PPT Parag Pallav Talks

CMSC38T

Introduction To Docker

Source: LinkedIn
Curated by: Parag Pallav Singh

Contents

1. Docker

Understanding Docker and
Containerized Applications.

Virtual Machines

- Isolates applications and allocates resources to run that application
- VMs can be shared as images
- Aren’t dependent on the Host OS
- Multiple VMs can be run simultaneously using a hypervisor

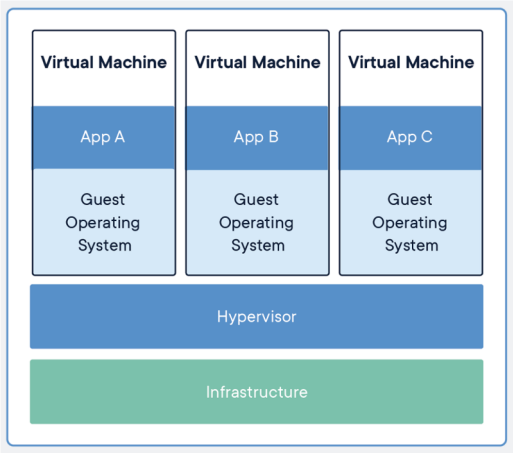


Image Source: docker.com

Docker Containers

- Standard unit of software
- Packages code and dependencies
- Can be shared as Docker Images
- Multiple containers can be run simultaneously
- Portable - Can be used with any OS
- Lightweight - Uses the host operating system
- Secure - Strong default isolation features
- Sometimes used with VMs

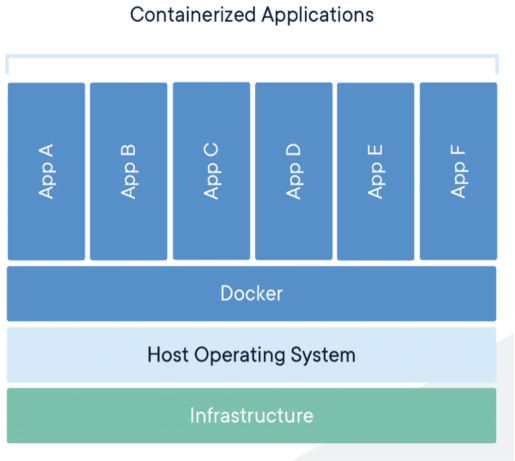


Image Source: docker.com

Microservices

- Breaks large applications down into smaller executable components
- Easy to maintain and test
- Loosely coupled and can be deployed independently
- Can be combined with serverless architecture (AWS Fargate)

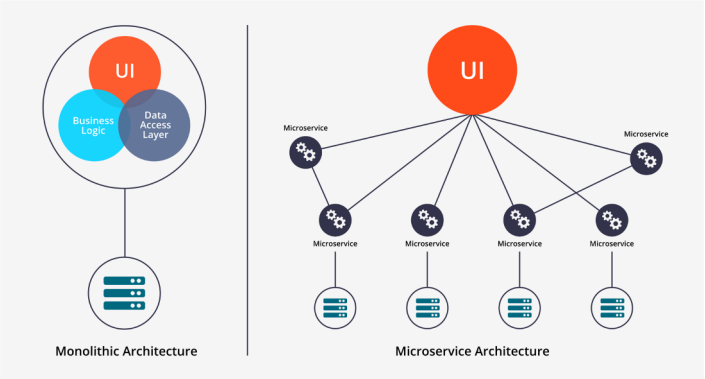


Image Source: hackernoon.com

Why Use Docker

- Develop applications that work on any OS
- Easy to share applications among teams
- Easy to scale across multiple servers
- Large applications can be broken into multiple containers - one for each microservice
- Great solution for Cloud Computing
- Big community and library of Docker Images

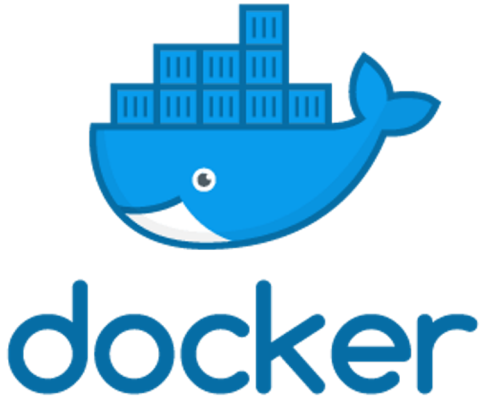


Image Source: docker.com

Serverless

- Removes Dependency on Infrastructure
- Allows developers to focus on application development
- Microservices can be decoupled with different cloud services
- Usually more cost effective
- Probably covered in more depth in a Cloud Computing class

Image Source: aws.amazon.com

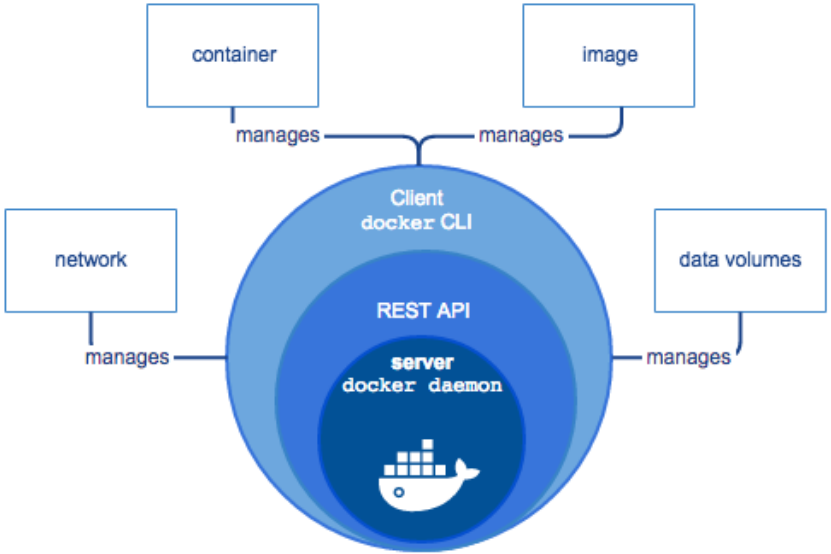
2. Using Docker CLI

Working with Gitlab to work in a DevOps environment with our existing Github Repositories

Install Docker Engine

Install Docker Engine

- Docker Engine is available on a variety of Linux platforms, macOS and Windows 10 through Docker Desktop



Pull An Image

There are many publicly available images that we can use to work with Docker.
The example below pulls a hello-world image using the **`docker pull`** command:

```
[ ~ ]docker pull hello-world
Using default tag: latest
latest: Pulling from library/hello-world
0e03bdcc26d7: Pull complete
Digest: sha256:31b9c7d48790f0d8c50ab433d9c3b7e17666d6993084c002c2ff1ca09b96391d
Status: Downloaded newer image for hello-world:latest
docker.io/library/hello-world:latest

[ ~ ]docker images
REPOSITORY    TAG       IMAGE ID       CREATED         SIZE
hello-world   latest    bf756fb1ae65   12 months ago   13.3kB
```

Create A Container

To create a container from an image we can use the **`docker create`** command

```
[ ~ ]docker create hello-world

2ffd5f2c5a7562fbf1d7b89a14c11a52e5843dd7938f380a8cd53f3952da99de
```

Run A Container

To run a container we can use the **`docker container start`** command to start a container. The **`-i`** runs the container interactively and allows us to see the output

```
[ ~ ] docker container start -i 2ffd5f2c5a7562fbf1d7…
```

Hello from Docker!
This message shows that your installation appears to be working correctly.

Run An Image

There is a shortcut for building a container from an image and running it with the **`docker run`** command. This will create a new container for an image and run it:

```
[ ~ ] docker run hello-world

Hello from Docker!
This message shows that your installation appears to be working correctly.
```

List Images

To see what images are already installed on your machine you can use the **`docker image ls`** command. We can see our hello-world image below:

```
[ ~ ]docker image ls
REPOSITORY    TAG       IMAGE ID       CREATED          SIZE
ubuntu        latest    f63181f19b2f   13 hours ago     72.9MB
hello-world   latest    bf756fb1ae65   12 months ago    13.3kB
```

List Containers

To list the containers that we have built, we can use the **`docker container ls`** command. The **`-a`** flag allows us to see both stopped and running containers. There are two containers below, one that was built with the **`docker create`** command and the other that was built with **`docker run`**:

```
[ ~ ]docker container ls -a

CONTAINER ID   IMAGE         COMMAND    CREATED          STATUS                       PORTS     NAMES
5017fd2b94c2   hello-world   "/hello"   7 minutes ago    Exited (0) 7 minutes ago               stoic_nobel
5f0cea57eacf   ubuntu        "bash"     10 minutes ago   Exited (127) 8 minutes ago             condescending_neumann
2ffd5f2c5a75   hello-world   "/hello"   14 minutes ago   Exited (0) 13 minutes ago              hungry_mclaren
```

Running Interactively

Running containers interactively allows you to run commands inside the container if it supports it. We can use the openjdk image that we used before:

```
[ ~ ]docker run -it openjdk
Unable to find image 'openjdk:latest' locally
latest: Pulling from library/openjdk
a73adebe9317: Pull complete
8b73bcd34cfe: Pull complete
1227243b28c4: Pull complete
Digest: sha256:7ada0d840136690ac1099ce3172fb02787bbed83462597e0e2c9472a0a63dea5
Status: Downloaded newer image for openjdk:latest
Jan 21, 2021 4:48:58 PM java.util.prefs.FileSystemPreferences$1 run
INFO: Created user preferences directory.
|  Welcome to JShell -- Version 15.0.2
|  For an introduction type: /help intro

jshell> System.out.println(”hello world");
Hello world
```

This allows us to execute java commands line by line in a Java shell

List Running Processes

To see what containers are currently running, we can use the **`docker ps`** command. This is useful when you are running containers in the background.

```
[ ~ ]docker ps
CONTAINER ID   IMAGE     COMMAND    CREATED          STATUS          PORTS     NAMES
55e4a7c3ddcc   openjdk   "jshell"   11 seconds ago   Up 10 seconds             affectionate_kowalevski
```

Interactive shell

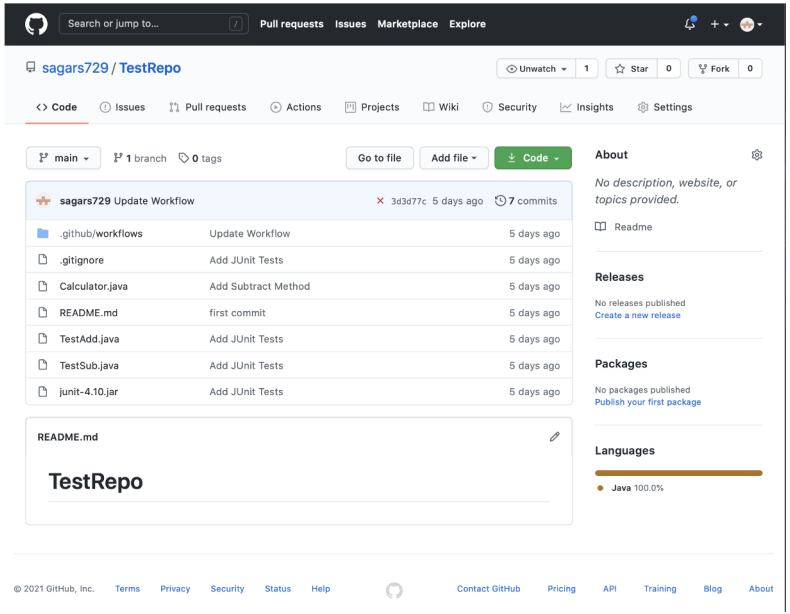
```
[ ~ ] docker run -it ubuntu bash
Unable to find image 'ubuntu:latest' locally
latest: Pulling from library/ubuntu
83ee3a23efb7: Pull complete
db98fc6f11f0: Pull complete
f611acd52c6c: Pull complete
Digest:
sha256:703218c0465075f4425e58fac086e09e1de5c340b12976ab9eb8ad26615c3715
Status: Downloaded newer image for ubuntu:latest
root@5f0cea57eacf:/#
```

3. Dockerize An Application

Dockerize your own application in a custom docker image

Revisiting the Calculator TestRepo

Let’s dockerize the code in the repository we created in Lecture 6. It has a calculator class and two JUnit tests for add and subtract.



Adding A Dockerfile

To create a custom docker image. We need to create a Dockerfile. The dockerfile specifies how our image should be built.

```
FROM openjdk

RUN useradd -ms /bin/bash ojdk

RUN mkdir -p /home/ojdk/app/ && chown -R ojdk:ojdk
/home/ojdk/app

WORKDIR /home/ojdk/app

COPY *.java ./
COPY junit-* ./

USER ojdk

RUN javac -cp "junit-4.10.jar:." *.java

COPY --chown=ojdk:ojdk . .

CMD [ "java", "-cp", "junit-4.10.jar:.",
"org.junit.runner.JUnitCore", "TestAdd", "TestSub"]
```

Dockerfile

Dockerfile Syntax

- **From** - The base image to use
- **Run** - Runs commands when building the docker image
- **Workdir** - Specifies the directory that commands are run from
- **User** - Switches users
- **Copy** - Copies Files
- **CMD** - Runs commands when running the container

```
FROM openjdk

RUN useradd -ms /bin/bash ojdk

RUN mkdir -p /home/ojdk/app/ && chown -R ojdk:ojdk
/home/ojdk/app

WORKDIR /home/ojdk/app

COPY *.java ./
COPY junit-* ./

USER ojdk

RUN javac -cp "junit-4.10.jar:." *.java

COPY --chown=ojdk:ojdk . .

CMD [ "java", "-cp", "junit-4.10.jar:.",
"org.junit.runner.JUnitCore", "TestAdd", "TestSub"]
```

Dockerfile

Dockerfile Explained

1. Use the OpenJDK image to have a pre-configured java environment
2. Add a new user “ojdk” that we will be using for executing scripts
3. Create a directory that will contain our files and give permission to our user
4. Change the working directory to the directory we created
5. Copy the java and junit files
6. Switch to user ojdk
7. Compile all of our code
8. Copy files to the working directory and give permissions to ojdk
9. Run all tests

```
FROM openjdk

RUN useradd -ms /bin/bash ojdk

RUN mkdir -p /home/ojdk/app/ && chown -R ojdk:ojdk
/home/ojdk/app

WORKDIR /home/ojdk/app

COPY *.java ./
COPY junit-* ./

USER ojdk

RUN javac -cp "junit-4.10.jar:." *.java

COPY --chown=ojdk:ojdk . .

CMD [ "java", "-cp", "junit-4.10.jar:.",
"org.junit.runner.JUnitCore", "TestAdd", "TestSub"]
```

Dockerfile

Build A Docker Image

To build a docker image using a Dockerfile we can use the **`docker image build`** command and provide it the directory where the Dockerfile exists. The **`--tag`** option allows us to name and tag the docker image.

```
TestRepo $ docker image build . --tag "calculator:latest"
Sending build context to Docker daemon  590.3kB
Step 1/9 : FROM openjdk
 ---> e105e26a0a75
Step 9/10 : COPY --chown=ojdk:ojdk . .
 ---> 47cff2b55e3c
Step 10/10 : CMD [ "java", "-cp", "junit-4.10.jar:.", "org.junit.runner.JUnitCore", "TestAdd", "TestSub"]
 ---> Running in c8395bc770b4
Removing intermediate container c8395bc770b4
 ---> 6b345c94e511
Successfully built 6b345c94e511
Successfully tagged calculator:latest
```

Run Our Custom Image

We can use the **`docker run`** command to run our image and we can see that our tests are being run:

```
TestRepo $ docker run calculator
JUnit version 4.10
..
Time: 0.006

OK (2 tests)

TestRepo $
```

Run Interactively

We can use **`docker run -it`** to run our image interactively and open a bash shell in our working directory:

```
[TestRepo $ docker run -it calculator bash
[[ojdk@419a727a1ca1 app]$ ls
Calculator.class  README.md      TestSub.class
Calculator.java   TestAdd.class  TestSub.java
Dockerfile        TestAdd.java   junit-4.10.jar
[ojdk@419a727a1ca1 app]$
```

Docker Compose File

Docker Compose files can be used to run multiple services at once and is great once you have many microservices as part of your application. For our project we create a single service calculator that

- is built using a custom Dockerfile
- tagged with the name calculator
- restarts unless it is stopped

```yaml
version: "3"

services:
  calculator:
    build:
      context: .
      dockerfile: Dockerfile
    image: calculator
    container_name: calculator
    restart: unless-stopped
```

docker-compose.yml

Run Docker Compose

To run our docker compose file, we use the **`docker-compose up`** command. This builds all images and and runs and containers.

```
TestRepo $ docker-compose up
Creating network "testrepo_default" with the default driver
Creating calculator ... done
Attaching to calculator
calculator    | JUnit version 4.10
calculator    | ..
calculator    | Time: 0.005
calculator    |
calculator    | OK (2 tests)
calculator    |
TestRepo $
```

Stop Containers

The calculator container will keep restarting unless its stopped. To stop all services, we can use the **`docker-compose down`** command:

```
TestRepo $ docker-compose down
Stopping calculator ... done
Removing calculator ... done
Removing network testrepo_default
TestRepo $
```

Clean Up

To remove all unused docker resources, we can use the **`docker system prune`** command with the **`--all`** flag:

```
TestRepo $ docker system prune --all
WARNING! This will remove:
  - all stopped containers
  - all networks not used by at least one container
  - all images without at least one container associated to them
  - all build cache

Are you sure you want to continue? [y/N]
```