1

Outline

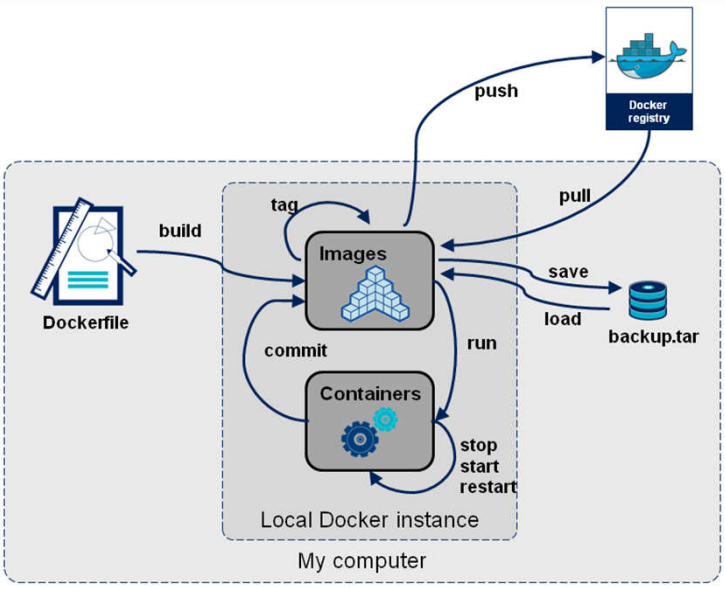
- Building and Shipping Docker Images
- Authoring manifests for deployment
- Exposing Applications with Services

2

Building and Shipping Docker Images

3

Docker containers & images



4

Dockerfile overview

- A Dockerfile is a build recipe for a Docker image.
- It contains a series of instructions telling Docker how an image is constructed.
- The docker build command builds an image from a Dockerfile.

5

docker build example

- This Dockerfile for creating a simple Java application:

```
FROM maven:3.5.2-jdk-9
COPY src /usr/src/app/src
COPY pom.xml /usr/src/app
RUN mvn -f /usr/src/app/pom.xml clean package

EXPOSE 8080
ENTRYPOINT ["java","-jar","/usr/src/app/target/myapp-1.0.0-SNAPSHOT.jar"]
```

- The image can be built with the following command:

```
$ docker build -t my-lighttpd .
```

- Notes:
  - The build has the current directory as context
  - Each command in the Dockerfile creates a new (temporary container)
  - Every creation step generates a layer that is is cached, so repeated builds are fast

6

Deploying and exposing applications

Dockerfile instructions

Instruction	Description
FROM	Parent image
ARG	Parameters for contructing the image
ENV	Specify Environnement variables
LABEL	Specify Label meta-data
VOLUME	Mount volumes
RUN	Run a command
COPY	Copy files to the image
ADD	Add files to the image
WORKDIR	Specify the working directory
EXPOSE	Expose ports to be accessed
USER	User name or UID to be used
ONBUILD	Instructions to execute when constructing child images
CMD	Command to execute when starting a container
ENTRYPOINT	The default entry point of the container

7

Deploying and exposing applications

Dockerfile best practices

- **Use official base images**
- **Prefer COPY over ADD**
- **Group RUN instructions in one line**
- **ADD a .dockerignore file**
- **Use Multi-stage builds**

8

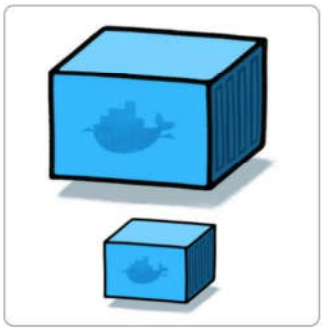
Deploying and exposing applications

What are multi stage builds ?

- **Multi-stage builds are a method of organizing a Dockerfile to minimize the size of the final container.**
  - This is made possible by the image building process into multiple stages
  - Each stage is a separate image, and can copy files from previous stages.

```
# Dockerfile
# build stage
FROM buildbase as build
...
...
...

# production ready stage
FROM runbase
...
COPY --from=build
/artifact /app
```



M.Romdhani, 2021 9

9

Deploying and exposing applications

Multi-stage builds in practice

- **Building a Java Spring Boot Application in a single image**

```
FROM maven:3.5.2-jdk-9
COPY src /usr/src/app/src
COPY pom.xml /usr/src/app
RUN mvn -f /usr/src/app/pom.xml clean package

EXPOSE 8080
ENTRYPOINT ["java","-jar","/usr/src/app/target/myapp-1.0.0-SNAPSHOT.jar"]
```

- **Building the Java Spring Boot Application using a multi-stage build**

```
FROM maven:3.5.2-jdk-9 AS build
COPY src /usr/src/app/src
COPY pom.xml /usr/src/app
RUN mvn -f /usr/src/app/pom.xml clean package

FROM openjdk:9-jre-alpine
COPY --from=build /usr/src/app/target/ myapp-1.0.0-SNAPSHOT.jar
/usr/app/myapp-1.0.0-SNAPSHOT.jar
EXPOSE 8080
ENTRYPOINT ["java","-jar","/usr/app/myapp-1.0.0-SNAPSHOT.jar"]
```

M.Romdhani, 2021 10

10

Docker-Compose for development stacks

- **Dockerfiles are great to build container images.**
  - But what if we work with a complex stack made of multiple containers?
  - Eventually, we will want to write some **custom scripts and automation to build, run, and connect our containers together**.
  - There is a better way: **using Docker Compose**
- **Compose is a tool for defining and running multi-container Docker applications**
  - Docker compose helps by defining and coordinating multiple containers.
- **The general idea of Compose is to enable a very simple, powerful onboarding workflow:**
  - Checkout your code.
  - Run docker-compose up.
  - Your app is up and running!

11

An example of Stack with Compose

- **This is an example of Docker Compose file**
- **To start the app:**
  - `> docker-compose up`
- **To stop the app:**
  - `> docker-compose stop`
- **Stop the app and remove containers, images,networks,**
  - `> docker-compose down`

```
version: '3'
services:
  db:
     image: mysql
     container_name: mysql_db
     restart: always
     environment:
        - MYSQL_ROOT_PASSWORD="secret"
        - MYSQL_USER_PASSWORD="secret"
  web:
    context: ./webapp
    dockerfile: Dockerfile

    depends_on:
       - db
    container_name: apache_web
    restart: always
    ports:
      - "8080:80"
networks:
   - mynetwork
volumes:
   - myNamedVolume
```


12

Useful Docker Commands

- **docker ps** list running containers.
- **docker ps -a** list all container including stopped container
- **docker pull** download a image from Docker Hub registry.
- **docker build .** to build a container based on the Dockerfile in the current directory (the dot). docker build -t "myimage:latest" . creates a container and stores the image under the given name
- **docker images** or docker image ls shows all local storage images
- **docker run** to run a container using the image given in parameter
- **docker logs** display the logs of a container, you specified. To continue showing log updates just use docker logs -f mycontainer
- **docker volume ls** lists the volumes, which are commonly used for persisting data of Docker containers.
- **docker network ls** - list all networks available for docker container
- **docker network** connect adds the container to the given container network. That enables container communication by simple container name instead of IP.
- **docker rm** removes one or more containers. docker rm mycontainer, but make sure the container is not running
- **docker rmi** removes one or more images. docker rmi myimage, but make sure no running container is based on that image
- **docker stop** stops one or more containers.

13

Shipping an Image to a registry

- **An account is required to push images to Dockerhub**
- **Ship an image to DockerHub.com**

```
> docker push myrepo/myimage:1.0
```

- **DockerHub has a limit of the number of Pulls, since 11/2020**
  - 100 pulls in 6 Hours /Anonymous
  - 200 pulls in 6 Hours/Logged in user
- **Alternatives to DockerHub**
  - RedHat Quay.io
  - Amazon Elastic Container Registry (ECR)
  - JFrog Artifactory.
  - Azure Container Registry.
  - Google Container Registry.
  - VMWare Harbor

14

How to create a local Docker Registry

- **Run the registry as a Container**

```
> docker run -p 5000:5000 --restart=always--name myregistry  -v
                          $(pwd):/var/lib/registry registry:2
                [https://docs.docker.com/registry/deploying/]
```

- **Ship an image to a local docker registry**

```
> docker image tag my-image localhost:5000/my-image
> docker push localhost:5000/my-image
```


15

Authoring manifests for deployment

16

Yaml manfiest structure

- **Required fields**
  - **apiVersion** - Which version of the Kubernetes API you're using to create this object
  - **kind** - What kind of object you want to create
  - **metadata** - Data that helps uniquely identify the object, including a name string, UID, and optional namespace
  - **spec** - What state you desire for the object. The precise format of the object spec is different for every Kubernetes object
- **The status field**
  - While spec describes the desired state, the status describes the current state. It is added and updated continuously by K8s control plane.

```
kubectl get deploy mydepl -o yaml
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```


17

kubectl apply vs create

- **`kubectl create -f whatever.yaml`**
  - creates resources if they don't exist
  - if resources already exist, don't alter them (and display error message)
- **`kubectl apply -f whatever.yaml`**
  - creates resources if they don't exist
  - if resources already exist, update them (to match the definition provided by the YAML file)
  - stores the manifest as an annotation in the resource

18

Simple Pod Deployment

- **Deployement steps**
  1. Describe the app using Kubernets YAML (**my-ngnix-pod.yaml**)
  2. Run the deployment Command **`kubectl apply –f my-ngnix-pod.yaml`**
  3. Make sure the pod has been created **`kubectl get pods`**
  4. Tear down your app **`kubectl apply –f my-ngnix-pod.yaml`**

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mynginxapp
  labels:
    name: mynginxapp
spec:
  containers:
    - name: mynginxapp
      image: nginx
      ports:
        - containerPort: 80
```


19

Simple Pod with namespace and labels

- **Additional information**
  - **Namespace**: Namespaces provide a scope for Kubernetes resources, splitting the cluster in smaller units.
  - **Labels**: Labels are intended to be used to specify identifying attributes of objects that are meaningful and relevant to users, but do not directly imply semantics to the core system.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mynginxapp
  namespace: default
  labels:
    name: mynginxapp
    profile: dev
spec:
  containers:
  - name: mynginxapp
    image: nginx
    ports:
      - containerPort: 80
```


20

A Multi container Pod: Main Container with Side Car Container

- Main Container and the Side Car Container share a Volume

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-sidecar
spec:
  # Create a volume called 'shared-logs' that the pp and sidecar share.
  volumes:
  - name: shared-logs
    emptyDir: {}
  containers:
  - name: app-container # Main application container
    # Simple application: write the current date  to the log file every 5 seconds
    image: alpine
    command: ["/bin/sh"]
    args: ["-c", "while true; do date >> /var/log/app.txt; sleep 5;done"]
    volumeMounts: # Mount the pod's shared log file into the app container
    - name: shared-logs
      mountPath: /var/log

  - name: sidecar-container # Sidecar container
    image: nginx:1.7.9
    ports:
      - containerPort: 80
    volumeMounts: # Mount the pod's shared log file into the sidecar
    - name: shared-logs
      mountPath: /usr/share/nginx/html # nginx-specific mount path
```


Using Deployments

- **Saving this manifest into `nginxdeploy.yaml` and submitting it to a Kubernetes cluster will create the defined Deployment, ReplicaSet and the Pods**
  - You can then get the current Deployments deployed:

    `kubectl get deployments`
  - You can then get the current ReplicaSets deployed:

    `kubectl get rs`
  - You can then get the current pods deployed:

    `kubectl get pods`

```yaml
# for versions before 1.9.0 use apps/v1beta2
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # tells deployment to run 2 pods
  template:                    Pod Template
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

Updating the deployment

- You can update the deployment by applying a new YAML file. This YAML file specifies that the deployment should be updated to use nginx 1.16.1

```yaml
# for versions before 1.9.0 use apps/v1beta2
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # tells deployment to run 2 pods
  template:                    Pod Template
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.16.1
        ports:
        - containerPort: 80
```


23

Scaling the application by increasing the replica count

- You can increase the number of pods in your Deployment by applying a new YAML file. This YAML file sets replicas to 4, which specifies that the Deployment should have four pods:

```yaml
# for versions before 1.9.0 use apps/v1beta2
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 4 # Update the replicas from 2 to 4
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.16.1
        ports:
        - containerPort: 80
```


24

Exposing Applications with Services

25

Services

- **Services give us a stable endpoint to connect to a pod or a group of pods**
  - Durable resource (unlike Pods)
    - static cluster-unique IP
  - Target Pods using equality based selectors
  - kube-proxy provides simple load-balancing.
- **A Kubernetes Service can select the pods it is supposed to abstract through a label selector**
- **We can create a service either using the command `kubectl expose` or using a Yaml manifest**
    - Services are automatically added to an internal DNS zone
- **A service has a number of "endpoints"**

26

Deploying and exposing applications

Service Types

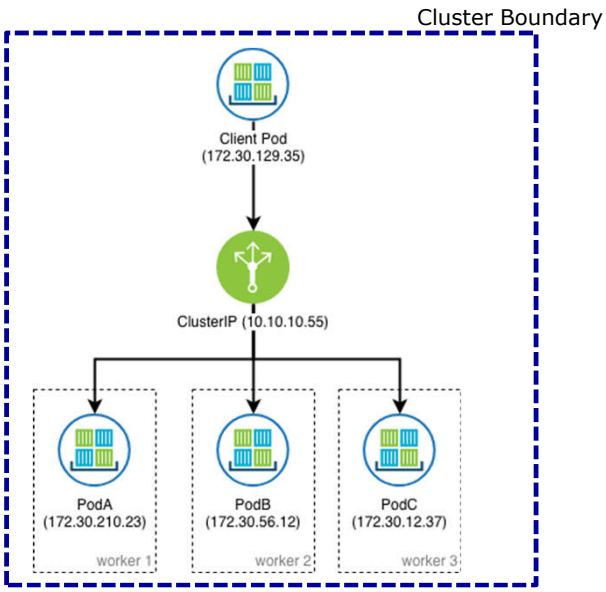
- There are 3 major service types:
  1. ClusterIP (default)
  2. NodePort
  3. LoadBalancer
- There is also another resource type called Ingress (specifically for HTTP services)

M.Romdhani, 2021 27

27

Deploying and exposing applications

ClusterIP Services



M.Romdhani, 2021 28

28

ClusterIP Services

- It is the default service type
- A virtual IP address is allocated for the service
- This IP address is reachable only from within the cluster (nodes and pods)
- Perfect for internal communication, within the cluster

```yaml
apiVersion: v1
kind: Service
metadata:
  name: example-prod
spec:
  type: ClusterIP
  selector:
    app: nginx
    env: prod
  ports:
  - protocol: TCP
    port: 8080
    targetPort: 80 # should meet the
                Container port
```


29

NodePort Services

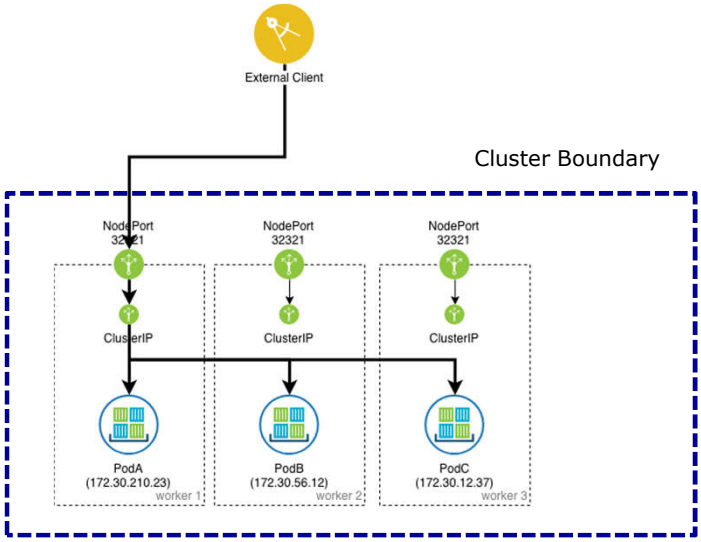
- NodePort services extend the ClusterIP service.
  - Exposes a port on every node's IP.
- Port can either be statically defined, or dynamically taken from a range between 30000-32767.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: example-prod
spec:
  type: NodePort
  selector:
    app: nginx
    env: prod
  ports:
  - nodePort:30008
    port: 80
    targetPort: 80
```


30

Deploying and exposing applications

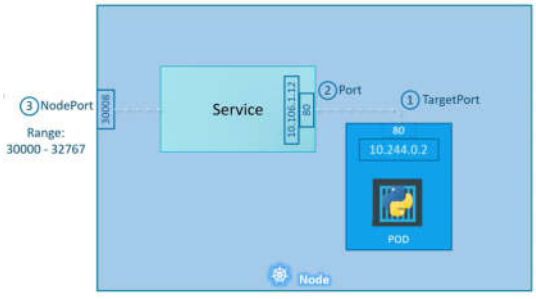
NodePort Services



31

Deploying and exposing applications

nodePort, port, targetPort




```
service-definition.yml
apiVersion: v1
kind: Service
metadata:
    name: myapp-service

spec:
    type: NodePort
    ports:
     - targetPort: 80
       port: 80
       nodePort: 30008
```


32

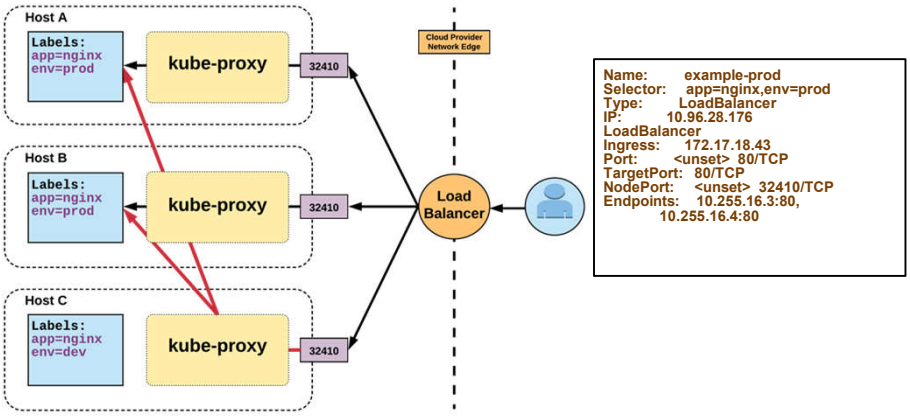
LoadBalancer Services

- LoadBalancer services extend NodePort.
- Works in conjunction with an external system to map a cluster external IP to the exposed service (typically a cloud load balancer, e.g. ELB on AWS, GLB on GCE ...)

```
apiVersion: v1
kind: Service
metadata:
  name: example-prod
spec:
  type: LoadBalancer
  selector:
    app: nginx
    env: prod
  ports:
    protocol: TCP
    port: 80
    targetPort: 80
```


33

LoadBalancer Services



- Gets external IP from provider

34