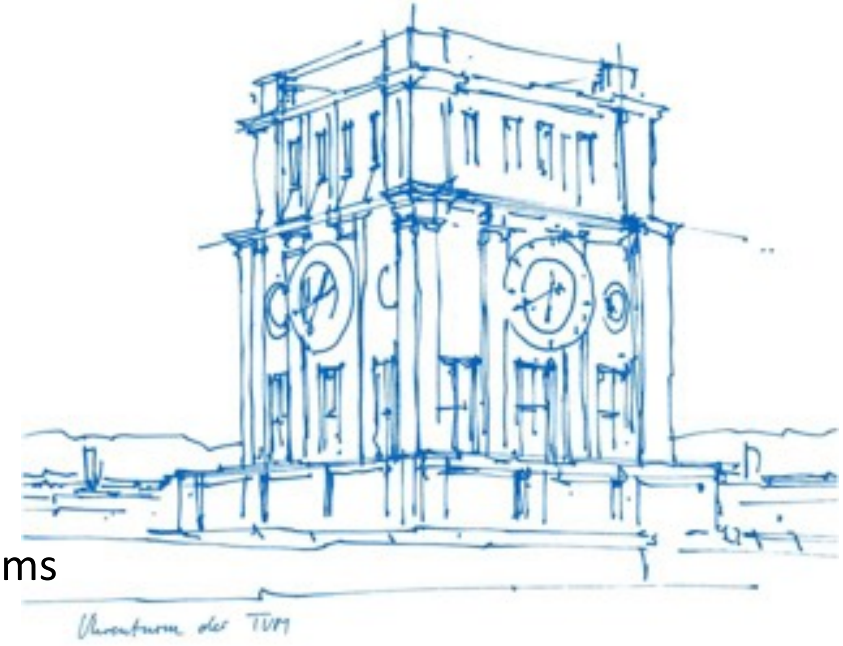
Cloud Computing Exercise – 2

Application Deployment Using Docker



Anshul Jindal (M.Sc. Informatics)

anshul.jindal@tum.de

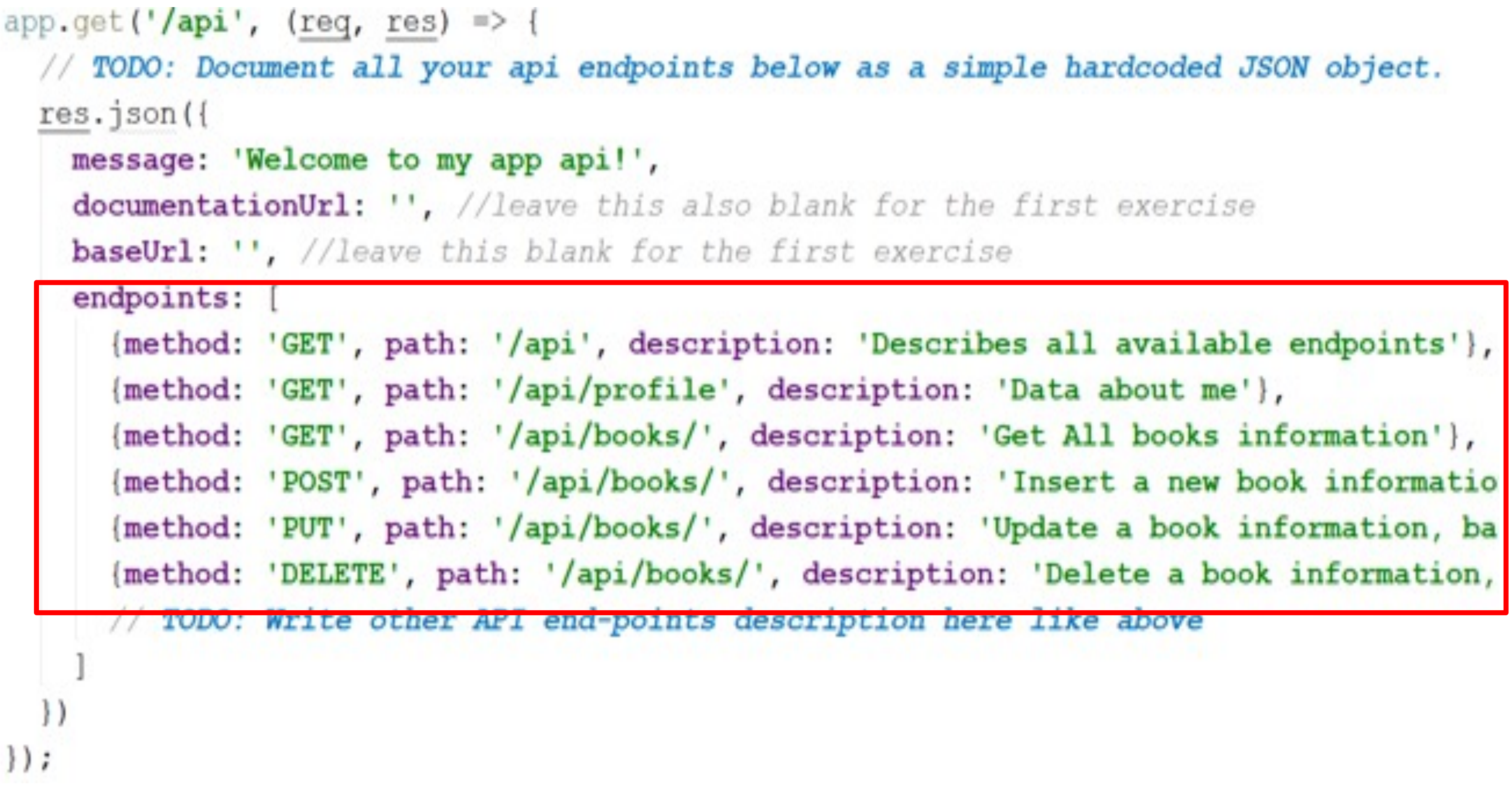
Chair of Computer Architecture and Parallel Systems

Technical University of Munich (TUM), Germany

Exercise 1 Solution

Tasks to be completed

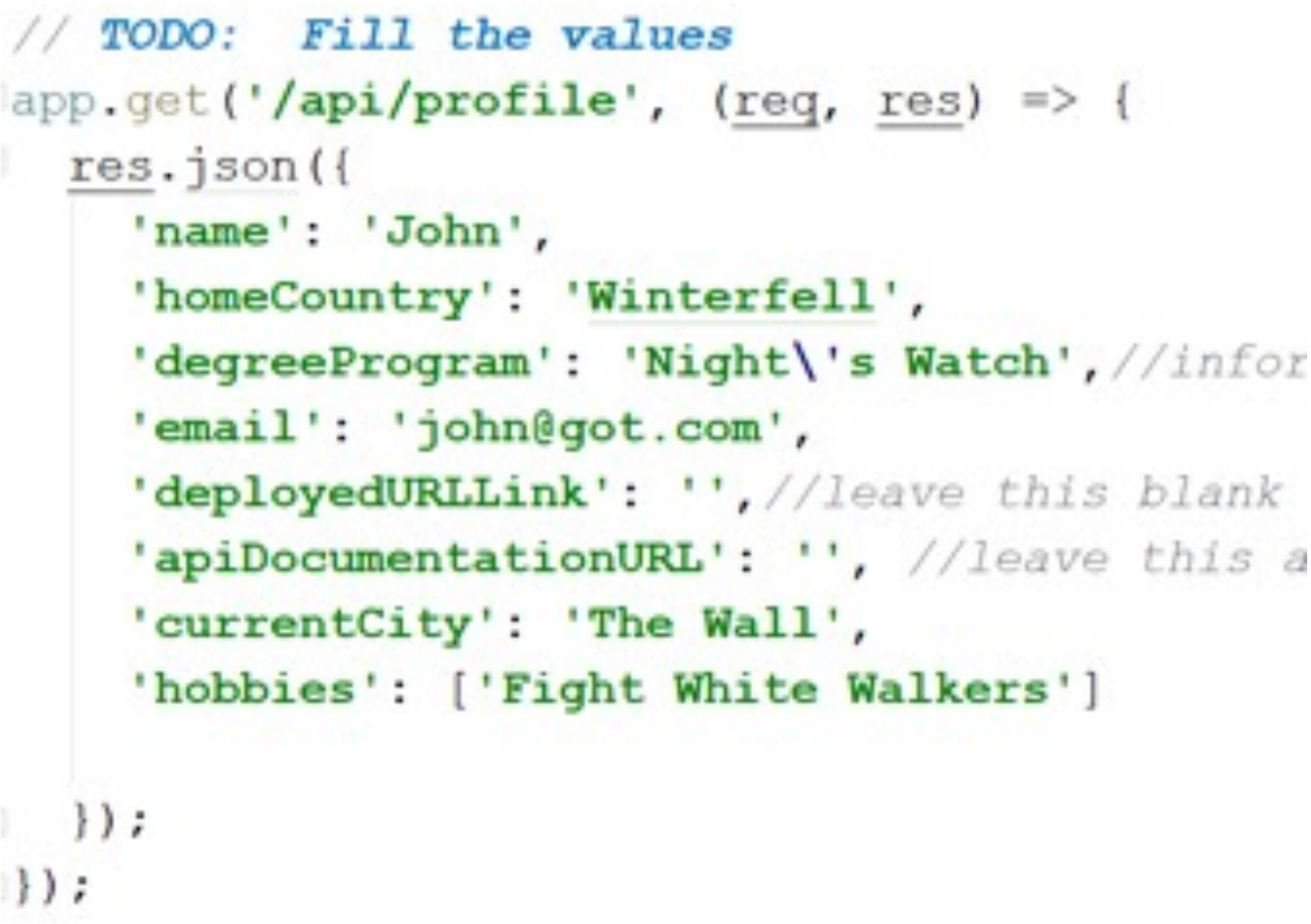
1. Document all your api endpoints in a simple hardcoded JSON object in the "/api" endpoint.



```
app.get('/api', (req, res) => {
  // TODO: Document all your api endpoints below as a simple hardcoded JSON object.
  res.json({
    message: 'Welcome to my app api!',
    documentationUrl: '', //leave this also blank for the first exercise
    baseUrl: '', //leave this blank for the first exercise
    endpoints: [
      {method: 'GET', path: '/api', description: 'Describes all available endpoints'},
      {method: 'GET', path: '/api/profile', description: 'Data about me'},
      {method: 'GET', path: '/api/books/', description: 'Get All books information'},
      {method: 'POST', path: '/api/books/', description: 'Insert a new book informatio
      {method: 'PUT', path: '/api/books/', description: 'Update a book information, ba
      {method: 'DELETE', path: '/api/books/', description: 'Delete a book information,
      // TODO: Write other API end-points description here like above
    ]
  })
});
```

Tasks to be completed Continue..

2. Complete the /api/profile endpoint. You can add here fake information too, to make it more interesting like Name as Jon Snow, homeCountry as winterfell ☺



```
// TODO:  Fill the values
app.get('/api/profile', (req, res) => {
  res.json({
    'name': 'John',
    'homeCountry': 'Winterfell',
    'degreeProgram': 'Night\'s Watch',//infor
    'email': 'john@got.com',
    'deployedURLLink': '',//leave this blank
    'apiDocumentationURL': '', //leave this a
    'currentCity': 'The Wall',
    'hobbies': ['Fight White Walkers']

  });
});
```

GET API Explanation

This is the first function which is called when you call **/api/books**

```
app.get('/api/books/', (req, res) => {

  db.books.find({}, function (err, books) {
    if (err) throw err;

    res.json(books);

  });
});
```

This is the callback function, when you get all the objects from mongodb.
See it has two arguments, one is the “**err”** and other the found array of objects “**books”**.
So now, you can return that found array inside that Callback function back to user.

Here it is returned.

Tasks to be completed Continue..

- /api/books [POST] : To store new book information and return the stored information as JSON.

First triggered function when you call
Post: /api/books

```
app.post('/api/books/', (req, res) => {
  /*
   * New Book information in req.body
   */
  console.log(req.body);

  db.books.create(req.body, (err, newBook) => {

    if (err) throw err;

    res.json(newBook);
  });
});
```

Insert information into mongodb, once it is inserted a callback function is called.

Call back function with two arguments **err** and **newBook** object

Return the **newBook** object back to user

Tasks to be completed Continue..

- /api/books/:id [PUT] : To Update a book information based upon the provided id and new information.

First triggered function when you call **PUT: /api/books/:id**

```
app.put('/api/books/:id', (req, res) => {

  const bookId = req.params.id;
  const bookNewData = req.body;
  console.log(`book ID = ${bookId} \n
 Book Data = ${bookNewData}`);

  db.books.findOneAndUpdate({_id: bookId},
  bookNewData, {new: true}, (err, updatedBookInfo) => {
    if (err) throw err;
    /*
     * Send the updated book information as a JSON object
     */
    res.json(updatedBookInfo);
  });
});
```

Update information into mongodb, based upon the **id**. Once it is inserted a callback function is called.

Call back function with two arguments **err** and **updatedBookInfo** object

Return the **updatedBookInfo** object back to user.

Tasks to be completed Continue..

- /api/books/:id [DELETE]: To delete a book information based upon the id.

First triggered function when you call **Delete: /api/books/:id**

```
app.delete('/api/books/:id', (req, res) => {

   const bookId = req.params.id;

   db.books.findOneAndRemove({_id: bookId},
  (err, deletedBookInfo) => {

     if (err) throw err;
     /*
      * Send the deleted book information as a JSON object
      */
     res.json(deletedBookInfo);
   });
});
```

Delete information from mongodb, based upon the **id**. Once it is deleted a callback function is called.

Call back function with two arguments **err** and **deletedBookInfo** object

Return the **deletedBookInfo** object back to user.

Introduction to Docker and Docker Hub

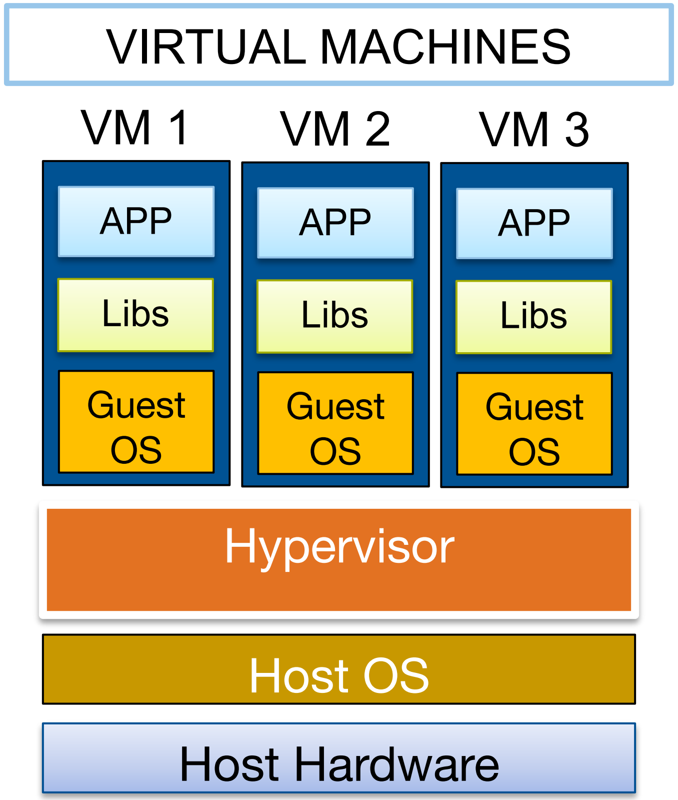
Problems with the deployment method of 1st Exercise

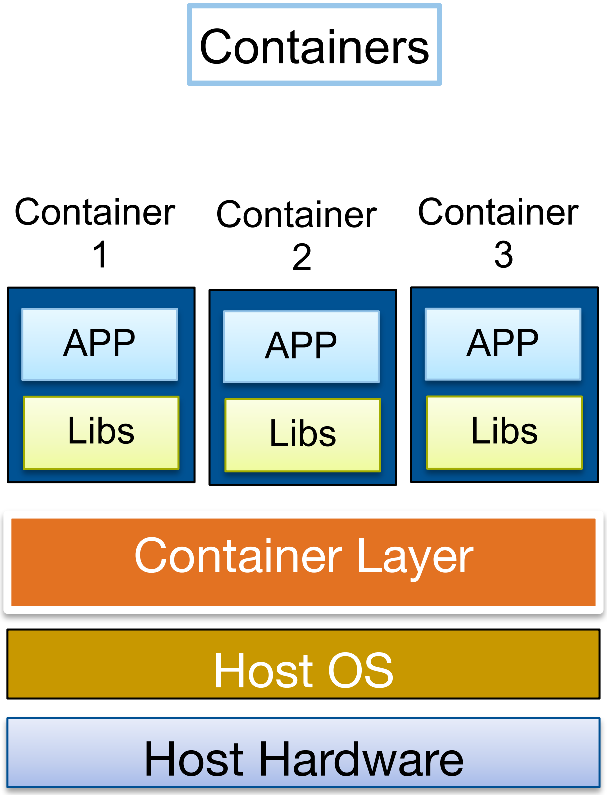
- **OS Dependent:** Different deployment procedure and requirements for different OS.
- **Not-Scalable:** Run on more Laptops or VMs ? Not an Ideal Solution
- **Not-Portable:** Running the same procedure from starting again on the new machine ? Time wastage.
- And many more….

Containerization (container-based virtualization)

What?

- is an OS-level virtualization method for deploying and running distributed applications without launching an entire VM for each application.
- share the same OS kernel as the host.



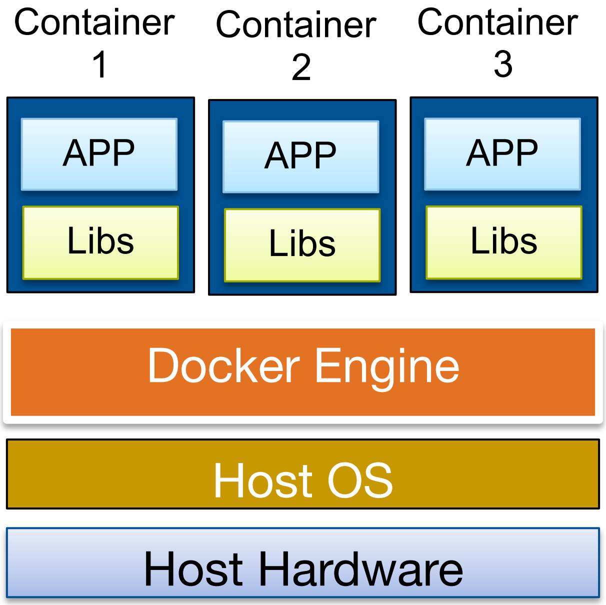


Docker

Docker provides a unified access to

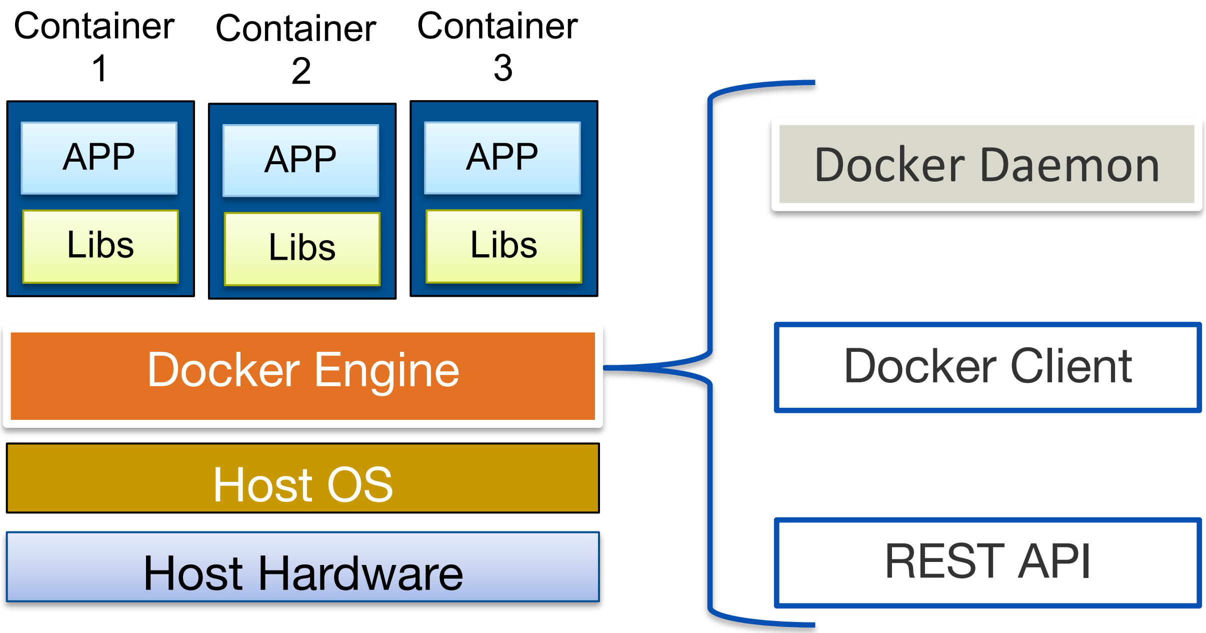
- Linux container technology (cgroups, namespaces)
- Various container implementations (lxc, libvirt, libcontainer, etc.)

'libcontainer' is Docker's implementation of container technology



Docker Continued…

The Docker *client* talks to the Docker *daemon*, which does the heavy lifting of building, running, and distributing your Docker containers.

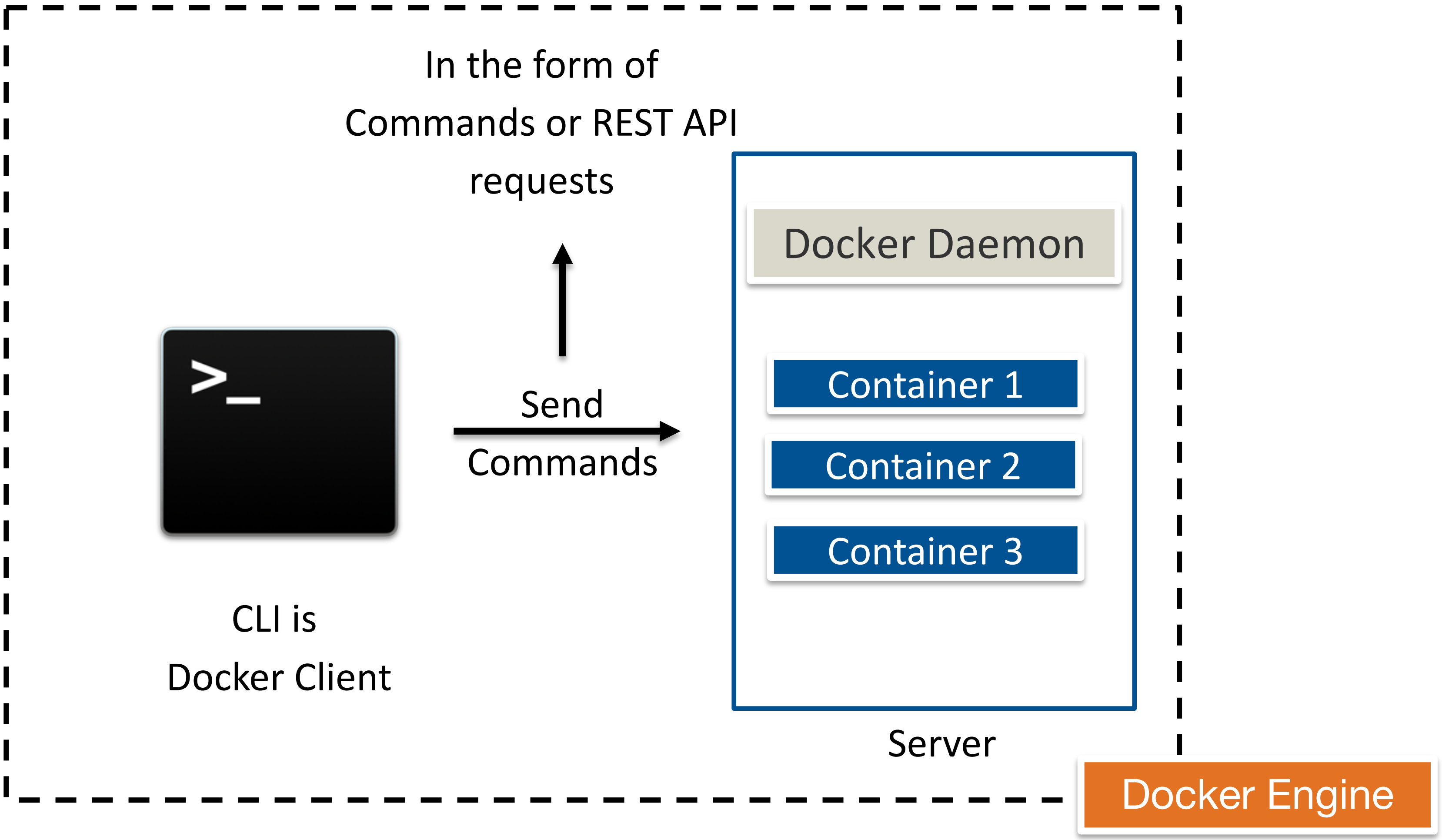


executes commands sent by the Docker Client

communicates with the Docker Daemon to execute commands

for interacting with the Docker Daemon remotely

Docker client-server Architecture



Docker client and daemon can be present on the same or different host machines

Advantages of containers

- packs your application in a container with all of your application's bins\libs and dependencies.
- makes it fully isolated from external environments regardless of where it is running.
- we can ship that container to any where i.e. to any other OS, to Docker Registry (such as Docker Hub) or to the cloud.

OS Independent

Scalable

Portable

Dockerfile

```
FROM node:alpine
```

Use a Docker base Image

Image based on **Alpine Linux**. Node is the **repository name (<your-username>/my-first-repo) in dockerhub** and **alpine** is the version

```
RUN mkdir -p /usr/src/server
```

Create Application Directory

```
WORKDIR /usr/src/server
```

Set the working directory of the container for all the RUN commands

```
COPY package.json /usr/src/server/
```

Copy the **package.json** file which contain all the dependencies required for application

```
RUN npm install
```

This command will install all the dependencies listed in **package.json**

```
COPY . /usr/src/server
```

Copy all other files from local machine to container

```
EXPOSE 3000
```

Expose container port to the host machine

```
CMD [ "node", “server.js"]
```

A start command to run the application

Images and layers

- A Docker image is built up from a series of layers.
- Each layer represents an instruction in the image’s Dockerfile.
- A new writable layer on top of the underlying layers often called as the “container layer” is added when a new container is created.

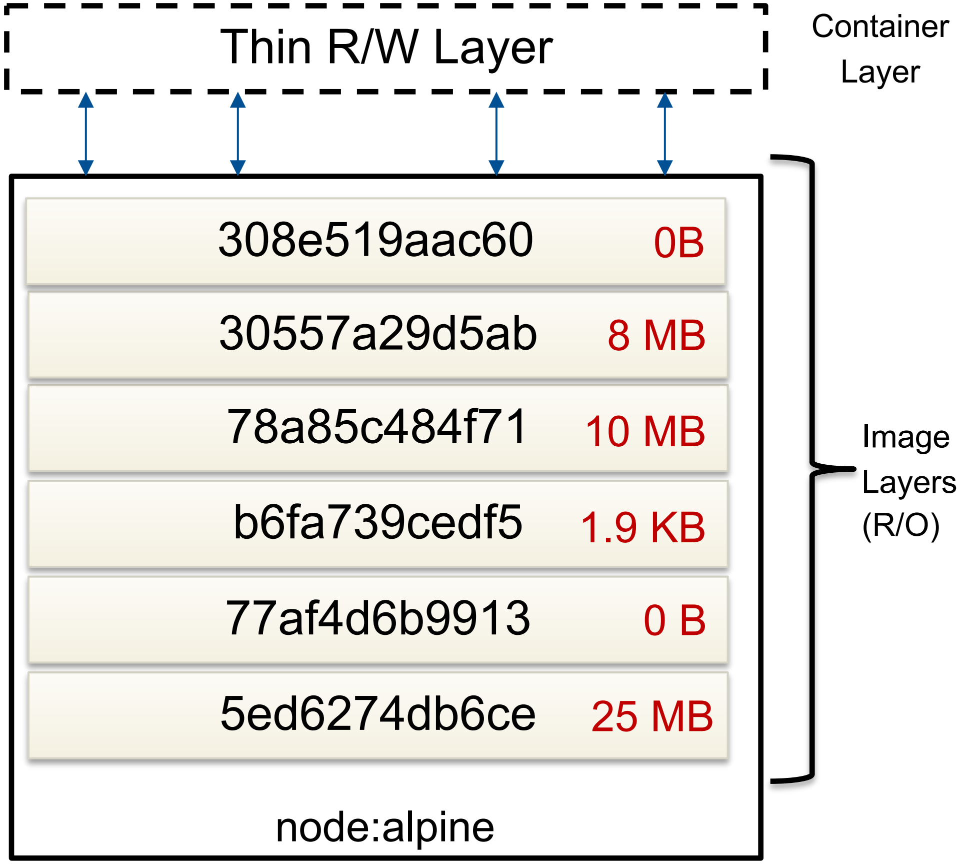
```
CMD [ "node", “server.js"]

COPY . /usr/src/server

RUN npm install
COPY package.json /usr/src/server/

RUN mkdir -p /usr/src/server

FROM node:alpine
```



Thin R/W Layer

Container Layer

308e519aac60	0B
30557a29d5ab	8 MB
78a85c484f71	10 MB
b6fa739cedf5	1.9 KB
77af4d6b9913	0 B
5ed6274db6ce	25 MB

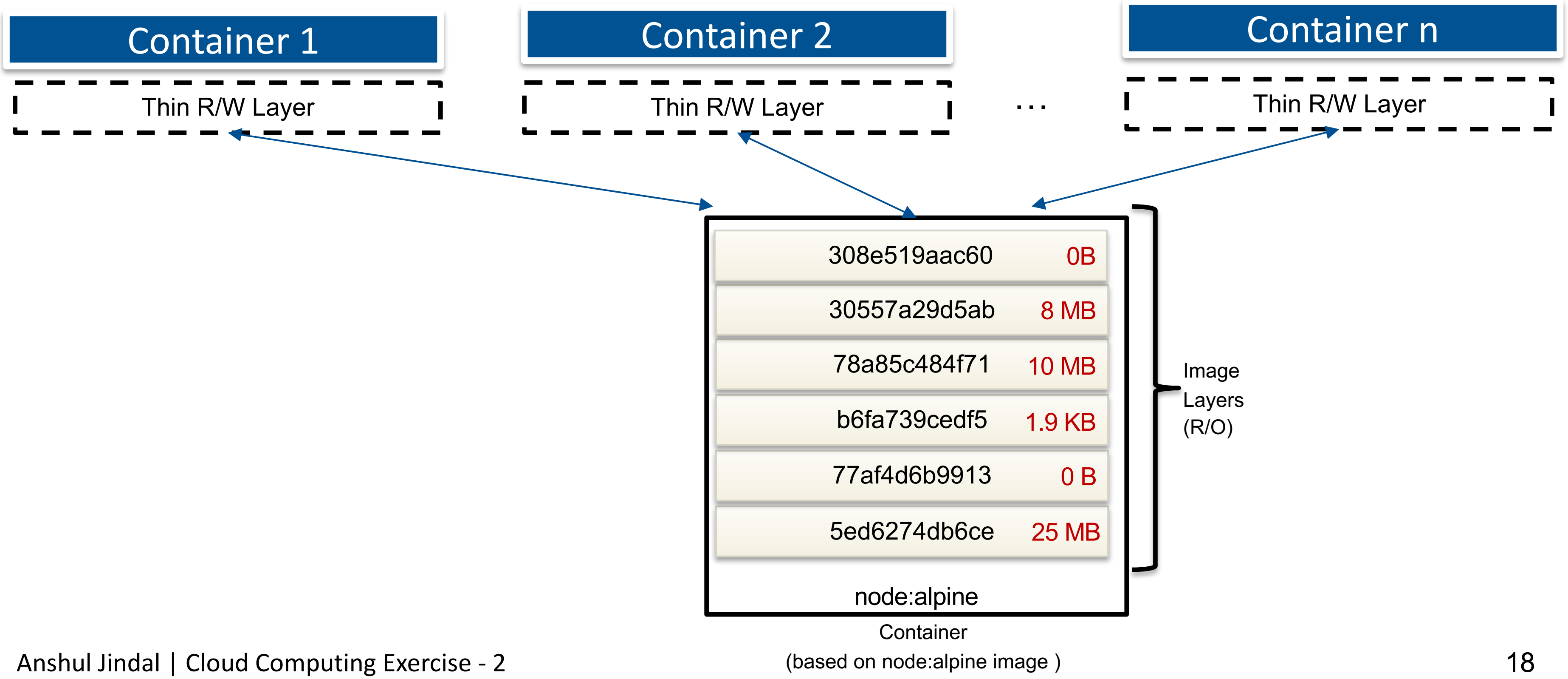
node:alpine

Image Layers (R/O)

Container
(based on node:alpine image)

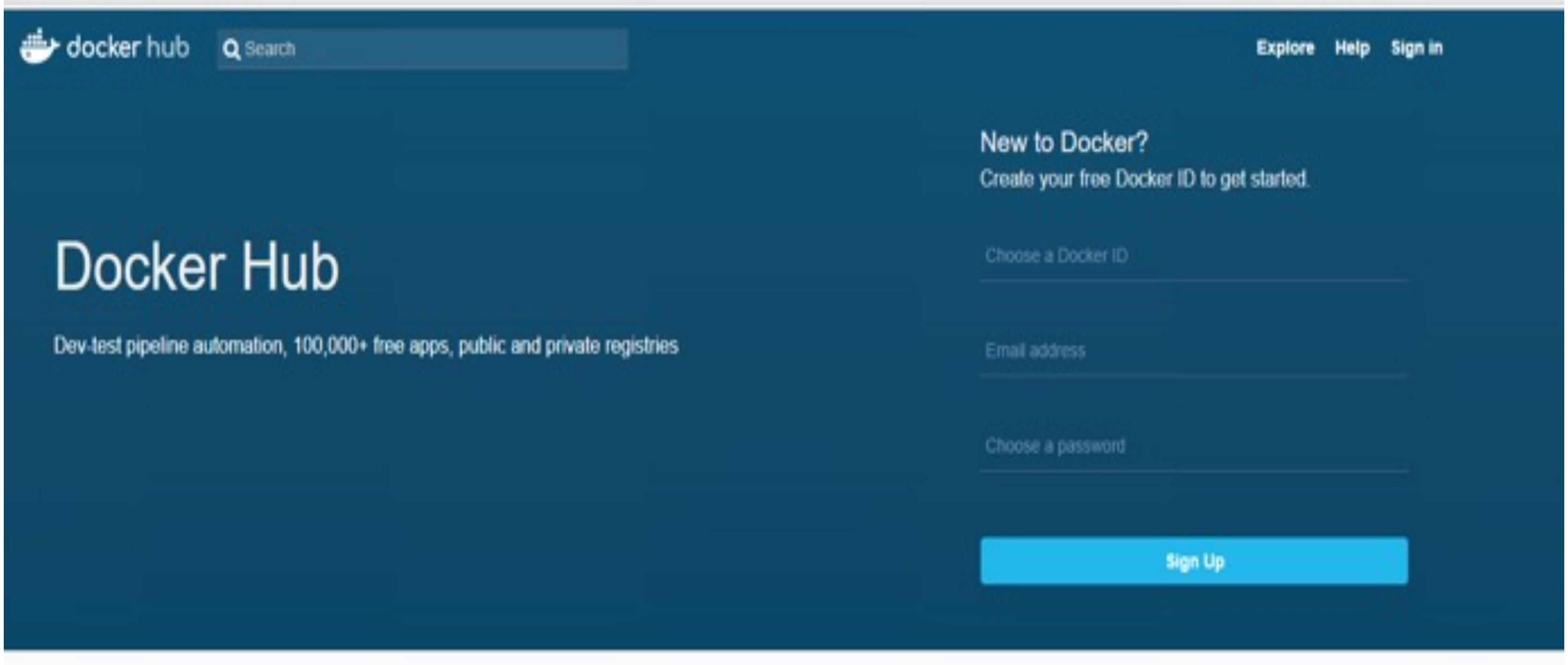
Container and layers

- The major difference between a container and an image is the top writable layer.
- All new/modification writes to the container are stored in this writable layer.
- Multiple containers can share access to the same underlying image and yet have their own data state.
- Multiple containers sharing the same image:



Docker Hub

- Cloud-based registry service. Link
- Allows you to link to code repositories, build your images, stores manually pushed images, and links to Docker Cloud so you can deploy images to your hosts.
- It provides a centralized resource for container image discovery, distribution and management



Docker Hub

Dev-test pipeline automation, 100,000+ free apps, public and private registries

New to Docker?

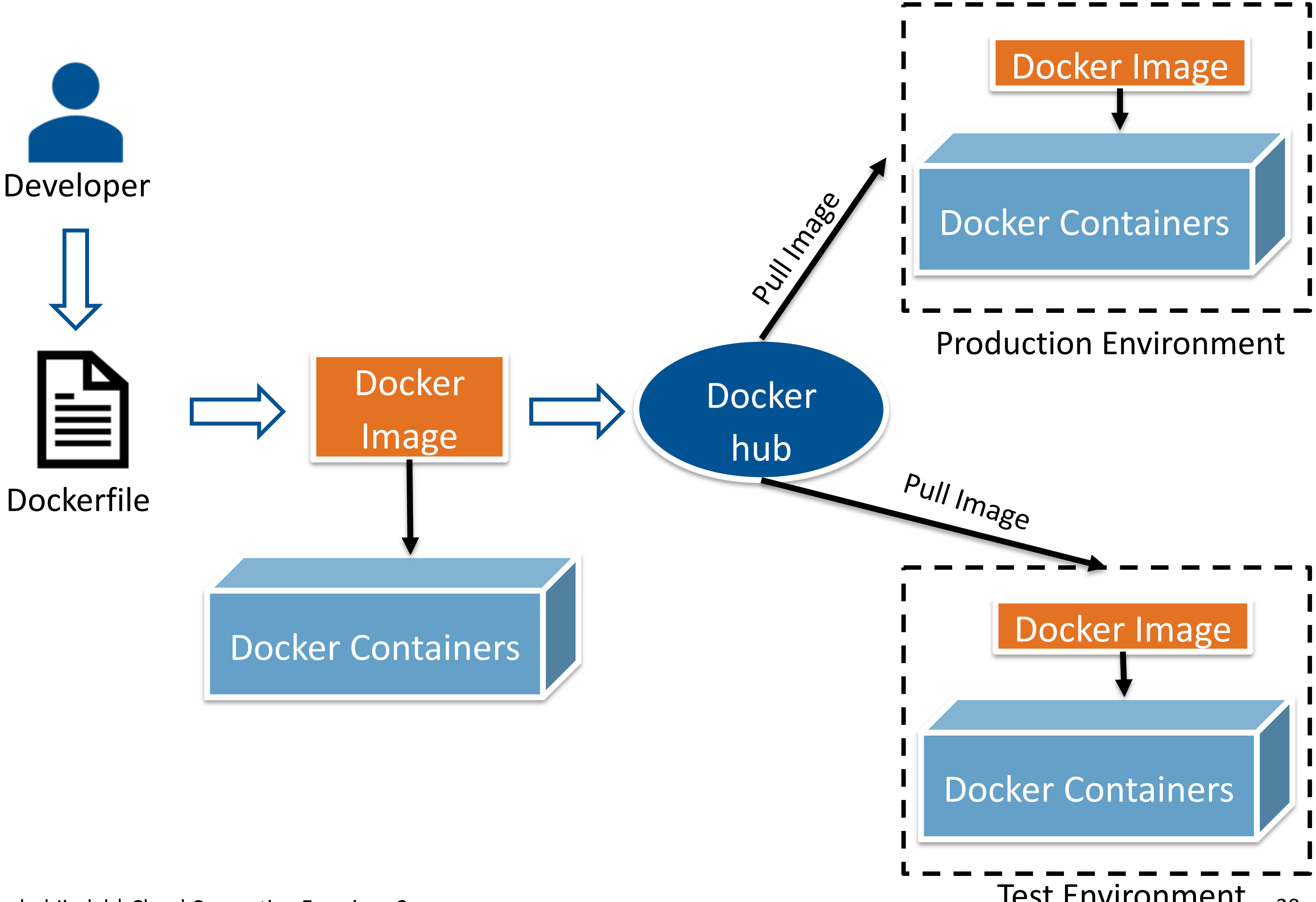
Create your free Docker ID to get started.

Choose a Docker ID

Email address

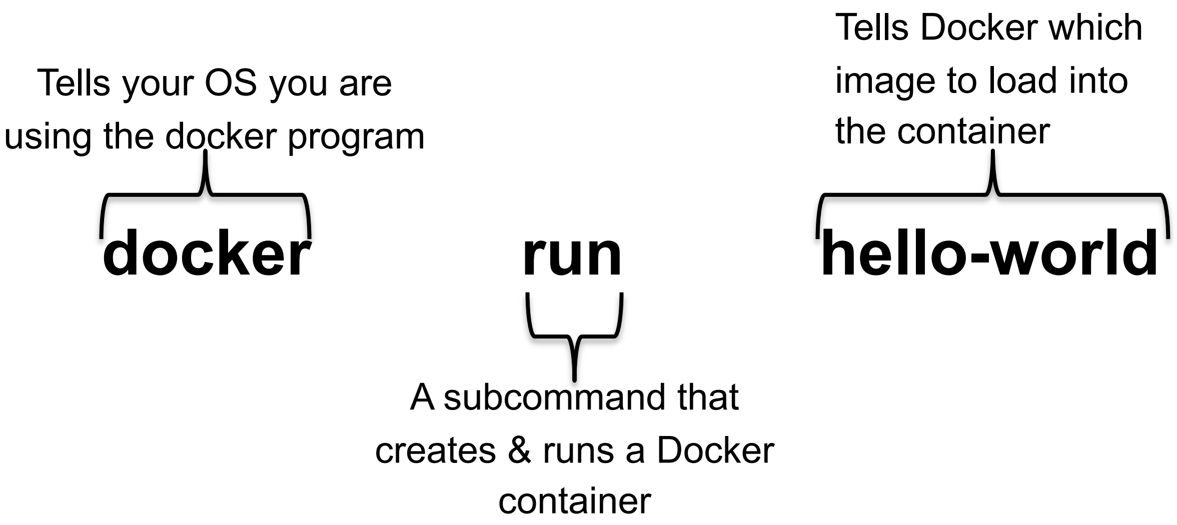
Choose a password

Docker Containers build flow



Docker Run Command

Docker Run Command



Tells your OS you are using the docker program

docker

run

A subcommand that creates & runs a Docker container

hello-world

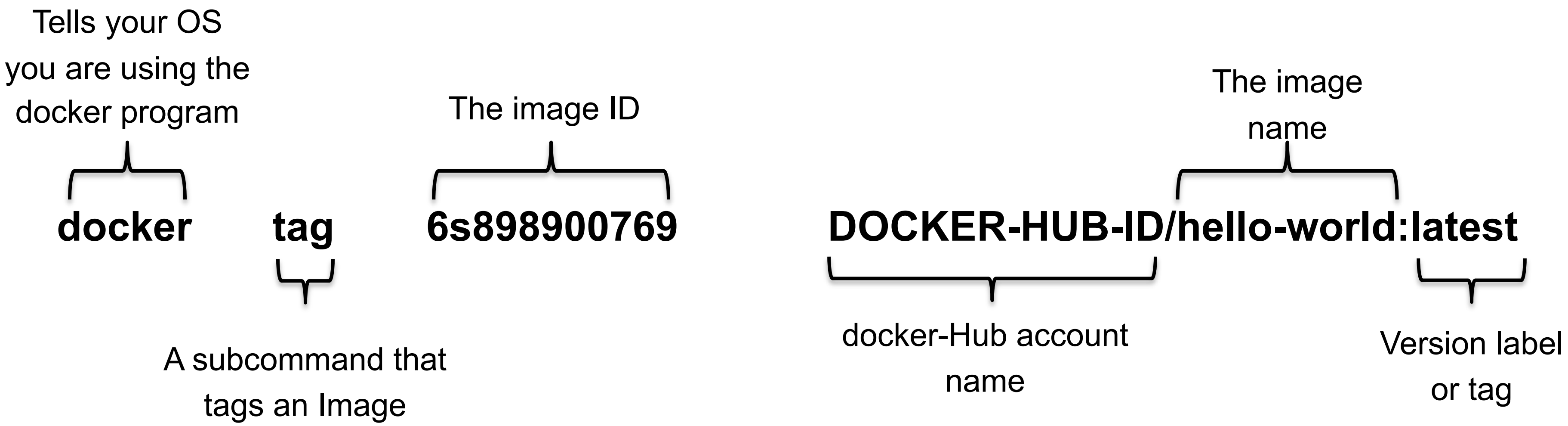
Tells Docker which image to load into the container

When we run the command, Docker Engine:

- check to see if we had the **hello-world** image
- download the image from the Docker Hub (if it’s not present)
- load the image into the container and “run” it

Tag, Push and Pull your Image

- You can tag your image and push it to your docker hub account.
- Tags are used to identify different versions.



Tells your OS you are using the docker program

The image ID

The image name

```
docker     tag     6s898900769          DOCKER-HUB-ID/hello-world:latest
```

A subcommand that tags an Image

docker-Hub account name

Version label or tag

Other Important Docker Commands

Build an image from the Dockerfile in the directory and tag the image

```
docker build -t <image-name>:<tag> <Directory>
```

List all images that are locally stored with the Docker engine

```
docker images
```

List Running Containers

```
docker ps
```

Stop a running container

```
docker stop <container-id>
```

Remove a running container

```
docker rm <container-id>
```

Delete an image from the local image store

```
docker rmi <image-name>
```

```
docker run
```

--rm	remove container automatically after it exits
-it	connect the container to terminal
--name <nam>	name the container
-p 5000:80	expose port 5000 externally and map to port 80
-v ~/dev:/code	create a host mapped volume inside the container
image:tag	the image from which the container is instantiated
/bin/sh	the command to run inside the container

Docker-compose

- Compose is a tool for defining and running multi-container Docker applications.
- It also helps to link multiple services.
- Uses a docker-compose.yml file, it is written in **YAML** format.
  - YAML (YAML Ain't Markup Language) is a human-readable data serialization language.
  - It uses Python-style indentation to indicate nesting, and uses [] for lists and {} for maps.
  - YAML 1.2 a superset of JSON.

docker-compose.yml file

```
version: '3'
services:

  server:
    build: ./server

    image: HUB_ID/cloudcomputinggroup#:latest
    container_name: cloudcomputinggroup#
    depends_on:
      - "mongodb"
    environment:
      - MONGODB_URI= mongodb://mongodb:27017/booksData
    ports:
      - ”3000:3000“

  mongodb:
    image: mongo:latest
    container_name: "mongodb"
    environment:
      - MONGO_DATA_DIR=/data/db
    volumes:
      - ./data/:/data/db
    ports:
        - "27017:27017"
```

Version of docker-compose file

Start of all services

Server service container

Path to make the image from

Image location on docker hub

Name of the container

This service depends on mongodb service.

Environment Variables

Mapping of VM port to container port

MongoDb container

Docker hub repo/image name of mongodb

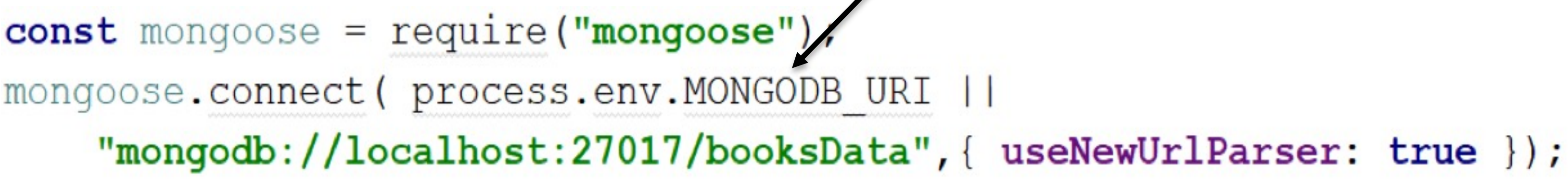
Environment Variables

Volume to be mounted

Mapping of VM port to container port

docker-compose.yml file continued..

Usage of environment variable



```
const mongoose = require("mongoose");
mongoose.connect( process.env.MONGODB_URI ||
    "mongodb://localhost:27017/booksData",{ useNewUrlParser: true });
```

Docker-compose commands

Build all the images

```
docker-compose build
```

Build only the selected image

```
docker-compose build <image-name>
```

Log in to a registry (the Docker Hub by default)

```
docker login
docker login <registry-host>
```

Push images to a registry

```
docker-compose push
```

To start all containers

```
docker-compose up
```

To start all containers in background

```
docker-compose up -d
```

Stop the containers

```
docker-compose stop
```

Kill the containers

```
docker-compose kill
```

Remove stopped containers

```
docker-compose rm
```

Installation and Running the application

Install docker and docker-compose

Docker Installation (use this official steps only):

https://docs.docker.com/install/linux/docker-ce/ubuntu/

Docker Compose Installation:

https://docs.docker.com/compose/install/#prerequisites

Enable Docker Remote API on the VM

1. Edit the file /lib/systemd/system/docker.service
2. Modify the line that starts with ExecStart to look like this

```
ExecStart=/usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock -H tcp://0.0.0.0:4243
```

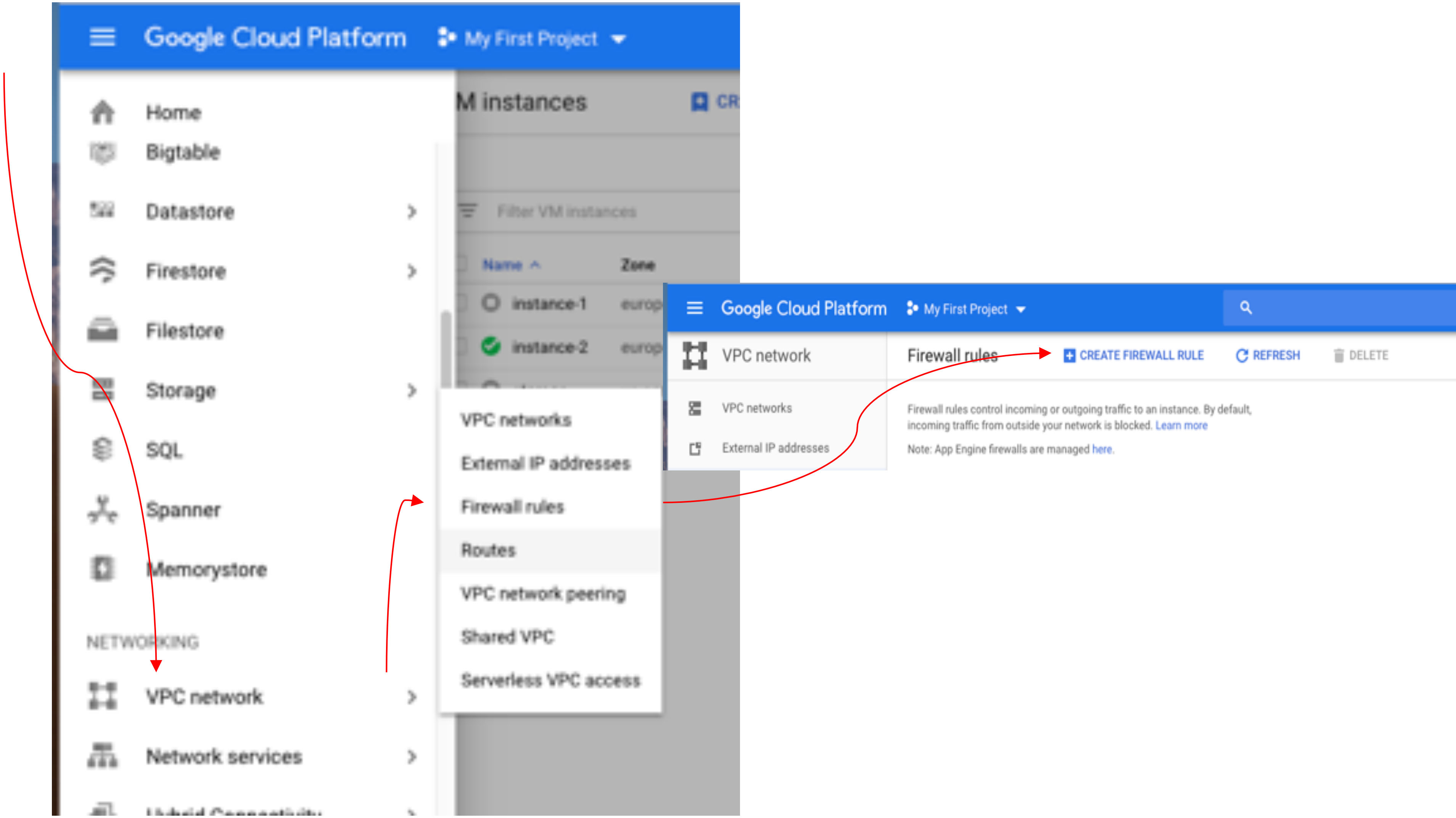
Where the addition is “-H tcp://0.0.0.0:4243”

3. Save the modified file
4. Run systemctl daemon-reload
5. Run sudo service docker restart
6. Test that the Docker API is accessible:

```
curl http://localhost:4243/version
```

Enable port **4243** on your VM so that docker API can be accessed from outside network using your VM IP.

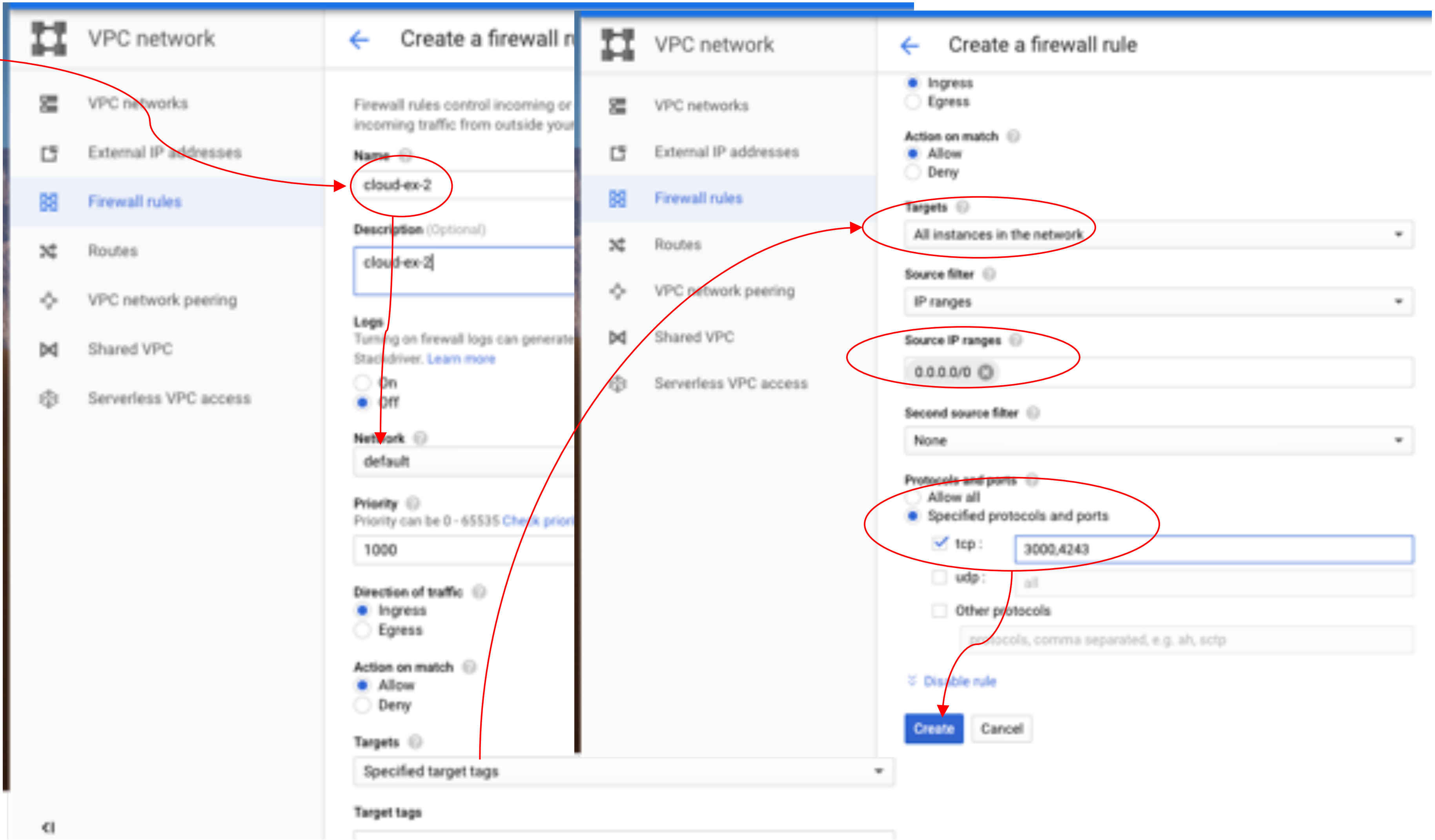
Enabling Ports on GCP



Firewall rules control incoming or outgoing traffic to an instance. By default, incoming traffic from outside your network is blocked. Learn more

Note: App Engine firewalls are managed here.

Enabling Ports on GCP



VPC network

- VPC networks
- External IP addresses
- Firewall rules
- Routes
- VPC network peering
- Shared VPC
- Serverless VPC access

← Create a firewall r

Firewall rules control incoming or
incoming traffic from outside you

Name: cloud-ex-2

Description (Optional): cloud-ex-2

Logs
Turning on firewall logs can generate
Stackdriver. Learn more

○ On
● Off

Network: default

Priority
Priority can be 0 - 65535 Check prior

1000

Direction of traffic
● Ingress
○ Egress

Action on match
● Allow
○ Deny

Targets: Specified target tags

Target tags

VPC network

- VPC networks
- External IP addresses
- Firewall rules
- Routes
- VPC network peering
- Shared VPC
- Serverless VPC access

← Create a firewall rule

● Ingress
○ Egress

Action on match
● Allow
○ Deny

Targets: All instances in the network

Source filter: IP ranges

Source IP ranges: 0.0.0.0/0

Second source filter: None

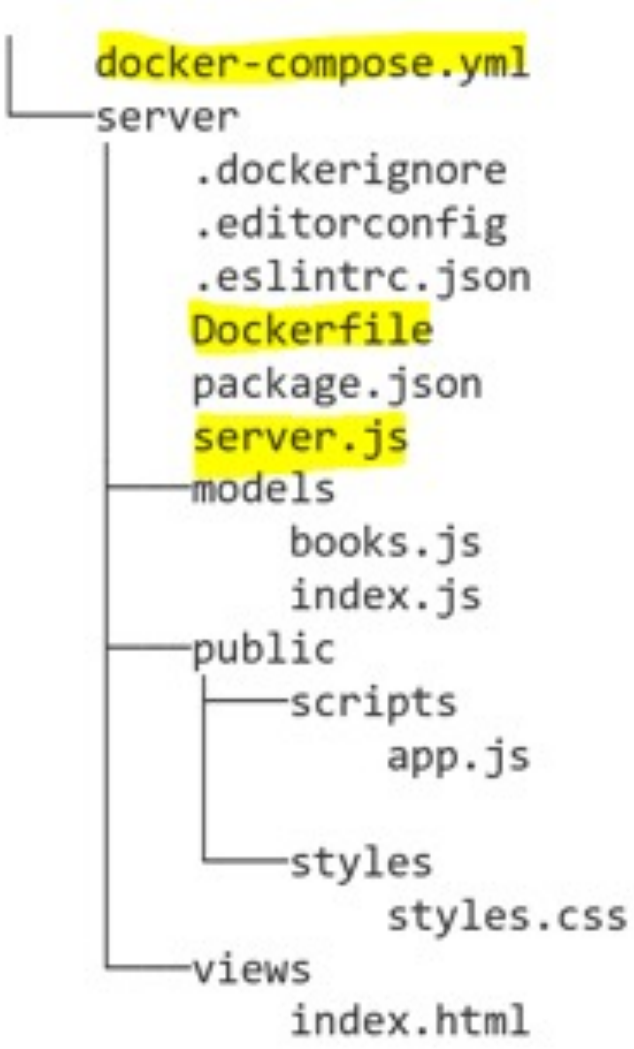
Protocols and ports
○ Allow all
● Specified protocols and ports

☑ tcp : 3000,4243
☐ udp : all
☐ Other protocols
protocols, comma separated, e.g. ah, sctp

Disable rule

Application Download and Make Changes

1. Download the provided application source zip file from Moodle.



```
│   docker-compose.yml
└───server
    │   .dockerignore
    │   .editorconfig
    │   .eslintrc.json
    │   Dockerfile
    │   package.json
    │   server.js
    ├───models
    │       books.js
    │       index.js
    ├───public
    │   ├───scripts
    │   │       app.js
    │   │
    │   └───styles
    │           styles.css
    └───views
            index.html
```

Highlighted are the ones which need to modified or added

Running and testing the Application

1. Do the changes to the application on your local laptop/computer.
2. Check the application is running or not locally.

```
sudo docker-compose up –build
```

If everything is working correctly now :

1. Create a repository on docker hub.
2. Login to your hub account using the command on your local machine :

```
sudo docker login
```

4. Push Images to docker hub (don’t forget to add your docker hub id and image name into **docker-compose.yml file**)

```
sudo docker-compose push
```

5. Copy the **docker-compose.yml** file to the VM and remove **build** line from it.
6. Run the application using docker-compose

```
sudo docker-compose up
```

7. Test the Application is running or not by going to the URLs:
   - http://YOUR_VM_PUBLIC_IP:3000/api/exercise2
   - http://YOUR_VM_PUBLIC_IP:3000/api/profile
   - http://YOUR_VM_PUBLIC_IP:3000/api/books

Tasks to be Completed

Tasks to be completed

As part of the exercise2, there are following tasks to be completed:

1. Add an API in your application:
   1. **/api/exercise2 :** Which sends a message **"group # application deployed using docker“** back to the user when called the API.
   2. Make sure to use the same completed application (all the first exercise tasks).
2. Create the missing docker file of your application and then build the image.
3. Create your docker hub account and push your application image to it. First do a **docker login** then **docker-compose push**.
4. Name of your application image should be as **cloudcomputinggroup#**
5. Start a VM and run the provided docker-compose file. This will pull this application image on it along with mongo image and run them using docker.
6. Enable docker remote API

Deadline for submission: Check the exercise page on server

*Replace # with your group number in above tasks.

Submission

Submission Instructions

To submit your application results you need to follow this :

1. Open the Cloud Class server url : https://cloudcom.caps.in.tum.de/
2. Login with your provided username and password.
3. After logging in, you will find the button for **exercise2**
4. Click on it and a form will come up where you must provide
   - VM ip on which your application is running
   - and the **dockerhub** image path name.

   Example:

   10.0.23.1

   dockerHubUserId/myImageName
5. Then click submit.
6. You will get the correct submission from server if everything is done correctly.

Remember no cheating and no Hacking ☺

Hints

- For sending message from the API **use res.send(“message here”) method.**
- First test locally then only submit on server.
- Enable ports on VM
  - 3000
  - 4243

Thank you for your attention!