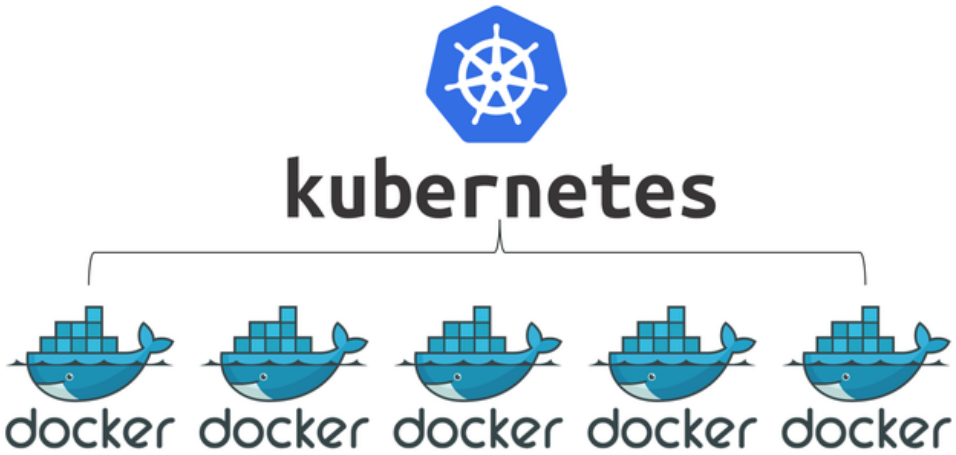
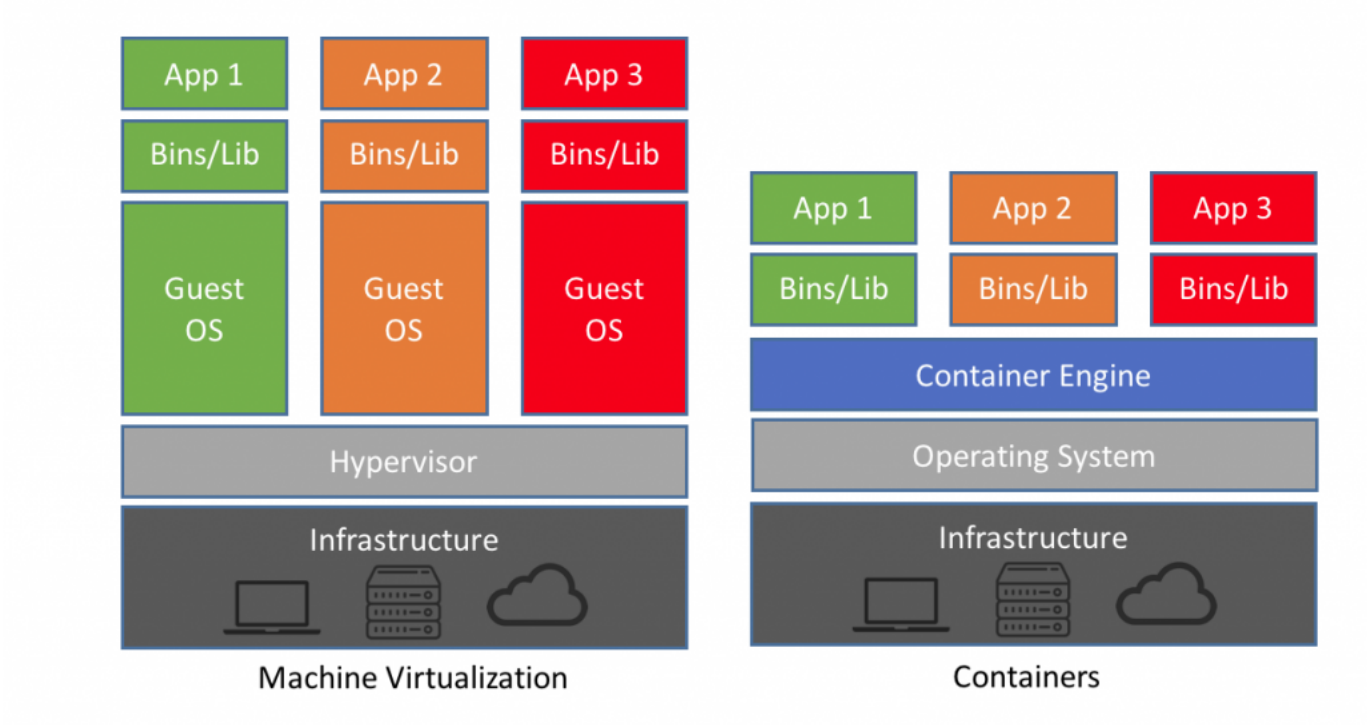
Docker and k8s



1. Introduction to Docker



Benefits of dockers:

- Deployment
- Reusability
- For Microservices

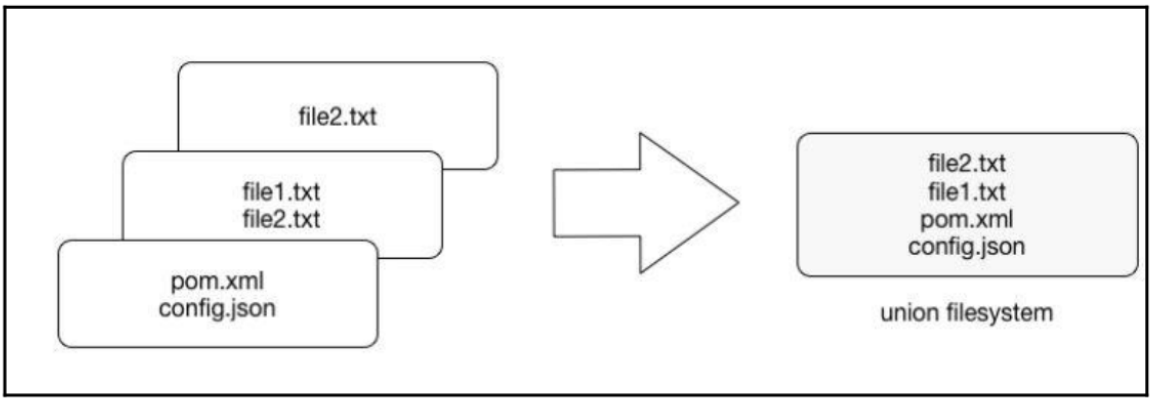
Docker concepts:

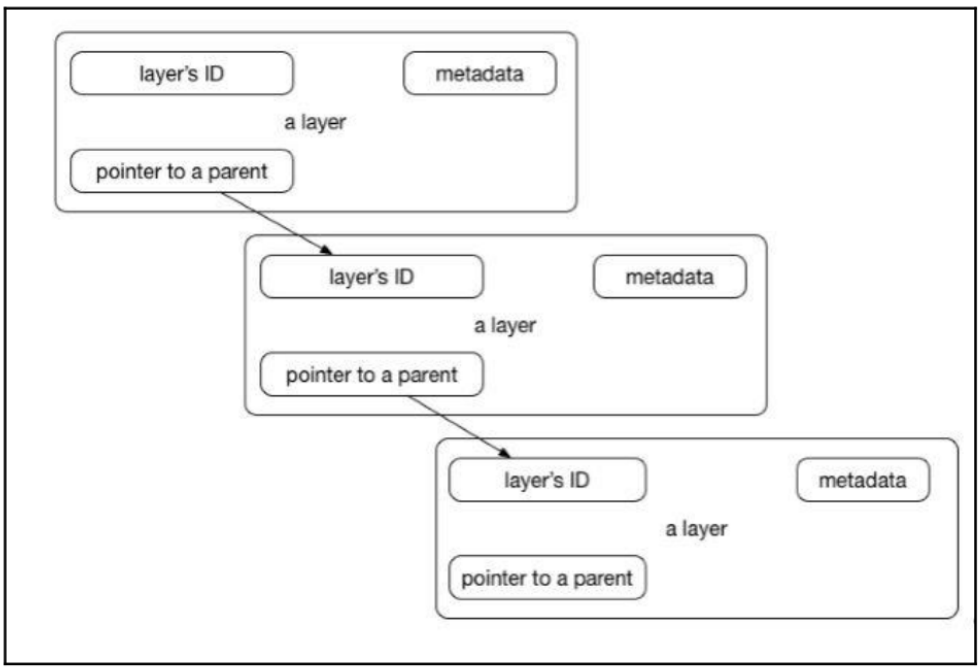
Image:

Think of an image as a read-only template which is a base foundation to create a container from. It's same as a recipe containing the definition of everything your application needs to operate. It can be Linux with an application server (such as Tomcat or Wildfly, for example)

Layers:

Each image consists of a series of layers which are stacked, one on top of the another. In fact, every layer is an intermediate image. By using the **union filesystem**, Docker combines all these layers into a single image entity. The union filesystem allows transparent overlaying files and directories of separate filesystems, giving a single, consistent filesystem as a result, as you can see the following diagram:





Layers have some interesting features. First, they are reusable and cacheable. The pointer to a parent layer you can see in the previous diagram is important. As Docker is processing your Dockerfile it's looking at two things: the Dockerfile instruction being executed and the parent image. Docker will scan all of the children of the parent layer and look for one whose command matches the current instruction. If a match is found, Docker skips to the next Dockerfile instruction and repeats the process. If a matching layer is not found in the cache, a new one is created. For the instructions that add files to your image (we will get to know them later in detail), Docker creates a checksum for each file contents. During the building process, this checksum is compared against the checksum of the existing images to check if the layer can be reused from the cache. If two different images have a common part, let's say a Linux shell or Java runtime for example, Docker, which tracks all of the pulled layers, will reuse the shell layer in both of the images. It's a safe operation; as you already know, layers are read-only. When downloading another image, the layer will be reused and only the difference will be pulled from the Docker Hub. This saves time, bandwidth, and disk space of course, but it has another great advantage. If you modify your Docker image, for example by modifying your containerized Java application, only the application layer gets modified. After you've successfully built an image from your Dockerfile, you will notice that subsequent builds of the same Dockerfile finish a lot faster. Once Docker caches an image layer for an instruction, it doesn't need to be rebuilt. Later on, instead of distributing the whole image, you push just the updated part. It makes the process simpler and faster. This is especially useful if you use Docker in your continuous deployment flow: pushing a Git branch will trigger building an image and then publishing the application for users. Due to the layer-reuse feature, the whole process is a lot faster.

> Layers and images are closely related to each other. As we have said before, Docker images are stored as a series of read-only layers. This means that once the container image has been created, it does not change. But having all the filesystem read-only would not make a lot of sense. What about modifying an image? Or adding your software to a base web server image? Well, when we start a container, Docker actually takes the read-only image (with all its read-only layers) and adds a writable layer on top of the layers stack. Let's focus on the containers now.

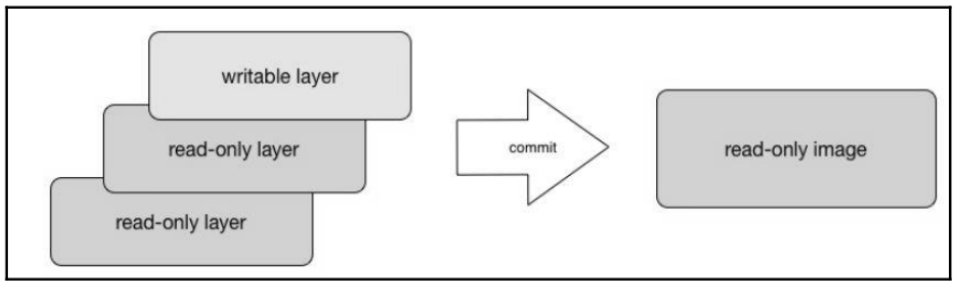
Container:

> A running instance of an image is called a container. Docker launches them using the Docker images as read-only templates. If you start an image, you have a running container of this image. Naturally, you can have many running containers of the same image. In fact, we will do it very often a little bit later, using Kubernetes.
>
> To run a container, we use the `docker run` command:
>
> **`docker run [OPTIONS] IMAGE [COMMAND] [ARG...]`**

```
docker run [OPTIONS] IMAGE [COMMAND] [ARG...]
docker stop
docker ps -a
docker rm $(docker ps -a -q -f status=exited)
docker commit <container-id> <image-name>
```

We have said that a Docker image is always read-only and immutable. If it did not have the possibility to change the image, it would not be very useful. So how's the image modification possible except by, of course, altering a Dockerfile and doing a rebuild? When the container is started, the writable layer on top of the layers stack is for our disposal. We can actually make changes to a running container; this can be adding or modifying files, the same as installing a software package, configuring the operating system, and so on. If you modify a file in the running container, the file will be taken out of the underlying (parent) read-only layer and placed in the top, writable layer. Our changes are only possible in the top layer. The union filesystem will then cover the underlying file. The original, underlying file will not be modified; it still exists safely in the underlying, read-only layer. By issuing the `docker commit` command, you create a new read-only image from a running container (and all it changes in the writable layer):

> The `docker commit` command saves changes you have made to the container in the writable layer. To avoid data corruption or inconsistency, Docker will pause a container you are committing changes into. The result of the `docker commit` command is a brand new, read-only image, which you can create new containers from:



> If your app is named `hello-world-java` and your username (or namespace) for the **Registry** is `dockerJavaDeveloper` then your image will be placed in the `dockerJavaDeveloper/hello-world-java` repository. You can tag an image and store multiple versions of that image with different IDs in a single named repository and access different tagged versions of an image with a special syntax such as `username/image_name:tag`. The `Docker` repository is quite similar to a Git repository. For example, `Git`, a `Docker` repository is identified by a URI and can either be public or private. The URI looks the same as the following:

```
{registryAddress}/{namespace}/{repositoryName}:{tag}
```

Docker registry, repo, index:

205

Docker registry is a service that is storing your docker images.

Docker registry could be hosted by a third party, as public or private registry, like one of the following registries:

- Docker Hub,
- Quay,
- Google Container Registry,
- AWS Container Registry

or you can host the docker registry by yourself
(see https://docs.docker.com/ee/dtr/ for more details).

Docker repository is a collection of different docker images with same name, that have different tags. Tag is alphanumeric identifier of the image within a repository.

For example see https://hub.docker.com/r/library/python/tags/. There are many different tags for the official python image, these tags are all members of the official python repository on the Docker Hub. Docker Hub is a Docker Registry hosted by Docker.

2. Network & Storage

2.1 Network

```
PS C:\Users\duyng> docker network ls
NETWORK ID     NAME      DRIVER    SCOPE
58f862059f56   bridge    bridge    local
9123d227b41a   host      host      local
5f022a05cb17   none      null      local
PS C:\Users\duyng>
```

Bridge

This is the default network type in Docker. When the Docker service daemon starts, it configures a virtual bridge, named `docker0`. If you do not specify a network with the `docker run -net=<NETWORK>` option, the Docker daemon will connect the container to the bridge network by default. Also, if you create a new container, it will be connected to the bridge network. For each container that Docker creates, it allocates a virtual Ethernet device which will be attached to the bridge. The virtual Ethernet device is mapped to appear as `eth0` in the container, using Linux namespaces, as you can see in the following diagram:

```
docker network create -d bridge mybridge
docker run -d --net mybridge --name db redis
docker run -d --net mybridge -e DB=db -p 8000:5000 --name web chrch/web
```

Network drivers

Docker's networking subsystem is pluggable, using drivers. Several drivers exist by default, and provide core networking functionality:

- `bridge` : The default network driver. If you don't specify a driver, this is the type of network you are creating. **Bridge networks are usually used when your applications run in standalone containers that need to communicate.** See bridge networks.
- `host` : For standalone containers, remove network isolation between the container and the Docker host, and use the host's networking directly. See use the host network.
- `overlay` : Overlay networks connect multiple Docker daemons together and enable swarm services to communicate with each other. You can also use overlay networks to facilitate communication between a swarm service and a standalone container, or between two standalone containers on different Docker daemons. This strategy removes the need to do OS-level routing between these containers. See overlay networks.
- `ipvlan` : IPvlan networks give users total control over both IPv4 and IPv6 addressing. The VLAN driver builds on top of that in giving operators complete control of layer 2 VLAN tagging and even IPvlan L3 routing for users interested in underlay network integration. See IPvlan networks.
- `macvlan` : Macvlan networks allow you to assign a MAC address to a container, making it appear as a physical device on your network. The Docker daemon routes traffic to containers by their MAC addresses. Using the `macvlan` driver is sometimes the best choice when dealing with legacy applications that expect to be directly connected to the physical network, rather than routed through the Docker host's network stack. See Macvlan networks.
- `none` : For this container, disable all networking. Usually used in conjunction with a custom network driver. `none` is not available for swarm services. See disable container networking.
- Network plugins: You can install and use third-party network plugins with Docker. These plugins are available from Docker Hub or from third-party vendors. See the vendor's documentation for installing and using a given network plugin.

Use bridge networks

Manage a user-defined bridge

Use the `docker network create` command to create a user-defined bridge network.

```
$ docker network create my-net
```

You can specify the subnet, the IP address range, the gateway, and other options. See the docker network create reference or the output of `docker network create --help` for details.

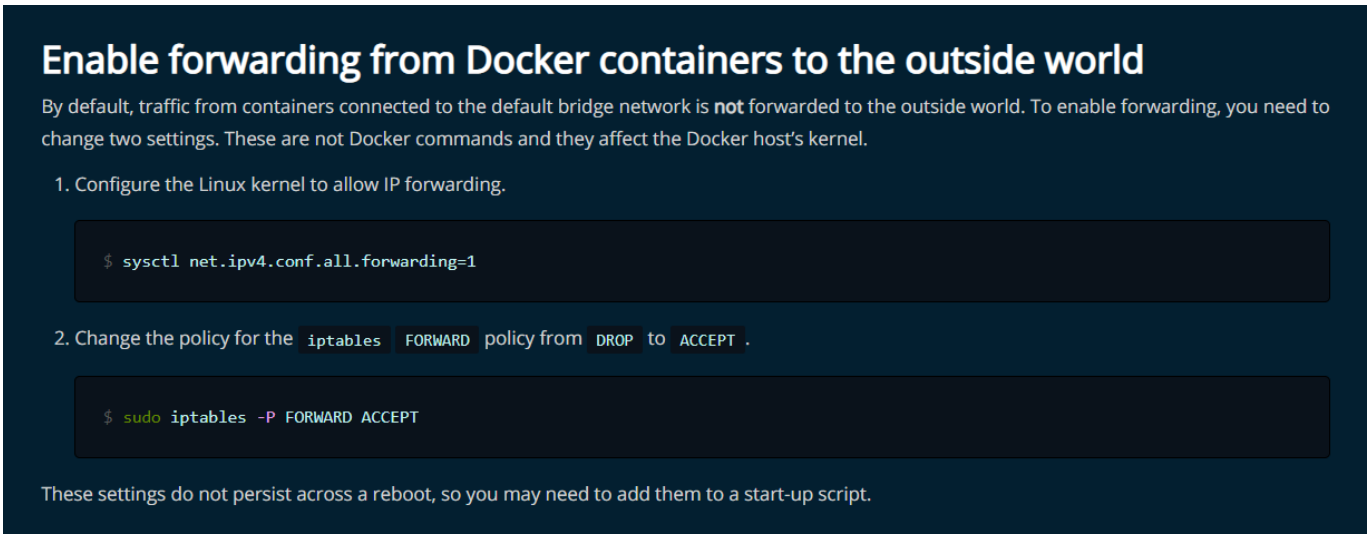
Use the `docker network rm` command to remove a user-defined bridge network. If containers are currently connected to the network, disconnect them first.

```
$ docker network rm my-net
```

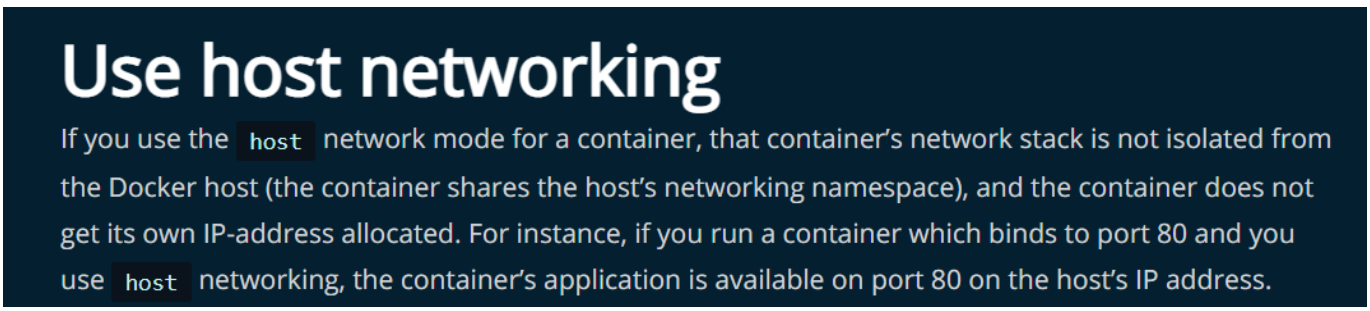
What's really happening?

When you create or remove a user-defined bridge or connect or disconnect a container from a user-defined bridge, Docker uses tools specific to the operating system to manage the underlying network infrastructure (such as adding or removing bridge devices or configuring `iptables` rules on Linux). These details should be considered implementation details. Let Docker manage your user-defined networks for you.

```
docker create --name d-nginx --network d-net --publish 8080:80 nginx:latest
docker network connect d-net d-nginx
```



Using host networking:



Networking command:

```
docker network create myNetwork
docker network inspect myNetwork
# connect container to network
docker run -it --net=myNetwork tomcat
# expose port
docker run -it --name myTomcat2 --net=myNetwork -p 8080:8080 tomcat
```

2.2 Persistent store

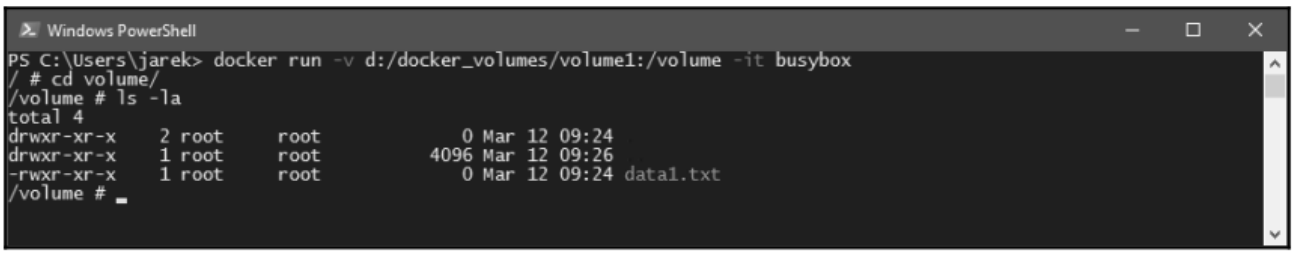
```
docker volume create: Creates a volume
docker volume inspect: Displays detailed information on one or more columes
docker volume ls: Lists volumes
docker volume rm: removes one or more volumes
docker volume prune: removes all unused volumes, which is all volume that are no
longer mapped into any container
```

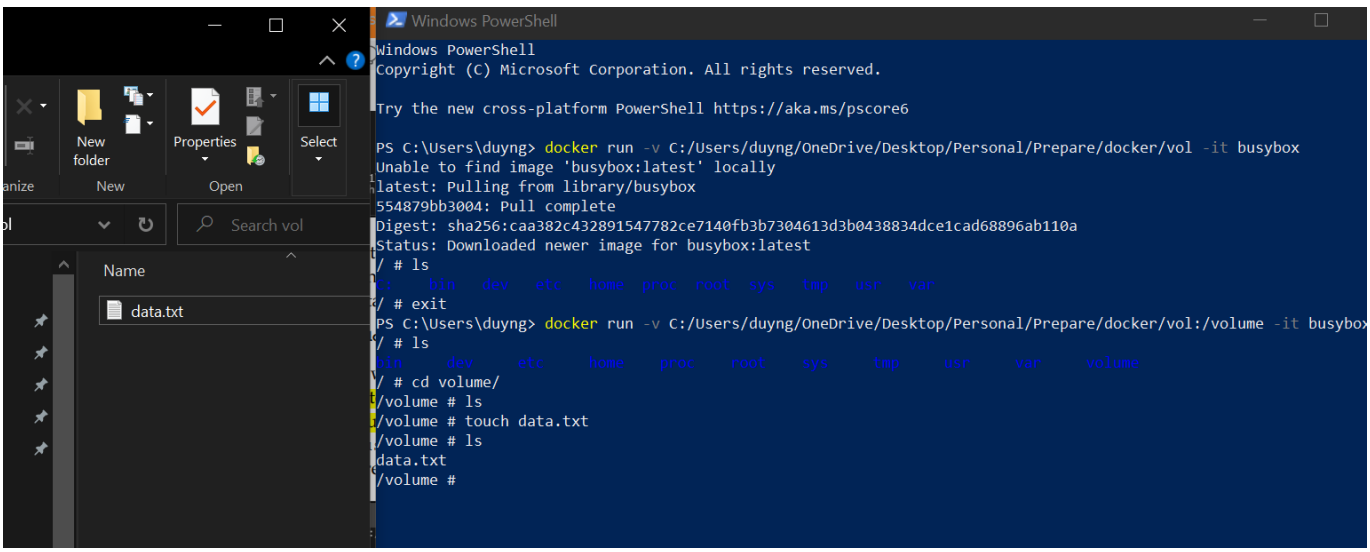
There are two ways to create volumes. The first one is to specify the `-v` option when running an image. Let's run the `busybox` image we already know and, at the same time, create a volume for our data:

```
$ docker run -v d:/docker_volumes/volume1:/volume -it busybox
```

```
docker run -v C:/Users/duyng/OneDrive/Desktop/Personal/Prepare/docker/vol:/volume -it
busybox
```

In the previous command, we have created a volume using the `-v` switch and instructed Docker that the `host` directory `d:/docker_volumes/volume1` should be mapped into the `/volume` directory in the running container. If we now list the contents of the `/volume` directory in the running `busybox` container, we can see our empty `data1.txt` file, as you can see in the following screenshot:





3. Image microservices

Dockerfile

As you will remember from `Chapter 1`, *Introduction to Docker*, the `Dockerfile` is kind of a recipe to build an image. It's a plain text file containing instructions which are executed by Docker in the order they are placed. Each `Dockerfile` has a base image that the Docker engine will use to build upon. A resulting image will be a specific state of a file system: a read-only, frozen immutable snapshot of a live container, composed of layers representing changes in the filesystem at various points in time.

The image creation flow in Docker is pretty straightforward and consists basically of two steps:

1. First, you prepare a text file named `Dockerfile`, which contains a series of instructions on how to build the image. The set of instructions you can use in the `Dockerfile` is not very broad, but sufficient to fully instruct Docker how to create an image.
2. Next, you execute the `docker build` command to create a Docker image based on the `Dockerfile` that you have just created. The `docker build` command runs within the context. The build's context is the files at a specified location, which can be a `PATH` or a URL. The `PATH` is a directory on your local filesystem and the URL is a Git repository location. A context is processed recursively. `PATH` will include any subdirectories. The URL will include the repository and its submodules.

Docker instruction:

Build instruction:

```
docker build . -t rest-example
docker image ls
 docker run -it --name rest-api -p 8080:8080 rest-example
```

Command	Description
FROM	This is the first instruction in the Dockerfile. It sets the base image for every subsequent instruction coming next in the file. The syntax for the `FROM` instruction is straightforward. It's just: `FROM <image>`, or `FROM <image>:<tag>`, or `FROM <image>@<digest>`
MAINTAINER	By using the `MAINTAINER` instruction, you set the `Author` field of the generated image. This can be your name, username, or whatever you would like as an author of the image that will be created by using the `Dockerfile` you are writing. This command can be placed
WORKDIR	The `WORKDIR` instruction adds a working directory for any `CMD`, `RUN`, `ENTRYPOINT`, `COPY`, and `ADD` instructions that comes after it in the Dockerfile. The syntax for the instruction is `WORKDIR /PATH`. You can have multiple `WORKDIR` instructions in one Dockerfile, if the relative path is provided; it will be relative to the path of the previous `WORKDIR` instruction.
ADD	For example, `ADD target/*.jar /` will add all files ending with `.jar` into the root directory in the image's file system.
COPY	As you can see, the functionality of `COPY` is almost the same as the `ADD` instruction, with one difference. `COPY` supports only the basic copying of local files into the container. On the other hand, `ADD` gives some more features, such as archive extraction, downloading files through URL, and so on. Docker's best practices say that you should prefer `COPY` if you do not need those additional features of `ADD`. The `Dockerfile` will be cleaner and easier to understand thanks to the transparency of the `COPY` command.

RUN	Consider the following example. Probably the most common usecase for `RUN` is an application of `apt-get`, which is a package manager command for downloading packages on Ubuntu. Let's say we have the following Dockerfile, installing Java runtime:
	
	`FROM ubuntu`
	`RUN apt-get update`
	`RUN apt-get install -y openjdk-8-jre`

CMD	![• `CMD ["executable","parameter1","parameter2"]`: This is a so called `exec` form. It's also the preferred and recommended form. The parameters are JSON array, and they need to be enclosed in square brackets. The important note is that the `exec` form does not invoke a command shell when the container is run. It just runs the executable provided as the first parameter. If the `ENTRYPOINT` instruction is present in the `Dockerfile`, `CMD` provides a default set of parameters for the `ENTRYPOINT` instruction. TIP `RUN` is a build-time instruction, the `CMD` is a runtime instruction.

Command	Description

ENTRYPOINT	The official Docker documentation says that the ENTRYPOINT instruction allows you to configure a container that will run as an executable. It's not very clear, at least for the first time. The ENTRYPOINT instruction is related to the CMD instruction. In fact, it can be confusing at the beginning. The reason for that is simple: CMD was developed first, then ENTRYPOINT was developed for more customization, and some functionality overlaps between those two instructions. Let's explain it a bit. The ENTRYPOINT specifies a command that will always be executed when the container starts. The CMD, on the other hand, specifies the arguments that will be fed to the ENTRYPOINT. Docker has a default ENTRYPOINT which is /bin/sh -c but does not have a default CMD. For example, consider this Docker command:
	`docker run ubuntu "echo" "hello world"`
	`docker run ubuntu "echo" "hello world"`
	In this case, the image will be the latest ubuntu, the ENTRYPOINT will be the default /bin/sh -c, and the command passed to the ENTRYPOINT will be echo "hello world".
	The syntax for the ENTRYPOINT instruction can have two forms, similar to CMD.
	ENTRYPOINT ["executable", "parameter1", "parameter2"] is the exec form, preferred and recommended. Exactly the same as the exec form of the CMD instruction, this will not invoke a command shell. This means that the normal shell processing will not happen. For example, ENTRYPOINT ["echo", "$HOSTNAME"] will not do variable substitution on the $HOSTNAME variable. If you want shell processing then you need either to use the shell form or execute a shell directly. For example:
	`ENTRYPOINT [ "sh", "-c", "echo $HOSTNAME" ]`
	Consider this simple Dockerfile based on BusyBox. BusyBox is software that provides several stripped-down Unix tools in a single executable file. To demonstrate ENTRYPOINT, we are going to use a ping command from BusyBox:
	`FROM busybox`
	`ENTRYPOINT ["/bin/ping"]`
	`CMD ["localhost"]`

EXPOSE	The EXPOSE instruction informs Docker that the container listens on the specified network ports at runtime. We have already mentioned the EXPOSE instruction in Chapter 2, *Networking and Persistent Storage*. As you will remember, EXPOSE in a Dockerfile is the equivalent to the --expose command-line option. Docker uses the EXPOSE command followed by a port number to allow incoming traffic to the container. We already know that

VOLUME	The parameter for VOLUME can be a JSON array, a plain string with one or more arguments. For example:
	`VOLUME ["/var/lib/tomcat8/webapps/"]`
	`VOLUME /var/log/mongodb /var/log/tomcat`

LABEL	The syntax of the LABEL instruction is straightforward:
	`LABEL "key"="value"`
	To have a multiline value, separate the lines with backslashes; for example:
	`LABEL description="This is my \`
	`multiline description of the software."`
	You can have multiple labels in a single image. Provide them separated with a space or a backslash; for example:
	`LABEL key1="value1" key2="value2" key3="value3"`
	`LABEL key1="value1" \`
	`key2="value2" \`
	`key3="value3"`

Command	Description
ENV	The first one, `ENV <key> <value>`, will set a single variable to a value. The entire string after the first space will be treated as the `<value>`. This will include any character, and also spaces and quotes. For example: `ENV JAVA_HOME /var/lib/java8`
USER	
ARG	The `ARG` instruction is being used to pass an argument to the Docker daemon during the `docker build` command. An `ARG` variable definition comes into effect from the line on which it is defined in the `Dockerfile`. By using the `--build-arg` switch, you can assign a value to the defined variable: `$ docker build --build-arg <variable name>=<value> .`

ONBUILD	This is useful if you are building an image which will be used as a base to build other images. For example, an application build environment or a daemon which may be customized with a user-specific configuration. The `ONBUILD` instruction is very useful (`https://docs.docker.com/engine/reference/builder/#onbuild` and `https://docs.docker.com/engine/reference/builder/#maintainer-deprecated`), for automating the build of your chosen software stack. Consider the following example with Maven and building Java applications (yes, Maven is also available as a Docker container). Basically, all your project's Dockerfile needs to do is reference the base container containing the `ONBUILD` instructions:
	`FROM maven:3.3-jdk-8-onbuild`
	`CMD ["java","-jar","/usr/src/app/target/app-1.0-SNAPSHOT-jar-with-dependencies.jar"]`
	There's no magic, and everything becomes clear if you look into the parent's Dockerfile. In our case, it will be a `docker-maven` Dockerfile available on GitHub:
	`FROM maven:3-jdk-8`
	`RUN mkdir -p /usr/src/app`
	`WORKDIR /usr/src/app`
	`ONBUILD ADD . /usr/src/app`
	`ONBUILD RUN mvn install`

HEALTHCHECK	`HEALTHCHECK --interval=5m --timeout=2s --retries=3 CMD curl -f http://localhost/ping || exit 1`

Using Maven:

```
            <plugin>
                <groupId>io.fabric8</groupId>
                <artifactId>docker-maven-plugin</artifactId>
                <version>0.20.1</version>
                <configuration>
                    <images>
                        <image>
                            <name>rest-example:${project.version}</name>
                            <alias>rest-example</alias>
                            <build>
                                <from>openjdk:latest</from>
                                <assembly>
                                    <descriptorRef>artifact</descriptorRef>
                                </assembly>
                                <cmd>java -jar
maven/${project.name}-${project.version}.jar</cmd>
                            </build>
                            <run>
                                <ports>
                                    <port>8080:8080</port>
                                </ports>
                            </run>
                        </image>
                    </images>
                </configuration>
            </plugin>
```

Build image

```
mvn clean package docker:build
```

4. Running Containers with Java Applications

```
docker run  [OPTIONS] IMAGE[:TAG|@DIGEST] [COMMAND] [ARG...]
docker stop [OPTIONS] CONTAINER [CONTAINER...]
docker kill CONTAINER [CONTAINER...]
REM List container
docker ps -a
docker rm [OPTIONS] CONTAINER [CONTAINER...]
```

- `docker stop`: The main process inside the container will first receive a `SIGTERM`, and after a grace period, a `SIGKILL`
- `docker kill`: The main process inside the container will be sent `SIGKILL` (by default) or any signal specified with option `--signal`

Foreground

In the foreground mode, the console you are using to execute `docker run` will be attached to standard input, output, and error streams. This is the default; Docker will attach `STDIN`, `STDOUT` and `STDERR` streams to your shell console. If you need to, you can change this behavior and use the `-a` switch for the `docker run` command. As a parameter for the `-a` switch, you use the name of the stream you want to attach to the console. For example:

```
$ docker run -a stdin -a stdout -i -t centos /bin/bash
```

The preceding command will attach both `stdin` and `stdout` streams to your console.

The useful `docker run` options are the `-i` or `--interactive` (for keeping `STDIN` stream open, even if not attached) and `-t` or `-tty` (for attaching a `pseudo-tty`) switches, commonly used together as `-it` which you will need to use to allocate a `pseudo-tty` console for the process running in the container. Actually, we used this option in `Chapter 5`, *Creating Images with Java Applications*, when we were running our REST service:

```
$ docker run -it rest-example
```

Detached

You can start a Docker container in detached mode with a `-d` option. It's the opposite of the foreground mode. The container starts up and runs in background, the same as a daemon or a service. Let's try to run our rest-example in the background, executing the following command:

```
$ docker run -d -p 8080:8080 rest-example
```

Attaching to running containers

To retain control over a detached container, use `docker attach` command. The syntax for `docker attach` is quite simple:

```
$ docker attach [OPTIONS] <container ID or name>
```

In our case this would be the ID we were given, when the container was started:

```
$ docker attach
5687bd611f84b53716424fd826984f551251bc95f3db49715fc7211a6bb23840
```

```
REM Foreground
docker run -a stdin -a stdout -i -t centos /bin/bash

REM Detached
docker run -d -p 8080:8080 rest-example

docker attach [OPTIONS] <container ID or name>

REM Viewing logs
docker logs -f <container id>
```

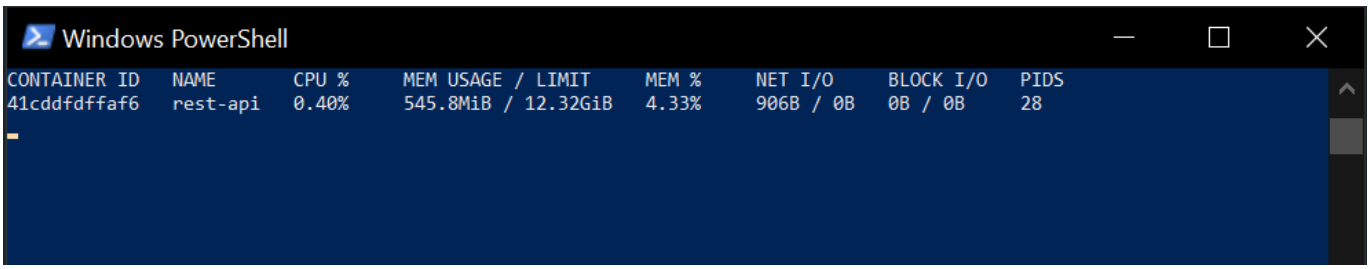
Log driver:

There are some other log drivers available. The list includes:

- `none`: It will just switch off logging completely
- `syslog`: It's a `syslog` logging driver for Docker. It will write log messages to the system `syslog`
- `journald`: Will log messages to `journald`. `systemd-journald` is a daemon responsible for event logging, with append-only binary files serving as its logfiles
- `splunk`: Provides the writing of log messages to Splunk using `Event Http` Collector. Splunk can be used as an enterprise-grade log analyzer. You can read more about it at `https://www.splunk.com`
- `gelf`: Will write log entries into a GELF endpoint such as Graylog or Logstash. Graylog, available at `https://www.graylog.org`, is an open source log management, supporting search, analysis, and alerting across all of your log files. Logstash, which you can find at `https://www.elastic.co/products/logstash`, is a pipeline for processing any data (including log data) from any source
- `fluentd`: Writes log messages to `fluentd`. Fluentd is an open source data collector for a unified logging layer. The main feature of Fluentd is that it separates data sources from backend systems by providing a unified logging layer in between. It's small, fast and has hundreds of plugins that make a very flexible solution out of it. You can read more about `fluentd` on its website at `https://www.fluentd.org`
- `gcplogs`: Will send the log entries to Google Cloud logging
- `awslogs`: This driver will write log messages to the Amazon CloudWatch logs.

```
docker run --log-driver=syslog rest-example
docker stats [OPTIONS] [CONTAINER...]
```

Stats command:



Resource handling:

```
REM Memory
docker run -it -m 512m ubuntu


REM --memory-reservation signal docker to minimize memory consume after 1G
docker run -it --memory-reservation 1G ubuntu /bin/bash

REM CPUs 0 -> 1024
--c  --cpu-shares

REM CPUs quota 0 -> 50000 (allowance of the CPU usage)
--cpu-quota=50000

REM It means that the container can get 50% of the CPU usage every 50 ms.
docker run -it --cpu-quota=25000 --cpu-period=50000 ubuntu /bin/bash

REM num cores allows
docker run -it --cpuset 4 ubuntu
```

Using maven:

```
mvn clean package docker:start
```

5. Introduction to K8s

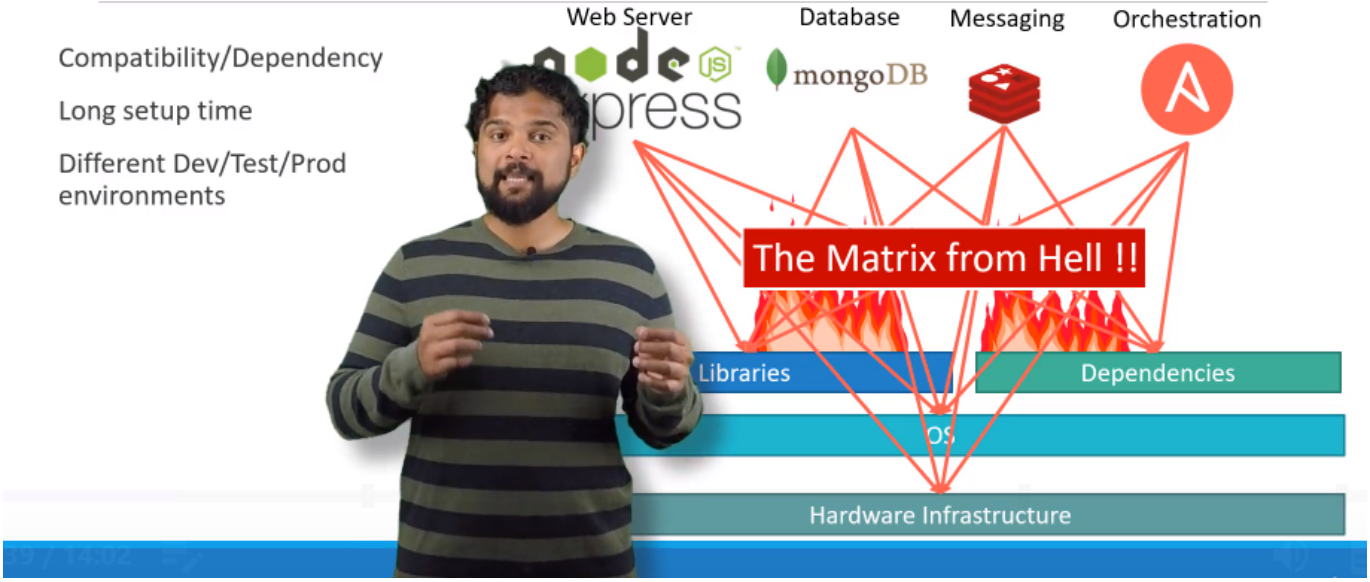
5.1 Basic Kubernetes concepts

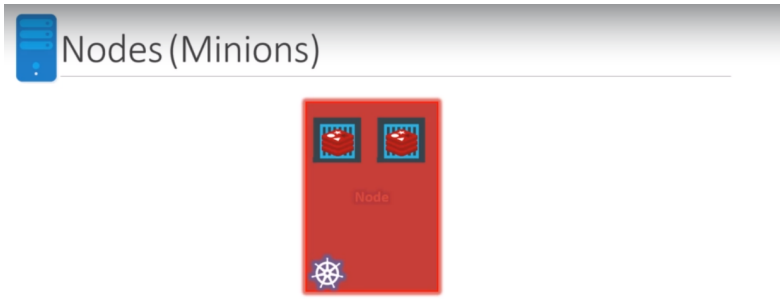
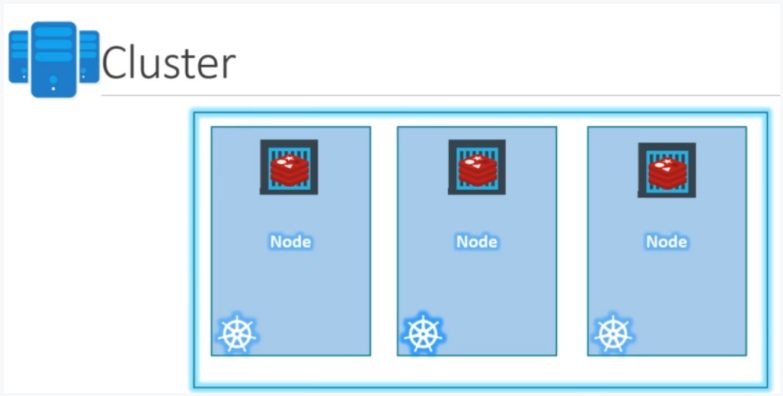
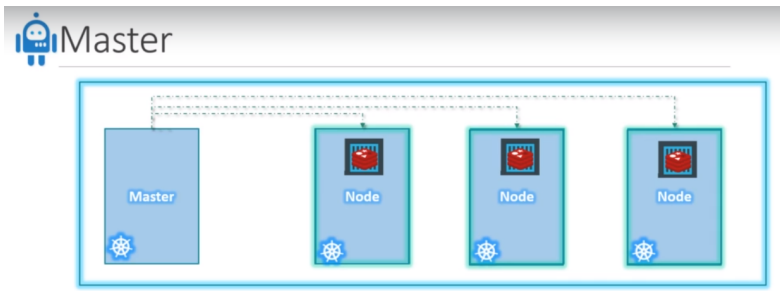
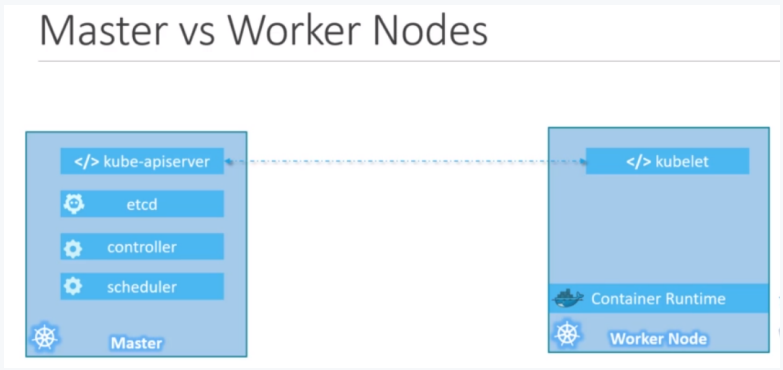
Why do you need containers?

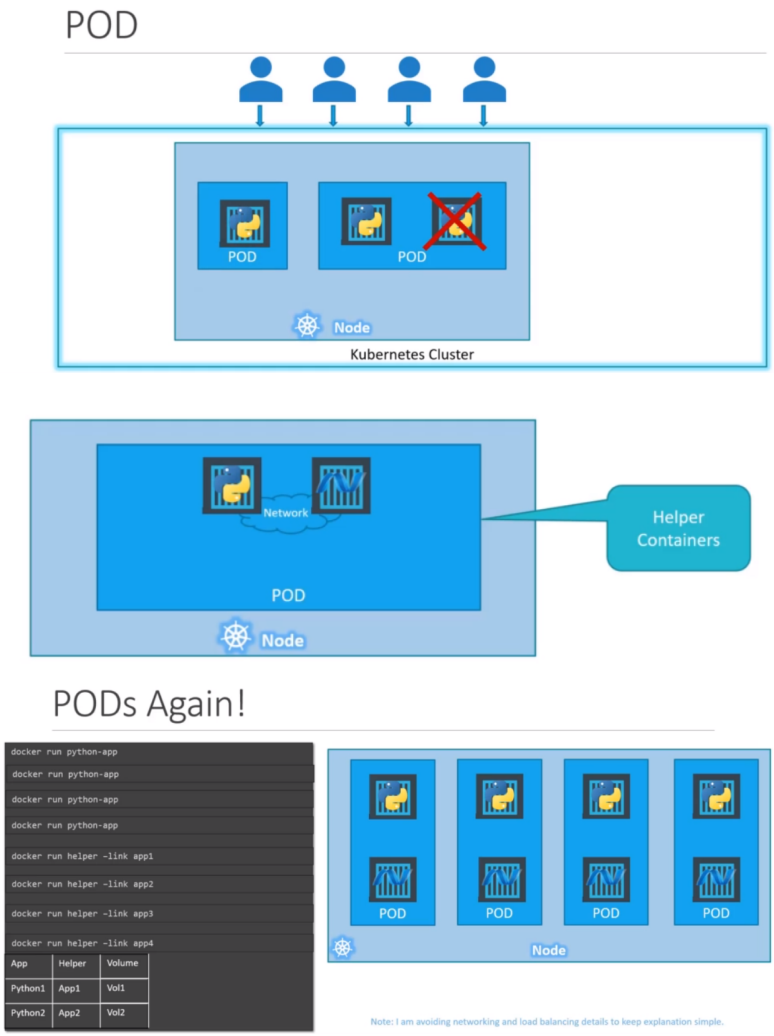
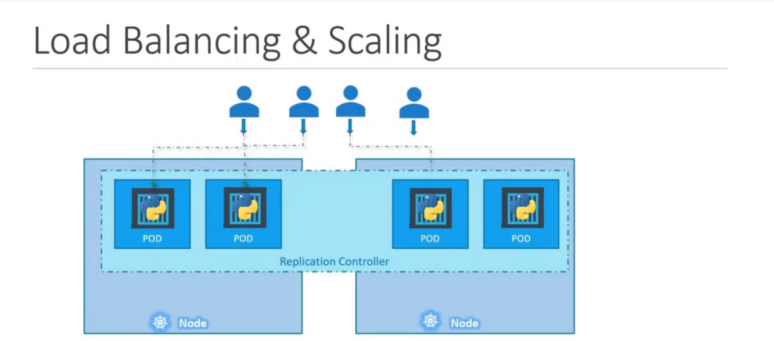
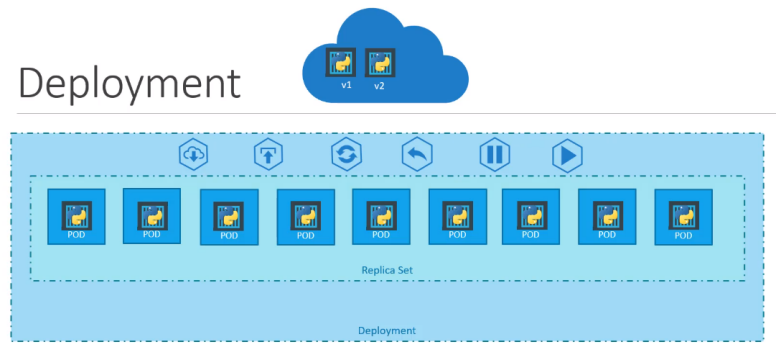
Compatibility/Dependency

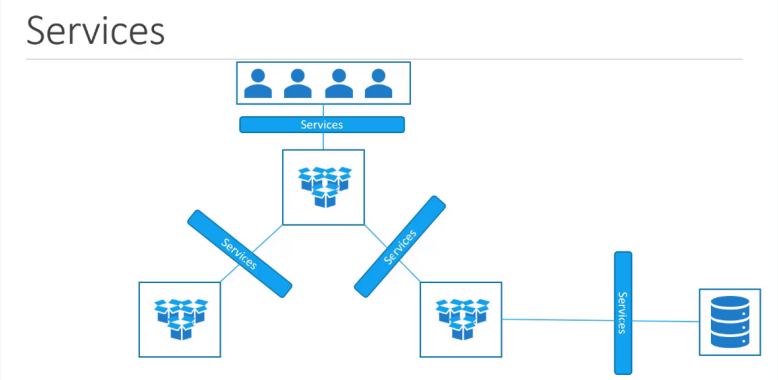
Long setup time

Different Dev/Test/Prod environments



Component	Description	
NODE	Virual or physical machine k8s running on	
CLUSTER	Sets of nodes	
MASTER	Node with k8s install as master	
WORKER		

Component	Description	
PODS	Instance of Application. 1 pod = 1 container	
REPLICA SETS	Sets of Pod	
DEPLOYMENT	Higher level of replica set support rolling update	

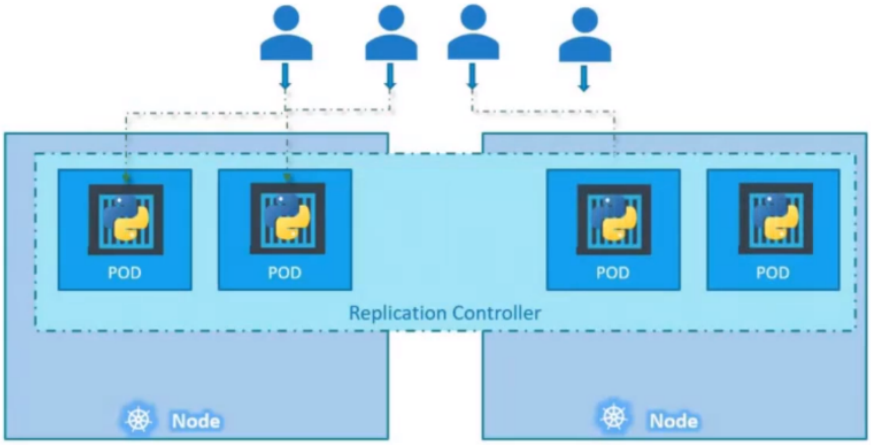
Component	Description	
SERVICES	Support communication between services and service to data sources	

Pod with YAML:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
    type: front-end
spec:
  containers:
    - name: nginx-container
      image: nginx
    - name: busy-container
      image: busy
# kubectl create -f pod-1.yaml
# kubectl apply -f pod-1.yaml
```

Replicate sets:

Load Balancing & Scaling



```yaml
# Replica Controller
apiVersion: v1
kind: ReplicationController
metadata:
  name: myapp-rc
  labels:
    app: myapp
    type: front-end
spec:
  template:
    metadata:
      name: myapp-pod
      labels:
        app: myapp
        type: pod
    spec:
      containers:
        - name: nginx-container
          image: nginx
        - name: busy-container
          image: busy

  replicas: 3


# Replica set
apiVersion: apps/v1 # Change here
kind: ReplicaSet
metadata:
  name: myapp-replicaset
  labels:
    app: myapp
    type: replica-set
spec:
  template:
    metadata:
      name: myapp-pod
      labels:
        app: myapp
        type: set
    spec:
      containers:
        - name: nginx-container
          image: nginx

  replicas: 3
  selector:  # Differ to replica controller
    matchLabels:
      type: set
```

Scale

```
> kubectl replace -f replicaset-definition.yml
```

replicaset-definition.yml

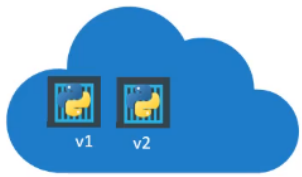
```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: myapp-replicaset
  labels:
      app: myapp
      type: front-end
spec:
  template:
    metadata:
      name: myapp-pod
      labels:
        app: myapp
        type: front-end
    spec:
      containers:
      - name: nginx-container
        image: nginx
  replicas: 6
  selector:
    matchLabels:
      type: front-end
```

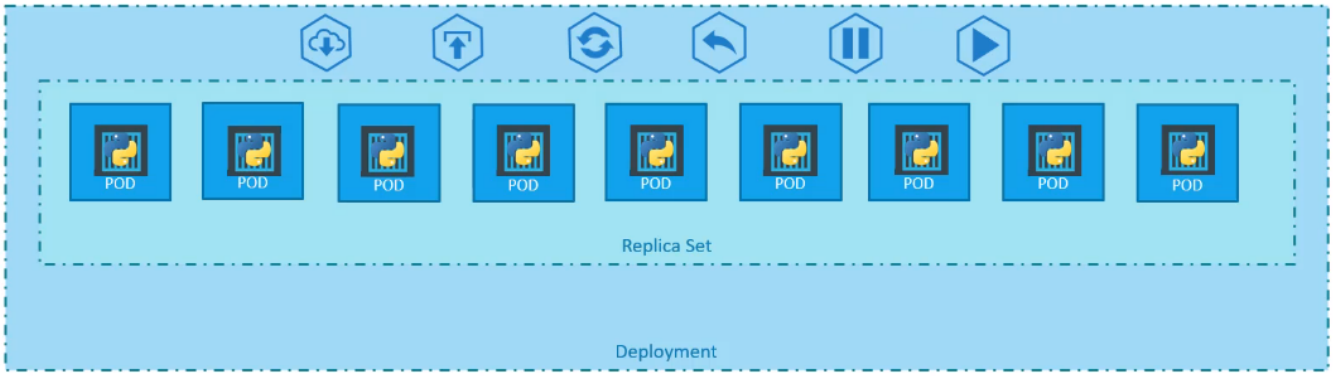
```
> kubectl replace -f replicaset-definition.yml

> kubectl scale --replicas=6 -f replicaset-definition.yml

> kubectl scale --replicas=6 replicaset myapp-replicaset
```

TYPE NAME

Deployments:

Deployment





When to use a ReplicaSet

A ReplicaSet ensures that a specified number of pod replicas are running at any given time. However, a Deployment is a higher-level concept that manages ReplicaSets and provides declarative updates to Pods along with a lot of other useful features. Therefore, we recommend using Deployments instead of directly using ReplicaSets, unless you require custom update orchestration or don't require updates at all.

This actually means that you may never need to manipulate ReplicaSet objects: use a Deployment instead, and define your application in the spec section.

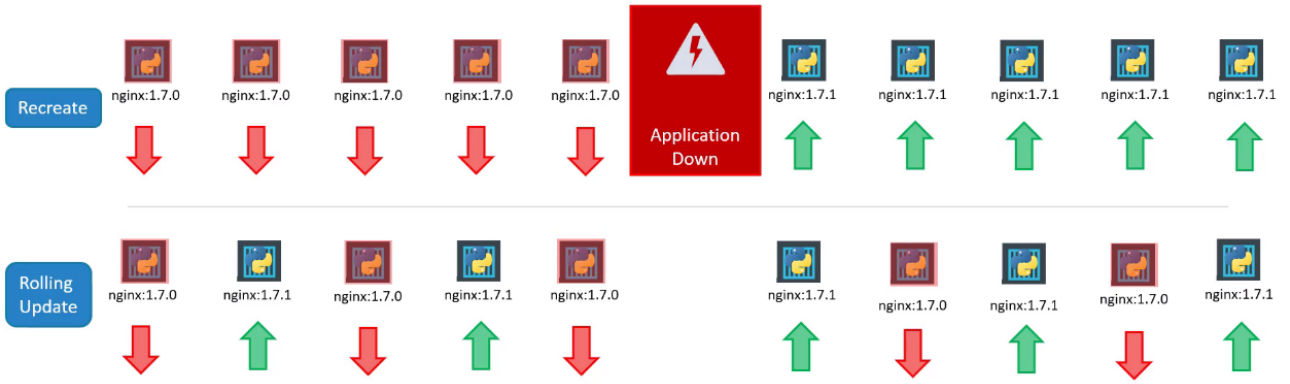
Update and roll back:


```
> kubectl rollout status deployment/myapp-deployment
deployment "myapp-deployment" successfully rolled out


> kubectl rollout history deployment/myapp-deployment
deployment.apps/myapp-deployment
REVISION  CHANGE-CAUSE
1         <none>
```

Deployment strategy:

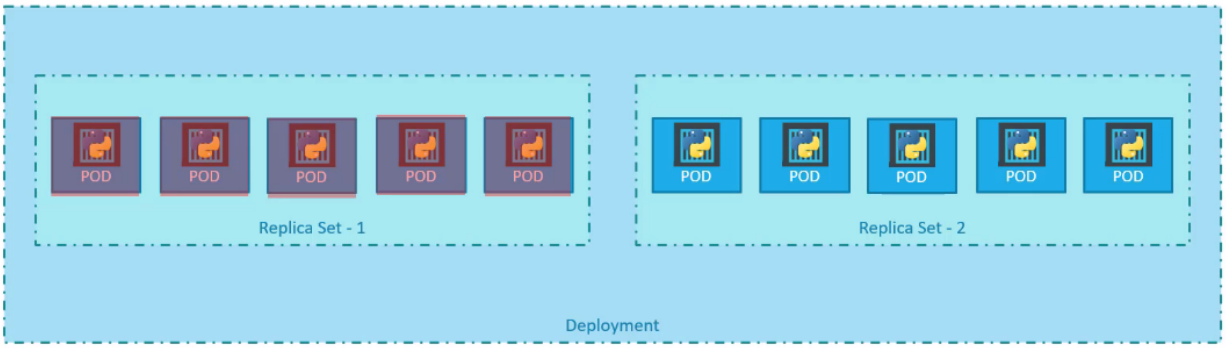
- Delete and create new (or recreate strategy)
- Rolling update

Deployment Strategy




```
kubectl.exe apply -f .\deployment.yaml
deployment.apps/myapp-deployment configured
```

Upgrades



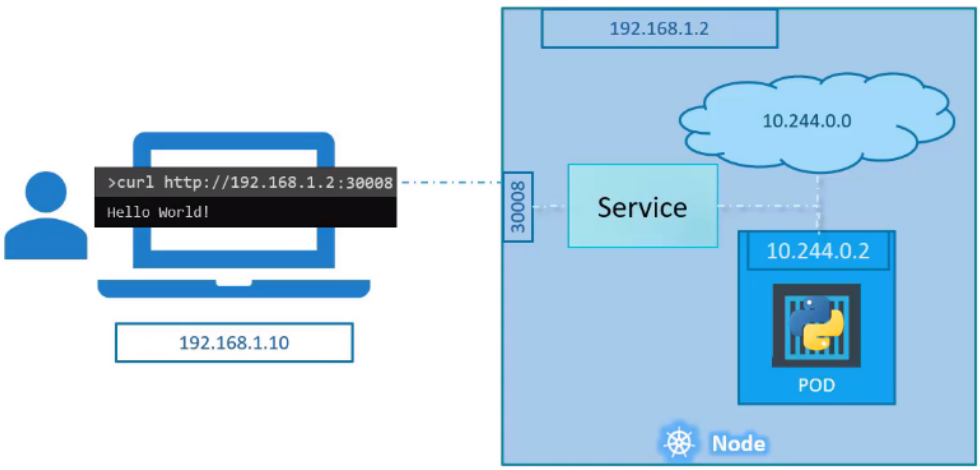

```
> kubectl get replicasets
NAME                          DESIRED   CURRENT   READY   AGE
myapp-deployment-67c749c58c   0         0         0       22m
myapp-deployment-7d57dbdb8d   5         5         5       20m
```

Summarize Commands

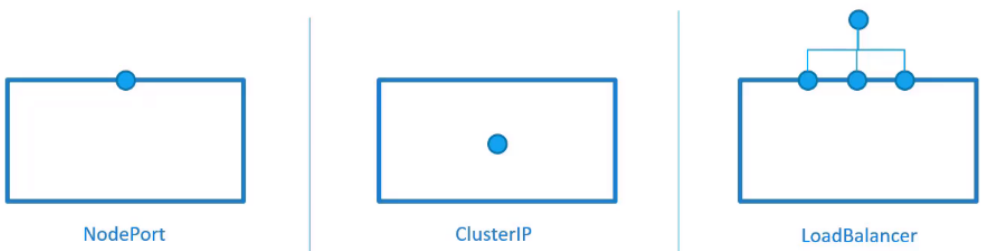
Create	`> kubectl create -f deployment-definition.yml`
Get	`> kubectl get deployments`
Update	`> kubectl apply -f deployment-definition.yml`
	`> kubectl set image deployment/myapp-deployment nginx=nginx:1.9.1`
Status	`> kubectl rollout status deployment/myapp-deployment`
	`> kubectl rollout history deployment/myapp-deployment`
Rollback	`> kubectl rollout undo deployment/myapp-deployment`

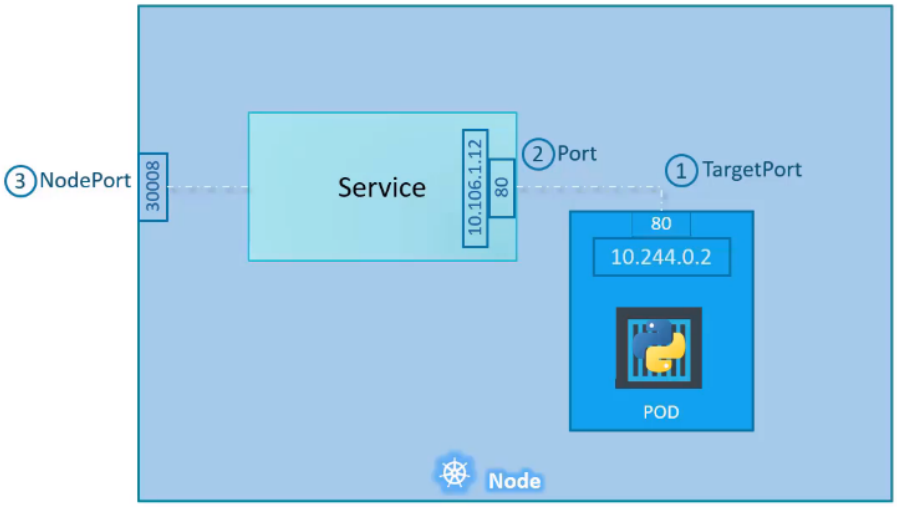
Services

External communication:



Services Types



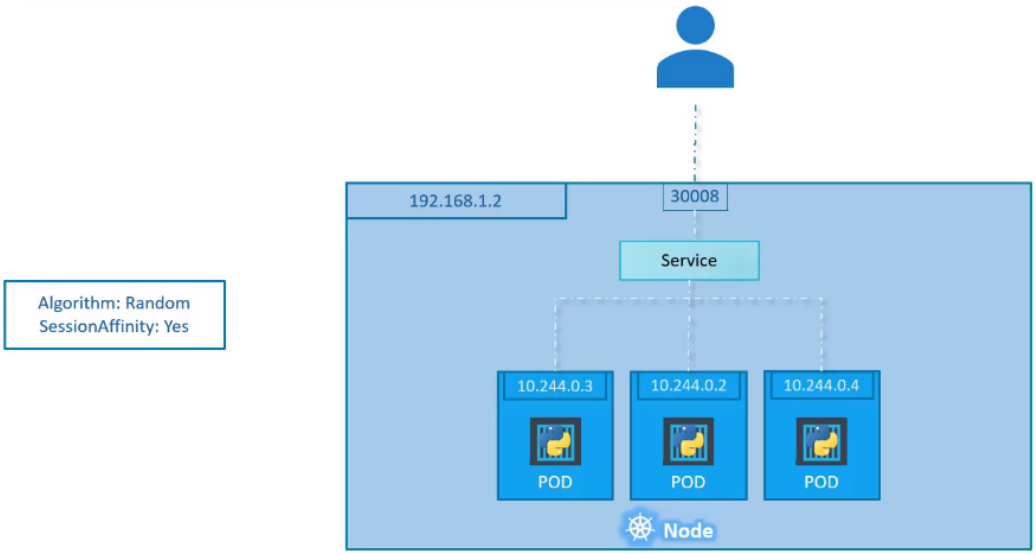



```yaml
apiVersion: v1
kind: Service
metadata:
  name: myapp-service
spec:
  type: NodePort
  ports:
  - port: 80
    targetPort: 80

  selector:
    app: myapp
    type: front-end
```

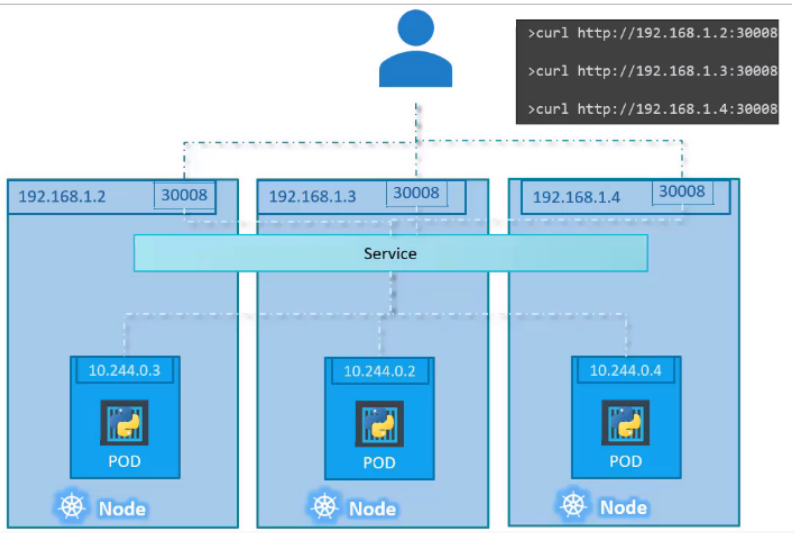
Service commands:

```
.\kubectl.exe get services
NAME            TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
kubernetes      ClusterIP   10.96.0.1       <none>        443/TCP        23h
myapp-service   NodePort    10.102.114.92   <none>        80:32565/TCP   23m


minikube ssh
ping <nodeip>:32565
```

NodePort



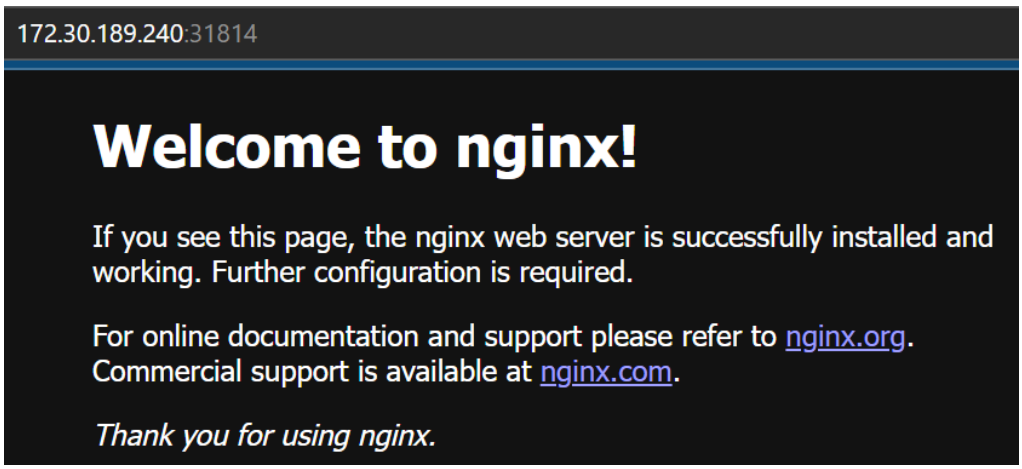
Service - NodePort



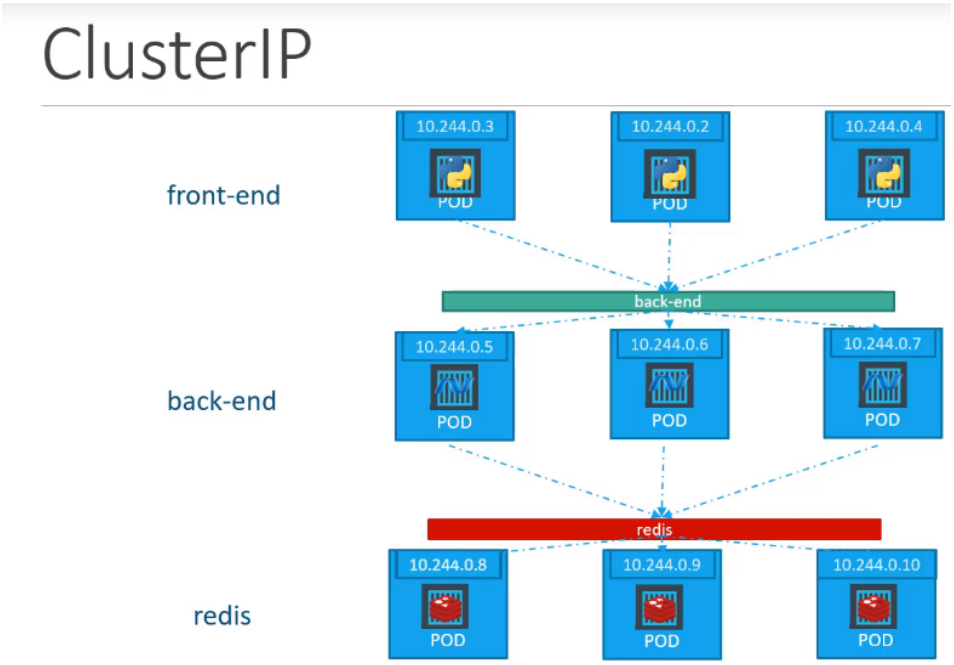

```
REM Debuging services
kubectl describe svc myapp-services
kubectl get pods --show-labels
```

Error Minikube port don't show up fix by using

```
sudo minikube start --driver=hyperv
```



ClusterIP



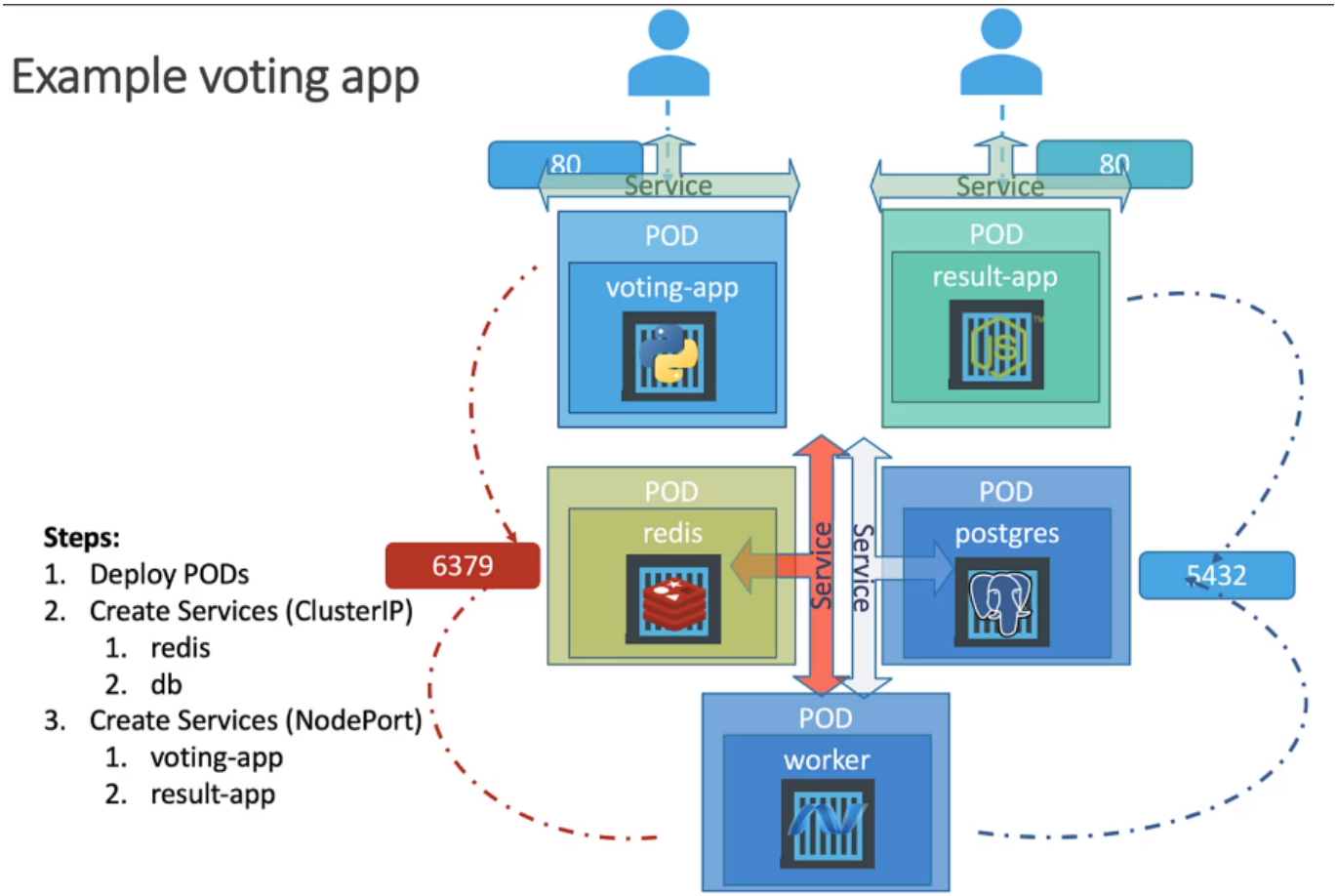
Cluster IP used for internal communication, to test using

```
minikube ssh
curl 10.97.32.132
```

```
apiVersion: v1
kind: Service
metadata:
  name: myapp-cluster
spec:
  type: ClusterIP
  selector:
    app: myapp
  ports:
  - port: 80
    targetPort: 80
```

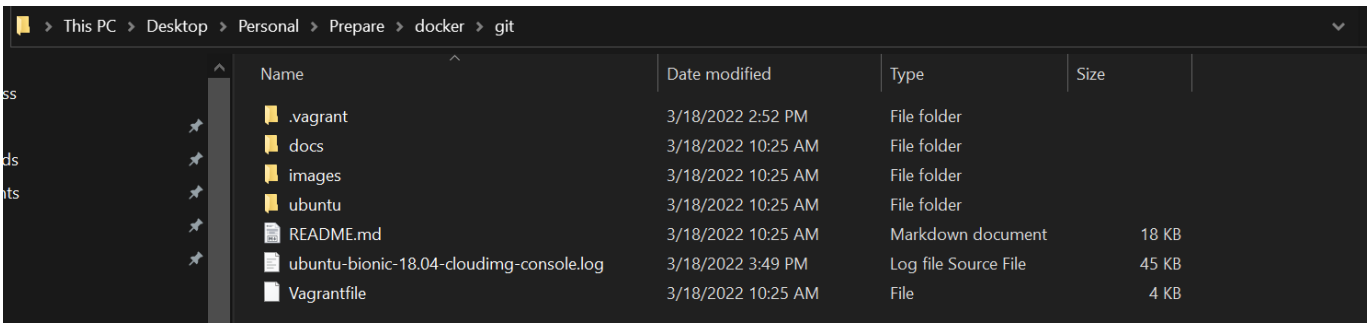
Load Balancer:

Microservices:



5.2 Kube adm

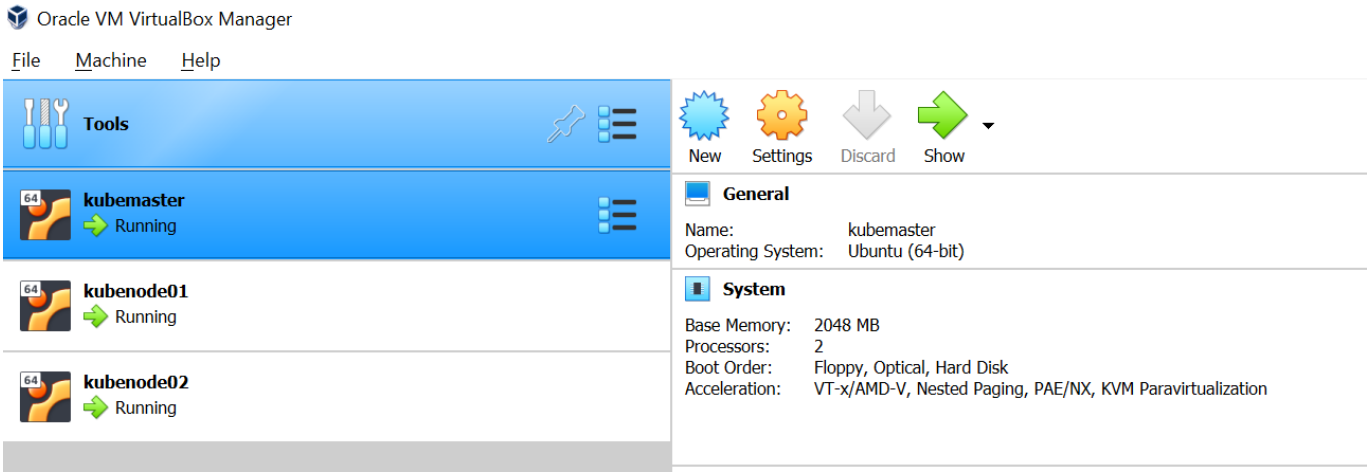
Go to directory configuration



Install Vagrant

Install VirtualBox

```
vagrant up
```



Install Docker Runtime on each machine

Installing kubeadm | Kubernetes

Creating a cluster with kubeadm | Kubernetes

```
sudo kubeadm init --pod-network-cidr 10.244.0.0/16 \
--apiserver-advertise-address=192.168.56.2 --config kubeadm-config.yaml
```

```
sudo journalctl -u kubelet
```

Error:

Failed to run kubelet" err="failed to run Kubelet: misconfiguration: kubelet cgroup driver: "systemd" is different from docker cgroup driver: "cgroupfs"

```
cat var/lib/kubelet/config.yaml
root@kubemaster:/etc/systemd/system# docker info
Client:
 Context:    default
 Debug Mode: false
 Plugins:
  app: Docker App (Docker Inc., v0.9.1-beta3)
  buildx: Docker Buildx (Docker Inc., v0.8.0-docker)
  scan: Docker Scan (Docker Inc., v0.17.0)

Server:
 Containers: 0
  Running: 0
  Paused: 0
  Stopped: 0
 Images: 7
 Server Version: 20.10.13
 Storage Driver: overlay2
  Backing Filesystem: extfs
  Supports d_type: true
  Native Overlay Diff: true
  userxattr: false
 Logging Driver: json-file
 Cgroup Driver: cgroupfs <---------- Here
 Cgroup Version: 1
 Plugins:
  Volume: local
  Network: bridge host ipvlan macvlan null overlay
  Log: awslogs fluentd gcplogs gelf journald json-file local logentries splunk syslog
 Swarm: inactive
 Runtimes: io.containerd.runc.v2 io.containerd.runtime.v1.linux runc
 Default Runtime: runc
 Init Binary: docker-init
 containerd version: 2a1d4dbdb2a1030dc5b01e96fb110a9d9f150ecc
 runc version: v1.0.3-0-gf46b6ba
 init version: de40ad0
 Security Options:
  apparmor
  seccomp
   Profile: default
 Kernel Version: 4.15.0-173-generic
 Operating System: Ubuntu 18.04.6 LTS
 OSType: linux
 Architecture: x86_64
 CPUs: 2
 Total Memory: 1.946GiB
 Name: kubemaster
 ID: UHFE:ASCK:GRWL:YDPO:HMFJ:LL6X:Q5BY:NB7E:22KW:UXAY:O4YQ:UKPD
 Docker Root Dir: /var/lib/docker
 Debug Mode: false
```

```
 Registry: https://index.docker.io/v1/
 Labels:
 Experimental: false
 Insecure Registries:
  127.0.0.0/8
 Live Restore Enabled: false

WARNING: No swap limit support


cat /usr/lib/systemd/system/docker.service
```

Fix by:

```
sudo vi /etc/systemd/system/kubelet.service.d/10-kubeadm.conf
```

```
vagrant@kubemaster:~$ sudo vi /etc/systemd/system/kubelet.service.d/10-kubeadm.conf
# Note: This dropin only works with kubeadm and kubelet v1.11+
[Service]
Environment="KUBELET_KUBECONFIG_ARGS=--bootstrap-kubeconfig=/etc/kubernetes/bootstrap-
kubelet.conf --kubeconfig=/etc/kubernetes/kubelet.conf"
Environment="KUBELET_CONFIG_ARGS=--config=/var/lib/kubelet/config.yaml"
Environment="KUBELET_CGROUP_ARGS=--cgroup-driver=cgroupfs" # <------ FIX HERE
# Environment="KUBELET_SYSTEM_PODS_ARGS=--pod-manifest-path=/etc/kubernetes/manifests
--allow-privileged=true --fail-swap-on=false"

# This is a file that "kubeadm init" and "kubeadm join" generates at runtime,
populating the KUBELET_KUBEADM_ARGS variable dynamically
EnvironmentFile=-/var/lib/kubelet/kubeadm-flags.env
# This is a file that the user can use for overrides of the kubelet args as a last
resort. Preferably, the user should use
# the .NodeRegistration.KubeletExtraArgs object in the configuration files instead.
KUBELET_EXTRA_ARGS should be sourced from this file.
EnvironmentFile=-/etc/default/kubelet
ExecStart=
ExecStart=/usr/bin/kubelet $KUBELET_KUBECONFIG_ARGS $KUBELET_CONFIG_ARGS
$KUBELET_KUBEADM_ARGS $KUBELET_EXTRA_ARGS $KUBELET_CGROUP_ARGS
```

Output of kubeadm init:

```
vagrant@kubemaster:~$ sudo kubeadm init --pod-network-cidr 10.244.0.0/16 --apiserver-advertise-address=192.168.56.2
[init] Using Kubernetes version: v1.23.5
[preflight] Running pre-flight checks
[preflight] Pulling images required for setting up a Kubernetes cluster
[preflight] This might take a minute or two, depending on the speed of your internet connection
[preflight] You can also perform this action in beforehand using 'kubeadm config images pull'
[certs] Using certificateDir folder "/etc/kubernetes/pki"
[certs] Using existing ca certificate authority
[certs] Using existing apiserver certificate and key on disk
[certs] Using existing apiserver-kubelet-client certificate and key on disk
[certs] Using existing front-proxy-ca certificate authority
[certs] Using existing front-proxy-client certificate and key on disk
[certs] Using existing etcd/ca certificate authority
[certs] Using existing etcd/server certificate and key on disk
[certs] Using existing etcd/peer certificate and key on disk
[certs] Using existing etcd/healthcheck-client certificate and key on disk
[certs] Using existing apiserver-etcd-client certificate and key on disk
[certs] Using the existing "sa" key
[kubeconfig] Using kubeconfig folder "/etc/kubernetes"
[kubeconfig] Using existing kubeconfig file: "/etc/kubernetes/admin.conf"
[kubeconfig] Using existing kubeconfig file: "/etc/kubernetes/kubelet.conf"
[kubeconfig] Using existing kubeconfig file: "/etc/kubernetes/controller-manager.conf"
[kubeconfig] Using existing kubeconfig file: "/etc/kubernetes/scheduler.conf"
[kubelet-start] Writing kubelet environment file with flags to file "/var/lib/kubelet/kubeadm-flags.env"
[kubelet-start] Writing kubelet configuration to file "/var/lib/kubelet/config.yaml"
[kubelet-start] Starting the kubelet
[control-plane] Using manifest folder "/etc/kubernetes/manifests"
[control-plane] Creating static Pod manifest for "kube-apiserver"
[control-plane] Creating static Pod manifest for "kube-controller-manager"
[control-plane] Creating static Pod manifest for "kube-scheduler"
[etcd] Creating static Pod manifest for local etcd in "/etc/kubernetes/manifests"
[wait-control-plane] Waiting for the kubelet to boot up the control plane as static Pods from directory "/etc/kubernetes/manifests". This can take up to 4m0s
[apiclient] All control plane components are healthy after 11.015170 seconds
[upload-config] Storing the configuration used in ConfigMap "kubeadm-config" in the "kube-system" Namespace
[kubelet] Creating a ConfigMap "kubelet-config-1.23" in namespace kube-system with the configuration for the kubelets in the cluster
NOTE: The "kubelet-config-1.23" naming of the kubelet ConfigMap is deprecated. Once the UnversionedKubeletConfigMap feature gate graduates to Beta the default name will become just "kubelet-config". Kubeadm upgrade will handle this transition transparently.
[upload-certs] Skipping phase. Please see --upload-certs
[mark-control-plane] Marking the node kubemaster as control-plane by adding the labels: [node-role.kubernetes.io/master(deprecated) node-role.kubernetes.io/control-plane node.kubernetes.io/exclude-from-external-load-balancers]
```

```
[mark-control-plane] Marking the node kubemaster as control-plane by adding the taints
[node-role.kubernetes.io/master:NoSchedule]
[bootstrap-token] Using token: j7bgf8.ipnizx19t6vq01ag
[bootstrap-token] Configuring bootstrap tokens, cluster-info ConfigMap, RBAC Roles
[bootstrap-token] configured RBAC rules to allow Node Bootstrap tokens to get nodes
[bootstrap-token] configured RBAC rules to allow Node Bootstrap tokens to post CSRs in
order for nodes to get long term certificate credentials
[bootstrap-token] configured RBAC rules to allow the csrapprover controller
automatically approve CSRs from a Node Bootstrap Token
[bootstrap-token] configured RBAC rules to allow certificate rotation for all node
client certificates in the cluster
[bootstrap-token] Creating the "cluster-info" ConfigMap in the "kube-public" namespace
[kubelet-finalize] Updating "/etc/kubernetes/kubelet.conf" to point to a rotatable
kubelet client certificate and key
[addons] Applied essential addon: CoreDNS
[addons] Applied essential addon: kube-proxy

Your Kubernetes control-plane has initialized successfully!

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

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

Alternatively, if you are the root user, you can run:

  export KUBECONFIG=/etc/kubernetes/admin.conf

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

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

kubeadm join 192.168.56.2:6443 --token j7bgf8.ipnizx19t6vq01ag --discovery-token-ca-
cert-hash sha256:2dfd649cef3a0d06875eabc63e8fbee97c6bf1d6addf888b979d72d9d657c41c
```

```
kubeadm join 192.168.56.2:6443 --token j7bgf8.ipnizx19t6vq01ag --discovery-token-ca-
cert-hash sha256:2dfd649cef3a0d06875eabc63e8fbee97c6bf1d6addf888b979d72d9d657c41c
```

Run this as regular user:

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

```
REM install network
kubectl apply -f "https://cloud.weave.works/k8s/net?k8s-version=$(kubectl version |
base64 | tr -d '\n')"


serviceaccount/weave-net created
clusterrole.rbac.authorization.k8s.io/weave-net created
clusterrolebinding.rbac.authorization.k8s.io/weave-net created
role.rbac.authorization.k8s.io/weave-net created
rolebinding.rbac.authorization.k8s.io/weave-net created
daemonset.apps/weave-net created
```