SUSE

Cloud Native Fundamentals Scholarship Program

Prepared by Sani Kamal

Lesson 3

Container Orchestration with Kubernetes

Learn how to use Docker to create an image and how to deploy an application to Kubernetes

CONCEPTS

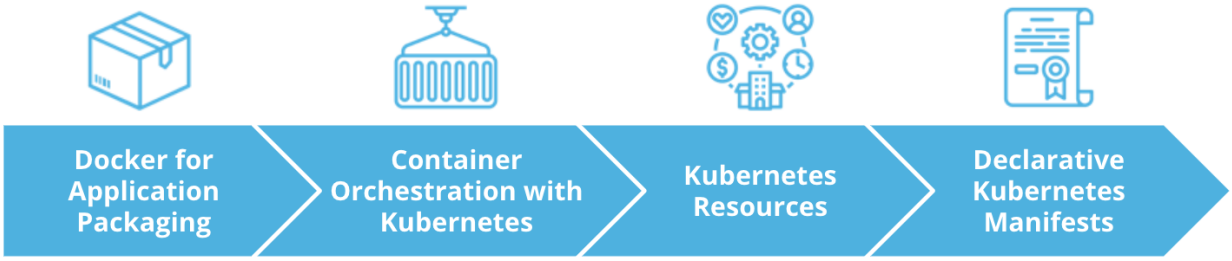
Introduction

Summary

Welcome to the Container Orchestration lesson!

Once a team has developed an application, the next phase is represented by the release process. This includes techniques for service packaging, containerization, and distribution. The end result of a product release is represented by a service that is deployed in a production environment and can be accessed by consumers.

In this lesson, we will discuss how an application can be packaged using Docker and released to a Kubernetes cluster.



Container Orchestration lesson outline

Overall, in this lesson we will explore:

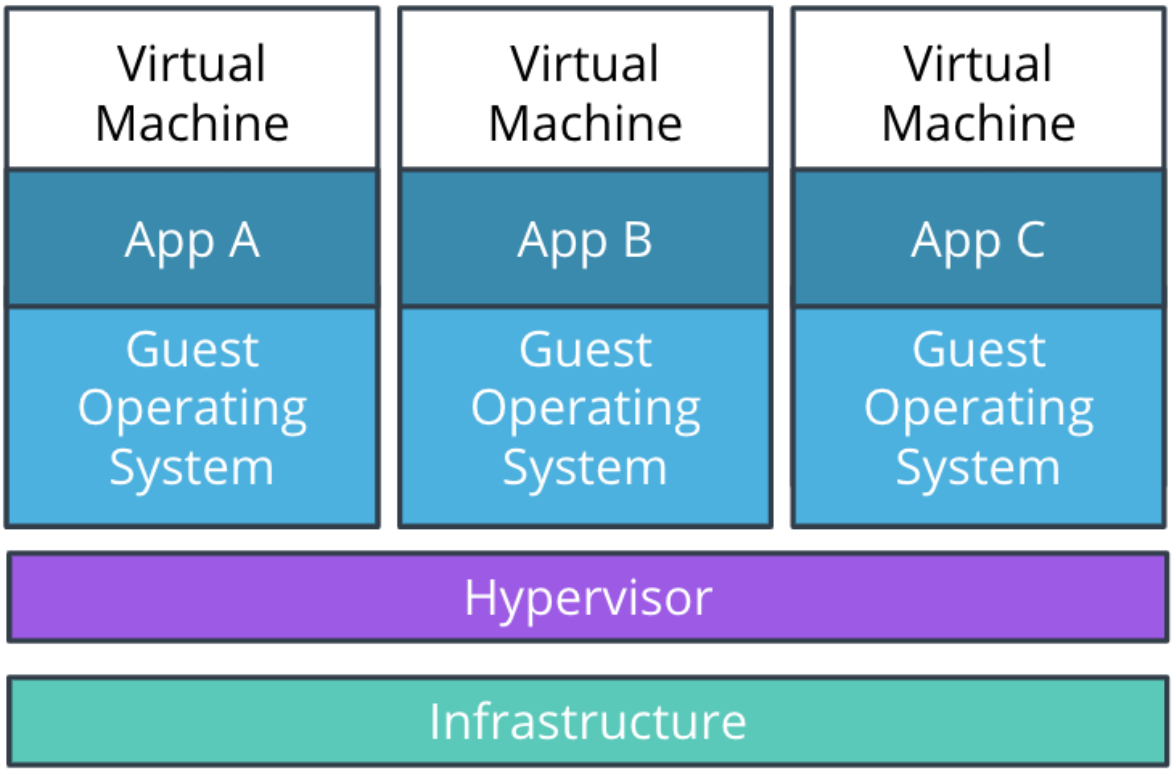
- Docker for Application Packaging
- Container Orchestration with Kubernetes
- Kubernetes Resources
- Declarative Kubernetes Manifests

Transitions from VMs to Containers

Summary

VMs

In the past years, VMs (Virtual Machines) were the main mechanism to host an application. VMs encapsulate the code, configuration files, and dependencies necessary to execute the application.



Multiple applications hosted on VMs

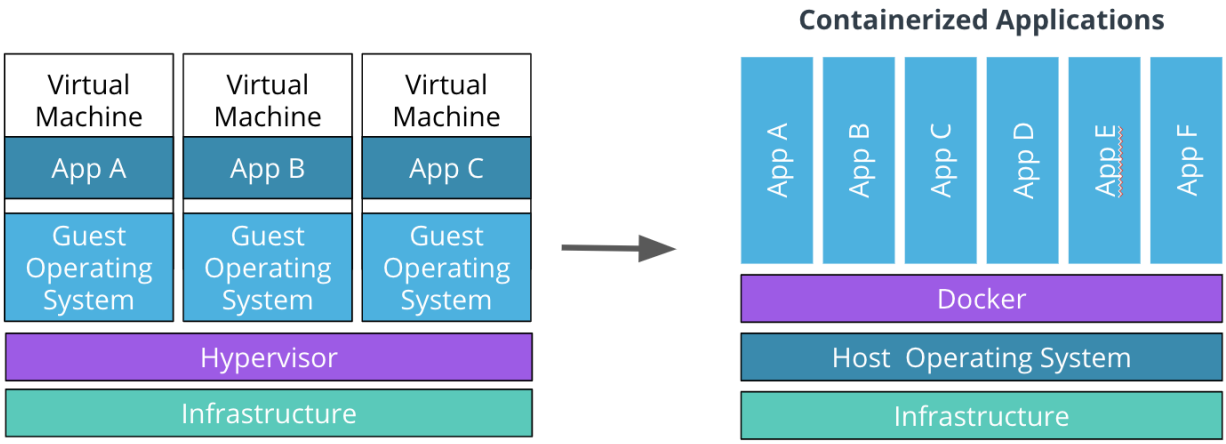
In essence, a VM is composed of an **operating system** (OS) with a set of pre-installed libraries and packages. During execution, an **application** utilizes an OS filesystem, resources, and packages.

A set of VMs is managed through a **hypervisor**. A hypervisor provides the virtualization of the **infrastructure** which is composed of physical servers. As a result, a hypervisor is capable of

creating, configuring, and managing multiple VMs on the available servers. For example, we are able to running applications A, B, and C on 3 separate VMs.

The utilization of VMs introduced standardization in infrastructure provisioning, in association with efficient use of available infrastructure. Instead of running an application per server, a hypervisor enables multiple VMs to run at the same time to host multiple applications. However, there is one downside to this mechanism: it is not efficient enough. For example, applications A, B, and C uses the same Operating System. Replicating an OS consumes a lot of resources, and the more applications we run the more space we allocate to the replication of the operating systems alone.

Containers



The transition from VMs to containers

There was a clear need to optimize the usage of the available infrastructure. As a result, the virtualization of the Operating System was introduced. This prompted the appearance of **containers**, which represent the bare minimum an application requires for a successful execution e.g. code, config files, and dependencies. By default, there is a better usage of available infrastructure.

Multiple VMs on a hypervisor are replaced by multiple containers running on a single host operating system. The processes in the containers are completely isolated but are able to access the OS filesystem, resources, and packages. The creation and execution of containers is delegated to a container management tool, such as Docker.

The appearance of containers is unlocked by OS-level virtualization and as a result, multiple applications can run on the same OS. By nature, containers are lightweight, as these encapsulate only the application code and essential dependencies. Consequently, there is a better usage of available infrastructure and a more efficient pathway to release a product to consumers.

Docker for Application Packaging

Summary

The appearance of containers enabled organizations to ship products using a lightweight mechanism, that would make the most of available infrastructure. There are plenty of tools used to containerize services, however, Docker has set the industry standards for many years.

To containerize an application using Docker, 3 main components are distinguished: Dockerfiles, Docker images, and Docker registries. Let's explore each component in a bit more detail!

Dockerfile

A Dockerfile is a set of instructions used to create a Docker image. Each instruction is an operation used to package the application, such as installing dependencies, compile the code, or impersonate a specific user. A Docker image is composed of multiple layers, and each layer is represented by an instruction in the Dockerfile. All layers are cached and if an instruction is modified, then during the build process only the changed layer will be rebuild. As a result, building a Docker image using a Dockerfile is a lightweight and quick process.

To construct a Dockerfile, it is necessary to use the pre-defined instructions, such as:

```
     FROM -  to set the base image
     RUN - to execute a command
     COPY & ADD  - to copy files from host to the container
     CMD - to set the default command to execute when the container
starts
     EXPOSE - to expose an application port
```

Below is an example of a Dockerfile that targets to package a Python hello-world application:

```
# set the base image. Since we're running
# a Python application a Python base image is used
FROM python:3.8
# set a key-value label for the Docker image
LABEL maintainer="Katie Gamanji"
# copy files from the host to the container filesystem.
# For example, all the files in the current directory
# to the  `/app` directory in the container
COPY . /app
```

```
#  defines the working directory within the container
WORKDIR /app
# run commands within the container.
# For example, invoke a pip command
# to install dependencies defined in the requirements.txt file.
RUN pip install -r requirements.txt
# provide a command to run on container start.
# For example, start the `app.py` application.
CMD [ "python", "app.py" ]
```

Docker Image

Once a Dockerfile is constructed, these instructions are used to build a **Docker image**. A Docker image is a read-only template that enables the creation of a runnable instance of an application. In a nutshell, a Docker image provides the execution environment for an application, including any essential code, config files, and dependencies.

A Docker image can be built from an existing Dockerfile using the docker build command. Below is the syntax for this command:

```

# build an image
# OPTIONS - optional;  define extra configuration
# PATH - required;  sets the location of the Dockefile and  any referenced
files
docker build [OPTIONS] PATH

# Where OPTIONS can be:
-t, --tag - set the name and tag of the image
-f, --file - set the name of the Dockerfile
--build-arg - set build-time variables

# Find all valid options for this command
docker build --help
```

For example, to build the image of the Python hello-world application from the Dockerfile, the following command can be used:

```
# build an image using the Dockerfile from the current directory
```

```
docker build -t python-helloworld .

# build an image using the Dockerfile from the `lesson1/python-app`
directory
docker build -t python-helloworld lesson1/python-app
```

Before distributing the Docker image to a wider audience, it is paramount to test it locally and verify if it meets the expected behavior. To create a container using an available Docker image, the docker ‘run command’ is available. Below is the syntax for this command:

```
# execute an image
# OPTIONS - optional;  define extra configuration
# IMAGE -  required; provides the name of the image to be executed
# COMMAND and ARGS - optional; instruct the container to run specific
commands when it starts
docker run [OPTIONS] IMAGE [COMMAND] [ARG...]

# Where OPTIONS can be:
-d, --detach - run in the background
-p, --publish - expose container port to host
-it - start an interactive shell

# Find all valid options for this command
docker run --help
```

For example, to run the Python hello-world application, using the created image, the following command can be used:

Note: To access the application in a browser, we need to bind the Docker container port to a port on the host or local machine. In this case, 5111 is the host port that we use to access the application e.g. http://127.0.0.1:5111/. The 5000 is the container port that the application is listening to for incoming requests.

```
# run the `python-helloworld` image, in detached mode and expose it on
port `5111`
docker run -d -p 5111:5000 python-helloworld
```

To retrieve the Docker container logs use the docker logs {{ CONTAINER_ID }} command. For example:

```

```

```
docker logs 95173091eb5e

## Example output from a Flask application
 * Serving Flask app "app" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production
deployment.
   Use a production WSGI server instead.
 * Debug mode: off
```

Docker Registry

The last step in packaging an application using Docker is to store and distribute it. So far, we have built and tested an image on the local machine, which does not ensure that other engineers have access to it. As a result, the image needs to be pushed to a **public Docker image registry**, such as DockerHub, Harbor, Google Container Registry, and many more. However, there might be cases where an image should be private and only available to trusted parties. As a result, a team can host private image registries, which provides full control over who can access and execute the image.

Before pushing an image to a Docker registry, it is highly recommended to tag it first. During the build stage, if a tag is not provided (via the -t or --tag flag), then the image would be allocated an ID, which does not have a human-readable format (e.g. 0e5574283393). On the other side, a defined tag is easily scalable by the human eye, as it is composed of a registry repository, image name, and version. Also, a tag provides version control over application releases, as a new tag would indicate a new release.

To tag an existing image on the local machine, the docker tag command is available. Below is the syntax for this command:

```
# tag an image
# SOURCE_IMAGE[:TAG]  - required and the tag is optional; define the name
of an image on the current machine
# TARGET_IMAGE[:TAG] -  required and the tag is optional; define the
repository, name, and version of an image
docker tag SOURCE_IMAGE[:TAG] TARGET_IMAGE[:TAG]
```

For example, to tag the Python hello-world application, to be pushed to a repository in DockerHub, the following command can be used:

```
# tag the `python-helloworld` image, to be pushed
```

```
# in the `pixelpotato` repository, with the `python-helloworld` image name
# and version `v1.0.0`
docker tag python-helloworld pixelpotato/python-helloworld:v1.0.0
```

Once the image is tagged, the final step is to push the image to a registry. For this purpose, the docker push command can be used. Below is the syntax for this command:

```
# push an image to a registry
# NAME[:TAG] - required and the tag is optional; name, set the image name
to be pushed to the registry
docker push NAME[:TAG]
```

For example, to push the Python hello-world application to DockerHub, the following command can be used:

```
# push the `python-helloworld` application in version v1.0.0
# to the `pixelpotato` repository in DockerHub
docker push pixelpotato/python-helloworld:v1.0.0
```

New terms

- **Dockerfile** - set of instructions used to create a Docker image
- **Docker image** - a read-only template used to spin up a runnable instance of an application
- **Docker registry** - a central mechanism to store and distribute Docker images

Further reading

Explore Dockerfiles best practices and valid list of instructions:

- Dockerfile reference
- Best practices for writing Dockerfiles

Explore how to build and run a Docker image, with a list of all available options:

- Docker Build command
- Docker Run command

Explore Docker registries, alternatives to package an application, and OCI standards:

- Introduction to Docker registry
- Docker Tag command
- Docker Push command
- Demystifying the Open Container Initiative (OCI) Specifications
- Buildpacks: An App's Brief Journey from Source to Image

Docker Walkthrough

This demo provides a step-by-step walkthrough of how to package, build, run, tag, and push a Docker image. You can follow this demo by referencing the Dockerfile from the course repository.

Useful Docker Commands

Summary

Docker provides a rich set of actions that can be used to build, run, tag, and push images. Below is a list of handy Docker commands used in practice.

Note: In the following commands the following arguments are used:

- **OPTIONS** - define extra configuration through flags
- **IMAGE** - sets the name of the image
- **NAME**- set the name of the image
- **COMMAND** and **ARG** - instruct the container to run specific commands associated with a set of arguments

Build Images

To build an image, use the following command, where PATH sets the location of the Dockerfile and referenced application files:

docker build [OPTIONS] PATH

Run Images

To run an image, use the following command:

docker run [OPTIONS] IMAGE [COMMAND] [ARG...]

Get Logs

To get the logs from a Docker container, use the following command:

docker logs CONTAINER_ID

Where the CONTAINER_ID is the ID of the Docker container that runs an application.

List Images

To list all available images, use the following command:

docker images

List Containers

To list all running containers, use the following command:

docker ps

Tag Images

To tag an image, use the following command, where SOURCE_IMAGE defines the name of an image on the current machine and TARGET_IMAGE defines the repository, name, and version of an image:

docker tag SOURCE_IMAGE[:TAG] TARGET_IMAGE[:TAG]

Login to DockerHub

To login into DockerHub, use the following command:

docker login

Push Images

To push an image to DockerHub, use the following command:

docker push NAME[:TAG]

Pull Images

To pull an image from DockerHub, use the following command:

docker pull NAME[:TAG]

Quizzes: Docker for Application Packaging

Exercise: Docker for Application Packaging

Package a Go web application using Docker capabilities. This exercise will involve the creation of a Docker image and pushing it to a public image registry, such as DockerHub.

Note: You will require a valid DockerHub account.

Environment Setup

Set up your environment to create a Docker image for an application:

Task List

-

Once you can access the application through the local browser, the next steps are to package the application using Docker.

Exercise

Create the Docker image for the Go web application and push it to DockerHub, considering the following requirements:

Dockerfile:

- use the golang:alpine base image
- set the working directory to /go/src/app
- make sure to copy all the files from the current directory to the container working directory (e.g. /go/src/app)
- to build the application, use go build -o helloworld command, where -o helloworld will create the binary of the application with the name helloworld
- the application should be accessible on port 6111
- and lastly, the command to start the container is to invoke the binary created earlier, which is ./helloworld

Docker image:

- should have the name go-helloworld
- should have a valid tag, and a version with a major, minor, and patch included
- should be available in DockerHub, under your username e.g. pixelpotato/go-helloworld

Docker container:

- should be running on your local machine, by referencing the image from the DockerHub with a valid tag e.g. pixelpotato/go-helloworld:v5.12.3

Note: *You will need to use docker login to login into Docker before pushing images to DockerHub.*

Solution: Docker for Application Packaging

The following snippet showcases the Dockerfile for the application:

```
FROM golang:alpine

WORKDIR /go/src/app

ADD . .
```

```
RUN go build  -o helloworld

EXPOSE 6111

CMD ["./helloworld"]
```

To build tag and push the image to DockerHub, use the following commands:

```
# build the image
docker build -t go-helloworld .

# tag the image
docker tag go-helloworld pixelpotato/go-helloworld:v1.0.0

# push the image
docker push pixelpotato/go-helloworld:v1.0.0
```

Kubernetes - The Container Orchestrator Framework

Summary

So far, in this lesson, we have traversed the packaging of an application using Docker and its distribution through DockerHub. The next phase in the release process is the deployment of the service. However, running an application in production implies that thousands and millions of customers might consume the product at the same time. For this reason, it is paramount to build for scale. It is impossible to manually manage thousands of containers, keeping these are up to date with the latest code changes, in a healthy state, and accessible. As a result, a **container orchestrator framework** is necessary.

A container orchestrator framework is capable to create, manage, configure thousands of containers on a set of distributed servers while preserving the connectivity and reachability of these containers. In the past years, multiple tools emerged within the landscape to provide these capabilities, including Docker Swarm, Apache Mesos, CoreOS Fleet, and many more. However, **Kubernetes** took the lead in defining the principles of how to run containerized workloads on a distributed amount of machines.

Kubernetes is widely adopted in the industry today, with most organizations using it in production. Kubernetes currently is a graduated CNCF project, which highlights its maturity and

reported success from end-user companies. This is because Kubernetes solutionizes portability, scalability, resilience, service discovery, extensibility, and operational cost of containers.

Portability

Kubernetes is a highly portable tool. This is due to its open-source nature and vendor agnosticism. As such, Kubernetes can be hosted on any available infrastructure, including public, private, and hybrid cloud.

Scalability

Building for scale is a cornerstone of any modern infrastructure, enabling an application to scale based on the amount of incoming traffic. Kubernetes has in-build resources, such as HPA (Horizontal Pod Autoscaler), to determine the required amount of replicas for a service. Elasticity is a core feature that is highly automated within Kubernetes.

Resilience

Failure is expected on any platform. However, it is more important to be able to recover from failure fast and build a set of playbooks that minimizes the downtime of an application. Kubernetes uses functionalities like ReplicaSet, readiness, and liveness probes to handle most of the container failures, which enables powerful self-healing capability.

Service Discovery

Service discovery encapsulates the ability to automatically identify and reach new services once these are available. Kubernetes provide cluster level DNS (or Domain Name System), which simplifies the accessibility of workloads within the cluster. Additionally, Kubernetes provides routing and load balancing of incoming traffic, ensuring that all requests are served without application overload.

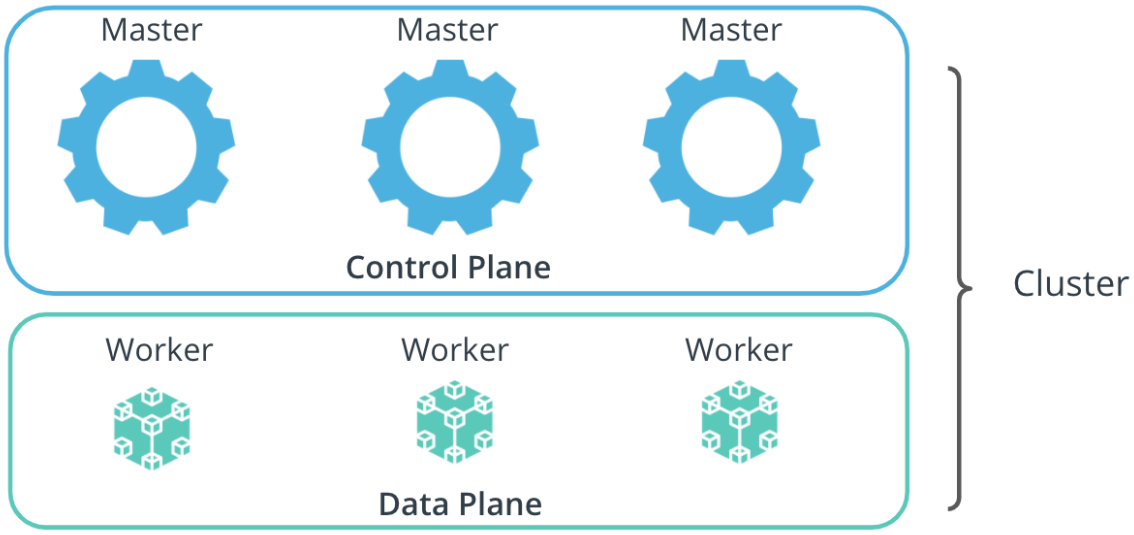
Extensibility

Kubernetes is a highly extensible mechanism that uses the building-block principle. It has a set of basic resources that can be easily adjusted. Additionally, it provides a rich API interface, that can be extended to accommodate new resources or CRDs (Custom Resource Definitions).

Operational Cost

Operational cost refers to the efficiency of resource consumption within a Kubernetes cluster, such as CPU and memory. Kubernetes has a powerful scheduling mechanism that places an application on the node with sufficient resources to ensure the successful execution of the service. As a result, most of the available infrastructure resources are allocated on-demand. Additionally, it is possible to automatically scale the size of the cluster based on the current incoming traffic. This capability is provisioned by the cluster-autoscaler, which guarantees that the cluster size is directly proportional to the traffic that it needs to handle.

Kubernetes Architecture

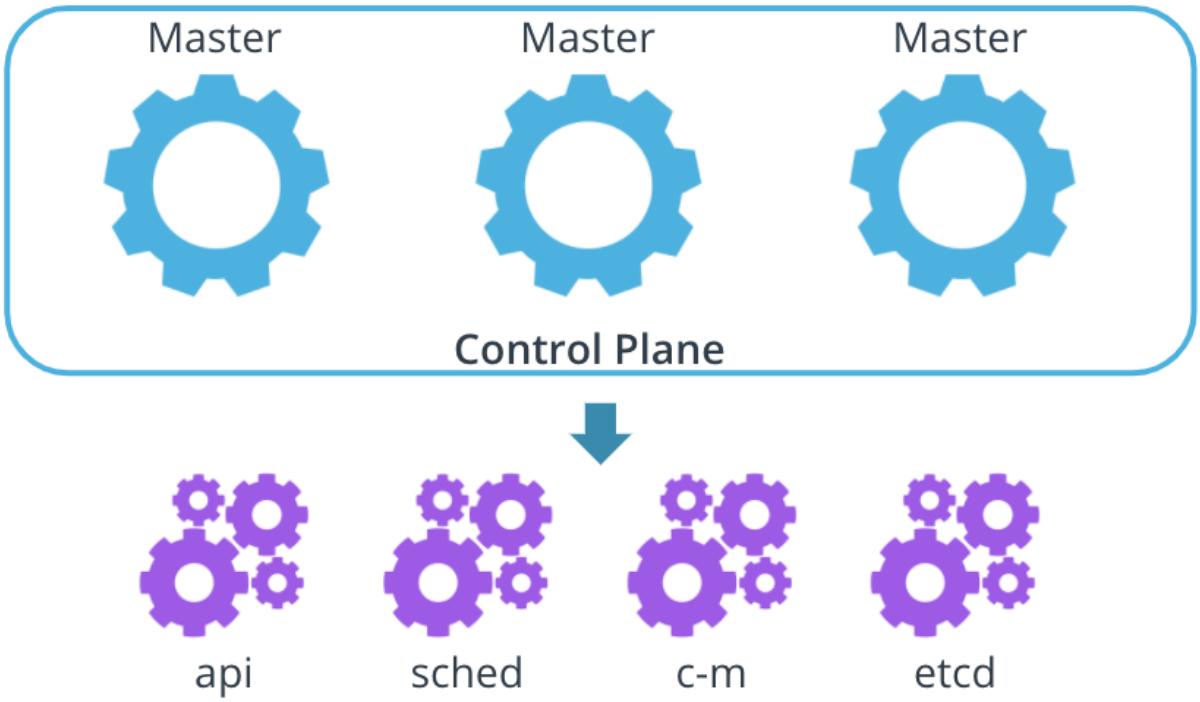


Kubernetes architecture, composed of control and data planes

A Kubernetes cluster is composed of a collection of distributed physical or virtual servers. These are called **nodes**. Nodes are categorized into 2 main types: master and worker nodes. The components installed on a node, determine the functionality of a node, and identifies it as a master or worker node.

The suite of master nodes, represents the **control plane**, while the collection of worker nodes constructs the **data plane**.

Control Plane



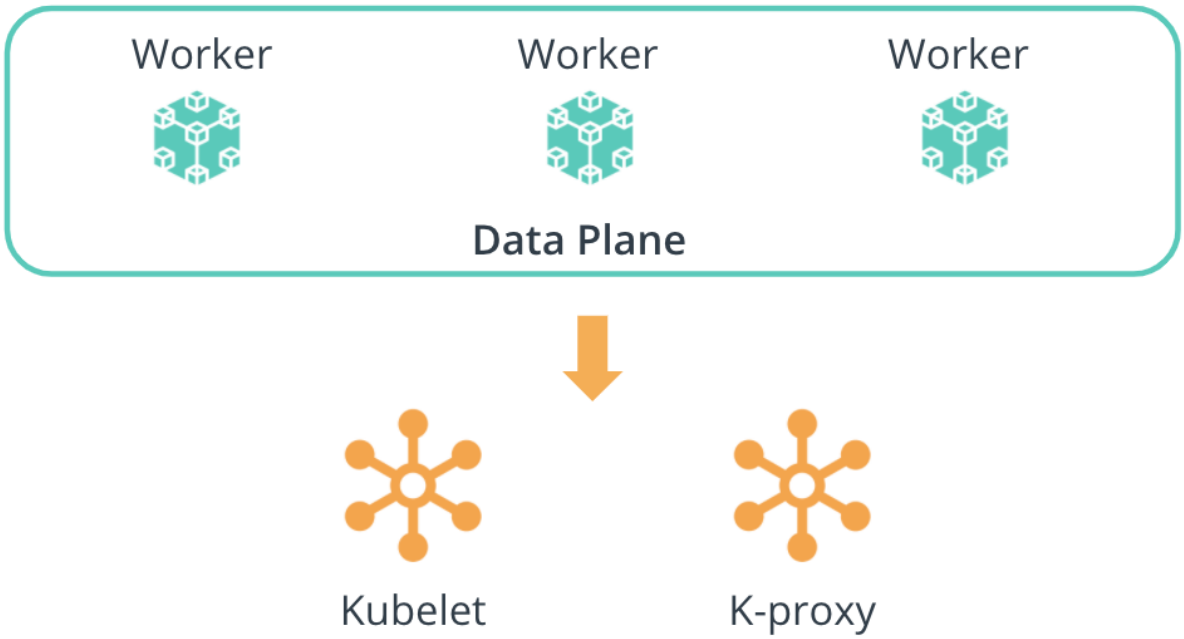
Control Plane components

The control plane consists of components that make global decisions about the cluster. These components are the:

- **kube-apiserver** - the nucleus of the cluster that exposes the Kubernetes API, and handles and triggers any operations within the cluster
- **kube-scheduler** - the mechanism that places the new workloads on a node with sufficient satisfactory resource requirements
- **kube-controller-manager** - the component that handles controller processes. It ensures that the desired configuration is propagated to resources
- **etcd** - the key-value store, used for backs-up and keeping manifests for the entire cluster

There are two additional components on the control plane, they are **kubelet** and **k-proxy**. These two are special and important as they are installed on **all** node. You can see the Data Plane below for more details.

Data Plane



Data Plane conponents

The data plane consists of the compute used to host workloads. The components installed on a worker node are the:

- **kubelet** - the agent that runs on **every** node and notifies the *kube- apiserver* that this node is part of the cluster
- **kube-proxy** - a network proxy that ensures the reachability and accessibility of workloads places on this specific node

Important Note: The **kubelet** and **kube-proxy** components are installed on **all** the nodes in the cluster (**master and worker** nodes). These components keep the *kube-apiserver* up-to-date with a list of nodes in the cluster and manages the connectivity and reachability of the workloads.

New terms

- **CRD** - Custom Resource Definition provides the ability to extend Kubernetes API and create new resources
- **Node** - a physical or virtual server
- **Cluster** - a collection of distributed nodes that are used to manage and host workloads
- **Master node** - a node from the Kubernetes control plane, that has installed components to make global, cluster-level decisions
- **Worker node** - a node from the Kubernetes data plane, that has installed components to host workloads

Further Reading

Explore Kubernetes features:

- Kubernetes DNS for Services and Pods
- Kubernetes CRDs
- Kubernete Cluster Autoscaler
- Kubernetes Architecture and Components

Deploy Your First Kubernetes Cluster

Cluster Creation

Summary

Provisioning a Kubernetes cluster is known as the bootstrapping process. When creating a cluster, it is essential to ensure that each node had the necessary components installed. It is possible to manually provision a cluster, however, this implied the distribution and execution of each component independently (e.g. kube-apiserver, kube-scheduler, kubelet, etc.). This is a highly tedious task that has a higher risk of misconfiguration.

As a result, multiple tools emerged to handle the bootstrapping of a cluster automatically. For example:

- Production-grade clusters
  - kubeadm
  - Kubespray
  - Kops
  - K3s
- Developmnet-grade clusters
  - kind
  - minikube
  - k3d

A good introduction to k3d, written by k3d's creator Thorsten Klein, can be found here

A comprehensive overview of currently existing lightweight kubernetes distros can be found here

Create Vagrant Box And Install Kubernetes with k3s

This demo is a step-by-step guide on how to create a vagrant box and install a Kubernetes cluster using k3s. To follow this demo, reference the Vagrantfile from the course repository.

A nice introduction to Vagrant can be found in this article

Before following the demo:

1. Make sure to have VirtualBox 6.1.16 or higher installed.
2. You also need to install vagrant on your machine.

Throughout the demo, the following kubectl commands are used:

```
# Inspect available vagrant boxes
vagrant status

# create a vagrant box using the Vagrantfile in the current directory
vagrant up

# SSH into the vagrant box
# Note: this command uses the .vagrant folder to identify the details of the vagrant box
vagrant ssh
```

New terms

- **Bootstrap** - the process of provisioning a Kubernetes cluster, by ensuring that each node has the necessary components to be fully operational

Further reading

- Bootstrapping clusters with kubeadm - a step-by-step guide on how to use kubeadm to provision a cluster

Kubeconfig

Summary

To access a Kubernetes cluster a **kubeconfig** file is required. A kubeconfig file has all the necessary cluster metadata and authentication details, that grants the user permission to query the cluster objects. Usually, the kubeconfig file is stored locally under the ~/.kube/config file. However, k3s places the kubeconfig file within /etc/rancher/k3s/k3s.yaml path. Additionally, the

location of a kubeconfig file can be set through the --kubeconfig kubectl flag or via the KUBECONFIG environmental variable.

A Kubeconfig file has 3 main distinct sections:

- **Cluster** - encapsulates the metadata for a cluster, such as the name of the cluster, API server endpoint, and certificate authority used to check the identity of the user.
- **User** - contains the user details that want access to the cluster, including the user name, and any authentication metadata, such as username, password, token or client, and key certificates.
- **Context** - links a user to a cluster. If the user credentials are valid and the cluster is up, access to resources is granted. Also, a current-context can be specified, which instructs which context (cluster and user) should be used to query the cluster.

Here is an example of a kubeconfig file:

```
apiVersion: v1
# define the cluster metadata
clusters:
- cluster:
    certificate-authority-data: {{ CA }}
    server: https://127.0.0.1:63668
  name: udacity-cluster
# define the user details
users:
# `udacity-user` user authenticates using client and key certificates
- name: udacity-user
  user:
    client-certificate-data: {{ CERT }}
    client-key-data: {{ KEY }}
# `green-user` user authenticates using a token
- name: green-user
  user:
    token: {{ TOKEN }}
# define the contexts
contexts:
- context:
    cluster: udacity-cluster
    user: udacity-user
  name: udacity-context
# set the current context
current-context: udacity-context
```

Once you start handling multiple clusters, you'll find a lot of useful information in [this article]

Kubeconfig Walkthrough

In this demo, the instructor uses a cluster bootstrapped with kind. Throughout this course, the students will use k3s to provision a cluser. However, in this demo *kind* is used to highlight how different tools provision the kubeconfig files.

If the students chose to follows this demo, these are the instructions to create a cluster using *kind*:

Note: kind can be installed directly on your local machine

- Ensure Docker is installed and running. Use the docker --version command to verify if Docker is installed.
- Install kind by using the official installation documentation
- Create a kind cluster using the kind create cluster --name demo command

Throughout the demo, the following kubectl commands are used:

```
# Inspect  the endpoints for the cluster and installed add-ons
kubectl cluster-info

# List all the nodes in the cluster.
# To get a more detailed view of the nodes, the `-o wide` flag can be passed
kubectl get nodes [-o wide]

# Describe a cluster node.
# Typical configuration: node IP, capacity (CPU and memory), a list of running pods on the node, podCIDR, etc.
kubectl describe node {{ NODE NAME }}
```

New terms

- **Kubeconfig** - a metadata file that grants a user access to a Kubernetes cluster

Further reading

- Organizing Cluster Access Using kubeconfig Files

Exercise: Deploy Your First Kubernetes Cluster

So far, you have learned that Kubernetes is a mechanism to manage containers at scale and that there are more than 160+ certified providers to bootstrap a cluster.

Now it's time for you to deploy your first Kubernetes cluster.

This exercise will focus on provisioning a vagrant box and installing a Kubernetes cluster using k3s.

Solution: Deploy Your First Kubernetes Cluster

The kubeconfig file and kubectl commands are the 2 main components that permits the interaction with a Kubernetes cluster.

Let's take a closer look at cluster configuration details.

- kubeconfig
    - K3s stores the kubeconfig file under /etc/rancher/k3s/k3s.yaml path
    - API server - https://127.0.0.1:6443
    - authentication mechanism - username (admin) and password
- kubectl commands

kubectl cluster-info to get the control plane and add-ons endpoints

Kubernetes master is running at https://127.0.0.1:6443
CoreDNS is running at
https://127.0.0.1:6443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy
Metrics-server is running at
https://127.0.0.1:6443/api/v1/namespaces/kube-system/services/https:metrics-server:/proxy

    -

kubectl get nodes - to get all the nodes in the cluster

NAME STATUS ROLES AGE VERSION
localhost Ready master 74m v1.18.9+k3s1

    -

kubectl get nodes -o wide - to get extra details about the nodes, including internal IP

```
 NAME        STATUS   ROLES    AGE   VERSION        INTERNAL-IP   EXTERNAL-IP
OS-IMAGE             KERNEL-VERSION            CONTAINER-RUNTIME
localhost   Ready    master   74m   v1.18.9+k3s1   10.0.2.15     <none>        openSUSE Leap
15.2   5.3.18-lp152.47-default   containerd://1.3.3-k3s2
```

-

kubectl describe node node-name - to get all the configuration details about the node, including the allocated pod CIDR

```
 kubectl describe node localhost | grep CIDR
PodCIDR:                      10.42.0.0/24
PodCIDRs:                     10.42.0.0/24
```

-

Kubernetes Resources Part 1

Summary

Up to this stage, we have explored how to package an application using Docker and how to bootstrap a cluster using k3s. In the next phase, we need to deploy the packaged application to a Kubernetes cluster.

Kubernetes provides a rich collection of resources that are used to deploy, configure, and manage an application. Some of the widely used resources are:

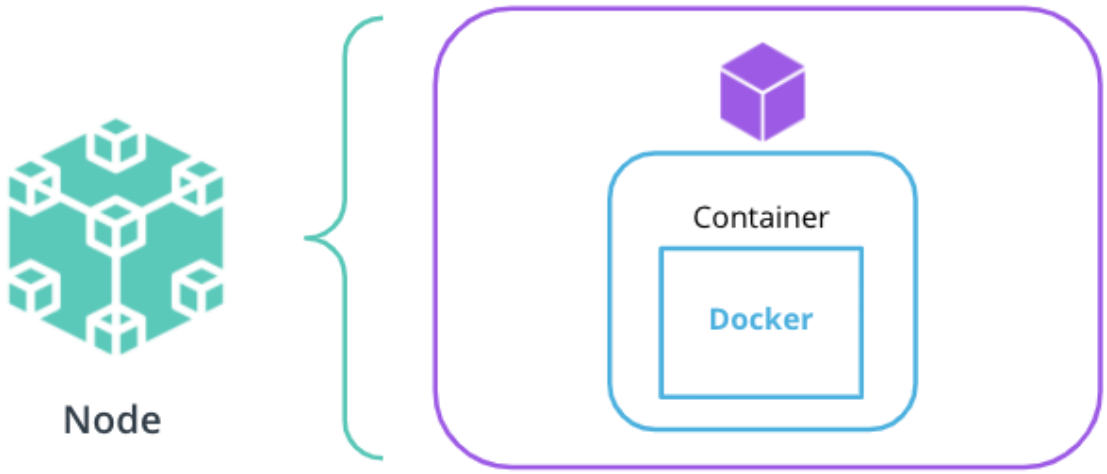
- Pods - the atomic element within a cluster to manage an application
- Deployments & ReplicaSets - oversees a set of pods for the same application
- Services & Ingress - ensures connectivity and reachability to pods
- Configmaps & Secrets - pass configuration to pods
- Namespaces - provides a logical separation between multiple applications and their resources
- Custom Resource Definition (CRD) - extends Kubernetes API to support custom resources

Application Deployment

Summary

A **pod** is the anatomic element within a cluster that provides the execution environment for an application. Pods are the smallest manageable units in a Kubernetes cluster. Every pod has a container within it, that executes an application from a Docker image (or any OCI-compliant image). There are use cases where 2-3 containers run within the same pod, however, it is highly recommended to keep the 1:1 ratio between your pods and containers.

All the pods are placed on the cluster nodes. A note can host multiple pods for different applications.



Pod architecture, showcasing a container running a Docker image

Deployments and ReplicaSets

Application management using a Deployment and ReplicaSet

To deploy an application to a Kubernetes cluster, a **Deployment** resource is necessary. A Deployment contains the specifications that describe the desired state of the application. Also, the Deployment resource manages pods by using a **ReplicaSet**. A ReplicaSet resource ensures that the desired amount of replicas for an application are up and running at all times.

To create a deployment, use the kubectl create deployment command, with the following syntax:

```
# create a Deployment resource
# NAME - required; set the name of the deployment
# IMAGE - required;  specify the Docker image to be executed
# FLAGS - optional; provide extra configuration parameters for the resource
# COMMAND and args - optional; instruct the container to run specific commands when it starts
kubectl create deploy NAME --image=image [FLAGS] -- [COMMAND] [args]

# Some of the widely used FLAGS are:
-r, --replicas - set the number of replicas
-n, --namespace - set the namespace to run
--port - expose the container port
```

For example, to create a Deployment for the Go hello-world application, the following command can be used:

```
# create a go-helloworld Deployment in namespace `test`
kubectl create deploy go-helloworld --image=pixelpotato/go-helloworld:v1.0.0 -n test
```

It is possible to create headless pods or pods that are not managed through a ReplicaSet and Deployment. Though it is not recommended to create standalone pods, these are useful when creating a testing pod.

To create a headless pod, the kubectl run command is handy, with the following syntax:

```
# create a headless pod
# NAME - required; set the name of the pod
# IMAGE - required;  specify the Docker image to be executed
# FLAGS - optional; provide extra configuration parameters for the resource
# COMMAND and args - optional; instruct the container to run specific commands when it starts
kubectl run NAME --image=image [FLAGS] -- [COMMAND] [args...]

# Some of the widely used FLAGS are:
--restart - set the restart policy. Options [Always, OnFailure, Never]
--dry-run - dry run the command. Options [none, client, server]
-it - open an interactive shell to the container
```

For example, to create a busybox pod, the following command can be used:

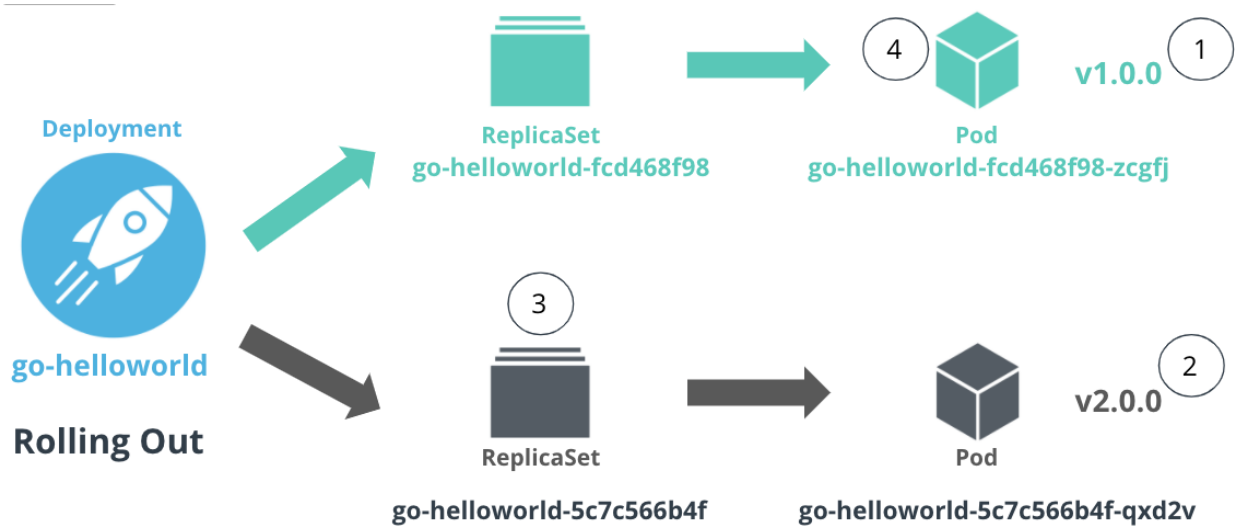
```
# example: create a busybox pod, with an interactive shell and a restart policy set to Never
kubectl run -it busybox-test --image=busybox --restart=Never
```

Rolling Out Strategy

The Deployment resource comes with a very powerful rolling out strategy, which ensures that no downtime is encountered when a new version of the application is released. Currently, there are 2 rolling out strategies:

- RollingUpdate - updates the pods in a rolling out fashion (e.g. 1-by-1)
- Recreate - kills all existing pods before new ones are created

For example, in this case, we upgrade a Go hello-world application from version 1.0.0 to version 2.0.0:



Rolling update of an application, between different versions

Where:

1. The Go hello-world application is running version v1.0.0 in a pod managed by a ReplicaSet
2. The version of Go hello-world application is set to v2.0.0
3. A new ReplicaSet is created that controls a new pod with the application running in version v2.0.0
4. The traffic is directed to the pod running v2.0.0 and the pod with the old configuration (v1.0.0) is removed

Application Development Demo

Summary

This demo showcases how an application can be deployed, configured, and managed within a Kubernetes cluster using Deployment, ReplicaSet, and pod resources.

The instructor uses the Go hello-world application in version v1.0.0 and v2.0.0. The difference between these 2 versions is the exposed port by the application. Below are the code snippets for both application versions (you can also refer to the go-hellowolrd application from the course repository):

```
# Application version: v1.0.0
# port exposed: 6111
package main

import (
    "fmt"
    "net/http"
)

func helloWorld(w http.ResponseWriter, r *http.Request){
    fmt.Fprintf(w, "Hello World")
}

func main() {
    http.HandleFunc("/", helloWorld)
    http.ListenAndServe(":6111", nil)
}
```

```
# Application version: v2.0.0
# port exposed: 6112
package main

import (
    "fmt"
    "net/http"
)

func helloWorld(w http.ResponseWriter, r *http.Request){
    fmt.Fprintf(w, "Hello World")
}

func main() {
    http.HandleFunc("/", helloWorld)
    http.ListenAndServe(":6112", nil)
}
```

New terms

- **Pod** - smallest manageable uint within a cluster that provides the execution environment for an application
- **ReplicaSet** - a mechanism to ensure a number of pod replicas are up and running at all times
- **Deployment** - describe the desired state of the application to be deployed

Further reading

Explore the Kubernetes resources in more detail:

- Kubernetes Pods
- Kubernetes Deployments
- Kubernetes ReplicaSets
- Kubernetes RollingOut Strategies

Kubernetes Resources Part 2

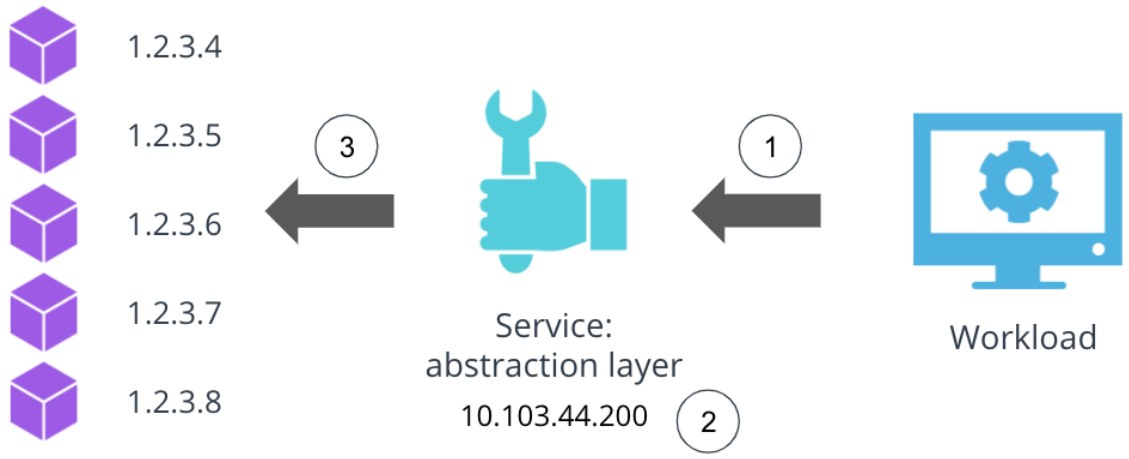
Application Reachability

Summary

Within a cluster, every pod is allocated 1 unique IP which ensures connectivity and reachability to the application inside the pod. This IP is only routable inside the cluster, meaning that external users and services will not be able to connect to the application.

For example, we can connect a workload within the cluster to access a pod directly via its IP. However, if the pod dies, all future requests will fail, as these are routes to an application that is not running. The remediation step is to configure the workload to communicate with a different pod IP. This is a highly manual process, which brings complexity to the accessibility of an application. To automate the reachability to pods, a Service resource is necessary.

Services



Pods accessibility through a Service resource

A **Service** resource provides an abstraction layer over a collection of pods running an application. A Service is allocated a cluster IP, that can be used to transfer the traffic to any available pods for an application.

As such, as shown in the above image, instead of accessing each pod independently, the workload (1) should access the service IP (2), which routes the requests to available pods (3).

There are 3 widely used Service types:

- **ClusterIP** - exposes the service using an internal cluster IP. If no service type is specified, a ClusterIP service is created by default.
- **NodePort** - expose the service using a port exposed on all nodes in the cluster.
- **LoadBalancer** - exposes the service through a load balancer from a public cloud provider such as AWS, Azure, or GCP. This will allow the external traffic to reach the services within the cluster securely.

To create a service for an existing deployment, use the kubectl expose deployment command, with the following syntax:

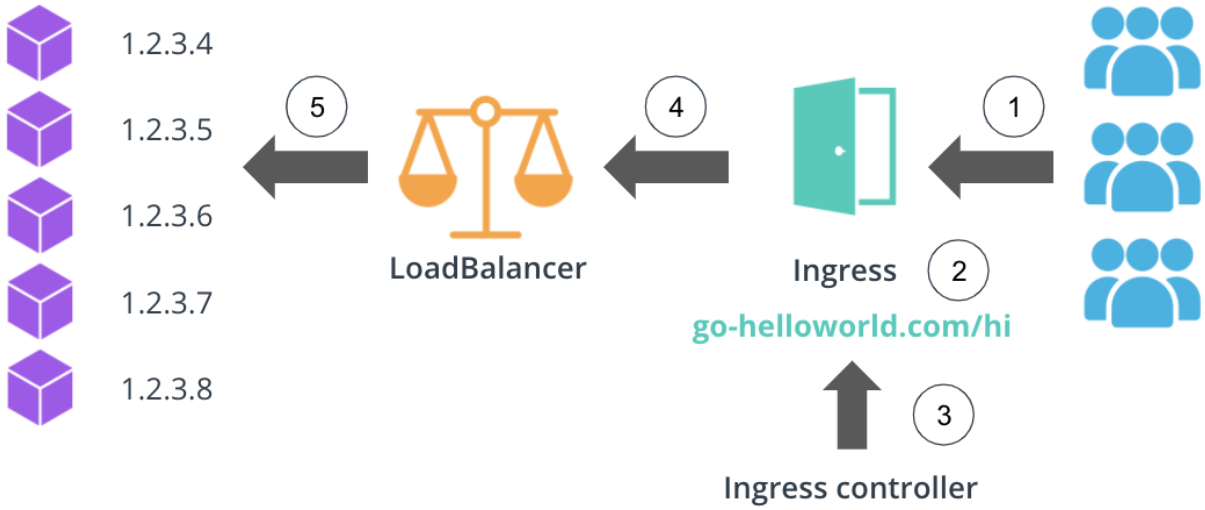
```
# expose a Deployment through a Service resource
# NAME - required; set the name of the deployment to be exposed
# --port - required; specify the port that the service should serve on
# --target-port - optional; specify the port on the container that the service should direct traffic to
# FLAGS - optional; provide extra configuration parameters for the service
kubectl expose deploy NAME --port=port [--target-port=port] [FLAGS]

# Some of the widely used FLAGS are:
--protocol - set the network protocol. Options [TCP|UDP|SCTP]
--type - set the type of service. Options [ClusterIP, NodePort, LoadBalancer]
```

For example, to expose the Go hello-world application through a service, the following command can be used:

```
# expose the `go-helloworld` deployment on port 8111
# note: the application is serving requests on port 6112
kubectl expose deploy go-helloworld --port=8111 --target-port=6112
```

Ingress



Ingress resources enabling access from the external users to services within the cluster

To enable the external user to access services within the cluster an **Ingress** resource is necessary. An Ingress exposes HTTP and HTTPS routes to services within the cluster, using a load balancer provisioned by a cloud provider. Additionally, an Ingress resource has a set of rules that are used to map HTTP(S) endpoints to services running in the cluster. To keep the Ingress rules and load balancer up-to-date an Ingress Controller is introduced.

For example, as shown in the image above, the customers will access the *go-helloworld.com/hi* HTTP route (1), which is managed by an Ingress (2). The Ingress Controller (3) examines the configured routes and directs the traffic to a LoadBalancer (4). And finally, the LoadBalancer directs the requests to the pods using a dedicated port (5).

Application Reachability Demo

Summary

This demo provides a step-by-step guide on how to expose the Go hello-world application through a ClusterIP service. Additionally, an *alpine* pod is used to showcase how a workload

can connect to the Go hello-world application through the service IP and port from within the cluster.

New terms

- **Service** - an abstraction layer over a collection of pods running an application
- **Ingress** - a mechanism to manage the access from external users and workloads to the services within the cluster

Further reading

Explore Kubernetes resources used to connect to an application:

- Kubernetes Services
- Kubernetes Ingress

Kubernetes Resources Part 3

Application Configuration And Context

Summary

In the implementation phase, a good development practice is to separate the configuration from the source code. This increased the portability of an application as it can cover multiple customer use cases. Kubernetes has 2 resources to pass data to an application: Configmaps and Secrets.

ConfigMaps

ConfigMaps are objects that store non-confidential data in key-value pairs. A Configmap can be consumed by a pod as an environmental variable, configuration files through a volume mount, or as command-line arguments to the container.

To create a Configmap use the kubectl create configmap command, with the following syntax:

```
# create a Configmap
# NAME - required; set the name of the configmap resource
# FLAGS - optional; define  extra configuration parameters for the configmap
kubectl create configmap NAME [FLAGS]
```

```
# Some of the widely used FLAGS are:
--from-file - set path to file with key-value pairs
--from-literal - set key-value pair from command-line
```

For example, to create a Configmap to store the background color for a front-end application, the following command can be used:

```
# create a Configmap to store the color value
kubectl create configmap test-cm --from-literal=color=yellow
```

Secrets

Secrets are used to store and distribute sensitive data to the pods, such as passwords or tokens. Pods can consume secrets as environment variables or as files from the volume mounts to the pod. It is noteworthy, that Kubernetes will encode the secret values using base64.

To create a Secret use the kubectl create secret generic command, with the following syntax:

```
# create a Secret
# NAME - required; set the name of the secret resource
# FLAGS - optional; define  extra configuration parameters for the secret
kubectl create secret generic NAME [FLAGS]
```

```
# Some of the widely used FLAGS are:
--from-file - set path to file with the sensitive key-value pairs
--from-literal - set key-value pair from command-line
```

For example, to create a Secret to store the secret background color for a front-end application, the following command can be used:

```
# create a Secret to store the secret color value
kubectl create secret generic test-secret --from-literal=color=blue
```

Namespaces

A Kubernetes cluster is used to host hundreds of applications, and it is required to have separate execution environments across teams and business verticals. This functionality is provisioned by the **Namespace** resources. A Namespace provides a logical separation between multiple applications and associated resources. In a nutshell, it provides the **application context**, defining the *environment* for a group of Kubernetes resources that relate to a project, such as the amount of CPU, memory, and access. For example, a project-green namespace

includes any resources used to deploy the Green Project. These resources construct the application context and can be managed collectively to ensure a successful deployment of the project.

Each team or business vertical is allocated a separate Namespace, with the desired amount of CPU, memory, and access. This ensures that the application is managed by the owner team and has enough resources to execute successfully. This also eliminates the "noisy neighbor" use case, where a team can consume all the available resources in the cluster if no Namespace boundaries are set.

To create a Namespace we can use the kubectl create namespace command, with the following syntax:

```
# create a Namespace
# NAME - required; set the name of the Namespace
kubectl create ns NAME
```

For example, to create a test-udacity Namespace, the following command can be used:

```
# create a `test-udacity` Namespace
kubectl create ns test-udacity

# get all the pods in the `test-udacity` Namespace
kubectl get po -n test-udacity
```

Demo - Application Configuration

Summary

This demo showcases how Configmap and Secret resources can be created using literal values from the command line.

Demo - Application Context

Summary

This demo is a step-by-step guide on how to create a Namespace resource and retrieve resources from a specific Namespace.

New terms

- **Configmap** - a resource to store non-confidential data in key-value pairs.
- **Secret** - a resource to store confidential data in key-value pairs. These are base64 encoded.
- **Namespace** - a logical separation between multiple applications and associated resources.

Further reading

Explore Kubernetes resources to pass configuration to pods:

- Kubernetes Configmap
- Kubernetes Secrets
- Kuebrnetes Namespaces

Useful kubectl Commands

Kubectl provides a rich set of actions that can be used to interact, manage, and configure Kubernetes resources. Below is a list of handy kubectl commands used in practice.

Note: In the following commands the following arguments are used:

- **RESOURCE** is the Kubernetes resource type
- **NAME** sets the name of the resource
- **FLAGS** are used to provide extra configuration
- **PARAMS** are used to provide the required configuration to the resource

Create Resources

To create resources, use the following command:

kubectl create RESOURCE NAME [FLAGS]

Describe Resources

To describe resources, use the following command:

kubectl describe RESOURCE NAME

Get Resources

To get resources, use the following command, where -o yaml instructs that the result should be YAML formated.

kubectl get RESOURCE NAME [-o yaml]

Edit Resources

To edit resources, use the following command, where -o yaml instructs that the edit should be YAML formated.

kubectl edit RESOURCE NAME [-o yaml]

Label Resources

To label resources, use the following command:

kubectl label RESOURCE NAME [PARAMS]

Port-forward to Resources

To access resources through port-forward, use the following command:

kubectl port-forward RESOURCE/NAME [PARAMS]

Logs from Resources

To access logs from a resource, use the following command:

kubectl logs RESOURCE/NAME [FLAGS]

Delete Resources

To delete resources, use the following command:

kubectl delete RESOURCE NAME

Exercise: Kubernetes Resources

Now you have learned many Kubernetes recourses, in this exercise, you will deploy the following resources using the kubectl command.

- a namespace

    - name: demo
    - label: tier: test
- a deployment:
    - image: nginx:alpine
    - name:nginx-apline
    - namespace: demo
    - replicas: 3
    - labels: app: nginx, tag: alpine
- a service:
    - expose the above deployment on port 8111
    - namespace: demo
- a configmap:
    - name: nginx-version
    - containing key-value pair: version=alpine
    - namespace: demo

Note: Nginx is one of the public Docker images, that you can access and use for your exercises or testing purposes.

Make sure the following tasks are completed:

Task List

-

Solution: Kubernetes Resources

Below is a snippet creating a namespace and labeling it, a deployment, a service, and a configmap using the kubectl operations.

```
# create the namespace
# note: label option is not available with `kubectl create`
kubectl create ns demo

# label the namespace
kubectl label ns demo tier=test

# create the nginx-alpine deployment
kubectl create deploy nginx-alpine --image=nginx:alpine  --replicas=3 --namespace demo
```

```
# label the deployment
kubectl label deploy nginx-alpine app=nginx tag=alpine --namespace demo

# expose the nginx-alpine deployment, which will create a service
kubectl expose deployment nginx-alpine --port=8111 --namespace demo

# create a config map
kubectl create configmap nginx-version --from-literal=version=alpine --namespace demo
```

Spoiler alert: in the next section, you will learn and practice how to deploy Kubernetes resources using a different approach.

Declarative Kubernetes Manifests

Summary

Kubernetes is widely known for its support for imperative and declarative management techniques. The **imperative** approach enables the management of resources using kubectl commands directly on the live cluster. For example, all the commands you have practiced so far (e.g. kubectl create deploy [...]) are using the imperative approach. This technique is best suited for development stages only, as it presents a low entry-level bar to interact with the cluster.

On the other side, the **declarative** approach uses manifests stored locally to create and manage Kubertenest objects. This approach is recommended for production releases, as we can version control the state of the deployed resources. However, this technique presents a high learning curve, as an in-depth understanding of the YAML manifest structure is required. Additionally, using YAML manifests unlocks the possibility of configuring more advanced options, such as volume mounts, readiness and liveness probes, etc.

YAML Manifest structure

A YAML manifest consists of 4 obligatory sections:

- **apiversion** - API version used to create a Kubernetes object
- **kind** - object type to be created or configured
- **metadata** - stores data that makes the object identifiable, such as its name, namespace, and labels
- **spec** - defines the desired configuration state of the resource

To get the YAML manifest of any resource within the cluster, use the kubectl get command, associated with the -o yaml flag, which requests the output in YAML format. Additionally, to

explore all configurable parameters for a resource it is highly recommended to reference the official Kubernetes documentation.

Deployment YAML manifest

In addition to the required sections of a YAML manifest, a Deployment resource covers the configuration of ReplicaSet, RollingOut strategy, and containers. Bellow is a full manifest of a Deployment explaining each parameter:

```
## Set the API endpoint used to create the Deployment resource.
apiVersion: apps/v1
## Define the type of the resource.
kind: Deployment
## Set the parameters that make the object identifiable, such as its name, namespace, and
labels.
metadata:
  annotations:
  labels:
    app: go-helloworld
  name: go-helloworld
  namespace: default
## Define the desired configuration for the Deployment resource.
spec:
  ## Set the number of replicas.
  ## This will create a ReplicaSet that will manage 3 pods of the Go hello-world application.
  replicas: 3
  ## Identify the pods managed by this Deployment using the following selectors.
  ## In this case, all pods with the label `go-helloworld`.
  selector:
    matchLabels:
      app: go-helloworld
  ## Set the RollingOut strategy for the Deployment.
  ## For example, roll out only 25% of the new pods at a time.
  strategy:
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
    type: RollingUpdate
  ## Set the configuration for the pods.
  template:
    ## Define the identifiable metadata for the pods.
    ## For example, all pods should have the label `go-helloworld`
    metadata:
      labels:
        app: go-helloworld
```

```
  ## Define the desired state of the pod configuration.
  spec:
    containers:
      ## Set the image to be executed inside the container and image pull policy
      ## In this case, run the `go-helloworld` application in version v2.0.0 and
      ## only pull the image if it's not available on the current host.
    - image: pixelpotato/go-helloworld:v2.0.0
      imagePullPolicy: IfNotPresent
      name: go-helloworld
      ## Expose the port the container is listening on.
      ## For example, exposing the application port 6112 via TCP.
      ports:
      - containerPort: 6112
        protocol: TCP
      ## Define the rules for the liveness probes.
      ## For example, verify the application on the main route `/`,
      ## on application port 6112. If the application is not responsive, then the pod will be
restarted automatically.
      livenessProbe:
        httpGet:
          path: /
          port: 6112
      ## Define the rules for the readiness probes.
      ## For example, verify the application on the main route `/`,
      ## on application port 6112. If the application is responsive, then traffic will be sent to this
pod.
      readinessProbe:
        httpGet:
          path: /
          port: 6112
      ## Set the resource requests and limits for an application.
      resources:
      ## The resource requests guarantees that the desired amount
      ## CPU and memory is allocated for a pod. In this example,
      ## the pod will be allocated with 64 Mebibytes and 250 miliCPUs.
        requests:
          memory: "64Mi"
          cpu: "250m"
      ## The resource limits ensure that the application is not consuming
      ## more than the specified CPU and memory values. In this example,
      ## the pod will not surpass 128 Mebibytes and 500 miliCPUs.
        limits:
          memory: "128Mi"
          cpu: "500m"
```

Service YAML manifest

In addition to the required sections of a YAML manifest, a Service resource covers the configuration of service type and ports the service should configure. Bellow is a full manifest of a Service explaining each parameter:

```
## Set the API endpoint used to create the Service resource.
apiVersion: v1
## Define the type of the resource.
kind: Service
## Set the parameters that make the object identifiable, such as its name, namespace, and
labels.
metadata:
  labels:
    app: go-helloworld
  name: go-helloworld
  namespace: default
## Define the desired configuration for the Service resource.
spec:
  ## Define the ports that the service should serve on.
  ## For example, the service is exposed on port 8111, and
  ## directs the traffic to the pods on port 6112, using TCP.
  ports:
  - port: 8111
    protocol: TCP
    targetPort: 6112
  ## Identify the pods managed by this Service using the following selectors.
  ## In this case, all pods with the label `go-helloworld`.
  selector:
    app: go-helloworld
  ## Define the Service type, here set to ClusterIP.
  type: ClusterIP
```

Useful command

Kubernetes YAML manifests can be created using the kubectl apply command, with the following syntax:

```
# create a resource defined in the YAML manifests with the name manifest.yaml
kubectl apply -f manifest.yaml
```

To delete a resource using a YAML manifest, the kubectl delete command, with the following syntax:

```
# delete a resource defined in the YAML manifests with the name manifest.yaml
kubectl delete -f manifest.yaml
```

Kubernetes documentation is the best place to explore the available parameters for YAML manifests. However, a support YAML template can be constructed using kubectl commands. This is possible by using the --dry-run=client and -o yamlflags which instructs that the command should be evaluated on the client-side only and output the result in YAML format.

```
# get YAML template for a resource
kubectl create RESOURCE [REQUIRED FLAGS] --dry-run=client -o yaml
```

For example, to get the template for a Deployment resource, we need to use the create command, pass the required parameters, and associated with the --dry-run and -o yaml flags. This outputs the base template, which can be used further for more advanced configuration.

```
# get the base YAML templated for a demo Deployment running a nxing application
 kubectl create deploy demo --image=nginx --dry-run=client -o yaml
```

Declarative Kubernetes Manifests Walkthrough

Summary

This demo showcases how a Namespace and Deployment resource can be deployed using YAML manifests.

New terms

- **Imperative configuration** - resource management technique, that operates and interacts directly with the live objects within the cluster.
- **Declarative configuration** - resource management technique, that operates and manages resources using YAML manifests stored locally.

Further reading

Explore more details about different management techniques and advanced configuration for Deployment resources:

- Managing Kubernetes Objects Using Imperative Commands
- Declarative Management of Kubernetes Objects Using Configuration Files
- Configure Liveness, Readiness Probes for a Deployment
- Managing Resources for Containers

Exercise: Declarative Kubernetes Manifests

Kubernetes is widely known for its imperative and declarative management techniques. In the previous exercise, you have deployed the following resources using the **imperative** approach. Now deploy them using the **declarative** approach.

- a namespace
    - name: demo
    - label: tier: test
- a deployment:
    - image: nginx:alpine
    - name:nginx-apline
    - namespace: demo
    - replicas: 3
    - labels: app: nginx, tag: alpine
- a service:
    - expose the above deployment on port 8111
    - namespace: demo
- a configmap:
    - name: nginx-version
    - containing key-value pair: version=alpine
    - namespace: demo

Note: Nginx is one of the public Docker images, that you can access and use for your exercises or testing purposes.

Make sure the following tasks are completed:

Solution: Declarative Kubernetes Manifests

Declarative Approach

The declarative approach consists of using a full YAML definition of resources. As well, with this approach, you can perform directory level operations.

Examine the manifests for all of the resources in the exercises/manifests.

To create the resources, use the following command:

kubectl apply -f exercises/manifests/

To inspect all the resources within the namespace, use the following command:

kubectl get all -n demo

```
NAME                            READY   STATUS    RESTARTS   AGE
pod/nginx-alpine-798fb5b8bb-8rzq9   1/1     Running   0          12s
pod/nginx-alpine-798fb5b8bb-ms28l   1/1     Running   0          12s
pod/nginx-alpine-798fb5b8bb-qgqb2   1/1     Running   0          12s

NAME                   TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
service/nginx-alpine   ClusterIP   10.109.197.180   <none>        8111/TCP   18s

NAME                           READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/nginx-alpine   3/3     3            3           12s

NAME                                      DESIRED   CURRENT   READY   AGE
replicaset.apps/nginx-alpine-798fb5b8bb   3         3         3       12s
```

Edge Case: Failing Control Plane for Kubernetes

Summary

Failure is expected in any technology stack. However, it is more important to have efficient remediations steps rather than plan for no-failure scenarios. Kubernetes provides methods to handle some of the low-level failures and ensure the application is healthy and accessible. Some of these resources are:

- ReplicaSets - to ensure that the desired amount of replicas is up and running at all times
- Liveness probes - to check if the pod is running, and restart it if it is in an errored state
- Readiness probes - ensure that traffic is routed to that pods that are ready to handle requests
- Services - to provide one entry point to all the available pods of an application

These services, prompt the automatic recovery of an application if an error is encountered. However, failure can happen at the cluster-level, for example, a control plane failure. In this case, a subset or all control plane components are compromised. While the situation is disastrous, the applications are still running and handling traffic. The downside of the control plane failure is that no new workloads can be deployed and no changes can be applied to the existing workloads.

The engineering team needs to recover the control plane components as a critical priority. However, they should not worry about recovering applications, as these will be intact and still handling requests.

Lesson Review

Summary

In this lesson, we have covered how to package an application using Docker and store it in DockerHub. Then, we practiced how to bootstrap a cluster using k3s and deploy an application using Kubernetes resources. For more advanced configurations, we have evaluated Kubernetes YAML manifests that are the basis of declarative management techniques.

Overall, in this lesson, we covered:

- Docker for Application Packaging
- Container Orchestration with Kubernetes
- Kubernetes Resources
- Declarative Kubernetes Manifests

Glossary

- **Dockerfile** - set of instructions used to create a Docker image

- **Docker image** - a read-only template used to spin up a runnable instance of an application
- **Docker registry** - a central mechanism to store and distribute Docker images
- **CRD** - Custom Resource Definition provides the ability to extend Kubernetes API and create new resources
- **Node** - a physical or virtual server
- **Cluster** - a collection of distributed nodes that are used to manage and host workloads
- **Master node** - a node from the Kubernetes control plane, that has installed components to make global, cluster-level decisions
- **Worker node** - a node from the Kubernetes data plane, that has installed components to host workloads
- **Bootstrap** - the process of provisioning a Kubernetes cluster, by ensuring that each node has the necessary components to be fully operational
- **Kubeconfig** - a metadata file that grants a user access to a Kubernetes cluster
- **Pod** - smallest manageable uint within a cluster that provides the execution environment for an application
- **ReplicaSet** - a mechanism to ensure a number of pod replicas are up and running at all times
- **Deployment** - describe the desired state of the application to be deployed
- **Service** - an abstraction layer over a collection of pods running an application
- **Ingress** - a mechanism to manage the access from external users and workloads to the services within the cluster
- **Configmap** - a resource to store non-confidential data in key-value pairs.
- **Secret** - a resource to store confidential data in key-value pairs. These are base64 encoded.
- **Namespace** - a logical separation between multiple applications and associated resources.
- **Imperative configuration** - resource management technique, that operates and interacts directly with the live objects within the cluster.
- **Declarative configuration** - resource management technique, that operates and manages resources using YAML manifests stored locally.

Further Reading

[Kubernetes kubectl cheetsheet]