
ADD hom* /mydir/

# ? is replaced with any single character, e.g., "home.txt"
ADD hom?.txt /mydir/

ADD http://example.com/foobar /

ADD --chown=bin files* /somedir/
```

3.20. CMD

- CMD instruction has 3 forms:
  - CMD [“executable”, “param1”, “param2”] (this is the preferred form)
  - CMD [“param1”, “param2”] - as default parameters to ENTRYPOINT

    - CMD command param1 param2 - shell form
- Only one CMD instruction is allowed in a Dockerfile
- If there is more than one CMD, only the last CMD will take effect.

3.21. ENTRYPOINT

- ENTRYPOINT instruction has two forms:
    - ENTRYPOINT ["executable", "param1", "param2"] (exec form, preferred)
    - ENTRYPOINT command param1 param2 (shell form)
- An ENTRYPOINT allows you to configure a container that will run as an executable.
- If this is the ENTRYPOINT defined in your Dockerfile:

```
ENTRYPOINT ["java","-jar","./app/lib/spring-boot-app.jar"]`
```

- ...then the following will start your Java application with its default content, listening on port 80:

```
docker run -it --rm -p 8080:8080 my_image
```

- You can override the ENTRYPOINT instruction by using the following command:

```
docker run --entrypoint <cmd>
```

3.22. CMD vs. ENTRYPOINT

- Both CMD and ENTRYPOINT instructions define what command gets executed when running a container.
- There are a few rules that describe their co-operation.
    - Dockerfile should specify at least one of CMD or ENTRYPOINT commands.
    - ENTRYPOINT should be defined when using the container as an executable.
    - CMD should be used as a way of defining default arguments for an ENTRYPOINT command or for executing an ad-hoc command in a container.

3.23. VOLUME

- The VOLUME instruction has various forms:
    - VOLUME [“/data”]
    - VOLUME /data
- The VOLUME instruction creates a mount point with the specified name.
- The docker run command initializes the newly created volume with any data that exists at the specified location within the base image.
- For example, consider the following Dockerfile snippet:

```
FROM ubuntu
```

```
RUN mkdir /myvol
RUN echo "hello world" > /myvol/greeting
VOLUME /myvol
```

- This Dockerfile results in an image that causes *docker run* to create a new mount point at */myvol* and copy the *greeting* file into the newly created volume.

3.24. Build the Image

- Once you have the Dockerfile created, you can build a custom image by running the following command:

```
# specify a repository and tag
docker build -t abcinc/myservice .
```

- OR

```
docker build .
```

- OR

```
docker build -f __/__path/to/a/Dockerfile .
```

3.25. Build the Image (contd.)

The build is run by the Docker daemon.

- The build process sends the entire context recursively to the daemon.
- It’s best to start with an empty directory as context and keep your Dockerfile in that directory. Add only the files needed for building the Dockerfile.
- Do not use your root directory, /, as the PATH as it causes the build to transfer the entire contents of your hard drive to the Docker daemon.
- To increase the build’s performance, exclude files and directories by adding a .dockerignore file to the context directory.
- Viewing image list

```
docker images
```

3.26. .dockerignore

- Before the Docker CLI sends the context to the Docker daemon, it looks for a file named .dockerignore in the root directory of the context.
- If this file exists, the CLI modifies the context to exclude files and directories that match patterns in it.
- Sample rules:
  - Comments are ignored, e.g. # this is a comment
  - /temp - Exclude files and directories whose names start with temp in any immediate subdirectory of the root.

- /*/temp - Exclude files and directories starting with temp from any subdirectory that is two levels below the root.
- temp? - Exclude files and directories in the root directory whose names are a one-character extension of temp, e.g. /tempa and /tempb

3.27. Dockerfile – Best Practices

- Set version numbers
  - By default, "latest" is used. e.g. "FROM openjdk" will download the latest version of OpenJDK image.
  - "FROM openjdk:8" downloads version 8. It makes it repeatable. You generally want your containers to be the same every time. If "latest" is used then you have no control over what’s going to download and execute next time.
- Set a user
  - If you don’t specify a user in your Dockerfile your container will run as the default user – root.
  - User for a container can be specified in the Dockerfile by using this syntax "USER <user>:[<group>]"
  - e.g.
    - RUN useradd -ms /bin/bash bob
    - USER bob
    - WORKDIR /home/bob

3.28. Dockerfile - Best Practices (contd.)

- Verify downloads
  - When using the FROM command to download the images, you should verify you have downloaded the right image which hasn’t been tampered. It’s done by using a hash, or digest.
  - FROM debian@sha256:nnnnnnnnnnnnnnnn

3.29. Published Ports

- By default, when you create a container, it does not publish any of its ports to the outside world.
- To make a port available to services outside of Docker, or to Docker containers which are not connected to the container’s network, use the --publish or -p flag.
- This creates a firewall rule which maps a container port to a port on the Docker host.
- Here are some examples.

-p 8080:80	Map TCP port 80 in the container to port 8080 on the Docker host.
-p 192.168.1.100:8080:80	Map TCP port 80 in the container to port 8080 on the Docker host for connections to host IP 192.168.1.100.
-p 8080:80/udp	Map UDP port 80 in the container to port 8080 on the Docker host.

-p 8080:80/tcp -p 8080:80/udp	Map TCP port 80 in the container to TCP port 8080 on the Docker host, and map UDP port 80 in the container to UDP port 8080 on the Docker host.

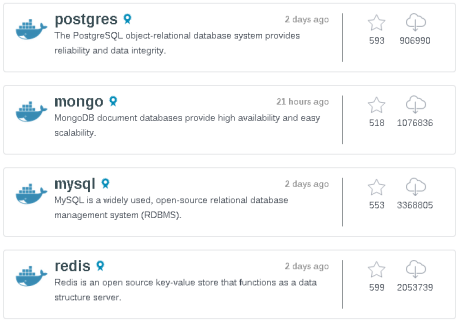
3.30. Docker Documentation

The official Docker documentation can be access by following this link:

- https://docs.docker.com/

3.31. Docker Registry

- By default, Docker pull/push commands access the Central Registry at index.docker.io (or, https://hub.docker.com)



- It is possible to create a local registry to make the images private.

3.32. Hosting a Local Registry

- Run a local registry.

```
$ docker run -d -p 5000:5000 --name registry registry:2
```

- Pull the ubuntu:16.04 image from Docker Hub

```
$ docker pull ubuntu:16.04
```

- Tag the image as localhost:5000/my-ubuntu

```
$ docker tag ubuntu:16.04 localhost:5000/my-ubuntu
```

- Push the image to the local registry running at localhost:5000

```
$ docker push localhost:5000/my-ubuntu
```

3.33. Hosting a Local Registry (contd.)

- Remove the locally-cached images, so that you can test pulling the image from your registry. (Note: This does not remove the localhost:5000/my-ubuntu image from your registry.)

```
$ docker image remove ubuntu:16.04
$ docker image remove localhost:5000/my-ubuntu
```

- Pull the localhost:5000/my-ubuntu image from your local registry.

`$ docker pull localhost:5000/my-ubuntu`

- Stop a local registry

```
$ docker container stop registry
```

- To remove the container, use docker container rm.

```
$ docker container stop registry && docker container rm -v
registry
```

A production-ready registry must be protected by TLS and should ideally use an access-control mechanism. The details are available at https://docs.docker.com/registry/configuration/

3.34. Deploying to Kubernetes

- **Method 1:** Using **Generators** (kubectl run, kubectl create (without .yaml), kubectl expose)
    - If you want to check quickly whether the container is working then you might use **Generators**.

```
# creates simple pods that you cannot scale after deployment.
kubectl run nginx-deployment --image=nginx

# creates a deployment that can be scaled after the
deployment has been performed
kubectl create deployment nginx-deployment --image=nginx

kubectl expose deployment nginx-deployment --port=80 --target
```

```
-port=8080
```

3.35. Deploying to Kubernetes (contd.)

- **Method 2:** Using **Imperative** way (kubectl create with .yaml)
  - can work with one object configuration file at a time that can be checked-in to a source control repository, such as Git.
  - It’s a one-time operation, i.e. you cannot run the same configuration file to make changes.

```
kubectl create -f deployment.yaml
```

3.36. Deploying to Kubernetes (contd.)

- **Method 3:** Using **Declarative** way (kubectl apply)
  - works with files, directories, and sub-directories containing object configuration YAML files.
  - you can modify the configuration file(s) and rerun them to update the deployment(s).

```
kubectl apply -f deployment.yaml
kubectl apply -f folder_name_with_a_bunch_of_yaml_files/
```

3.37. Running Commands in a Container

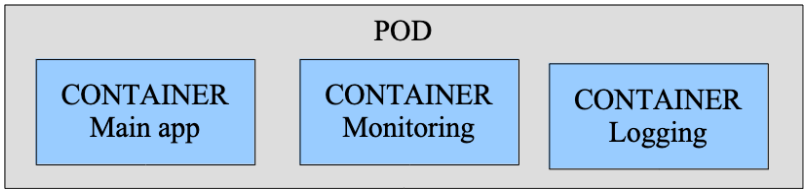
- You can run commands in a container for various reasons, such as inspect environment variables in a container and view logs.
- Running commands in a container using Docker

```
docker exec -it <container_name_or_id> <command_to_execute>
```

- Running commands in a container using Kubernetes

```
kubectl exec -it <pod_name> <command_to_execute>
```

3.38. Multi-Container Pod



- In a multi-container pod, containers run on a “logical host” (the same IP address and port space).
- Containers in a pod can also use shared volumes.
- These properties make it possible for these containers to efficiently communicate, ensuring data locality.
- Pods enable you to manage several tightly coupled application containers as a single unit.

3.39. Multi-Container Pod (contd.)

- The main purpose of a multi-container Pod is to support co-located, co-managed helper processes for a primary application.
- For example, the sidecar pattern uses containers that assist the main container. Some examples include log or data change watchers, monitoring adapters, and so on. The Sidecar and other patterns will be covered in more detail in a different chapter.
- Typically you will want to have one container per pod
    - For example, if you have a multi-tier application (such as WordPress with MySQL and Apache Web Server+PHP), the recommended way is to use separate Pods for each tier so you can scale tiers up independently and distribute them across cluster nodes.

3.40. Summary

In this chapter, you learned about the following:

- Containerizing an Application
- Creating the Dockerfile
- Hosting a Local Registry
- Creating a Deployment
- Running Commands in a Container
- Multi-Container Pod

Chapter 4. Design

Objectives

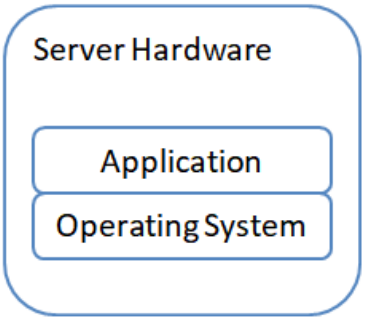
Key objectives of this chapter

- Traditional Applications
- Decoupled Resources
- Transience
- Flexible Framework
- Managing Resource Usage
- Using Label Selectors
- Multi-Container Pods
- Sidecar Container
- Adapter Container

4.1. Traditional Applications

Traditionally applications ran on dedicated hardware which has some drawbacks:

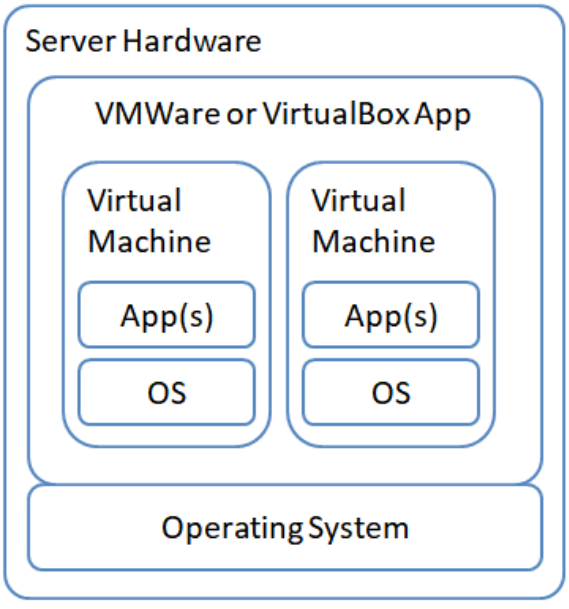
- Applications are not isolated from one another
- Server environments have to be manually and individually installed
- Hardware had to be dedicated to specific uses



Traditional/Bare-Metal

4.2. Virtual Machines

- Virtual machines allow:
  - Running on different OSs
  - Fast creation of server environments
  - Easier scaling to generic server hardware
  - Isolation of applications from one another
- One downside to virtual machines is that they use up a lot of RAM and CPU

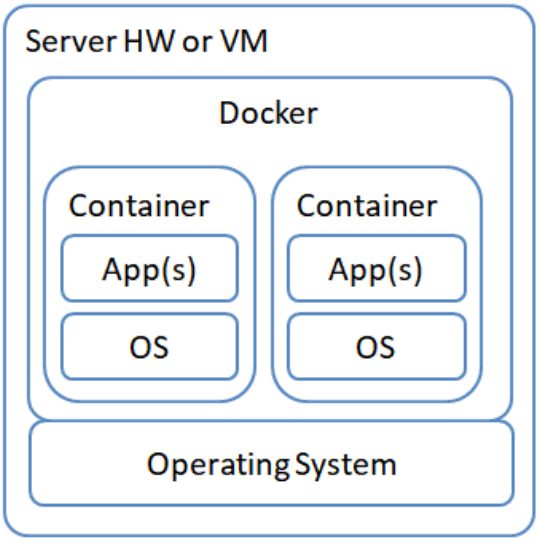


Virtual Machine(s)

4.3. Containerized Applications

Containerized applications have some of the same advantages of Virtual Machines with even more flexibility and much less resource usage.

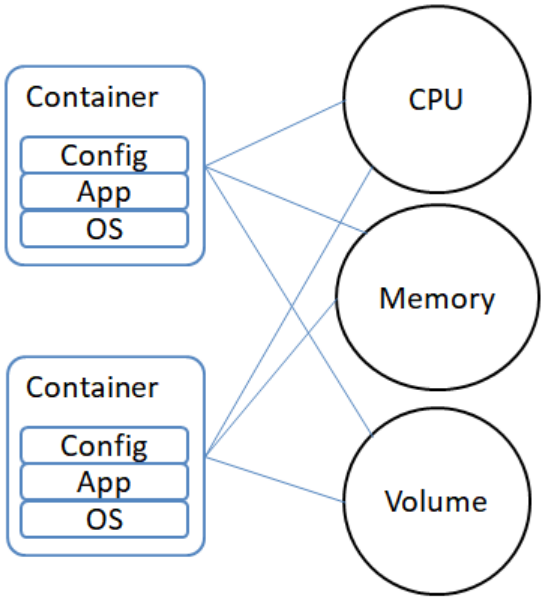
- Containers start up faster
- Containers are smaller and take less resources to run
- Containers require less configuration
- Containerized applications can be scaled quickly and automatically



Containerization

4.4. Decoupled Resources

- With traditional server environments resources such as file systems, disk space, memory and CPU are coupled
- With Containers resources are separate entities that can be requested and allocated as needed.
- For example a container can request and then be allocated a portion of disk space based on its needs.
- Scaling of applications can be done by increasing resources



4.5. Transience

- Transient things are temporary in nature. They are meant to exist is a given state only for a period of time which is not permanent or fixed
- **Containers** are transient. They can be created quickly and used just as long as they are needed.
- This is in contrast to traditional server environments which were meant to be set up and used on a permanent basis.
- The transient nature of containers allows for more efficient use of resources.
- Containers only hold onto resources during their lifetime after

which those resources can be allocated elsewhere.

4.6. Flexible Framework

- Application have various needs:
  - Applications need to be configured to access various resources
  - Applications need to be scaled in response to varying workflows
  - Configuration or Scaling can be manual or automated
- To achieve application requirements a flexible framework is needed
- A flexible framework requires:
  - A variety of resources (FS, DB, SaaS, PaaS, IaaS)
  - Virtualization
  - Containerization
  - Ability to mix and match components
- Various cloud platforms seek to offer this type of capability

4.7. Application Resource Usage

- Efforts should be made to balance the allocation of resources against their actual usage by an application. *For example:*
  - Allocating 3GB RAM to an application that only ever uses 1GB of RAM wastes 2GB that could be used for other purposes

- The solution is typically to allocate a given amount of a resource to start with and then more as the need increases. *For example:*
    - Allocate 512MB of disk space to an application when it starts up and then 512MB more as it gets close to using up the initial space.
- In some cases you also want to be able to de-allocate a resource. *For Example:*
    - A messaging server application is allocated 2 CPU cores during peak usage. The # of cores is then de-allocated to 1 CPU during off-peak hours.
- In order for resources to be allocated/de-allocated one needs to know
    - How much of a resource is being used at the moment
    - applications run and use resources

4.8. Measuring Resource Usage

- To plan for allocation and de-allocation we need to know:
    - How much of each resource an app uses
    - How that requirement changes
    - What causes the requirement to change
- Most computing platforms provide some type of real-time metrics/monitoring that can be used to gather information about an application or service's usage.
- Cloud platforms go one step further and use this information to automatically scale an application's resource usage up or down as

needed.

`docker stats`

4.9. Docker Resource Usage Statistics

- The Docker stats command returns a live data stream for running containers that included resource usage statistics

```
docker stats [options] [container(s)]
```

- By default it returns data for all running containers.
- Data can be limited to a specific containers by adding their names at the end of the command.
- Some of the data output includes:
  - CPU%
  - Memory Usage & Limits
  - Mem%
  - Network and File System I/O stats
  - Process ID Count
- Output can be restricted to certain columns and formatted using a Go template

4.9.1. Notes

For more information on the Docker stats command see:
https://docs.docker.com/engine/reference/commandline/stats/

4.10. Docker Container Resource Constraints

- By default containers have no resource constraints
- Constraints can be added if desired to control Memory and CPU usage
- Constraints can be enabled:
  - Using flags of the docker run command
  - Using service-level constraints an node labels for Docker services

4.11. Docker Run Command Resource Flags

- Flags added to the run command affect how memory/cpu are handled. These commands allow you to set limits on the amount of CPU and Memory used by the container

```
--cpus=<value>
--memory=<value>
```

- Example:

```
docker run -it --cpus=".5" --memory="512m" ubuntu /bin/bash
```

- For a list of all the available flags and details of their usage see: https://docs.docker.com/config/containers/resource_constraints/

4.12. Using Label Selectors

- Label selectors are used in Kubernetes to select sets of objects
  - Labels are added to one or more objects
  - Label selectors are used to choose objects based on their labels
- Two types of selectors are available:
  - **Equality** based selectors allow filtering by key and value
  - **Set based** selectors allow filtering by keys

4.13. Equality Based Label Selector

- Allows selection by label keys and values
- Given an object with these labels...

```
metadata: {
  labels: {
    app : nginx,
    environment : dev
  }
}
```

- The object(s) with the above labels could be selected with the following which are looking for objects with labels that are **equal**

to the one given:

```
selector:
  matchLabels:
    app: nginx
```

- or

```
selector:
  matchLabels:
    environment : dev
```

- You might find the selector above in a deployment where it is used to associate a pod or pods with the deployment

4.14. Set Based Label Selector

- Selects label values from a set of values
- Given an object with these labels...

```
metadata: {
  labels: {
    app : "nginx,
    environment : dev
  }
}
```

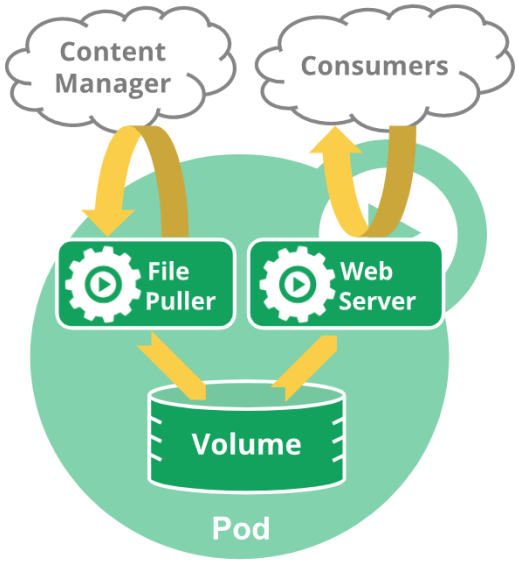
- The following selector will choose object that have the label key=app with a value equal to either nginx or httpd:

```
selector:
  matchLabels:
    app in (nginx, httpd)
```

- You might find the selector above in a deployment where it is used to associate a pod or pods with the deployment

4.15. Multi-Container Pods

- While many Pods hold only a single run-time container others may hold multiple containers
- Using the same pod for multiple containers allows the containers to be managed together
- One possible scenario where this might be done is when you need to run a server app in one container and run another app in a second container that is used to maintain the configuration or files used by the server app

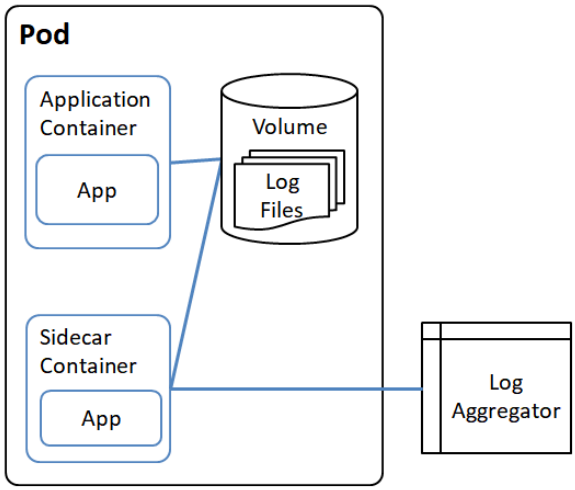


4.15.1. Notes

An example of multiple containers might include an nginx container and an alpine Linux container. The nginx container runs a web server while the alpine container runs a program that creates HTML files and maintains the server's root directory. In this case both containers are accessing the same directory from a shared volume.

4.16. Sidecar Container

- When a second container is added to a Pod that already has a container that second container is referred to as a 'Sidecar' Container
- Sidecar containers are used to implement the Sidecar Pattern
- All containers in the same Pod share the Pod's volume and network interface.



4.17. Sidecar Container Uses

- A typical use for a sidecar is to house an application that manages files used by the main application container.
- One specific use case for the sidecar might be to house an

application that distributes the main application's log files to a log aggregation server.

- Another use case is for the application in the sidecar to house an application that allows a specific type of access to the main application. One example of this might be having a database as the main application and providing REST access to the database through the application in the sidecar

4.18. Adapter Container

- Sidecar containers that implement additional ways to access the main application are called Adapter containers.
- The REST interface example mentioned on the previous slide is an example of an Adapter container.
- Another example might be to have an app log or metrics log conversion application in an Adapter container that provides the logs in a format required by another application.

4.19. Summary

In this chapter we covered:

- Traditional Applications
- Decoupled Resources
- Transience
- Flexible Framework

- Managing Resource Usage
- Using Label Selectors
- Multi-Container Pods
- Sidecar Container
- Adapter Container

Chapter 5. Deployment Configuration

Objectives

Key objectives of this chapter

- Introduction to Volumes
- Volume Spec and Types
- Persistent Volumes and Claims
- Dynamic Provisioning
- Secrets
- ConfigMaps
- SecurityContexts
- Deployment Configuration Status
- Replicas and ReplicaSets
- Scaling
- Rolling Updates

5.1. Introduction to Volumes

Volume Storage

- Container OS file system storage
- Docker Volumes

- Kubernetes Volumes

5.2. Container OS file system storage

- Containers run operating system images that include file systems.
- It is possible to create or modify files in the OS image fs and have them used by applications.
- This can cause problems though If container crashes. Any files added/modified since it was started will be lost
- Another downside to using the image fs for file storage is that files in the container can’t be shared with other containers if needed.

5.3. Docker Volumes

- The Docker container manager includes storage volumes.
- Docker storage volumes:
    - Consist of a directory on disk with some data
    - Do not have a managed lifecycle
    - Can use volume drivers but they are of limited functionality
    - Impose a limit of one vol driver per container

5.4. Kubernetes Volumes

- Kubernetes has its own implementation of volumes that:

- Also consist of a directory on disk with some data
- Has its volumes created in Pods
- Have a lifetime tied to Pod that encloses them
- Allows data to be persisted across container restarts
- Supports many volume types through 'volume drivers'
- Lets Pods use multiple volumes at the same time

5.5. Volume Specs

- Volumes are defined and used in Kubernetes specification yaml files:
- Volumes are created in a spec under the '**volumes:**' section:

```
spec:
    ...
    volumes:
    - name: temp-volume
      emptyDir: {}
```

- Volumes are used by containers and specified under '**volumeMounts**':

```
spec:
    containers:
    - image: nginx
      name: web-container
      volumeMounts:
      - mountPath: /temp
```

```
        name: temp-volume
```

5.6. K8S Volume Types

A few examples of Kubernetes volume are listed here:

- awsElasticBlockStore
- azureDisk
- azureFile
- gcePersistentDisk
- configMap
- emptyDir
- hostPath
- local
- nfs

5.7. Cloud Resource Types

- Must be created on cloud platform before being used in K8s
- Volume is independent of Pod, when Pod goes away the volume is simply unmounted and not deleted
- Pods typically need to be running on the cloud provider
- awsElasticBlockStore
    - Mounts Amazon Web Services(AWS) EBS volumes into a Pod.

    - Can only be used with Pods that are running on AWS EC2 nodes
- azureDisk & azureFile
    - An azureDisk is used to mount a Microsoft Azure Data Disk into a Pod.
    - An azureFile is used to mount a Microsoft Azure File Volume (SMB 2.1 and 3.0) into a Pod.
- gcePersistentDisk
    - Mounts a Google Compute Engine (GCE) Persistent Disk into Pod.

5.8. emptyDir

- Used to hold temporary files
- An emptyDir volume is created when a Pod is assigned to a Node
- Is deleted when a Pod is removed
- The dir exists as long as the Pod is running on a specific node
- Any container in the Pod can read and write to the emptyDir volume

5.9. Using an emptyDir Volume

- Pod Specification

```
# empty-dir-pod.yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  name: empty-dir-pod
spec:
    containers:
    - image: nginx:latest
      name: empty-dir-container
      volumeMounts:
      - mountPath: /cache
        name: cache-volume
    volumes:
    - name: cache-volume
      emptyDir: {}
```

- Notice how the 'cache-volume' is created and then used in the volumeMount
- The empty volume is available at the mountPath '/cache' inside the container

5.9.1. Notes

Full instructions for creating the pod and investigating the volume

```
# empty-dir-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: empty-dir-pod
spec:
  containers:
  - image: nginx:latest
    name: empty-dir-container
```

```
    volumeMounts:
    - mountPath: /cache
      name: cache-volume
    volumes:
    - name: cache-volume
      emptyDir: {}
kubectl apply -f empty-dir-pod.yaml
kubectl describe pods empty-dir-pod
kubectl get pods
kubectl exec -it empty-dir-pod -- bash
/# ls cache
/# echo "some file content" > cache/file1.txt
/# cat cache/file1.txt
some file content
exit
```

5.10. Other Volume Types

- **hostPath** - A hostPath volume mounts a file or directory from the host node's filesystem into your Pod. Any given hostPath may no longer be available if a Pod get re-scheduled to a different node.
- **local Volume** - A local volume represents a mounted local storage device such as a disk, partition or directory. Is know to the scheduler so pods using it always gets placed on the correct node.
- **nfs Volume** - An nfs volume allows an existing NFS (Network File System) share to be mounted into your Pod.

5.11. Persistent Volumes

- persistentVolumeClaims allow pods to request storage space

without knowing how or from where the storage will be provided

- The following K8s objects are created separately when setting up a Persistent Volume
  - PersistentVolume (pv) - Defines a volume
  - PersistentVolumeClaim (pvc) - Defines a need for persistent vol space
  - Pod - Uses a claim to request a portion of volume space from k8s
- PVCs are associated with PVs through storageClassName, they simply request a type and size of storage needed
- K8s picks the PV to use by storageClassName and resource availability
- The same PV can supply space to multiple Pods
- K8s decides which node to place the PV on based on its nodeAffinity settings

5.12. Creating a Volume

- The following operations are required when setting up a Pod to use a Persistent Volume
  - Create a yaml file for
    - The claim (lpvc.yaml)
    - The persistent volume (lpv.yaml)
    - The Pod (lvppod.yaml)
  - Apply each yaml file in the above order

- K8s will detect that your claim(pvc) and volume (pv) are related and bind them.
- K8s will provide storage space from the volume (pv) for the pod to use.

> The yaml names above can be anything you want. In practice it is helpful to name them in such a way that you can easily determine their purpose from the name later on.

5.13. Persistent Volume Claim

- Specification file: lpvc.yaml

```
# lpvc.yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: lpv-claim
spec:
  storageClassName: local-storage
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

- Commands to apply and check:

```
kubectl apply -f lpvc.yaml
```

```
kubectl get pvc
```

- The above spec requests 1Gi of local persistent storage from K8s

More information about Persistent Volume Claims can be found here: https://kubernetes.io/docs/concepts/storage/persistent-volumes/

5.14. Persistent Volume

- Specification file: lpv.yaml

```
#lpv.yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: local-volume
spec:
  capacity:
    storage: 2Gi
  volumeMode: Filesystem
  accessModes:
  - ReadWriteOnce
  persistentVolumeReclaimPolicy: Delete
  storageClassName: local-storage
  local:
    path: /mnt/disks/ssd1
  ...
```

- Commands to apply and check:

```
kubectl apply -f lpv.yaml
kubectl get pv
```

- The above defines 2Gi of local persistent storage on a node's filesystem

5.14.1. Notes

More information about PersistentVolumes can be found here: https://kubernetes.io/docs/concepts/storage/persistent-volumes/

```
// Full Yaml file, including nodeAffinity

#lpv.yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: local-volume
spec:
  capacity:
    storage: 2Gi
  volumeMode: Filesystem
  accessModes:
  - ReadWriteOnce
  persistentVolumeReclaimPolicy: Delete
  storageClassName: local-storage
  local:
    path: /mnt/disks/ssd1
  nodeAffinity:
    required:
      nodeSelectorTerms:
      - matchExpressions:
        - key: kubernetes.io/hostname
```

```
          operator: In
          values:
          - example-node
```

5.15. Pod that uses Persistent Volume

- Specification file: lpvpod.yaml

```
# lpvpod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: task-pv-pod
spec:
  volumes:
    - name: lpv-storage
      persistentVolumeClaim:
        claimName: lpv-claim
  containers:
    - name: lpv-container
      image: nginx
      ports:
        - containerPort: 80
            name: "http-server"
      volumeMounts:
        - mountPath: "/usr/share/nginx/html"
          name: lpv-storage
```

- Commands to apply and check:

```
kubectl apply -f lpvpod.yaml
kubectl get pods
```

- The above spec requests storage using the lpv-claim and makes it available at a mount point within the image

5.16. Dynamic Volume Provisioning

- Dynamic volume provisioning automatically provisions storage when requested by users.
- It eliminates the need to pre-provision storage.
- The first step in dynamic provisioning is to create a StorageClass object

```
kubectl create -f volume-storage-class.yaml
```

- StorageClass spec yaml

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: *disk-storage*
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-standard
```

- Types that can be specified include:
  - **pd-standard** - Standard Hard Disk Storage
  - **pd-ssd** - SSD Solid State Disk storage

5.17. Requesting Dynamic Storage

- Dynamically Provisioned storage is requested using a PersistentVolumeClaim.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: claim1
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: *disk-storage*
  resources:
    requests:
      storage: 512Mb
```

- The 'storageClassName' should match the 'name:' in the StorageClass spec you plan to use.
- The amount of storage requested is defined under 'resources:'

5.18. Secrets

- Secrets are used to store and manage sensitive information like passwords, tokens and keys.
- Secrets provide an alternative to placing sensitive information in a Pod definition or a container image.
- Pods can consume secrets:
    - From files in a mounted volume

    - A container environment variables
- Pods can be created from private images that are pulled by the kubelet. In this case the required secret comes from the imagePullSecrets field

5.19. Creating Secrets from Files

- Create a secret from a directory of files:
    - Create a directory for the secrets
    - In the secrets dir create file whose name is the secret's name and whose contents is the value of the secret

```
Filename: userid
Contents: jane@abc.com
```

    - Create another secrets file:

```
Filename: password
Contents: p@ssW0rd
```

    - Run this command:

```
kubectl create secret generic db-creds-1 --from-file=secrets
```

    - Check the secrets

```
kubectl get secrets db-creds-1 -o=json
```

5.20. Creating Secrets from Literals

- Create a secret by passing the data to the create command:
  - These are the secrets we want to save

```
userid=jane@abc.com
password=$ecretp@ss
```

  - Run this command:

```
kubectl create secret generic db-creds-2 --from
-literal=userid=jane@abc.com --from
-literal=password=$ecretp@ss
```

  - Check the secrets

```
kubectl get secrets db-creds-2 -o=json
```

5.21. Using Secrets

- Pod spec that mounts the secret as a volume

```
# nginx-dbcreds-pod.yaml (partial)
```

```
volumeMounts:
 - name: dbcredsvol
  mountPath: "/tmp/dbcreds"
  readOnly: true

volumes:
- name: dbcredsvol
  secret:
    secretName: db-creds-1
```

- Create the pod and open a shell into it

```
kubectl apply -f nginx-dbcreds-pod.yaml
kubectl exec -it consumesec -c shell -- bash
```

- Access the secret from its mount point

```
cat /tmp/dbcreds/userid
jane@abc.com
```

5.21.1. Notes

```
// Pod definition - Mount secret as volume
apiVersion: v1
kind: Pod
metadata:
  name: consumesec
spec:
  containers:
  - name: shell
    image: nginx:latest
```

```
    command:
      - "bin/bash"
      - "-c"
      - "sleep 10000"
    volumeMounts:
      - name: dbcredsvol
        mountPath: "/tmp/dbcreds"
        readOnly: true
  volumes:
  - name: dbcredsvol
    secret:
      secretName: db-creds-1
```

5.22. configMaps

- Provides a way to inject configuration data into Pods
- Data consists of names and values:

```
color.text=black
color.background=litegrey
```

- configMaps must be created before they can be used.
- configMaps can be created from literals or files

```
kubectl create configmap my-config --from-file=path
```

- configMaps are mounted and accessed as volumes

```
volumes:
```

```
- name: foo
  configMap:
    name: myconfigmap
```

For more info on configMaps see the following link: https://kubernetes.io/docs/concepts/configuration/configmap/

5.23. Creating configMaps from Literals

- configMaps can be created from literals
- This is a name/value pair we want to access

```
background=litegrey
```

- Execute this command:

```
kubectl create configmap config1 --from
-literal=background=litegrey
```

Multiple 'from-literal' sections can be added in the same command

- Verify that the configMap was created:

```
kubectl describe configmap config1
```

5.24. Creating configMaps from files

- Create a directory for the config file

```
mkdir configs
```

- Create a file named 'myconfig' in the above dir with these values:

```
color.text=black
color.background=litegrey
```

- Execute this command to create the configMap

```
kubectl create configmap myconfig --from-file=configs
```

- Verify that the configMap was created

```
kubectl describe configmap myconfig
```

5.25. Using configMaps

- This Pod definition creates a pod named myconfig that mounts the configMap named "myconfig"

```
# myconfig-pod.yaml
apiVersion: v1
kind: Pod
```

```
metadata:
  name: myconfig
spec:
  containers:
  - name: mypod
    image: redis
    volumeMounts:
    - name: config
      mountPath: "/etc/config"
      readOnly: true
  volumes:
  - name: config
    configMap:
      name: myconfig
```

- Create the pod and open a shell into it and access the config file:

```
kubectl apply -f myconfig-pod.yaml
kubectl exec -it myconfig -- bash
cat /etc/config/myconfig
```

5.26. Security Context

- A security context defines privilege and access control settings for Pods and Containers.
- securityContext settings at the pod level apply to all containers in the pod as well.
- There are many aspects of security that the context can set. Some examples include:
    - runAsUser

- runAsGroup
- fsGroup
- Linux Capabilities
- readOnlyRootFilesystem
- ...

5.27. Security Context Usage

Usage:

- Design a security context

```
securityContext:
runAsUser: 1000
runAsGroup: 3000
fsGroup: 2000
```

- Add the context to a pod specification

```
#security-context-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo
spec:
  securityContext:
    runAsUser: 1000
    runAsGroup: 3000
    fsGroup: 2000
  volumes:
```

```
  - name: sec-ctx-vol
    emptyDir: {}
  containers:
  - name: sec-ctx-demo
    image: busybox
    command: [ "sh", "-c", "sleep 1h" ]
```

- Create the pod

```
kubectl apply -f security-context.yaml
```

- The created pod will have the context applied

5.28. Deployment Configuration Status

- Deployments have a lifecycle that includes various states
- Deployment States:
  - progressing
  - complete
  - fail to progress
- Progress of a deployment can be monitored using the command:

```
kubectl rollout status {deployment-name}
```

5.28.1. Notes

More information on deployment status can be found here: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#deployment-status

5.29. Replicas

- Deployments specify the creation of pods where applications are run
- The number of identical Pods to create for the application is based on the deployment's 'replicas' field.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-world
spec:
  replicas: 5
```

- The number of Pods in a deployments can be increased or decreased(scaled) as needed
- A ReplicaSet that includes values that affect scaling is created for each deployment - even when replicas=1
- Kubernetes uses the ReplicaSet to help manage the process of scaling

5.29.1. Notes

Example replicaset:

```
Name: hello-nginx-654d688b4f
Namespace: default
Selector: app=hello-nginx,pod-template-hash=654d688b4f
Labels: app=hello-nginx
pod-template-hash=654d688b4f
Annotations: deployment.kubernetes.io/desired-replicas: 1
deployment.kubernetes.io/max-replicas: 2
deployment.kubernetes.io/revision: 1
Controlled By: Deployment/hello-nginx
Replicas: 1 current / 1 desired
Pods Status: 1 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
    Labels: app=hello-nginx
            pod-template-hash=654d688b4f
    Containers:
     nginx:
      Image: nginx:latest
      Port: <none>
      Host Port: <none>
      Environment: <none>
      Mounts: <none>
    Volumes: <none>

Events:
Type    Reason          Age     From                   Message
----    ------          ----    ----                   -------
Normal SuccessfulCreate 6m8s    replicaset-controller Created pod:
hello-nginx-654d688b4f-2kddc
```

5.30. Scaling

- Increasing or decreasing a deployments replicas can be done using the Kubernetes **scale** command.

```
kubectl scale deployment {deployment-name} --replicas #
```

- The command:
  - Changes the deployment's replicas value
  - Changes replica values in the associated replica set
  - Starts the process of adding or removing Pods from the deployment
- Example:

```
kubectl scale deployment hello-nginx --replicas 3
```

- The autoscale command lets Kubernetes decide when to perform scaling based on conditions such as CPU usage

```
kubectl autoscale deployment hello-nginx --max 6 --min 4 --cpu
-percent 50
```

More information regarding scaling is available here: https://cloud.google.com/kubernetes-engine/docs/how-to/scaling-apps

5.31. Rolling Updates

- Applications frequently need to be updated to include bug fixes or add new features.
- In some cases deployments are rolled back to a previous version to avoid issues that started when the application was updated.
- Rolling Updates in Kubernetes allow these changes to take place with zero downtime by incrementally updating Pods instances with new ones.
- Updates can be done by:
    - Modifying aspects of the original deployment.yaml and Applying the yaml spec with Kubectl
        - Increase or decrease the number of replicas
        - Set it to use an updated image
        - Applying the yaml spec

          ```
          kubectl apply -f deployment.yaml
          ```

    - Using commands like **scale** that modify the deployment

5.32. Summary

In this chapter we covered:

- Introduction to Volumes
- Volume Spec and Types

- Persistent Volumes and Claims
- Dynamic Provisioning
- Secrets
- ConfigMaps
- SecurityContexts
- Deployment Configuration Status
- Replicas and ReplicaSets
- Scaling
- Rolling Updates

Chapter 6. Security

Objectives

Key objectives of this chapter

- Security Overview
- Accessing the API
- Authentication
- Authorization
- Admission Controller
- ABAC and RBAC
- Network Policies

6.1. Security Overview

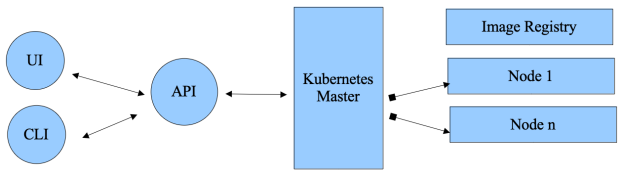
- Security is critical to production deployments.
- Kubernetes offers several features to secure your environment:
    - authentication
    - authorization
        - ABAC
        - RBAC
            - Role, ClusterRole, RoleBinding, ClusterRoleBinding
    - network policies

6.2. API Server

- Kubernetes has a built-in API server that provides access to objects, such as nodes, pods, deployments, services, secrets, configmaps, and namespaces.
- These objects are exposed via simple REST API through which basic CRUD operations are performed.
- API Server acts as the gateway to the Kubernetes platform.
- Components such as kubelet, scheduler, and controller access the API via the API Server for orchestration and coordination.
- The distributed key/value database, etcd, is accessible only through the API Server.
- In the Kubernetes API, most resources are represented and accessed using a string representation of their object name, such as pods for a Pod.
    - Some Kubernetes APIs involve a subresource, such as the logs for a Pod. A request for a Pod's logs looks like:

```
GET /api/v1/namespaces/{namespace}/pods/{name}/log
```

6.3. API & Security



Both the kubectl CLI tool and the web portal talks to the API

- Server.
- Before an object is accessed or manipulated within the Kubernetes cluster, the request needs to be authenticated by the API Server.
- The REST endpoint uses TLS based on X.509 certificate to secure and encrypt the traffic.
- The CA certificate and client certificate information is stored in **~/.kube/config**.
- You can view the file using any text editor or you can also view it by running the following command:

```
kubectl config view
```

6.4. ~/.kube/config

- Sample ~/.kube/config file

```
apiVersion: v1
clusters:
- cluster:
    certificate-authority: /Users/test/.minikube/*ca.crt*
    server: https://192.168.99.100:8443
  name: minikube
contexts:
- context:
    cluster: minikube
    user: minikube
  name: minikube
current-context: minikube
kind: Config
preferences: {}
```

```
users:
- name: minikube
  user:
    client-certificate: /Users/test/.minikube/client.crt
    client-key: /Users/test/.minikube/client.key
```

6.5. ~/.kube/config (contd.)

- The file **ca.crt** represent the CA used by the cluster.
- The **client.crt** and **client.key** files map to the user minikube that is the default cluster-admin.
- Kubectl uses these certificates and keys from the current context to encode the request.

6.6. Kubernetes Access Control Layers

- When a valid request hits the API Server, it goes through three stages before it is either allowed or denied.
    - Authentication
    - Authorization
    - Admission Controller

      IMAGE MISSING IN ASSETS

6.7. Authentication

- After the request gets past TLS, it passes through the authentication phase that involves authentication modules.
- Authentication modules are configured by the administrator during the cluster creation process.
    - Examples of authentication modules: client certificates, password, plain tokens, bootstrap tokens, and JWT tokens (used for service accounts).
    - Details of authentication modules are available on the Kubernetes website: https://kubernetes.io/docs/reference/access-authn-authz/authentication/
- Client certificates is the default and most common scenario.
- External authentication mechanisms provided by OpenID, Github, or even LDAP can be integrated with Kubernetes through one of the authentication modules.

6.8. Authorization

- After authentication, the next step is to determine whether the operation is allowed or not.
- For authorizing a request, Kubernetes looks at three aspects:
    - **the username of the requester** - extracted from the token embedded in the header
    - **the requested action** - one of the HTTP verbs like GET, POST, PUT, DELETE mapped to CRUD operations
    - **the object affected by the action** - one of the valid Kubernetes objects such as a pod or a service.

- Kubernetes determines the authorization based on an existing policy. By default, Kubernetes follows the philosophy of closed-to-open, which means an explicit allow policy is required to even access the resources.
- Like authentication, authorization is configured based on one or more modes/modules, such as:
    - RBAC
    - ABAC

6.9. ABAC Authorization

- Attribute-based access control (ABAC) defines an access control paradigm whereby access rights are granted to users through the use of policies that combine attributes.
- ABAC uses a policy file where one JSON object is listed per line.
- Each line in the JSON policy file is a policy object.
- If you are using the Minikube distribution, you can enable ABAC authorization like this:

```
minikube start --extra-config=apiserver.AuthorizationMode=ABAC
--extra
-config=apiserver.AuthorizationPolicyFile=/path/to/your/abac/poli
cy.json
```

6.10. ABAC - Policy Format

- Versioning properties:
  - **apiVersion**: "abac.authorization.kubernetes.io/v1beta1"
  - **kind**: "Policy"
- **spec:** property set to a map with the following properties:
  - Subject-matching properties:
    - **user:** "userName"
    - **group:** “groupName” | system:authenticated | system:unauthenticated
  - Resource-matching properties:
    - **apiGroup:** "*" | "extensions"
    - **namespace:** "*" | "your_custom_namespace"
  - **resource:** "*" | "pods" | "deployments" | "services", ...
  - Non-resource-matching properties:
    - nonResourcePath: "/version" | "*"
  - **readonly: true | false**, type boolean, when true, means that the Resource-matching policy only applies to get, list, and watch operations, Non-resource-matching policy only applies to get operation.

6.11. ABAC - Examples

- Alice can do anything to all resources:

```
{"apiVersion": "abac.authorization.kubernetes.io/v1beta1",
"kind": "Policy", "spec": {"user": "alice", "namespace": "*",
"resource": "*", "apiGroup": "*"}}
```

- The Kubelet can read any pods:

```
{"apiVersion": "abac.authorization.kubernetes.io/v1beta1",
"kind": "Policy", "spec": {"user": "kubelet", "namespace": "*",
"resource": "pods", "readonly": true}}
```

- The Kubelet can read and write events:

```
{"apiVersion": "abac.authorization.kubernetes.io/v1beta1",
"kind": "Policy", "spec": {"user": "kubelet", "namespace": "*",
"resource": "events"}}
```

- Bob can just read pods in namespace "projectCaribou":

```
{"apiVersion": "abac.authorization.kubernetes.io/v1beta1",
"kind": "Policy", "spec": {"user": "bob", "namespace":
"projectCaribou", "resource": "pods", "readonly": true}}
```

- Anyone can make read-only requests to all non-resource paths:

```
{"apiVersion": "abac.authorization.kubernetes.io/v1beta1",
"kind": "Policy", "spec": {"group": "system:authenticated",
"readonly": true, "nonResourcePath": "*"}}
{"apiVersion": "abac.authorization.kubernetes.io/v1beta1",
"kind": "Policy", "spec": {"group": "system:unauthenticated",
```

```
 "readonly": true, "nonResourcePath": "*"}}
```

6.12. RBAC Authorization

- Role-based access control (RBAC) is a method of regulating access to a computer or network resources based on the roles of individual users within your organization.
- RBAC authorization uses the rbac.authorization.k8s.io API group to drive authorization decisions, allowing you to dynamically configure policies through the Kubernetes API.
- RBAC authorization involves the following resources:
    - Role
    - CluserRole
    - RoleBinding
    - ClusterRoleBinding
- RBAC is the default authorization mode. If you want to explicitly specify this mode, you can use the following command with the Minikube distribution:

```
minikube start --extra-config=apiserver.Authorization.Mode=RBAC
```

6.13. Role and ClusterRole

- An RBAC Role or ClusterRole contains rules that represent a set of permissions.

- **Role** - always sets permissions within a particular namespace
  - when you create a Role, you have to specify the namespace it belongs in.
  - treat it as a project-based role where a user will have to access to a specific namespace.
- **ClusterRole -** is a non-namespaced resource.
  - use it to create admin users who can define permissions on namespaced resources and be granted within an individual namespace(s)
  - define permissions on cluster-scoped resources, such as nodes.
  - For example, you can use a ClusterRole to allow a particular user to run kubectl get pods --all-namespaces.
- The resources have different names (Role and ClusterRole) because a Kubernetes object always has to be either namespaced or not namespaced; it can’t be both.

6.14. Role - Example

Here’s an example Role in the "marketing" namespace that can be used to grant read access to pods:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: marketing
  name: marketing-pod-reader
rules:
```

```
- apiGroups: [""] # "" indicates the core API group
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

6.15. ClusterRole - Example

Here is an example of a ClusterRole that can be used to grant read access to nodes:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  # "namespace" omitted since ClusterRoles are not namespaced
  name: nodes-reader
rules:
- apiGroups: [""]
  #
  # at the HTTP level, the name of the resource for accessing Secret
  # objects is "nodes"
  resources: ["nodes"]
  verbs: ["get", "watch", "list"]
```

6.16. RoleBinding and ClusterRoleBinding

- A role binding grants the permissions defined in a role to a user or set of users.
- It holds a list of subjects (users, groups, or service accounts), and a reference to the role being granted.
- A RoleBinding grants permissions within a specific namespace

whereas ClusterRoleBinding grants that access cluster-wide.

- A RoleBinding may reference any Role in the same namespace.
- If you want to bind a ClusterRole to all the namespaces in your cluster, you use a ClusterRoleBinding.

6.17. RoleBinding - Example

Here is an example of a RoleBinding that grants the "pod-reader" Role to the user "alice" within the "sales" namespace. This allows "alice" to read pods in the "default" namespace.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-pods
  namespace: sales
subjects:
- kind: User
  name: alice
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

6.18. ClusterRoleBinding - Example

The following ClusterRoleBinding allows any user in the group "manager" to read deployments in any namespace.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: read-deployment-global
subjects:
- kind: Group
  name: manager
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: deployment-reader
  apiGroup: rbac.authorization.k8s.io
```

6.19. Authorization Modes - Node

A special-purpose authorization mode that grants permissions to kubelets based on the pods they are scheduled to run. To learn more about using the Node authorization mode

6.20. Authorization Modes - ABAC

- In Attribute-based access control (ABAC), access rights are granted to users through the use of policies that combine attributes.
- The policies can use any type of attributes (user attributes, resource attributes, object, environment attributes, etc).
- To enable ABAC mode, specify --authorization-policy -file=SOME_FILENAME and --authorization-mode=ABAC on startup.

6.21. Admission Controller

- After authorization, the request goes through the final stage: Admission Controller.
- Admission controllers limit requests to create, delete, modify, or connect to (proxy). They do not support read requests.
- For example, an admission control module may be used to enforce the pulling of images policy each time a pod is created.
- There are various admission controllers compiled into the kube-apiserver binary. Here are some of them:
    - **AlwaysPullImages:** When this admission controller is enabled, images are always pulled before starting containers, which means valid credentials are required
    - **CertificateApproval:** This admission controller observes requests to 'approve' CertificateSigningRequest resources
- For more details, refer to Kubernetes doc: https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/

6.22. Network Policies

- Network policies are the equivalent of a firewall that specify how groups of pods are allowed to communicate with each other and other network endpoints.
- Each network policy has a **podSelector** field, which selects a group of pods.
- When a pod is selected by a network policy, the network policy is applied to it.

- Each network policy also specifies a list of allowed (ingress and egress) connections. (NOTE: ingress/egress is covered in detail in another chapter).
- When the network policy is created, all the pods that it applies to are allowed to make or accept the connections listed in it.
- If no network policies are applied to a pod, then no connections to or from it would be permitted.
- Network policies require a network plugin that enforces network policies.
- Although Kubernetes allows the creation of network policies they aren't enforced unless a plugin is installed and configured.
- There are various plugins, such as Calico, Cilium, Kube-router, Romana and, Weave Net.

6.23. Network Policies - Examples

- You can apply various network policies, such as:
  - Limit access to services
  - Pod isolation
  - Allow internet access for pods
  - Allow pod-to-pod communication within the same or different namespaces.
- You can get various useful network policy recipes available from the following sites:
  - https://github.com/ahmetb/kubernetes-network-policy-recipes

- https://github.com/stackrox/network-policy-examples

6.24. Network Policies - Pod Isolation

- Pods are “isolated” if at least one network policy applies to them; if no policies apply, they are “non-isolated”.
- Network policies are not enforced on non-isolated pods.
- This behavior exists to make it easier to get a cluster up and running
  - a user who does not understand network policies can run their applications without having to create one.
- It’s recommended you start by applying a “default-deny-all” network policy.
- The effect of the default-deny-all policy specification is to isolate all pods, which means that only connections explicitly listed by other network policies will be allowed.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-all
spec:
  podSelector: {}
  policyTypes:
  - Ingress
```

Since network policies are namespaced resources, you will need to create this policy for each namespace. You can do

> so by running kubectl -n <namespace> create -f <filename> for each namespace.

6.25. Network Policies - Internet Access for Pods

- With just the default-deny-all policy in place in every namespace, none of your pods will be able to talk to each other or receive traffic from the Internet.
- For most applications to work, you will need to allow some pods to receive traffic from outside sources.
- The following network policy allows traffic from all sources for pods having the custom networking/allow-internet-access=true label:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: internet-access
spec:
  podSelector:
    matchLabels:
      networking/allow-internet-access: "true"
  policyTypes:
  - Ingress
  ingress:
  - {}
```

6.26. Network Policies - New Deployments

- When you create new deployments, they will not be able to talk to anything by default until you apply a network policy.
- You can create custom network policies that allow deployments/pods labeled networking/allow-all-connections=true to talk to all other pods in the same namespace.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-ingress-from-new
spec:
  podSelector: {}
  policyTypes:
  - Ingress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          networking/allow-all-connections: "true"
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-ingress-to-new
spec:
  podSelector:
    matchLabels:
      networking/allow-all-connections: "true"
policyTypes:
- Ingress
ingress:
- from:
  - podSelector: {}
```

6.27. Summary

In this chapter, you learned the following:

- Security Overview
- Accessing the API
- Authentication
- Authorization
- ABAC and RBAC
- Admission Controller
- Network Policies

Chapter 7. Exposing Applications

Objectives

Key objectives of this chapter

- Kubernetes Services
- Service Type
- ClusterIP
- NodePort
- LoadBalancer
- ExternalName
- Accessing Applications
- Services Without a Selector
- Ingresses
- Service Mesh

7.1. Kubernetes Services

- Kubernetes services expose applications running on a set of Pods as network services
- Services are a solution to the following situation:
    - Instances of an application run in separate Pods
    - Each Pod gets its own IP address
    - Pods come and go based on load

    - How can we access applications when we don’t know which instances are running and what their IP addresses are?

7.2. Service Resources

- Services are:
    - Defined using Kubernetes Service Resource Objects (yaml files)
    - Added to an managed on a cluster using kubectl commands
- Example:

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

- This spec creates the 'my-service" service which exposes "MyApp" on the internal cluster IP address at port 9376

7.3. Service Type

- Services can have one of the following types:

- ClusterIP (default) - IP address internal to the cluster
- NodePort - Port on the parent node's internal cluster IP address. Can be exposed outside the cluster.
- LoadBalancer - Provides access through a load balancer's IP address
- ExternalName - Maps to the external DNS name in the externalName field

7.4. ClusterIP

- ClusterIP is the default service type. Services created without a type field are assigned this type.
- The following command creates a ClusterIP type service:

```
kubectl expose deployment hello-nginx --port=8080 --targetPort=80
```

- Details of the created service

```
kubectl get services hello-nginx
NAME            TYPE        CLUSTER-IP      EXTERNAL-IP
PORT(S)     AGE
hello-nginx     ClusterIP   10.98.16.57     <none>
8080/TCP    4m59s
```

- Service can be accessed by shelling into the cluster

```
minikube ssh
```

```
curl http://10.98.16.57:8080
```

7.5. NodePort

- With a NodePort type of service Kubernetes exposes the service through a randomly generated port number off of the node where the application pods reside.
- Node ports can also be created using the **expose** command:

```
kubectl expose deployment hello-nginx --type=NodePort --port=80
```

- Details of the created service

```
kubectl get services hello-nginx
NAME            TYPE        CLUSTER-IP      EXTERNAL-IP
PORT(S)         AGE
hello-nginx     NodePort    10.110.148.227  <none>
80:31164/TCP    36s
```

- The service can be accessed through the node IP address and service's port number:

```
minikube service hello-nginx --url=true
http://192.168.99.100:31164
curl http://192.168.99.100:31164
```

The `minikube service {service-name} --url=true` command combines the node ip address and the service's

> port to create a URL that can be used to address the given service

7.6. NodePort from Service Spec

- NodePort services can also be created by applying Service Resource specifications like this one:

```
# my-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  type: NodePort
  selector:
    app: MyApp
  ports:
    - port: 80
      targetPort: 80
      nodePort: 30007
```

- Command creates/updates the service:

```
kubectl apply service -f my-service.yaml
```

7.7. LoadBalancer

- This service type provisions an external LoadBalancer to direct

traffic to the selected Pods

- This type is often used when running Kubernetes on cloud providers that include their own load balancers.
- The IP address of the external load balancer populates the service's status.loadBalancer.ip field
- The following command can be used to create this type of service:

```
kubectl expose deployment hello-nginx --name hello-nginx-lb
--type=LoadBalancer --port=80
```

- Details of the created service

```
kubectl get services hello-nginx
NAME        TYPE         CLUSTER-IP     EXTERNAL-IP     PORT(S)
AGE
hello-nginx LoadBalancer 10.111.8.184   192.168.122.1
80:30725/TCP    51m
```

- The service can be accessed through the node IP address and service's port number:

```
minikube service hello-nginx --url=true
http://192.168.122.1:31164
curl http://192.168.122.1:31164
```

7.8. LoadBalancer from Service Spec

LoadBalance type services can also be created from yaml file

- specs:

```
# example-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: example-service
spec:
  selector:
    app: example
  ports:
    - port: 8765
      targetPort: 9376
  type: LoadBalancer
```

- Command to apply the spec:

```
kubectl apply -f example-service.yaml
```

- Remember: the above requires an external load balancer to work

7.9. ExternalName

- ExternalName type services map the service to a DNS name
- Redirection to the service happens at the DNS level rather than by proxy or forwarding.
- This type maps 'my-service' in the 'prod' namespace to a server identified by a name (my.database.example.com) that can be resolved by an external network DNS.

- Example spec:

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
  namespace: prod
spec:
  type: ExternalName
  externalName: my.database.example.com
```

- This type might be used for example to map a service name inside the cluster to a database server outside the cluster

More information on the external name type is available here: https://kubernetes.io/docs/concepts/services-networking/service/#externalname

7.10. Accessing Applications

- Applications can be accessed in a variety of ways:
  - Shell into the app's container and use localhost
  - Shell into any container in the same pod as the app and again use localhost to access the app
  - Apps on the same node can access apps in different pods using the app pod's name or IP address
  - Apps outside a cluster can access a service in a k8s cluster through a service-defined port on the cluster's IP address.

- The 'kubectl port-forward' command can be used to forward a port on the local machine where the cluster is running to a pod running inside the cluster.

7.11. Service Without a Selector

- The selector in a service specification is used to identify Pods for a service to point to

```
spec:
  selector:
    app: MyApp
```

- When a service is meant to point to some other type of back-end (as opposed to Pods) the selector is omitted. You might do this:
  - To access an external production database cluster
  - To point to a service in a different namespace or cluster
  - When only some of your back-ends are hosted by Kubernetes
- When the service points to one or more Pods an Endpoint is automatically created for the service.
- For non-Pod services the endpoint must be added manually:

```
apiVersion: v1
kind: *Endpoints*
metadata:
  name: my-service
subsets:
  - addresses:
    - ip: 192.0.2.42
```

```
  ports:
    - port: 9376
```

- This endpoint then allows the service to point to the IP address and port of any server you choose.

For more information see: https://kubernetes.io/docs/concepts/services-networking/service/#services-without-selectors

7.12. Ingress

- An Ingress Resource is a Kubernetes object that acts as the entry point for your cluster.
- An Ingress sits between the external network and the services in the K8s Cluster and lets you expose multiple services under the same IP address.
- Ingresses can be used to provide:
  - Routes from network traffic outside the cluster to services within the cluster
  - Load balancing
  - SSL/TLS Termination
  - Name-based virtual hosting

7.12.1. Ingress Resource Example

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: example-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /$1
spec:
  rules:
  - host: hello-world.info
    http:
      paths:
      - path: /
        backend:
          serviceName: web
          servicePort: 8080
```

7.12.2. Notes

Documentation for Ingress' can be found here: https://kubernetes.io/docs/concepts/services-networking/ingress/

An ingress tutorial can be found here: https://kubernetes.io/docs/tasks/access-application-cluster/ingress-minikube/

7.13. Ingress Controller

- An Ingress Controller implements the features defined in the Ingress object.
- Ingress objects will have NO EFFECT unless an ingress controller has been deployed.

- Two Ingress Controllers supported by the Kubernetes project are:
  - nginx ingress controller for Kubernetes
  - GCE ingress-gce
- Several other ingress controllers are also available
- Setting up to use the **ingress-nginx** ingress controller on minikube:

```
minikube addons enable ingress
```

- Setting up the ingress-nginx controller on other platforms:
  - See: https://kubernetes.github.io/ingress-nginx/deploy/

Documentation on Ingress Controllers can be found here: https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/

7.14. Service Mesh

- A service mesh can be thought of as an infrastructure layer for service-to-service communication.
- A service mesh can implement a variety of cross-cutting features including: *Load Balancing, Circuit Breaking Retry & Timeout, Seciurity(TLS), Monitoring, Traffic Metrics*, ...
- A Service Mesh is typically implemented as proxy deployed in a sidecar container next to each instance container of an application
- Examples of Service Mesh include:

    - Open Service Mesh (OSM) (backed by Microsoft)
    - Istio (backed by Google & IBM)
    - Maesh
    - Kuma
    - AWS App Mesh
- For more information on Service Mesh see: https://servicemesh.es/

7.15. Summary

In this chapter we covered:

- Kubernetes Services
- Service Type
- ClusterIP
- NodePort
- LoadBalancer
- ExternalName
- Accessing Applications
- Services Without a Selector
- Ingresses
- Service Mesh

Chapter 8. Troubleshooting Kubernetes

Objectives

Key objectives of this chapter

- Troubleshooting Overview
- Kubernetes Objects, Relationships and Operations
- Understanding the Issue
- Tools and Commands
- Troubleshooting Pods
- Troubleshooting the Cluster
- Cluster Failure Modes
- Monitoring
- Monitoring Applications
- System Logs
- Logging Tools
- Conformance Testing

8.1. Troubleshooting Overview

- There are many possible places for things to go wrong when hosting, running and accessing applications on Kubernetes:
    - Deployments/ Pods/ Containers/ Volumes/ Services/ … can be

configured incorrectly.
    - It may not be possible to schedule applications to existing nodes
    - The application running in a Pod may not be functioning as expected
    - You may not be able to access an application from outside the cluster
    - ...
- In this chapter we will go over some techniques for discovering problems, determining their cause and correcting the issue

8.2. Objects in Kubernetes

- Issues that arise are commonly related to settings in one or more of the various objects being managed by Kubernetes:
    - Deployments
    - Nodes
    - Pods
    - Containers
    - Images
    - Applications
    - Logs
    - ReplicaSets
    - Services

    - Ingresses

8.3. Relationships in Kubernetes

There are some key relationships between objects in k8s that are helpful to remember when troubleshooting:

Deployments → ReplicaSet → Pods
Pods → Containers
Containers → Images → Applications
Network → Services → Pods
Ingresses → Services

8.4. Operations in Kubernetes

- In order to troubleshoot an issue you should know ahead of time what you expect to happen when creating or updating objects in Kubernetes
- Various operations are expected to unfold in different ways:
    - Create or Update a Deployment
        - Pods auto-created
        - Pods scheduled to node
        - Containers instantiated
        - Applications in Image are run
    - Create or Update a Service

        - Pods are selected
        - IP Addresses Configured
        - Cluster DNS is updated
    - Etc.

8.5. Understanding the Issue

- Before you begin troubleshooting it is essential that you understand what you are looking for. The steps you take to troubleshoot will depend on:
    - Something that was done manually to the cluster
        - Deployment added, scaled or updated
        - Service added
    - Some condition that triggered a change in the cluster
        - Usage increase triggered auto-scaling
    - A certain result which was expected
    - Something occurred that was not the expected result
- Having a clear understanding of the points above will help you decide how to proceed with troubleshooting.
- For example:
    - You deployed a web server, You expected 4 replicas to be created, but only three were created.
    - Based on this information it would be reasonable to suspect and look for a problem with resource allocation.

8.6. Troubleshooting Tools

- Many commands exist in Kubernetes that can be used to look into issues:
  - minikube dashboard - runs a gui for inspecting and modifying the cluster’s objects
  - kubectl - a command line tool for inspecting and modifying the cluster’s objects
- We will make use of these tools throughout this chapter

8.7. Troubleshooting Commands

- Common Kubernetes Commands:
  - kubectl get {object-type}
  - kubectl describe {object-type} {object-name}
  - kubectl exec -it {pod-name} — /bin/bash
  - kubectl logs {object-type} {object-name}
  - kubectl get events
- Get help for any command
  - Add --help to the end of the command

8.8. Troubleshooting Pods

- Get Pods

```
kubectl get pods
```

    - Check READY/STATUS
        - If STATUS != Running Continue with Describe
        - READY should show all Pods running (for example: 3/3 not 0/3)
    - Check RESTARTS
        - If RESTTARTS > 0 and growing
            - If AGE is not large
                - Check logs

- Describe A Specific Pod

```
kubectl describe pods {pod-name}
```

    - Check State [Pending|Waiting|Running]
        - If Pod is in 'Pending' state
            - Check resources (CPU/Memory)
        - If Pod is in 'Waiting' state - Pod has been scheduled but can’t start
            - Check for problems pulling the image

8.9. Troubleshooting the Cluster

- Troubleshooting the cluster begins with checking the nodes and

often involves checking various cluster level logs as well as logs at the individual worker nodes.

- Get nodes - All nodes should be present and in Ready state

```
kubectl get nodes
kubectl desceibe nodes
```

- Cluster Log Locations
  - API Server Log: /var/log/kube-apiserver.log
  - Scheduler Log: /var/log/kube-scheduler.log
  - Replication Log: /var/log/kube-controllller-manager.log
- Worker Node Log Locations
  - Kubelet Log - /var/log/kubelet.log
  - Kube Proxy - /var/log/kube-proxy.log

8.10. Cluster Failure Modes

- Problems at the cluster level that are typically the responsibility of the cluster administrator include:
  - Node VM Shuts down
  - Kubernetes Software Crashes
  - Data Volume unavailable
  - Mis-configuration of Kubernetes
  - Mis-configuration of Applications

For more information on troubleshooting cluster failures see: https://kubernetes.io/docs/tasks/debug-application-cluster/debug-cluster/

8.11. Monitoring

- Before scaling an application you need to understand how the application behaves with regards to loading and resources after deployment.
- Kubernetes allows monitoring of performance characteristics(CPU, Memory, Volume Info) for Pods, Services and Containers through existing tools like:
    - kubectl get
    - kubectl describe
- In addition to these basic commands the following metrics pipelines can be enabled:
    - **Resource Metrics Pipeline** - Limited metrics related to cluster components
    - **Full Metrics Pipeline** - Provides access to additional metrics
- As part of the Resource Metrics Pipeline the metrics-server discovers all nodes on the cluster and queries each node's kubelet for CPU and memory usage. The collected information is published through the /metrics/resource/v1beta1 endpoint on the Kubernetes API .
- The Full Metrics Pipeline exposes metrics via either the custom.metrics.k8s.io or external.metrics.k8s.io API. Metrics

applications such as Prometheus access Kubernetes metrics through these APIs

More information about metrics can be found at: https://kubernetes.io/docs/tasks/debug-application-cluster/resource-usage-monitoring/

8.12. Monitoring Applications

- Prometheus is an open-source monitoring framework.
- It can be used to monitor a variety of systems including Servers, VMs, and Databases as well as Kubernetes clusters.
- The data Prometheus gathers can:
  - Be used to create real-time dashboards
  - Be analyzed to help with scaling and performance tuning
- Prometheus:
  - Sends an HTTP scrape request for metrics to the system being monitored
  - Stores the results
  - Provides access to the results
- Prometheus is often paired with:
  - Grafana for visualization
  - Alertmanager for generating notifications

More information on setting up prometheus on Kubernetes: https://devopscube.com/setup-prometheus-

> monitoring-on-kubernetes/

8.13. Accessing Logs

- Kubernetes includes the versatile log command for displaying *application log information* for containers, pods and other resources:

```
kubectl logs
```

- The command takes several flags providing a variety of features.
- Display logs for containers:

```
kubectl logs {pod-name} --container={container-name}
```

- Display logs for all containers with the label app=nginx

```
kubectl logs -lapp=nginx --all-container=true
```

- Stream logs for a container in a given pod

```
kubectl logs -f -c {container-name} {pod-name}
```

- Display the most recent 20 log lines for a given pod

```
kubectl logs --tail=20 {pod-name}
```

For more information on "kubectl logs" use the command with the --help flag.

8.14. Logging Tools

- Kubernetes outputs many *system related* logs which are distributed across nodes in the cluster. Accessing these logs individually can be difficult and time consuming.
- Logging tools such as those listed here make it easier to work with logs:
- The EFK Stack is made up of:
    - Elastic search - object store and search engine
    - FluentD - log routing and aggregation
    - Kibana - visualization (GUI interface)
- An alternative setup makes use of the following applications:
    - Promtail - log transport
    - Grafana Loki - log aggregation and search system
    - Grafana - visualization (GUI interface)

8.15. Conformance Testing

- The Cloud Native Computing Foundation (CNCF) runs the Certified

Kubernetes Conformance Program.

- The Program ensures that every vendor's version of Kubernetes, or open source community version, conforms to specifications and supports required APIs.
- Certification provides consistency across different commercial and open source implementations of Kubernetes.
- To remain certified vendors must submit their versions of Kubernetes every year.
- End users can download and run the same conformance test suite against the Kubernetes version they use.
- See the following link for more information on certification and conformance testing:
    - https://github.com/cncf/k8s-conformance/blob/master/faq.md

8.16. Summary

In this chapter we covered:

- Troubleshooting Overview
- Kubernetes Objects, Relationships and Operations
- Understanding the Issue
- Tools and Commands
- Troubleshooting Pods
- Troubleshooting the Cluster

  Cluster Failure Modes

- Monitoring
- Monitoring Applications
- System Logs
- Logging Tools
- Conformance Testing