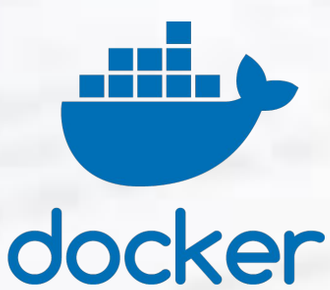
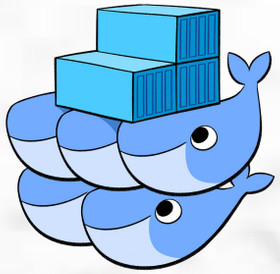




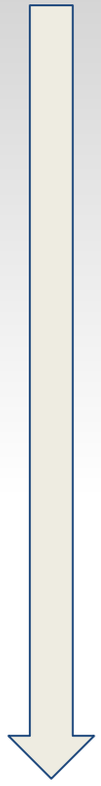
Docker- Docker Swarm Kubernetes

Msc.Eng. Anh Duy Tran
taduy@fit.hcmus.edu.vn

Docker Basics

History....

MAJOR
INFRASTRUCTURE
SHIFTS



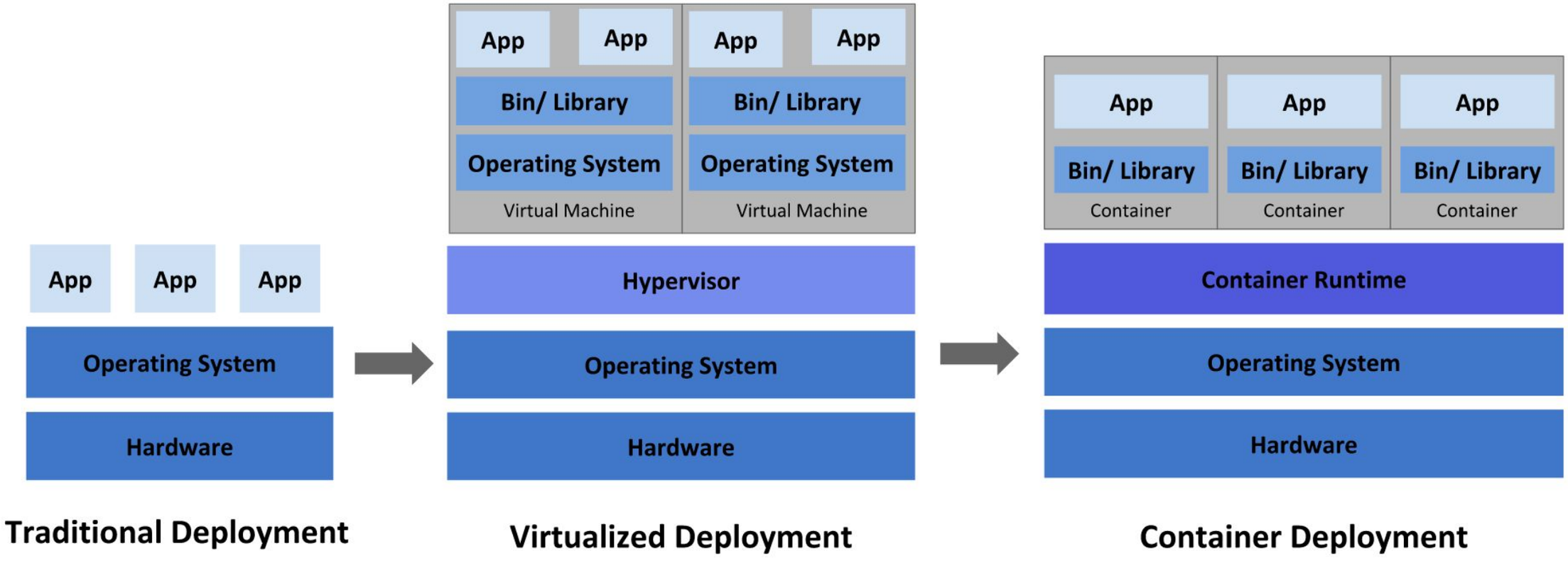
Mainframe to PC
90'S

Baremetal to Virtual
00'S

Datacenter to Cloud
10'S

Host to Container
(Serverless)

History....



https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/

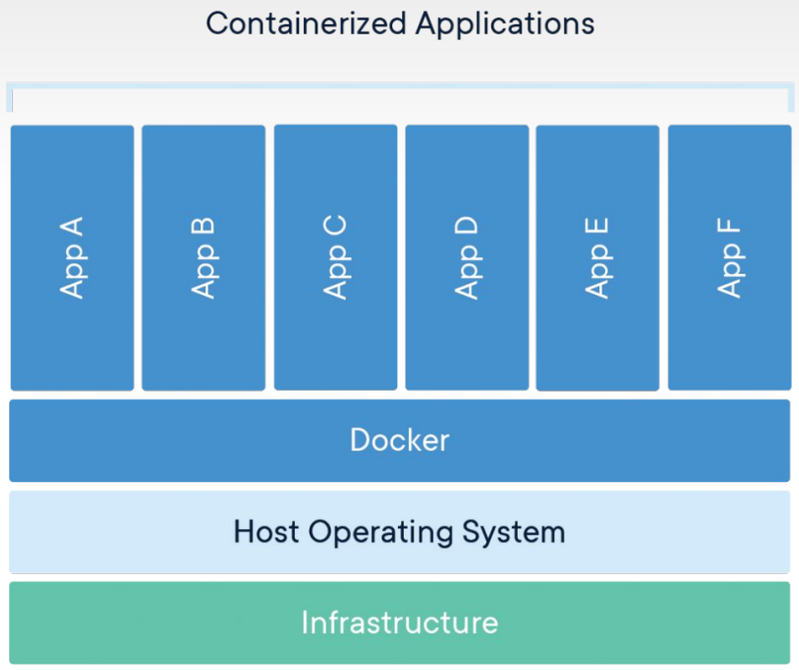
Why Docker?

- Problem in Software Development?
  - install all the necessary dependencies in our local machine.
  - then, we would have our application project ready to be deployed.
  - But, do you remember we had to install all the dependencies in our local machine?
  - Consuming several hours in configuring every single detail…
- Containerized applications was born

A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another

Why Docker?

- A containerized application will run on its own context, using a minimum set of binaries.
- Any container itself is exportable, destroyable and immutable.



https://www.docker.com/resources/what-container

Docker Images

- A Docker Image, in a logical way, is a container's template.
  - contains all the instructions and binaries necessary to build and run whatever containerized app.
  - You can find docker images for any technology, language, or framework.
- All primary images are stored in a large repository called Docker Hub

Docker Hello World

- Run docker hello-world
  - **`$ docker run hello-world`**

Docker Image Command

- Pull a docker image from remote repository
  - **`$ docker pull <image>`**
    - `Ex: $ docker pull nginx`
    - It downloads the latest official nginx image from the docker hub.
- You can specify the tag (simply version) of the image.
  - **`$ docker pull <image:tag>`**
    - `Ex: $ docker pull nginx:1.14.2`
- List all the images in local machine
  - **`$ docker image ls`**

Docker Run Container

- Run an *instance* of an image: Container.
  - **`$ docker container run --rm -p <host_port>:<container_port> --name <container_name> <image_name>`**
    - `--rm: force remove after shutdown container.`
    - `-p: port forwarding.`
  - `Ex: docker container run --rm -p 80:80 --name myNginx nginx`
  - *We build images, we run containers*
  - A container is just a process in your host machine.
- List all the containers are running in local machine
  - **`$ docker container ls`**
- You can also detach container from foreground and put them in background.
  - `Ex: $ docker container run --rm` **`-d`** `-p 81:80 --name myNginxBG nginx`

Docker Run Container

- Check the logs from the detached container
  - **`$ docker container logs <container>`**
    - `-f:` follow, keep the logs running.
- Docker run container with interactive mode, help to run command inside a container
  - **`$ docker container exec -it <container> <command>`**
    - `Ex: docker container exec -it myNginx bash`
  - **`-it:`** makes possible to interact with the bash. The letter “i” is for making our shell interactive and “t” is for getting the output from that shell (TTY).
  - we can also perform action without get shell access.
    - `Ex: $ docker container exec myNginx ls`

Docker Run Container

- There are some command to control the container.
  - **`$ docker container <action> <container_name>`**
    - `action can be restart, stop, start`
- There are some inactive container, you should remove it before starting the new one with the same name.
  - **`$ docker container ls -la`**
  - **`$ docker container rm -f <container_name>`**
    - `-f:` force remove active container.

Docker Storage

- How to tweak this image and include our files, code, and configurations. => Storage
- By using Storage in Docker we can persist data or files from the host system into our containers and vice-versa.
- There are two techniques:
  - **Bind Mounts:** are used to storing files and directories into the container. Most of the time you will use them for your project files.
  - **Volumes:** are used to persist data. You will use them for your Databases.

Docker Storage - Bind Mounts

- Take an example with nginx.
- Prepare your own source code in your machine, put them in the folder you want to mount.
  - **`$ docker container run --rm -p 80:80 --name myNginx -v $(pwd)/html:/usr/share/nginx/html nginx`**
  - It mount the current directory html to /usr/share/nginx/html inside container.

Docker Storage - Volumes

- Docker containers is destroyable. How can we keep persistent data? Like databases? => Use Volumes.
- Take the an example with mongodb
  - **$ docker container run -d --name myMongo mongo**
  - There are two volumes are assigned by default. Check this by using inspect command.
  - **$ docker container inspect myMongo**

```
...
"Mounts": [
            {
                "Type": "volume",
                "Name": "e0d5c4ad6b9d354518af3fc8b7e2acbfaad05661ba380471f87a7be570faafa0",
                "Source":
"/var/lib/docker/volumes/e0d5c4ad6b9d354518af3fc8b7e2acbfaad05661ba380471f87a7be570faafa0/_data",
                "Destination": "/data/configdb",
            ...
            },
            {
                "Type": "volume",
                "Name": "b919181b745c0547cda1ea73b2ab1a4b3fe942e13eab9d08c03a38a0cd37d84b",
                "Source":
"/var/lib/docker/volumes/b919181b745c0547cda1ea73b2ab1a4b3fe942e13eab9d08c03a38a0cd37d84b/_data",
                "Destination": "/data/db",
                        }
        ]
```

Docker Storage - Volumes

- Docker containers is destroyable. How can we keep persistent data? Like databases? => Use Volumes.
- Take the an example with mongodb
  - **`$ docker container run -d --name myMongo mongo`**
  - There are two volumes are assigned by default. Check this by using inspect command.
  - **`$ docker container inspect myMongo`**
  - You can list those volumes by:
    - **`$ docker volume ls`**
  - Try to access the container and modify that database:
    - **`$ docker container exec -it myMongo mongo`**
    - **`> db.getCollection('my_collection').insert({message:'Hello Docker'})`**
    - **`$ docker container restart myMongo`**

Docker Storage - Volumes

- Take the an example with mongodb
  - Then you can login to mongo shell and find this record
  - `> db.getCollection('my_collection').find({})`

    `{ "_id" : ObjectId("5c68415ce741bb40f9d91291"), "message" : "Hello Docker" }`
- When you delete the container. The volumes will still be there,
- … but if we start new DB containers, these are going to use new volumes instead of the old ones.
- Create your own volume and mount it to your container.
  - `$ docker volume create my-mongo-volume`
  - `$ docker container run -d -v my-mongo-volume:/data/db --name myNewMongo mongo`

Docker Networking

- In software development, for instance - web, a minimum stack project consists of an API, a frontend app, and a Database entity.
- We can also work with microservices architecture with many artifacts running simultaneously.
- *How can we connect all these pieces using Docker?*
- If you want to connect two or more containers, so that they can communicate with each other, **they have to share the same network**.
- In Docker, there are multiple ways to achieve this, but not all of them are good.

Docker Networking

- In Docker, there are multiple ways to achieve this, but not all of them are good.
  - Creating a specific bridge network
  - Publishing all the required ports and running the containers with the -- net=”host” option. (Avoid this).
  - Using the “link” option (--link). It still works, but it’s planned to be removed soon.

Docker Networking - Example

- Create your own network
  - `$ docker network create my-network`
- Start a PHP-apache container and link it to our new network created.
  - `$ docker container run -d --rm --network my-network --name php-server php:apache`
- Start MySQL database.
  - `$docker container run -d --network my-network -e MYSQL_ROOT_PASSWORD=123456 --name mysql-db mysql`
- These two containers should be able to communicate with each other.

Docker Networking - Example

- Let's get shell access into the PHP instance:
  - `$ docker container exec -it php-server bash`
- Install ping command inside this container
  - `$ apt-get update && apt-get install iputils-ping`
- Try to ping mysql-db
  - `$ ping -c 3 mysql-db`

Dockerfiles

- So far, we run the container of *built-in* image. How about create your own images or custom images. => You need to *code* :)
- Now, you'll learn to write Dockerfile to custom image.
- List of common commands in Dockerfile:
  - **FROM <image>:<tag>** (from an image, first instruction in the Dockerfile.)
  - **RUN <command>** (shell form, the command is run in a shell)
  - **CMD ["<executable>","<param1>","<param2>"]** (exec form, this is the preferred form)
  - **EXPOSE <port> [<port> ...]** (the container listens on the specified network port(s) at runtime)
  - **ENV <key> <value>** (sets the environment variable <key> to the value <value>)
  - **COPY <src> [<src> ...] <dest>** (copy file from src (in host) to dest (in container))
  - **VOLUME ["<path>", ...]** (mount point)
  - **WORKDIR </path/to/workdir>** (set working directory)

https://kapeli.com/cheat_sheets/Dockerfile.docset/Contents/Resources/Documents/index

Dockerizing a static website - Example

- “Dockerize” is the process to take a web application or element of your stack, and make it work using a Docker container.
- Clone this repo, a simple web
- We try to copy the source code of this project into new custom image.
- Create a custom Nginx image that uses the “default.conf” file and the “src” folder from our web project.
- Dockerfile:

```
FROM nginx
COPY src /usr/share/nginx/html
COPY default.conf /etc/nginx/conf.d/default.conf
```

Dockerizing a static website - Example

- Build that image:
  - **`$ docker build -t custom-nginx .`**
  - **`-t:`** indicate the name of the image
  - dot at the end indicates the Dockerfile location.
- Run the container of that image:
  - **`$ docker container run --rm -p 80:8080 --name myCustomNginx custom-nginx`**

Dockerizing a NodeJS Express - Example

- This time, we'll make more complex Dockerfile.
- Clone this repo, an Express app, which is a popular web framework for NodeJS.
- To dockerizing them, we need:
  - install the dependencies defined on the "package.json" file
  - copy all the files and folders required by our application.
  - The application runs using the command: **node ./bin/www**
  - It runs through the port 3000.

Dockerizing a NodeJS Express - Example

- Dockerfile:

```
FROM node:10.15.1

EXPOSE 3000

RUN npm i npm@latest -g

RUN mkdir /opt/app && chown node:node /opt/app

WORKDIR /opt/app

USER node

COPY package.json package-lock.json* ./

RUN npm install --no-optional && npm cache clean --force

COPY . .

CMD [ "node", "./bin/www" ]
```

Dockerizing a NodeJS Express - Example

- Dockerignore (prevent specific unnecessary files or directories to be copied into our custom image):

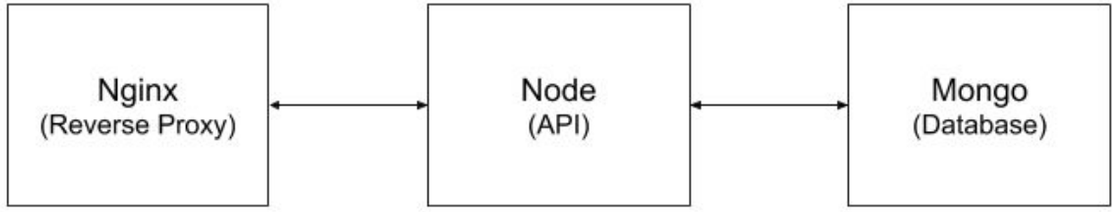
```
.git
.gitignore
.dockerignore
*Dockerfile*
node_modules
LICENSE
README.md
```

Dockerizing a NodeJS Express - Example

- Build that image:
  - **`$docker build -t express-app .`**
- Run the container of that image:
  - **`$ docker container run --rm -p 80:3000 --name myExpressApp express-app`**

Docker Compose

- Can we do all the steps so far by simple one file :) => Docker Compose.
- With “Docker Compose” we can forget of using all the “build”, “run” and “create volume” instructions and centralize those steps within a file called ***docker-compose.yml***, which is pretty much a recipe for our application.
- Suppose you have to compose many containers and organize them as a complete software project. Example: web application



Docker Compose - Example

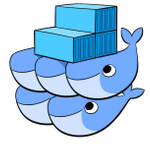
- Clone this repo, an example of a project which contains, nginx, nodeAPI (a common web project). (Actually, you've done it in previous two steps).
- Some files you need to consider:
  - ***Dockerfile (root directory)***: dockerizing a NodeJS application
  - ***Dockerfile (nginx directory)***: dockerizing a static website, only copying the "default.conf" file.
  - ***default.conf (nginx directory)***: Nginx config file which has the proxy-reverse configuration.
  - ***index.js (routes directory)***: database connection
- http://localhost/app and should visualize the Express index page printing some Mongo information.

Docker Compose - Example

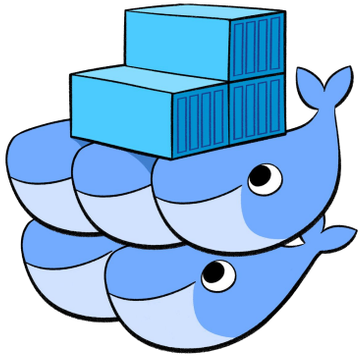
- **docker-compose.yml** file:

```
version: "3"
services:
  proxy-server:
    build: ./nginx
    ports:
      - 80:80
  app:
    build: .
    volumes:
      - ./public:/opt/app/public
      - ./routes:/opt/app/routes
      - ./views:/opt/app/views
  db:
    image: mongo:latest
    environment:
      MONGO_INITDB_ROOT_USERNAME: "root"
      MONGO_INITDB_ROOT_PASSWORD: "123456"
    volumes:
      - "my-mongo-volume:/data/db"
volumes:
  my-mongo-volume:
```

Docker Compose - Example

- Run the compose:
  - `$ docker-compose up --build`
- Access ***http://localhost/***:
  - you should see should see your regular Nginx default page.
- Access ***http://localhost/app***:
  - you should see the Express homepage showing some Mongo DB information.
- Docker-compose is a utility designed for **local development**, and its usage in **production** environments is highly discouraged.
- Then… :), here we go, my friends  

Docker Swarm



Why Docker Swarm?

- How to Building →Shipping → Running containers at scale?
- Docker Swarm is one of the tools to do this kind thing -> Container Orchestration
- Why do we want a Container Orchestration System? Suppose you have to run hundreds of containers, in a distributed mode.
  - Health Checks on the Containers
  - Launching a fixed set of Containers for a particular Docker image
  - Scaling the number of Containers up and down depending on the load
  - Performing rolling update of software across containers
  - and more…

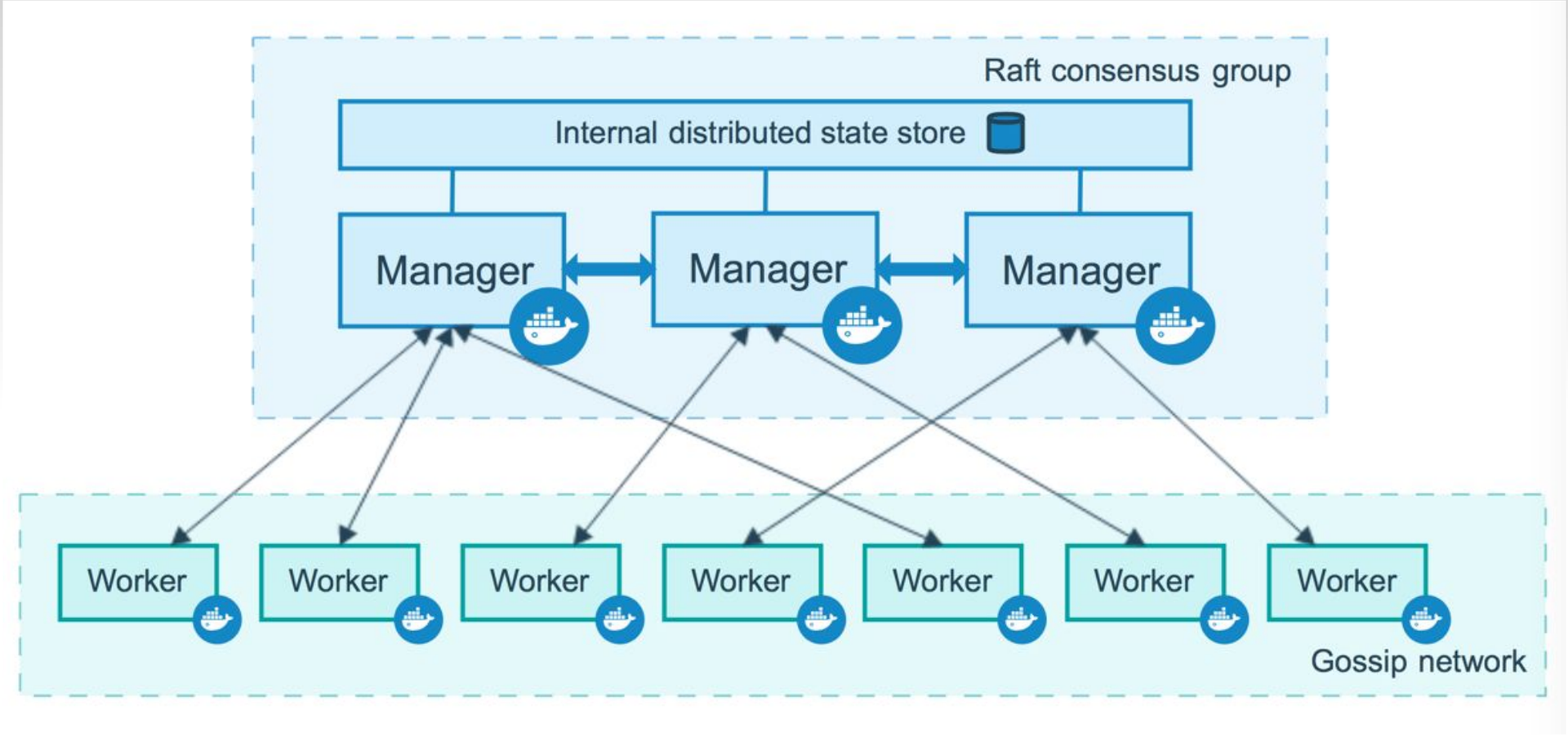
What is Docker Swarm

- The cluster management and orchestration features embedded in the Docker Engine are built using swarmkit
- A swarm consists of multiple **Docker hosts** which run in swarm mode and act as **managers** (to manage membership and delegation) and **workers** (which run swarm services).
- When you create a service, you define its optimal state (number of replicas, network and storage resources available to it, ports the service exposes to the outside world, and more). Docker works to maintain that desired state.

What is Docker Swarm

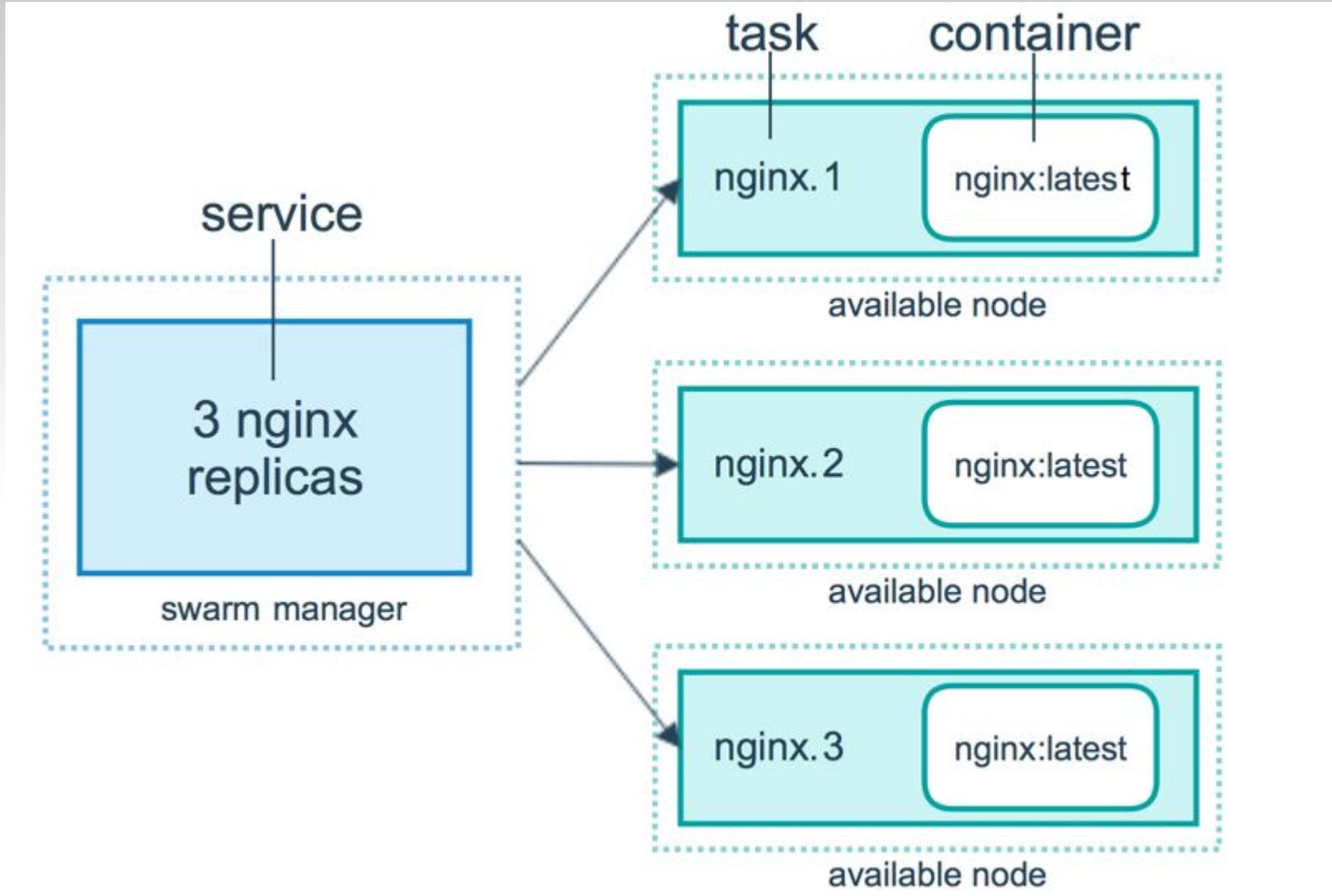
- A **node** is an instance of the Docker engine participating in the swarm.
- To deploy your application to a swarm, you submit a service definition to a **manager node**.
- **Worker nodes** receive and execute tasks dispatched from manager nodes.
- **A service** is the definition of the tasks to execute on the manager or worker nodes. It is the central structure of the swarm system and the primary root of user interaction with the swarm**.**
- In the **replicated services** model, the swarm manager distributes a specific number of replica tasks among the nodes based upon the scale you set in the desired state.
- A **task** carries a Docker container and the commands to run inside the container.

How nodes work



https://docs.docker.com/engine/swarm/how-swarm-mode-works/nodes/

How service work



https://docs.docker.com/engine/swarm/how-swarm-mode-works/services/

Docker Swarm - Example

- Create a set of docker-machine (nodes) in Docker Swarm: 1 manager and 5 worker.
  - **`$ docker-machine create --driver virtualbox manager1`**
  - **`$ docker-machine create --driver virtualbox worker1   #and 2,3,4,5,`**
  - You can use other driver for simulation.
- Check set of machine with ***docker-machine ls***.
- Check IP Address of manager1.
  - **`$ docker-machine ip manager1`**
- If you want to access the machine, you can simply use ***ssh*** command.
  - **`$ docker-machine ssh <machine-name>`**

Setting up the Swarm - Example

- First, you need to access the *manager1* and initialize swarm in there. (Swarm is not enable by default)
  - **`$ docker-machine ssh manager1`**
  - **`$ docker swarm init --advertise-addr MANAGER_IP`** *`#inside manager1`*
  - Where **`MANAGER_IP`** is the public IP Address of *manager1*
- After initialize a swarm, you can see the information for other nodes to join this swarm, for example:

```
To add a worker to this swarm, run the following command:

docker swarm join \
— token SWMTKN-1-5mgyf6ehuc5pfbmar00njd3oxv8nmjhteejaald3yzbef7osl1-ad7b1k8k3bl3aa3k3q13zivqd \
192.168.1.8:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.
docker@manager1:~$
```

Setting up the Swarm - Example

- You need token for other nodes to join swarm as manager or worker. Check this token by:
  - **$ docker swarm join-token worker**
  - **$ docker swarm join-token manager**
- Now you can join other nodes to swarm. Go ssh to each node and use provided command in previous step to join as worker:
  - **$ docker swarm join **
    – token
    **SWMTKN-1-5mgyf6ehuc5pfbmar00njd3oxv8nmjhteejaald3yzbef7osl1-ad7b1k8k3bl3aa3k3q13zivqd **
    192.168.1.8:2377
  - *Note: the token and IP Address may be different in your case.*
- In manager1, recheck the result with ***docker node ls.*** Then use ***docker info*** to show more

Create a Service - Example

- Now that we have our swarm up and running, it is time to schedule our containers on it
- All we are going to do is tell the manager to run the containers for us and it will take care of scheduling out the containers, sending the commands to the nodes and distributing it.
- To start a service, you would need to have the following:
  - What is the Docker image that you want to run. *In our case, we will run the standard **nginx** image that is officially available from the Docker hub.*
  - We will expose our service on **port** 80.
  - We can specify the number of containers (or instances) to launch. This is specified via the **replicas** parameter.
  - We will decide on the **name** for our service. And keep that handy.

Create a Service - Example

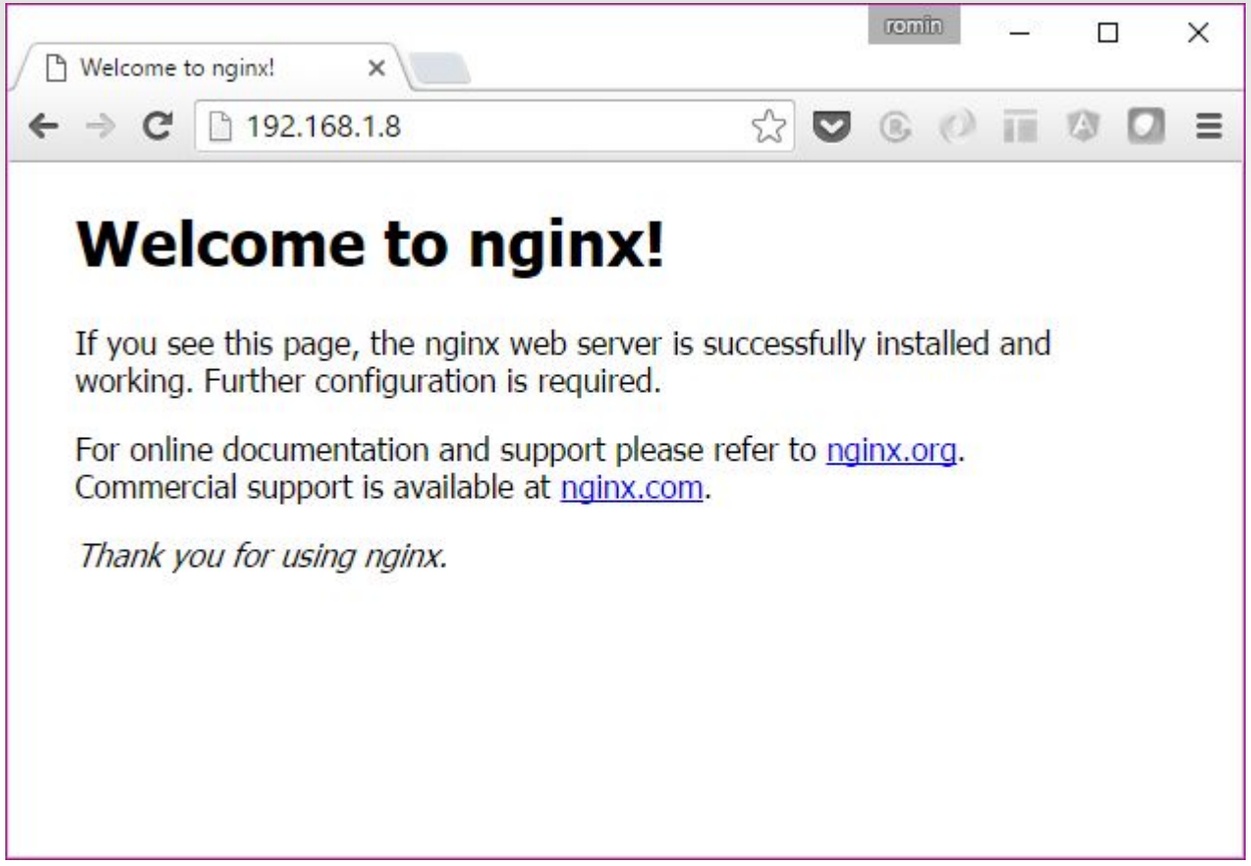
- Launch 5 replicas of the ***nginx*** container; ssh to manager1 beforehand. Then create a service:
  - **`$ docker service create --replicas 5 -p 80:80 --name web nginx`**
- Check status of the service:
  - **`$ docker service ls`**
  - **`$ docker service ps web`**
- It takes time for our service to run in all 5 replicas.

```
docker@manager1:~$ docker service ps web
ID  NAME   IMAGE  NODE      DESIRED STATE  CURRENT STATE      ERROR
7i*  web.1  nginx  worker3   Running        Running 4 minutes ago
17*  web.2  nginx  manager1  Running        Running 7 minutes ago
ey*  web.3  nginx  worker2   Running        Running 9 minutes ago
bd*  web.4  nginx  worker5   Running        Running 8 minutes ago
dw*  web.5  nginx  worker4   Running        Running 9 minutes ago
```

- If you do a docker ps on the manager1 node right now, you will find that the nginx daemon has been launched.

Accessing the Service - Example

- You can access the service by hitting any of the manager or worker nodes, just type their IP in your host machine's browser.



Scaling up and Scaling down - Example

- To scale your service, use **docker service scale**. So far, we have 5 containers, how about enlarge it to 8. Again, ssh to manager1 :
  - **$ docker service scale web=8**
- Check status of the service:
  - **$ docker service ls**

```
ID            NAME REPLICAS IMAGE COMMAND
ctolq1t4h2o8 web  5/8      nginx
```

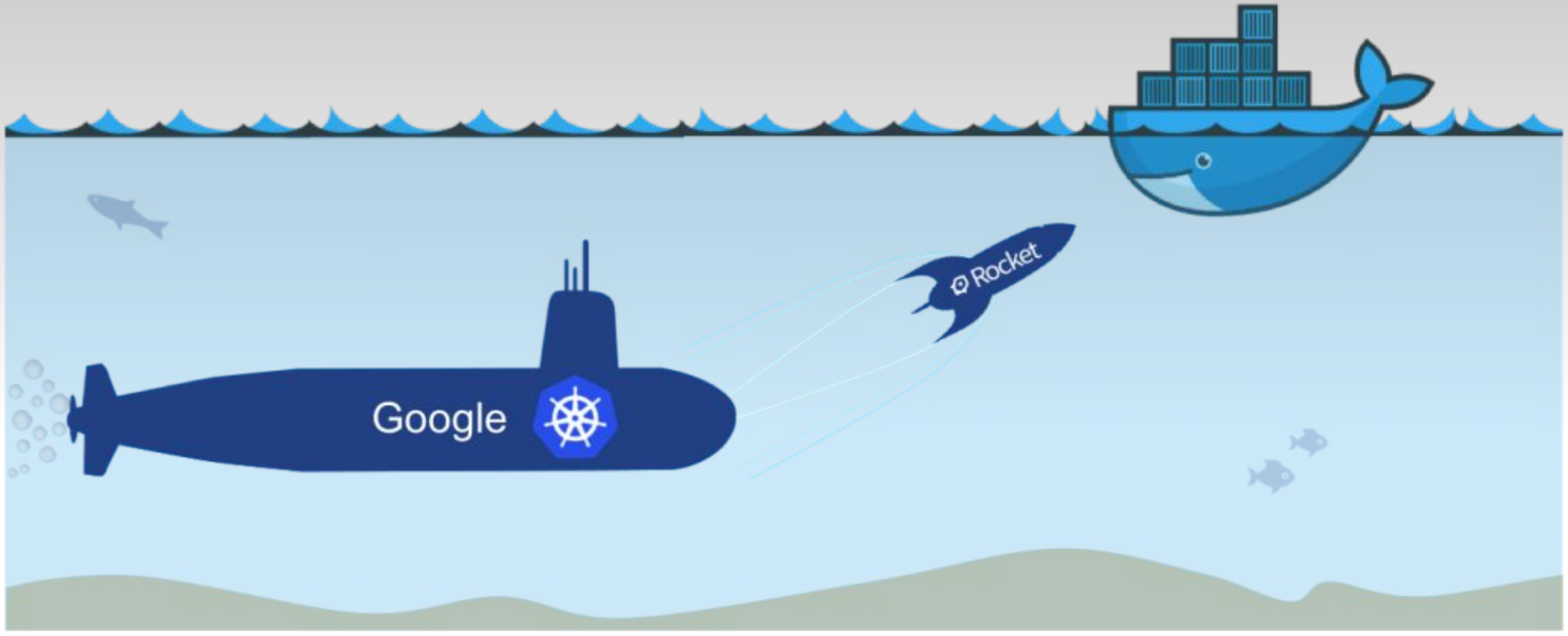
- It takes time for our service to schedule new containers.

```
docker@manager1:~$ docker service ps web
ID   NAME   IMAGE  NODE      DESIRED STATE  CURRENT STATE                ERROR
7i*  web.1  nginx  worker3   Running    Running 14 minutes ago
17*  web.2  nginx  manager1  Running    Running 17 minutes ago
ey*  web.3  nginx  worker2   Running    Running 19 minutes ago
bd*  web.4  nginx  worker5   Running    Running 17 minutes ago
dw*  web.5  nginx  worker4   Running    Running 19 minutes ago
8t*  web.6  nginx  worker1   Running    Starting about a minute ago
b8*  web.7  nginx  manager1  Running    Ready less than a second ago
0k*  web.8  nginx  worker1   Running    Starting about a minute ago
```

Some other commands in Docker Swarm

- Inspecting nodes :
  - `$ docker node inspect self  #or worker1`
- Draining a node, sometime you should shut down a node for maintenance:
  - `$ docker node update --availability drain worker1`
- Remove the service
  - `$ docker service rm web`
- Applying Rolling Updates
  - `$ docker service update --image <imagename>:<version> web`
- Learn more with *Docker Swarm on Google Computer Engine*

Kubernetes


Rocket
Google


Docker
Docker Compose
Docker Swarm
Kubernetes
meme-arsenal.ru

Why Kubernetes?

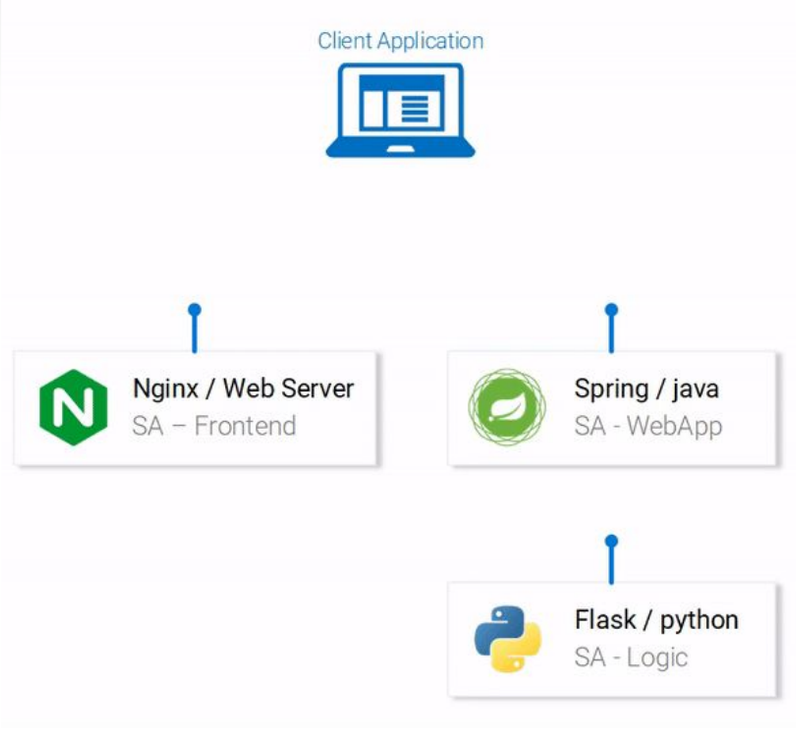
- How to Building →Shipping → Running containers at scale?
- Like Docker Swarm, it's a Container Orchestration
- Kubernetes is a portable, extensible, open-source platform for **managing containerized** workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available.
- The name Kubernetes originates from Greek, meaning helmsman or pilot. **Google** open-sourced the Kubernetes project in 2014.
  - Service discovery and load balancing
  - Storage orchestration
  - Automated rollouts and rollbacks
  - Automatic bin packing
  - Self-healing
  - Secret and configuration management

The Hands on Kubernetes

- Running a Microservice based application on your computer.
- Building container images for each service of the Microservice application.
- Introduction to Kubernetes. Deploying a Microservice based application into a Kubernetes Managed Cluster.

The Hands on Kubernetes

- The application (clone this) consists of three microservices. Each has one specific functionality:
  - **SA-Frontend**: a Nginx web server that **serves our ReactJS** static files.
  - **SA-WebApp**: a Java Web Application that **handles requests** from the frontend.
  - **SA-Logic**: a python application that **performs Sentiment Analysis**.



Setting up React App - Frontend

- Setting up React for Local Development. You need NodeJS và NPM installed on your computer.
- In **sa-frontend** directory, Downloads all the Javascript dependencies of the React application and places them in the folder node_modules:
  - `$ npm install`
- Start our react application and by default you can access it on **localhost:3000**:
  - `$ npm start`

Setting React App + Nginx - Frontend

- Build our application into static files and serve them using a web server for production. In **sa-frontend** directory, build the React application:
  - `$ npm run build`
- Start the Nginx WebServer:
  - `$ sudo systemctl restart nginx`
- Move the contents of the **sa-frontend/build** folder to [nginx_dir]/html
  - `$ sudo cp -r build/* /var/www/html/`
- Now you can access this web by default port 80 of Nginx server in browser: localhost:80

Setting up React App - Frontend

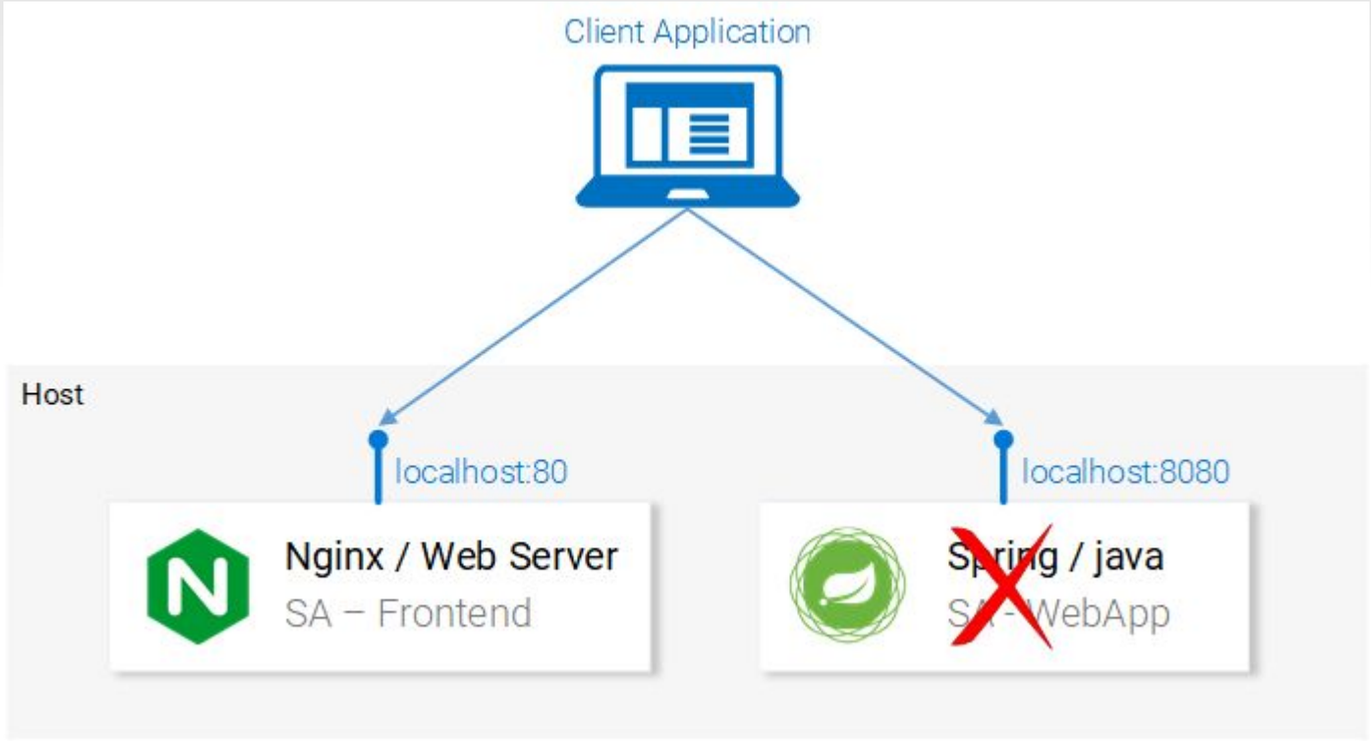
- Inspect the Code in src/App.js, check the *analyzeSentence()* method which is called when you press the Send button.

```
analyzeSentence() {
    fetch('http://localhost:8080/sentiment', {  // #1
        method: 'POST',
        headers: {
            'Content-Type': 'application/json'
        },
        body: JSON.stringify({
                        sentence: this.textField.getValue()})// #2
    })
        .then(response => response.json())
        .then(data => this.setState(data));  // #3
}
```

  - *#1: URL at which a POST call is made.*
  - *#2: The Request body sent to that application as displayed below:*
  - *#3: The response updates the component state. This triggers a re-render of the component.*

Setting up React App - Frontend

- The problem now is, we don't have anything to listen on localhost:8080



Setting up the Spring Web Application - Backend

- Install JDK8 and Maven beforehand:
- Packaging the Application into a Jar. In **sa-webapp**:
  - `$ mvn install`
- This will generate a folder named **target**. In the folder **target** we have our Java application packaged as a jar: '**sentiment-analysis-web-0.0.1-SNAPSHOT.jar**'
- Starting the Java Application, in target directory:
  - `java -jar sentiment-analysis-web-0.0.1-SNAPSHOT.jar`
- There is an error: *Error creating bean with name 'sentimentController': Injection of autowired dependencies failed; nested exception is java.lang.IllegalArgumentException: Could not resolve placeholder 'sa.logic.api.url' in value "${sa.logic.api.url}"*

Setting up the Spring Web Application - Backend

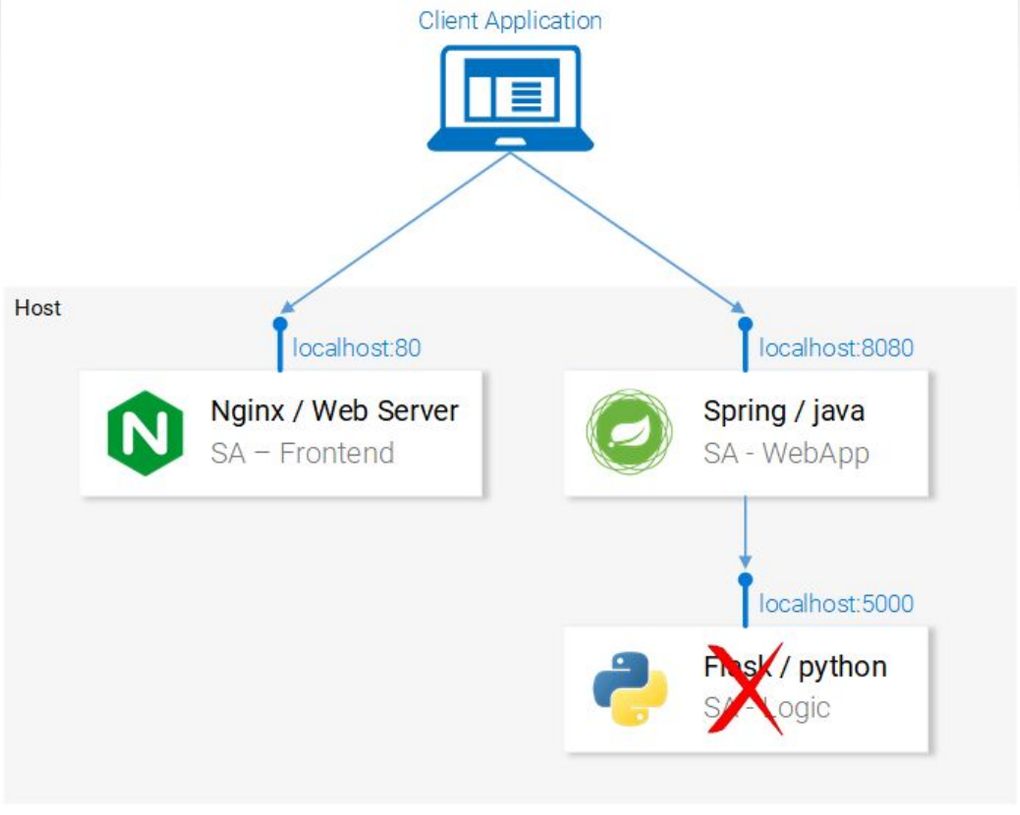
- Inspect the Code in SentimentController

```
@CrossOrigin(origins = "*")
@RestController
public class SentimentController {

    @Value("${sa.logic.api.url}")    // #1
    private String saLogicApiUrl;

    @PostMapping("/sentiment")
    public SentimentDto sentimentAnalysis(
        @RequestBody SentenceDto sentenceDto)
    {
        RestTemplate restTemplate = new RestTemplate();

        return restTemplate.postForEntity(
                saLogicApiUrl + "/analyse/sentiment",    // #2
                sentenceDto, SentimentDto.class)
                .getBody();
    }

}
```

- *#1: The SentimentController has a field named saLogicApiUrl. The field get's defined by the property sa.logic.api.url*
- *#2: The String saLogicApiUrl is concatenated with the value "/analyse/sentiment". Together they form the URL to make the request for Sentiment Analysis.*

Setting up the Spring Web Application - Backend

- To fix this error, we should define the Properties. It can be done in command, and suppose we chose ***localhost:5000*** where the *next* Python application will be run:
  - **`java -jar sentiment-analysis-web-0.0.1-SNAPSHOT.jar --sa.logic.api.url=http://localhost:5000`**



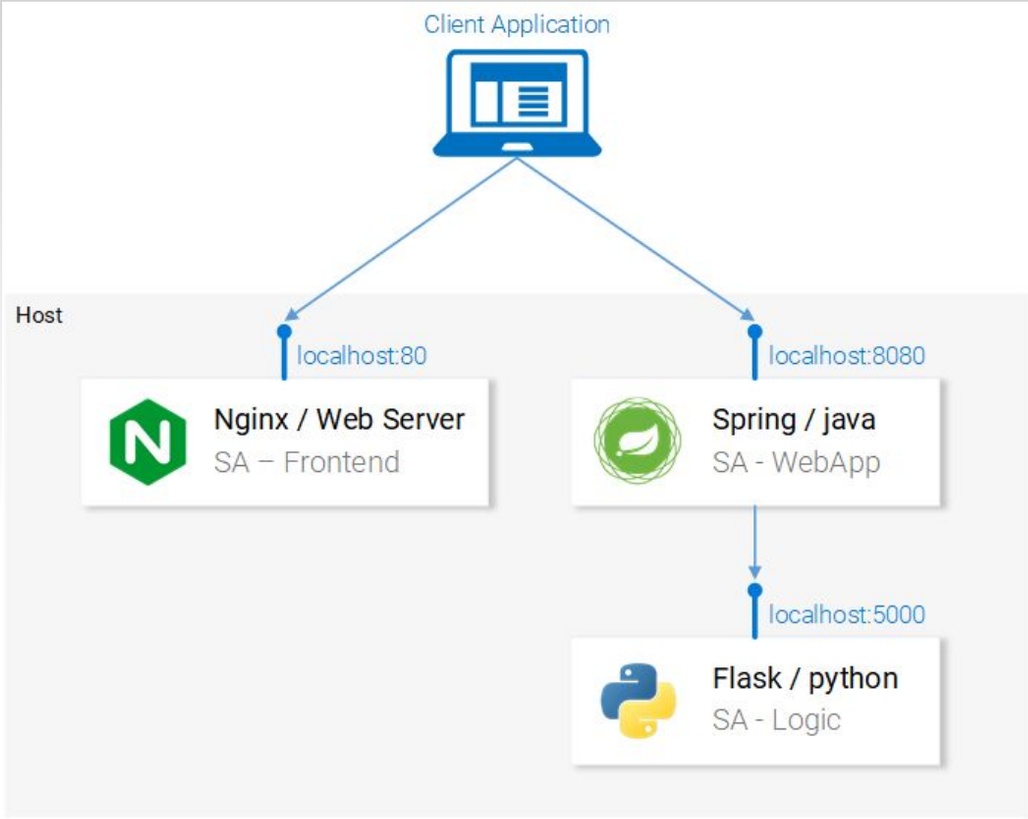
Setting up the Python Application - Logic Component

- Install Python3 and Pip beforehand:
- Install all requirements. In **sa-logic/sa**:
  - `$ python3 -m pip install -r requirements.txt`
  - `$ python3 -m textblob.download_corpora`
- Starting the Python Application, in target directory:
  - `$ python3 sentiment_analysis.py`

Setting up the Python Application - Logic Component

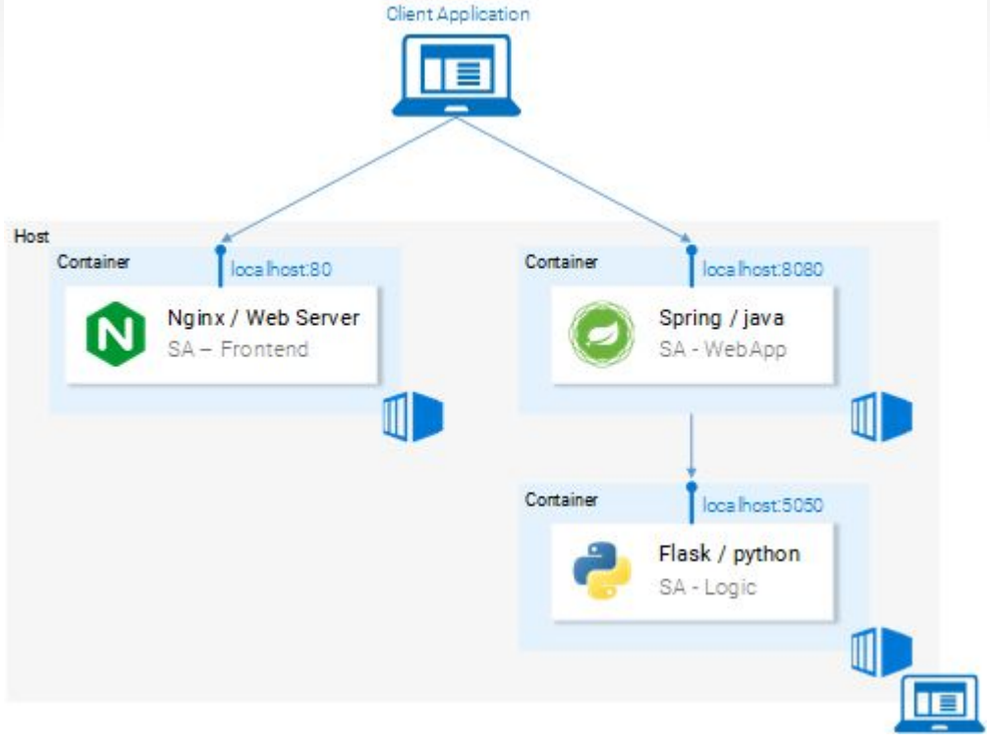
- Inspect the Code in sentiment_analysis.py

```
from textblob import TextBlob
from flask import Flask, request, jsonify

app = Flask(__name__)                                          #1

@app.route("/analyse/sentiment", methods=['POST'])            #2
def analyse_sentiment():
    sentence = request.get_json()['sentence']                  #3
    polarity = TextBlob(sentence).sentences[0].polarity        #4
    return jsonify(                                            #5
        sentence=sentence,
        polarity=polarity
    )

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)                         #6
```



Building container images

- Build that docker image for Python Application:
  - `$ docker build -f Dockerfile -t $DOCKER_USER_ID/sentiment-analysis-logic .`
- Run that docker container:
  - `$ docker run -d -p 5050:5000 $DOCKER_USER_ID/sentiment-analysis-logic`



Building container images

- Dockerfile for Java Application

```
ENV SA_LOGIC_API_URL http://localhost:5000
...
EXPOSE 8080
```

- Build that docker image:
  - **`$ docker build -f Dockerfile -t $DOCKER_USER_ID/sentiment-analysis-web-app .`**
- Run that docker container:
  - **`docker run -d -p 8080:8080 -e SA_LOGIC_API_URL='http://<container_ip or docker machine ip>:5000' $DOCKER_USER_ID/sentiment-analysis-web-app`**

Building container images

- Dockerfile for SA-Frontend

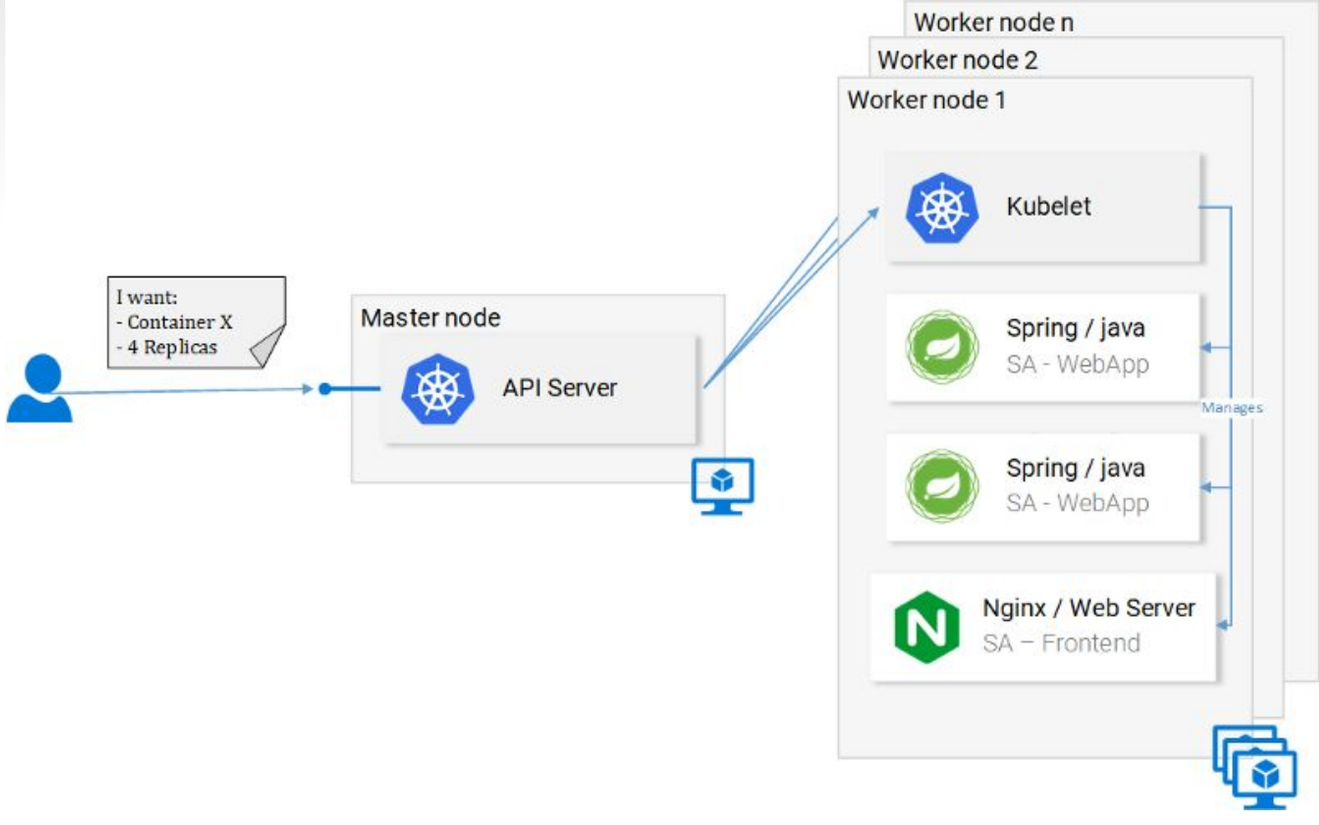
```
FROM nginx
COPY build /usr/share/nginx/html
```

- Build that docker image:
  - **`docker build -f Dockerfile -t $DOCKER_USER_ID/sentiment-analysis-frontend .`**
- Push that docker image:
  - **`docker push $DOCKER_USER_ID/sentiment-analysis-frontend`**
- Run that docker container:
  - **`docker run -d -p 80:80 $DOCKER_USER_ID/sentiment-analysis-frontend`**

Kubernetes… actually comes in

Abstracting the underlying infrastructure

- Kubernetes abstracts the underlying infrastructure by providing us with a simple API to which we can send requests.
- For example, it is as simple as requesting “Kubernetes spin up **4 containers** of the **image x**”. Then Kubernetes will find **under-utilized nodes** in which it will spin up the new containers

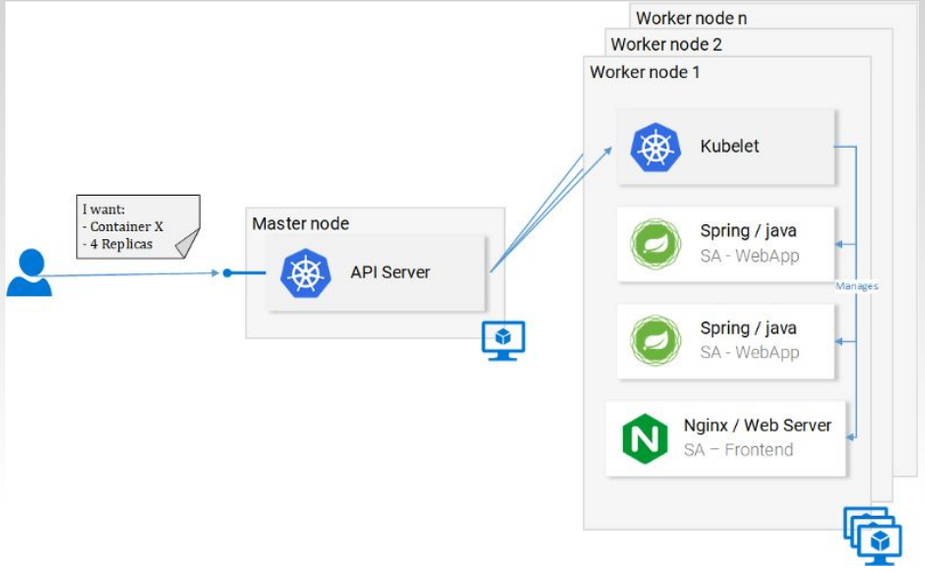


Abstracting the underlying infrastructure

- We don’t have to care about the number of nodes, where containers are started and how they communicate
- We don’t deal with hardware optimization or worry about nodes going down (and they will go down Murphy’s Law), because new nodes can be added to the Kubernetes cluster.
- In the meantime Kubernetes will spin up the containers in the other nodes that are still running

Abstracting the underlying infrastructure

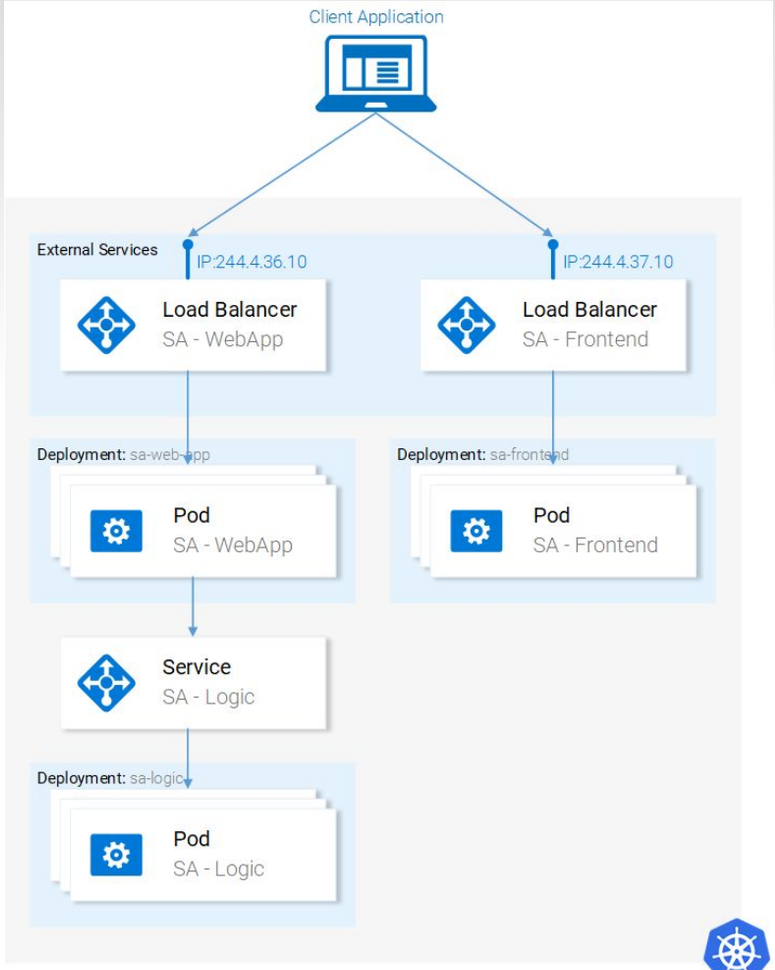
- **API Server**: Our only way to interact with the Cluster. Be it starting or stopping another container (err *pods) or checking current state, logs, etc.
- **Kubelet**: monitors the containers (err *pods) inside a node and communicates with the master node.
- ***Pods**: Initially just think of pods as containers.



Kubernetes help to standardize the Cloud Service Providers

Kubernetes in Practice — Pods

- As we know, manage containers is cumbersome process, and how about scalable or resilient too? Kubernetes will help this
- The Containers are orchestrated by Kubernetes



Minikube

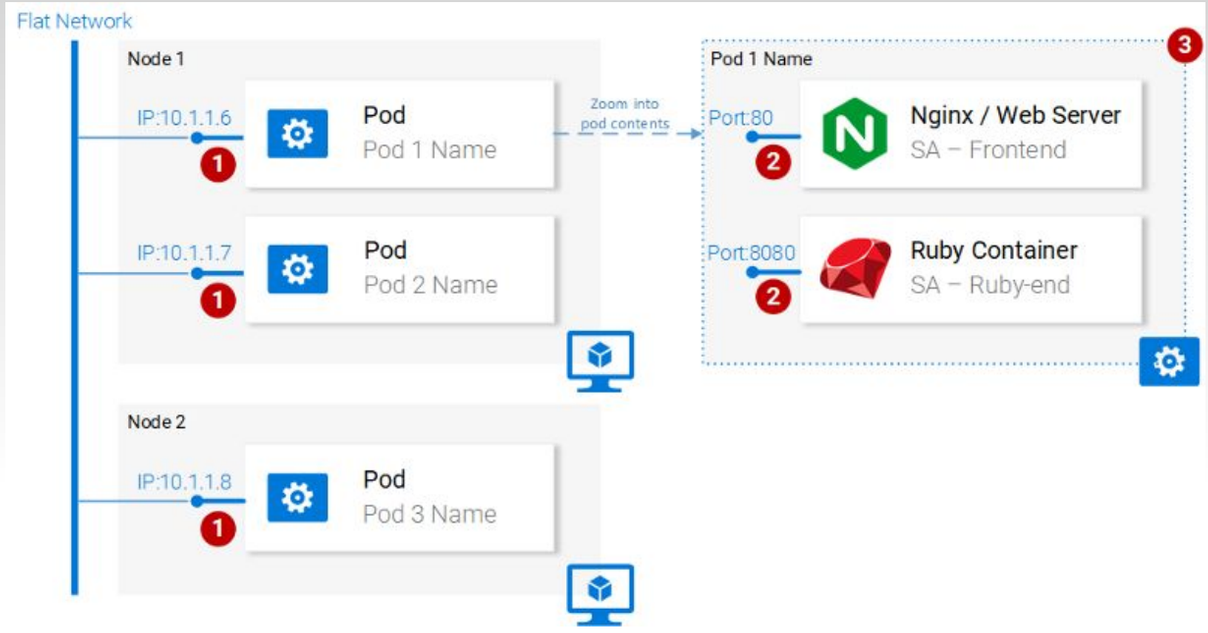
- Use Minikube for debugging locally. (Kubectl too)
- Start Minikube using **`minikube start`**
- Check the status with **`kubectl get nodes`**
- Minikube provides us with a Kubernetes Cluster that has only one node.
- We do not care how many Nodes there are, Kubernetes abstracts that away.

Pods

- **Pods** is the smallest deployable compute unit in Kubernetes
- **Pods** can be composed of one or even a group of containers that share the same execution environment.

Pods - Properties

- Each pod has a **unique IP address** in the Kubernetes cluster
- Pod can have multiple containers. The containers share the same port space, as such they can communicate via localhost and communicating with containers of the other pods has to be done in conjunction with the pod ip.
- Containers in a pod share the same volume*, same ip, port space, IPC namespace.



Pods definition

- Check resource-manifests/sa-frontend-pod.yaml

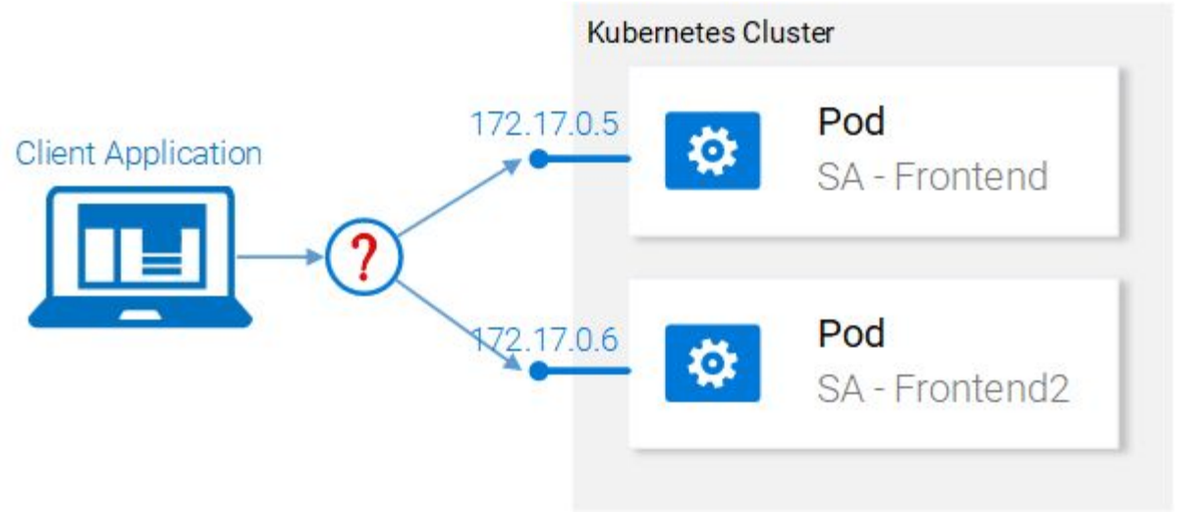
```
apiVersion: v1
kind: Pod                                                    # 1
metadata:
  name: sa-frontend
  labels:
    app: sa-frontend                                         # 2
spec:                                                        # 3
  containers:
    - image: pakkunandy/sentiment-analysis-frontend # 4
      name: sa-frontend                                      # 5
      ports:
        - containerPort: 80                                  # 6
```

Creating the SA Frontend pod

- Navigate to resource-manifests/sa-frontend-pod.yaml.
- Create the pod:
  - `$ kubectl create -f sa-frontend-pod.yaml`
- Check the status:
  - `$ kubectl get pods`
- Accessing the application externally (quick way):
  - `kubectl port-forward sa-frontend 8800:80`
- Open browser in 127.0.0.1:8800.

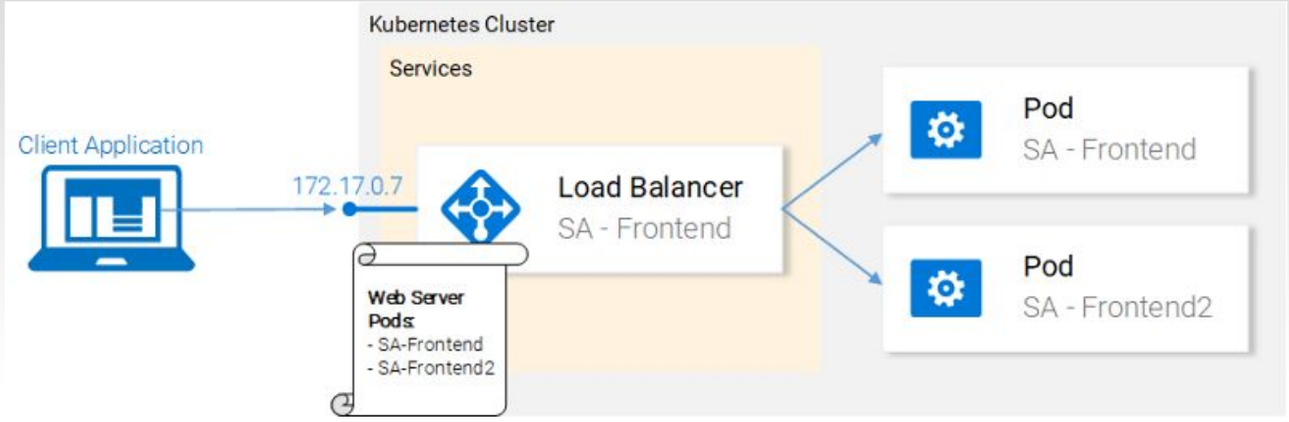
The wrong way to scale up

- Navigate to resource-manifests/sa-frontend-pod2.yaml.
- Create the pod:
  - `$ kubectl create -f sa-frontend-pod2.yaml`
- Check the status:
  - `$ kubectl get pods`
- We have two pods running, and it has many flaws. We'll improve it later on!



Kubernetes - Services

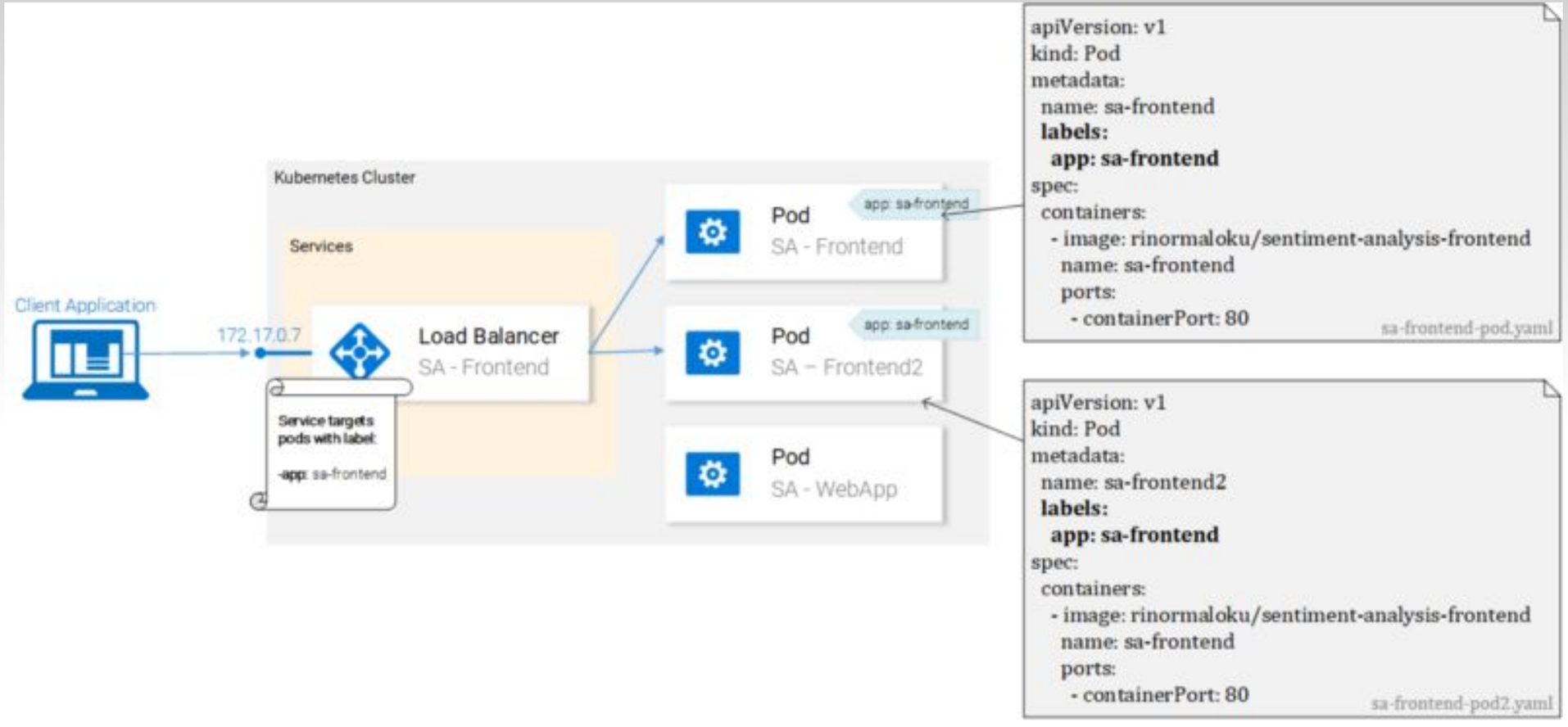
- The Kubernetes **Service** resource acts as the **entry point** to a set of pods that provide the same functional service.



- How does a **service** know which pods to target? I.e. how does it generate the list of the endpoints for the pods?

- This is done using Labels, and it is a two-step process:
  1. Applying a label to all the pods that we want our Service to target and
  2. Applying a “selector” to our service so that defines which labeled pods to target.

Kubernetes - Services



Labels

- Labels provide a simple method for organizing your Kubernetes Resources.
- They represent a key-value pair and can be applied to every resource.
- Verify that the pods were labeled by filtering the pods that we want to display:
  - `$ kubectl get pod -l app=sa-frontend --show-labels`

Service definition

- The YAML definition of the Loadbalancer Service (***service-sa-frontend-lb.yaml***) is shown below

```yaml
apiVersion: v1
kind: Service                 # 1
metadata:
  name: sa-frontend-lb
spec:
  type: LoadBalancer          # 2
  ports:
  - port: 80                  # 3
    protocol: TCP             # 4
    targetPort: 80            # 5
  selector:                   # 6

      app: sa-frontend        # 7
```

- Create the service execute:
  - **`$ kubectl create -f service-sa-frontend-lb.yaml`**
- Check the status:
  - **`$ kubectl get svc`**

Service definition

- You will see the External-IP is PENDING
- This is only because we are using Minikube. If we would have executed this in a cloud provider like Azure or GCP, we would get a Public IP, which makes our services worldwide accessible.
- For local debugging:
  - `$ minikube service sa-frontend-lb`

Deployment

- Kubernetes Deployments help us with one constant in the life of every application, and that is **change**.
- The Deployment resource automates the process of moving from one version of the application to the next, with **zero downtime** and in case of failures, it enables us to quickly **roll back** to the previous version.

Deployment

- So far, we have two pods and a service.
- The previous deploying the pods separately is far from perfect.
- It requires separately managing each (create, update, delete and monitoring their health)
- What we want:
  1. Two pods of the image pakkunandy/sentiment-analysis-frontend
  2. Zero Downtime deployments,
  3. Pods labeled with `app: sa-frontend` so that the services get discovered by the Service **sa-frontend-lb.**

Deployment definition

```yaml
apiVersion: apps/v1
kind: Deployment                                    # 1
metadata:
  name: sa-frontend
spec:
  selector:                                         # 2
    matchLabels:
      app: sa-frontend
  replicas: 2                                       # 3
  minReadySeconds: 15
  strategy:
    type: RollingUpdate                             # 4
    rollingUpdate:
      maxUnavailable: 1                             # 5
      maxSurge: 1                                   # 6
  template:                                         # 7
    metadata:
      labels:
        app: sa-frontend                            # 8
    spec:
      containers:
        - image: pakkunandy/sentiment-analysis-frontend
          imagePullPolicy: Always                   # 9
          name: sa-frontend
          ports:

            - containerPort: 80
```

Deployment definition

- Start the deployment:
  - **`$ kubectl apply -f sa-frontend-deployment.yaml`**
- Check the status:
  - **`$ kubectl get pods`**

```
NAME                              READY     STATUS    RESTARTS   AGE
sa-frontend                       1/1       Running   0          2d
sa-frontend-5d5987746c-ml6m4      1/1       Running   0          1m
sa-frontend-5d5987746c-mzsgg      1/1       Running   0          1m
sa-frontend2                      1/1       Running   0          2d
```

- We have 4 running pods, two pods created by the Deployment and the other two are the ones we created manually.
- Try delete the ones we created manually using the command ***kubectl delete pod <pod-name>***. Deleting one pod made the Deployment notice that the current state (1 pod running) is different from the desired state (2 pods running) so it started another pod.

Benefit #1: Rolling a Zero-Downtime deployment

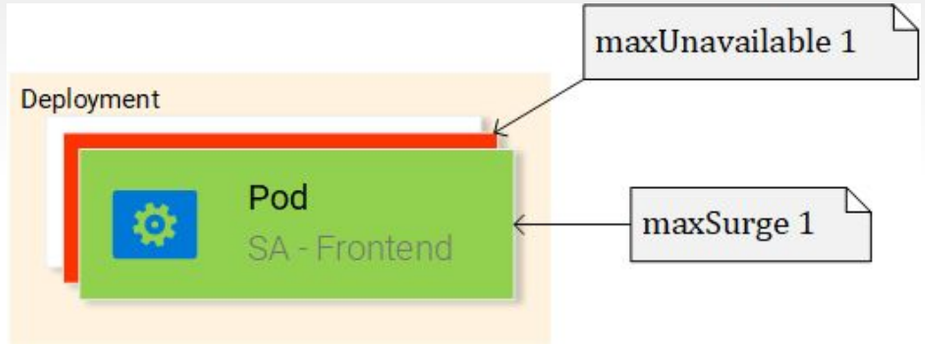
- Take an example, Our Product manager came to us with a new requirement, our clients want to have a green button in the frontend.
- The developers shipped their code and provided us with the only thing we need, the container image ***pakkunandy/sentiment-analysis-frontend:green***
- Now it's our turn, we the DevOps have to roll a Zero-Downtime deployment, Will the hard work pay off?
- So we change the value of container image in yaml file to a new image. Check the file ***sa-frontend-deployment-green.yaml*** then execute:
  - `$ kubectl apply -f sa-frontend-deployment-green.yaml --record`

Benefit #1: Rolling a Zero-Downtime deployment

- Check the status of the rollout using the following command:
  - **$ kubectl rollout status deployment sa-frontend**

```
Waiting for rollout to finish: 1 old replicas are pending termination...
Waiting for rollout to finish: 1 old replicas are pending termination...
Waiting for rollout to finish: 1 old replicas are pending termination...
Waiting for rollout to finish: 1 old replicas are pending termination...
Waiting for rollout to finish: 1 old replicas are pending termination...
Waiting for rollout to finish: 1 of 2 updated replicas are available...
deployment "sa-frontend" successfully rolled out
```

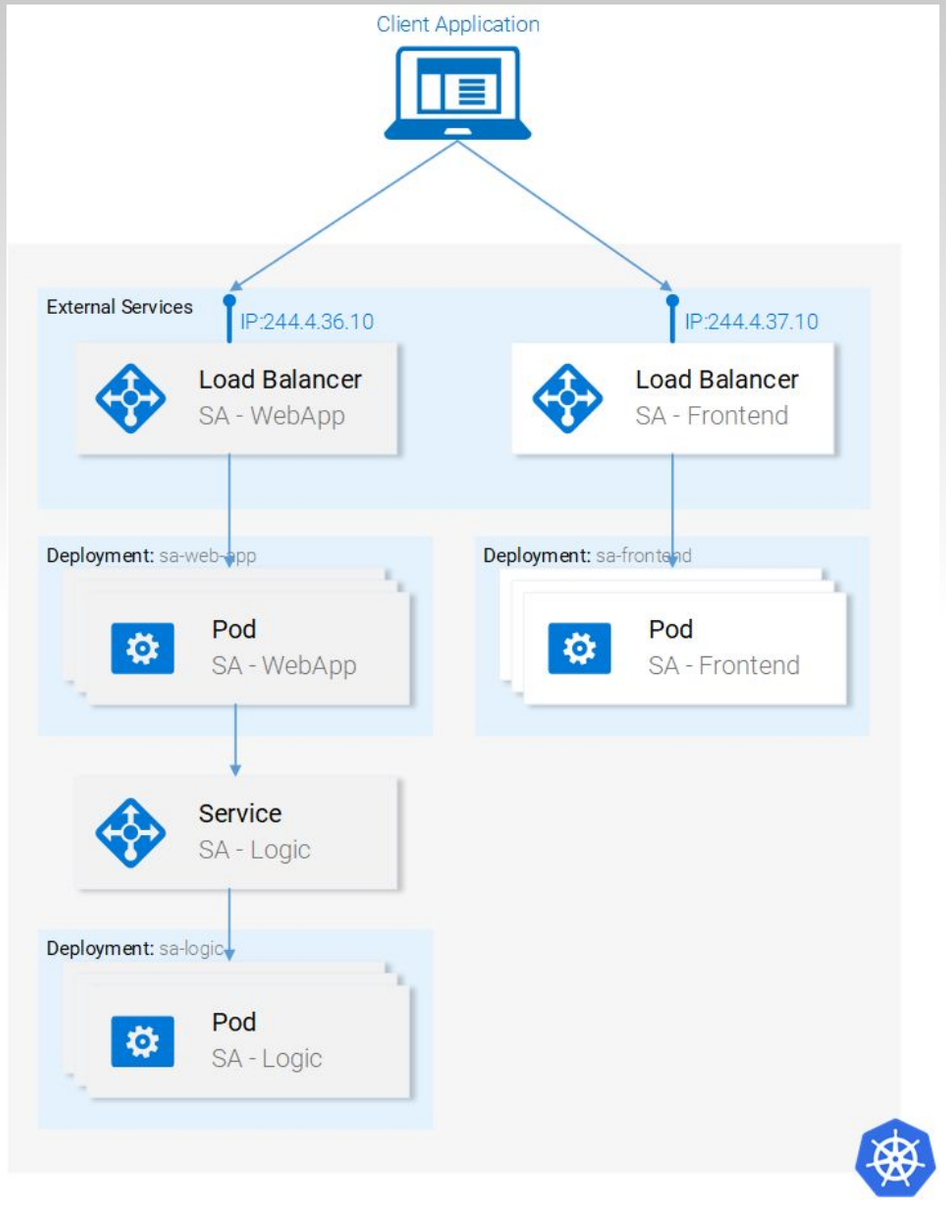
- Verify the deployment:
  - **$ minikube service sa-frontend-lb**
- Behind the scene:
  - Kubernetes compares the new state with the old one. In our case, the new state requests two pods with the image new image. **RollingUpdate** is kicked in.
  - The RollingUpdate acts according to the rules we specified, those being "**maxUnavailable**: 1" and "**maxSurge**: 1". This means that the deployment can terminate only one pod, and can start only one new pod.



Benefit #2: Rolling back to a previous state

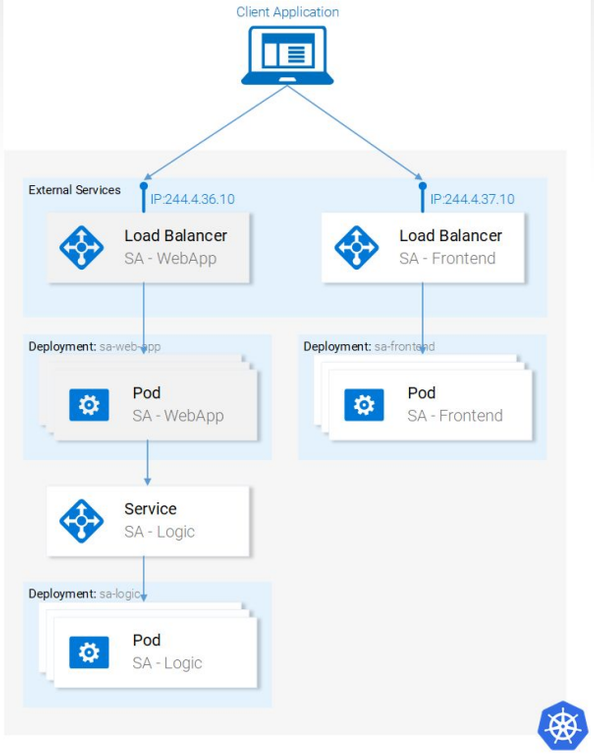
- Take another example, new our deployment have bug -_-, we need to rollback previous version.
- Well! Simply check the history of that deployment :)
  - `$ kubectl rollout history deployment sa-frontend`
- And, easily undo to the version we want:
  - `$ kubectl rollout undo deployment sa-frontend --to-revision=1`
  - `$ kubectl rollout status deployment sa-frontend`
- **For the next step, remember remove all the sa-frontend node which was created manually**

Completely deploy our system



Completely deploy our system

- Deployment SA-Logic (3 pods - Running the container of our python application):
  - **`$ kubectl apply -f sa-logic-deployment.yaml --record`**
- We need a Service for SA-Logic that “acts as the entry point to a set of pods that provide the same functional service”.
  - **`$ kubectl apply -f service-sa-logic.yaml`**



Completely deploy our system

- Deployment SA-WebApp (note: SA_LOGIC_API_URL is "http://sa-logic" which is KUBE-DNS) - (however, use the IP Address of Logic Server to be sure):
  - **`$ kubectl apply -f sa-web-app-deployment.yaml --record`**
- We need a Service SA-WebApp for Load Balancing.
  - **`$ kubectl apply -f service-sa-web-app-lb.yaml`**

Completely deploy our system

- When we deployed the SA-Frontend pods our container image was pointing to our SA-WebApp in http://localhost:8080/sentiment..
- But now we need to update it to point to the IP Address of the SA-WebApp Loadbalancer.
  - Get the SA-WebApp Loadbalancer IP
    - `$ minikube service list`
  - Use the SA-WebApp LoadBalancer IP in the file sa-frontend/src/App.js
  - Build the static files **npm run build**
  - Build the container image (remember push this image later)
    - `$ docker build -f Dockerfile -t $DOCKER_USER_ID/sentiment-analysis-frontend:minikube .`
  - Push the image to Docker hub.
    - `docker push $DOCKER_USER_ID/sentiment-analysis-frontend:minikube`
  - Edit the sa-frontend-deployment.yaml to use the new image and
  - Execute the command kubectl apply -f sa-frontend-deployment.yaml

Docker Swarm vs. Kubernetes

Features	Kubernetes	Docker Swarm
Installation & Cluster Configuration	Installation is complicated; but once setup, the cluster is very strong	Installation is very simple; but cluster is not very strong
GUI	GUI is the Kubernetes Dashboard	There is no GUI
Scalability	Highly scalable & scales fast	Highly scalable & scales 5x faster than Kubernetes
Auto-Scaling	Kubernetes can do auto-scaling	Docker Swarm cannot do auto-scaling
Load Balancing	Manual intervention needed for load balancing traffic between different containers in different Pods	Docker Swarm does auto load balancing of traffic between containers in the cluster
Rolling Updates & Rollbacks	Can deploy Rolling updates & does automatic Rollbacks	Can deploy Rolling updates, but not automatic Rollbacks
Data Volumes	Can share storage volumes only with other containers in same Pod	Can share storage volumes with any other container
Logging & Monitoring	In-built tools for logging & monitoring	3rd party tools like ELK should be used for logging & monitoring

https://medium.com/edureka/kubernetes-vs-docker-45231abeeaf1

References

- https://medium.com/faun/learn-docker-in-5-days-day-1-general-concepts-fcac5a4cf0a6
- https://rominirani.com/docker-swarm-tutorial-b67470cf8872
- https://www.freecodecamp.org/news/learn-kubernetes-in-under-3-hours-a-detailed-guide-to-orchestrating-containers-114ff420e882/
- https://medium.com/@dpaunin/the-best-architecture-with-docker-and-kubernetes-myth-or-reality-77b4f8f3804d