Achieving Continuous DevOps – CI/CD

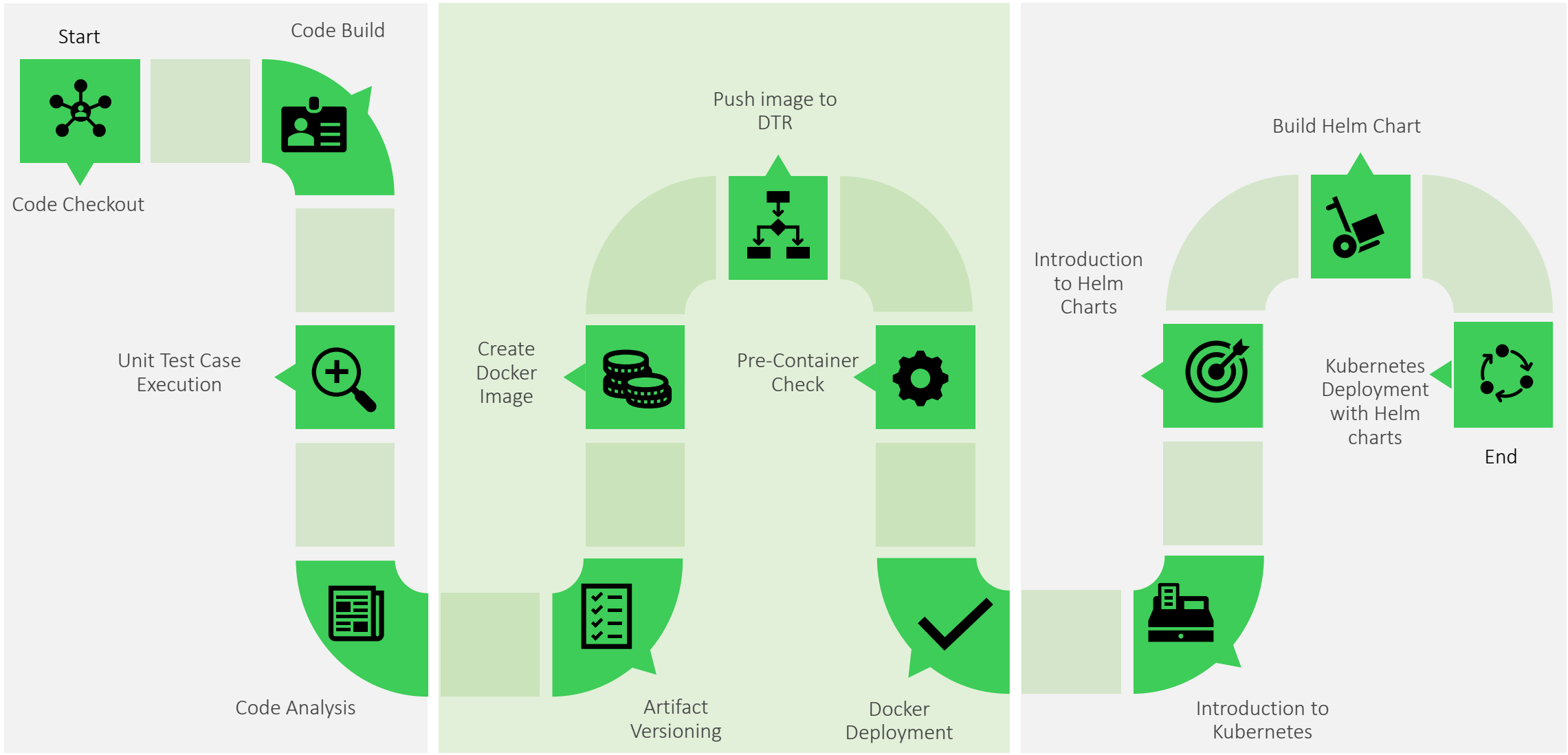
NAGP – Experienced Batch'20

nagarro

THINKING
BREAKTHROUGHS

Achieving Continuous DevOps …Recap

Agenda



Workshop 1 - 3
Continuous Integration *pipeline*

Workshop 4
Continuous delivery *pipeline*

Workshop 5
Continuous Deployment *pipeline*

Sample Jenkinsfile

An example Jenkinsfile looks like below

Jenkinsfile (Declarative Pipeline)

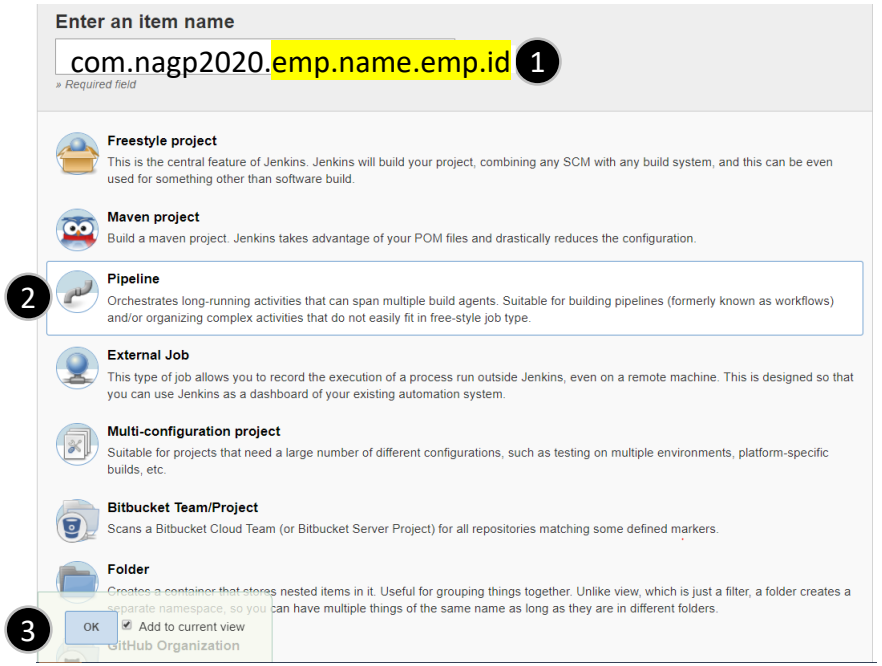
```
pipeline { (1)
    agent any (2)
    options {
        skipStagesAfterUnstable()
    }
    stages {
        stage('Build') { (3)
            steps { (4)
                sh 'make' (5)
            }
        }
        stage('Test'){
            steps {
                sh 'make check'
                junit 'reports/**/*.xml' (6)
            }
        }
        stage('Deploy') {
            steps {
                sh 'make publish'
            }
        }
    }
}
```

1. `pipeline` is Declarative Pipeline-specific syntax that defines a "block" containing all content and instructions for executing the entire Pipeline.
2. `agent` is Declarative Pipeline-specific syntax that instructs Jenkins to allocate an executor (on a node) and workspace for the entire Pipeline.
3. `stage` is a syntax block that describes a stage of this Pipeline. Read more about `stage` blocks in Declarative Pipeline syntax on the Pipeline syntax page. As mentioned above, `stage` blocks are optional in Scripted Pipeline syntax.
4. `steps` is Declarative Pipeline-specific syntax that describes the steps to be run in this `stage`.
5. `sh` is a Pipeline step (provided by the Pipeline: Nodes and Processes plugin) that executes the given shell command.
6. `junit` is another a Pipeline step (provided by the JUnit plugin) for aggregating test reports.
7. `node` is Scripted Pipeline-specific syntax that instructs Jenkins to execute this Pipeline (and any stages contained within it), on any available agent/node. This is effectively equivalent to `agent` in Declarative Pipeline-specific syntax.

Achieving Continuous Integration Pipeline (CI)

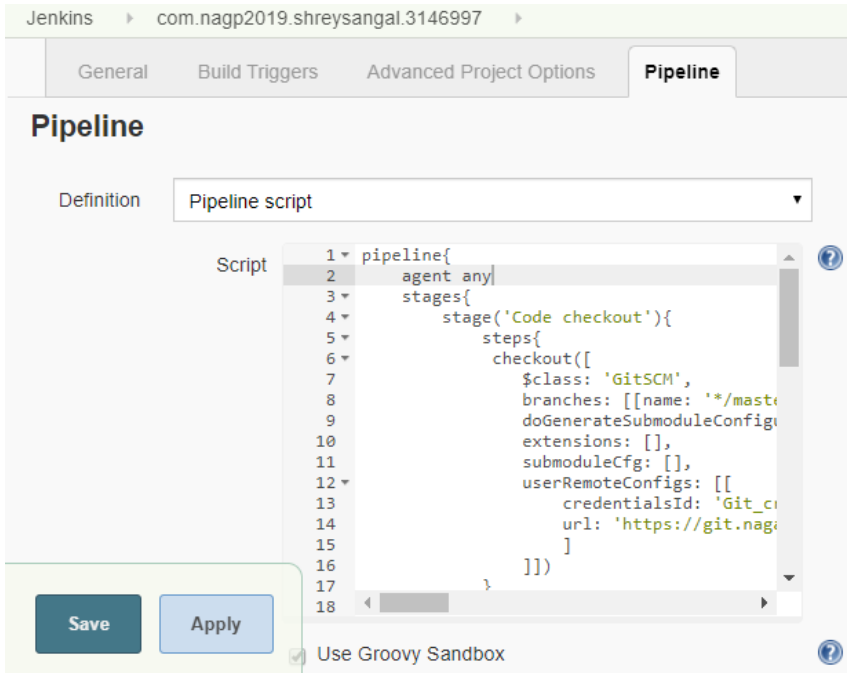
Code checkout → code build → AUT → sonar code coverage → Artifact upload

A Click on new Item on the left side of jenkins,



1. Type pipeline name following above naming convention
2. Select pipeline as type of job
3. Click OK with checked Add to current view

B Write pipeline script, and click Save

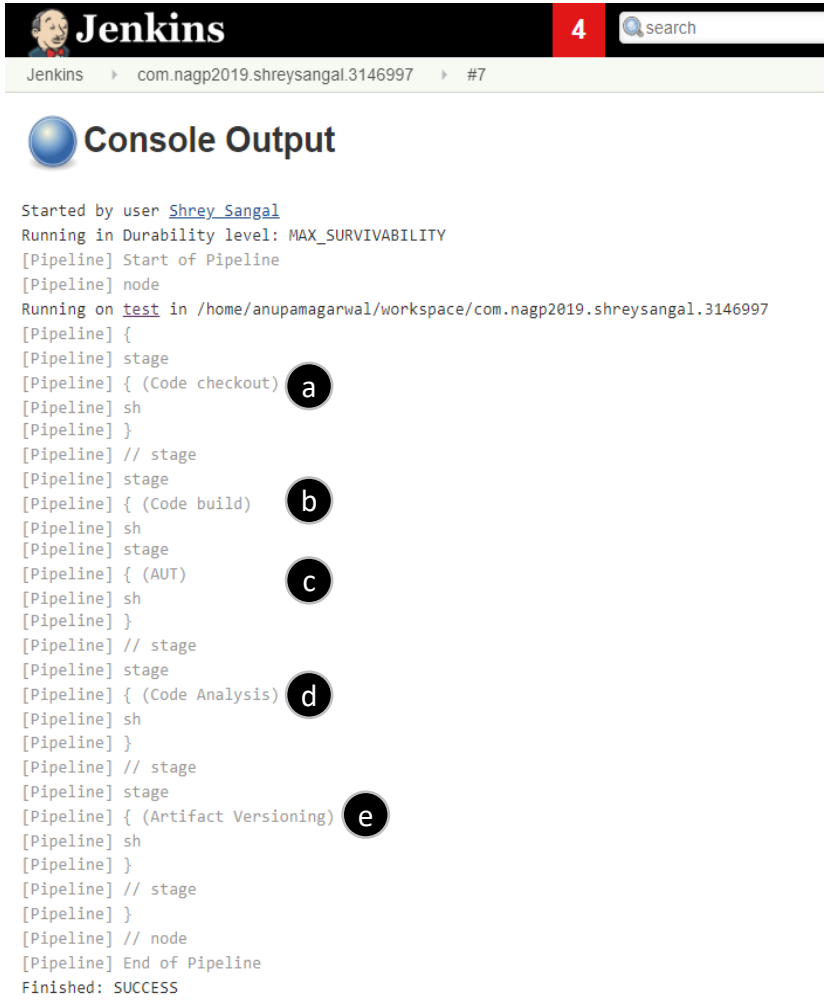


a. Code checkout
b. Code build
c. Automated unit testing
d. Code Analysis
e. Artifact versioning

Achieving Continuous Integration Pipeline (CI).. contd

Code checkout → code build → AUT → sonar code coverage → Artifact upload

c Click Build now, and check console logs

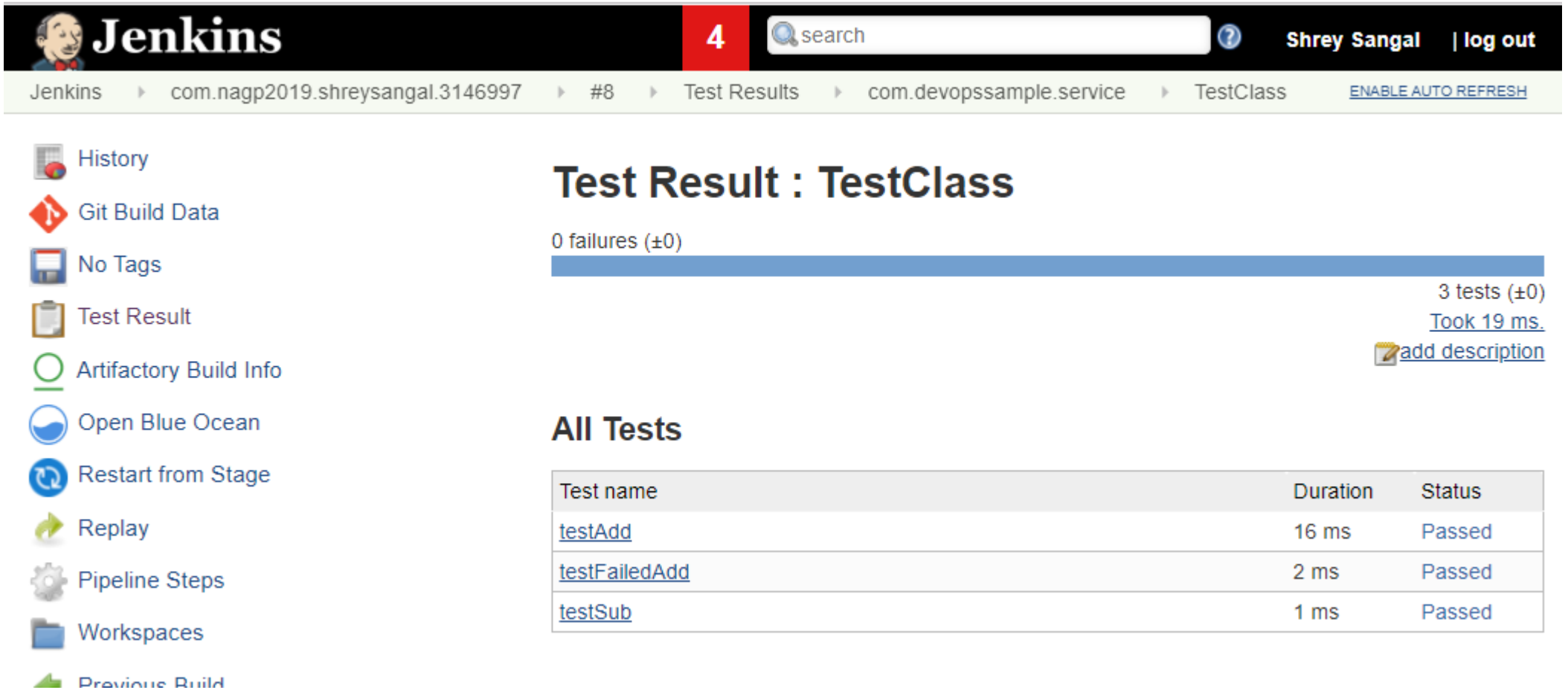


Automated unit testing

Jenkins pipeline triggers automated unit testing

Navigation steps:

1. Goto test results on the left pane
2. Click on test module

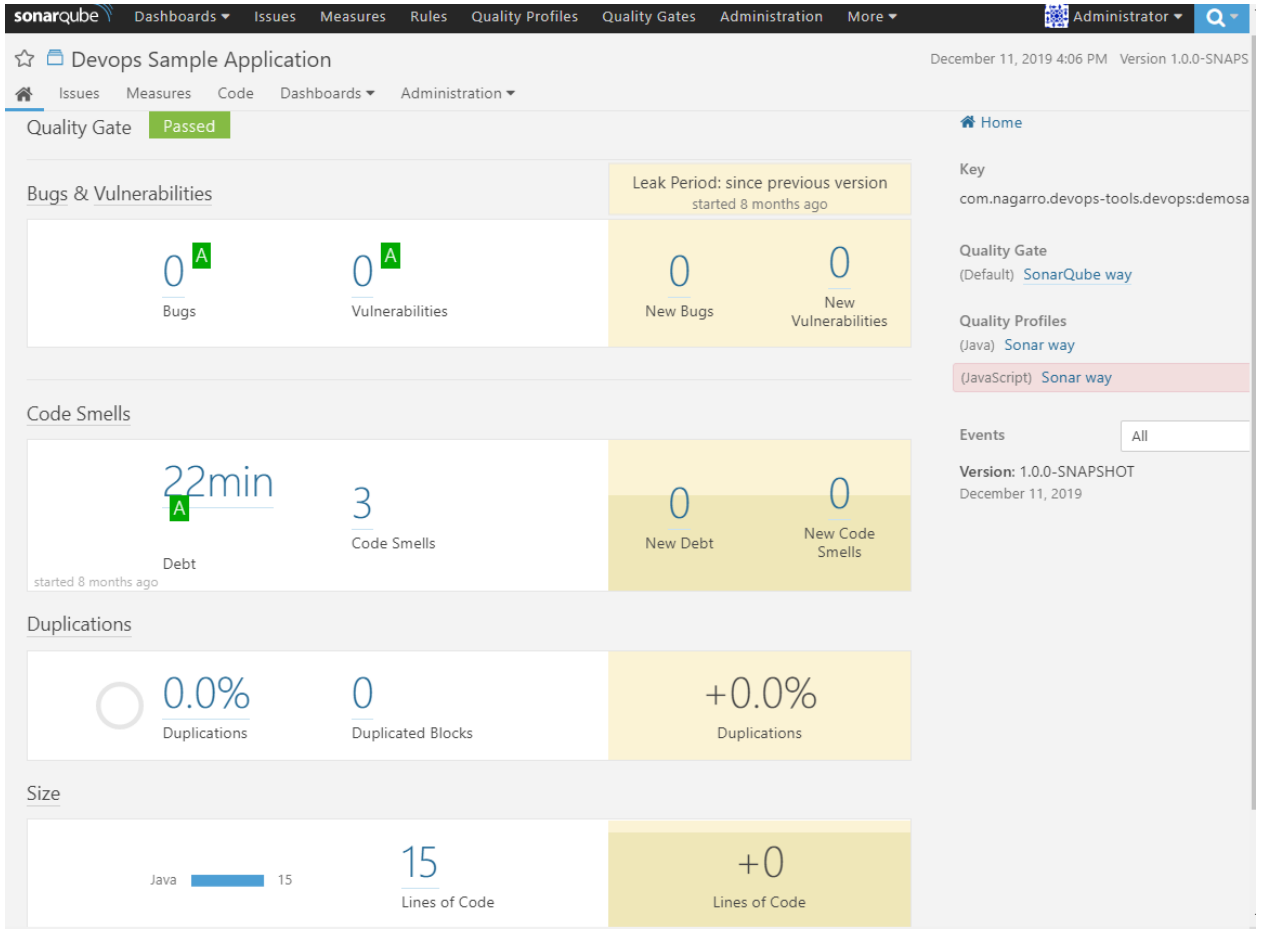


Code Coverage

Sonarqube code coverage execution within Jenkins pipeline

Navigation steps:

1. Navigate to URL
2. Login using your AD credentials

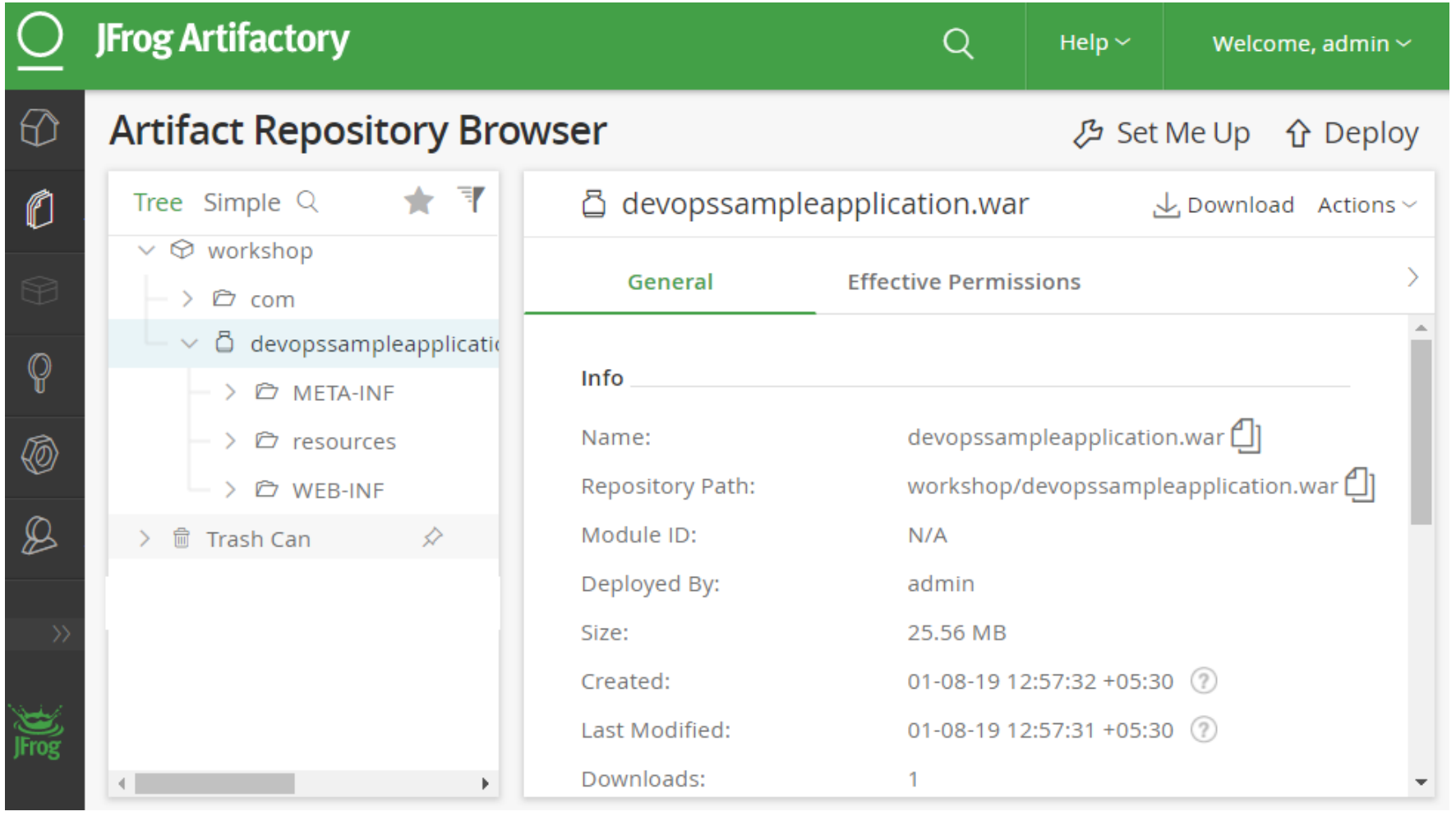


Artifact Versioning

Artifact get uploaded from jenkins build pipeline

Navigation steps:

1. Navigate to URL
2. Login using your AD credentials



Overview

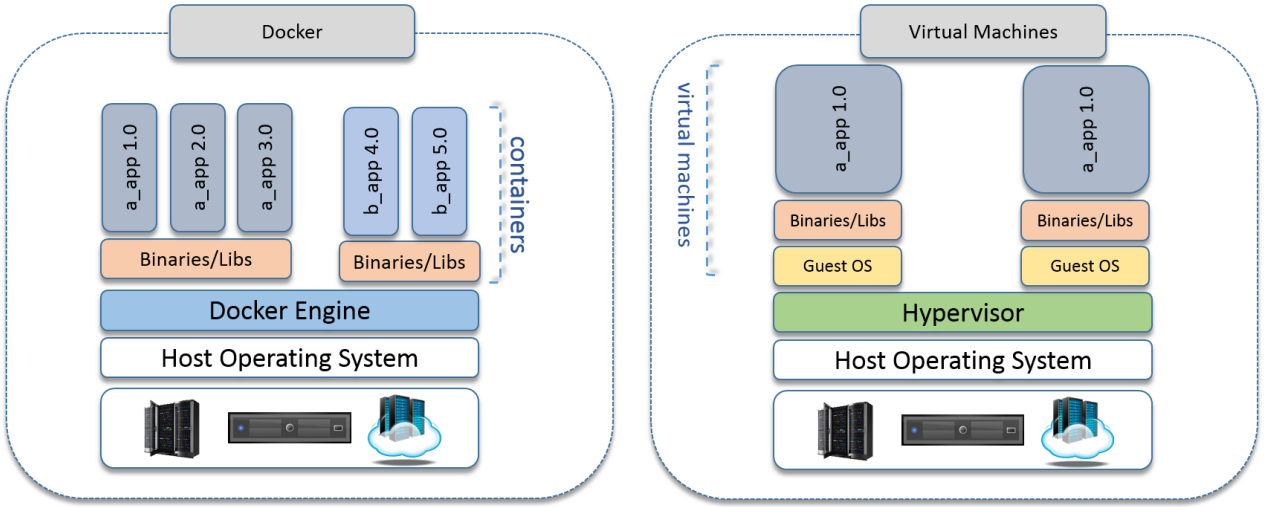
What is Docker

Docker is a container management service. The keywords of Docker are **develop**, **ship** and **run** anywhere. The whole idea of Docker is for developers to easily develop applications, ship them into containers which can then be deployed anywhere. The initial release of Docker was in **March 2013** and since then, it has become the buzzword for modern world development, especially in the face of Agile-based projects.

How Docker is different from VMs

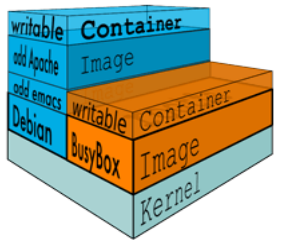
Docker allows to package an application with all of its dependencies into a standardized unit for software development.

- Virtual machines
  - Have a full **OS** with its own memory management installed with the associated overhead of virtual device drivers.
  - In a virtual machine, valuable resources are **emulated** for the guest OS and **hypervisor**, which makes it possible to run many instances of one or more operating systems in parallel on a single machine (or host).
  - Every guest OS runs as an **individual entity** from the host system.



- On the other hand Docker containers
  - Are executed with the **Docker engine** rather than the hypervisor.
  - Containers are therefore **smaller** than Virtual Machines and enable **faster** start up with better **performance**, less isolation and greater compatibility possible due to sharing of the host's kernel.
  - Docker Containers are able to **share** a single **kernel** and share application **libraries**

Docker images



A Docker image is built up from a series of layers and each layer contains a change..

Each layer represents an instruction in the image's Dockerfile.

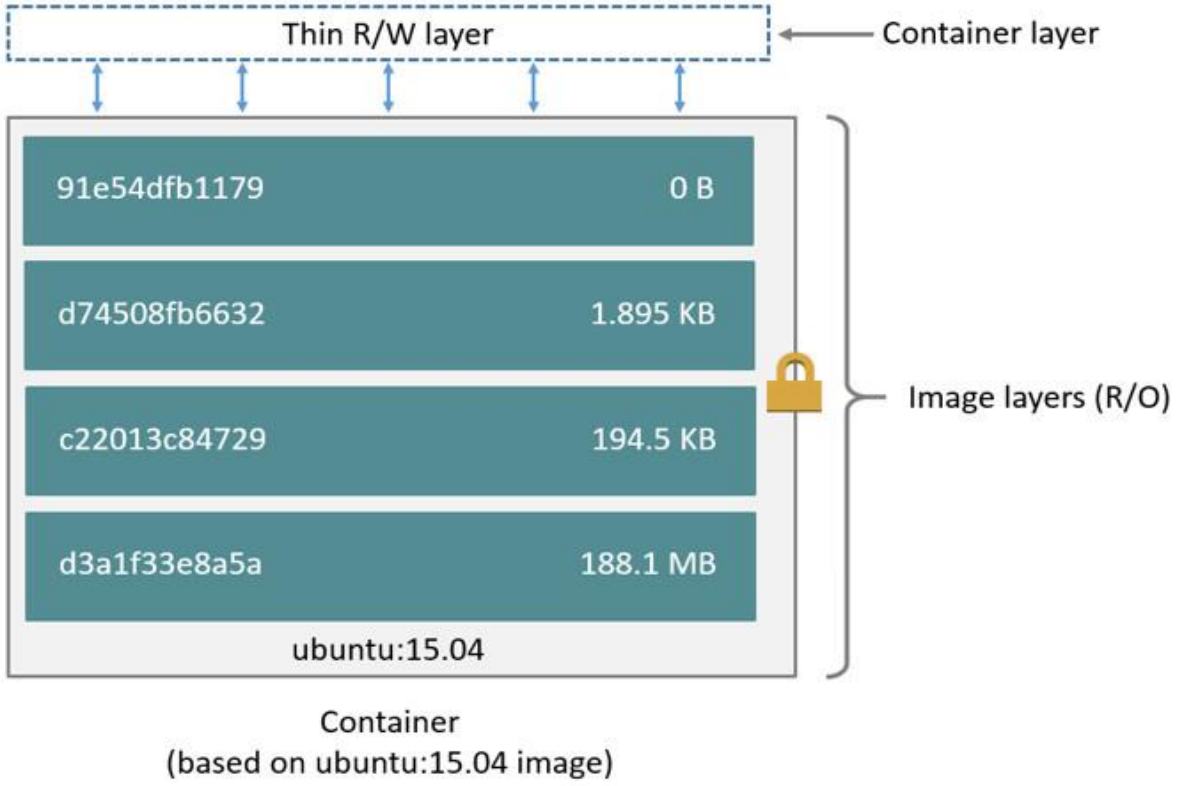
Consider the following Dockerfile:

FROM ubuntu:15.04

COPY . /app

RUN make /app

CMD python /app/app.py



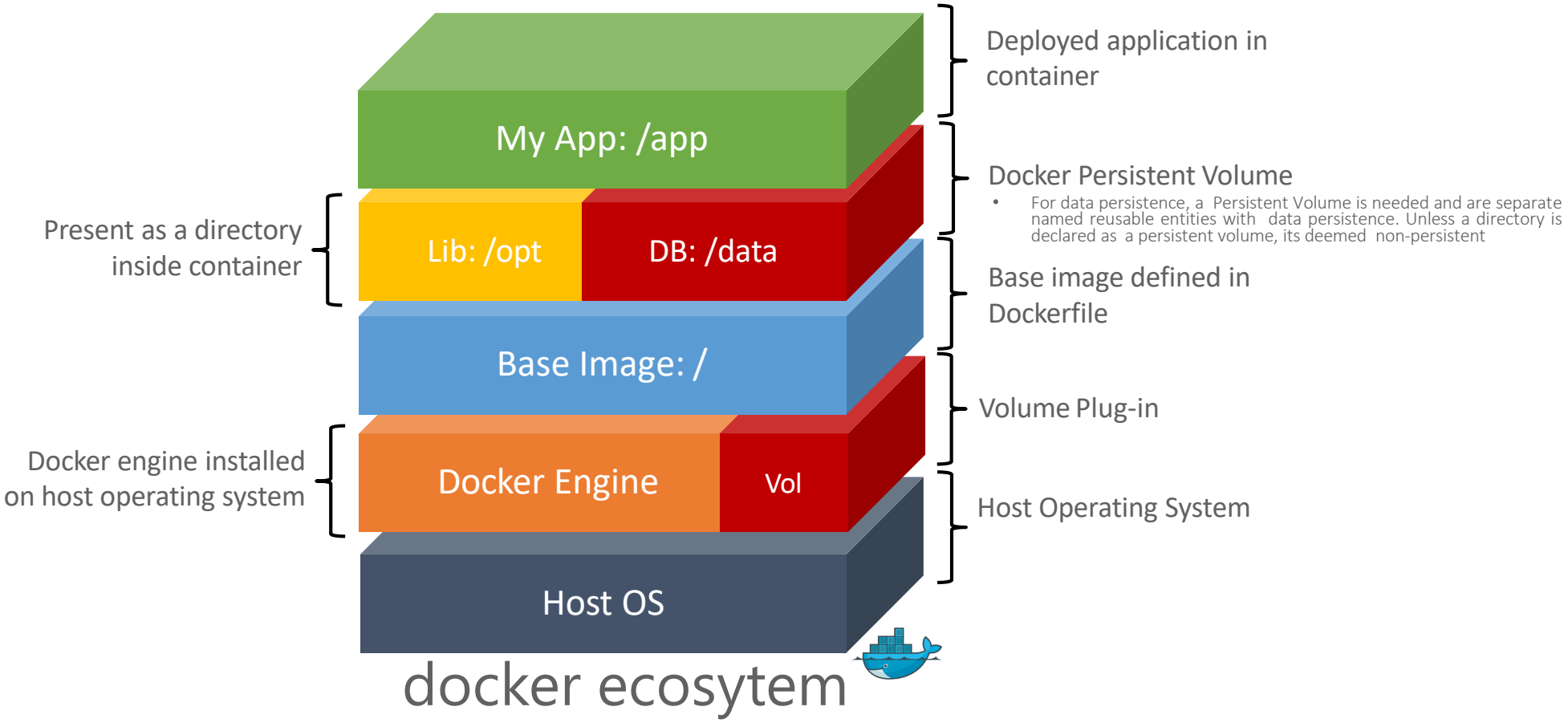
This Dockerfile contains four commands, each of which creates a layer. Each layer is only a set of differences from the layer before it. The layers are stacked on top of each other. When you create a new container, you add a new writable layer on top of the underlying layers.

TAR (archive) file containing all the binary and configuration files needed to run a container eg: docker save/load.

Containers

Each container has its own
- Root filesystem
- Processes
- Memory
- Devices
- Network ports

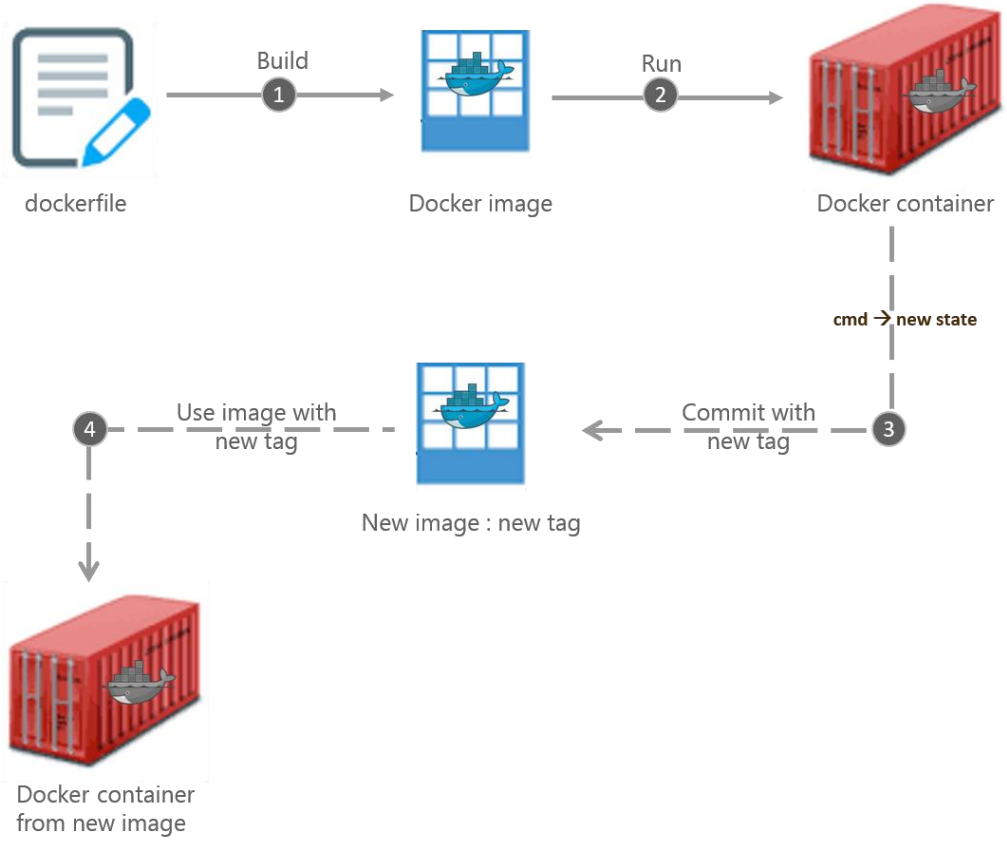
Containers are an operating system level virtualization technology for providing multiple isolated environments on a single host. Unlike virtual machines (VMs), containers do not run dedicated guest operating systems. Rather, they share the host operating system kernel and make use of the guest operating system libraries for providing the required OS capabilities. Since there is no dedicated operating system, containers start much faster than VMs.



Note *that whilst persistent volumes can be assigned to more than one container concurrently; you need to manage data change collisions/conflicts as there is no locking mechanism in Docker*

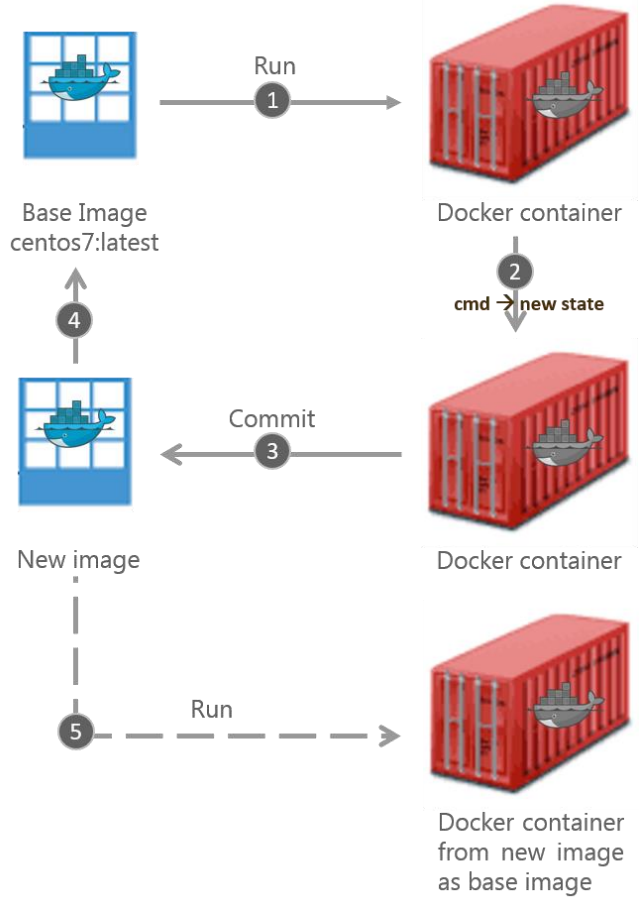
Image vs container

Scenario 1



Source: Dockerfile

Scenario 2

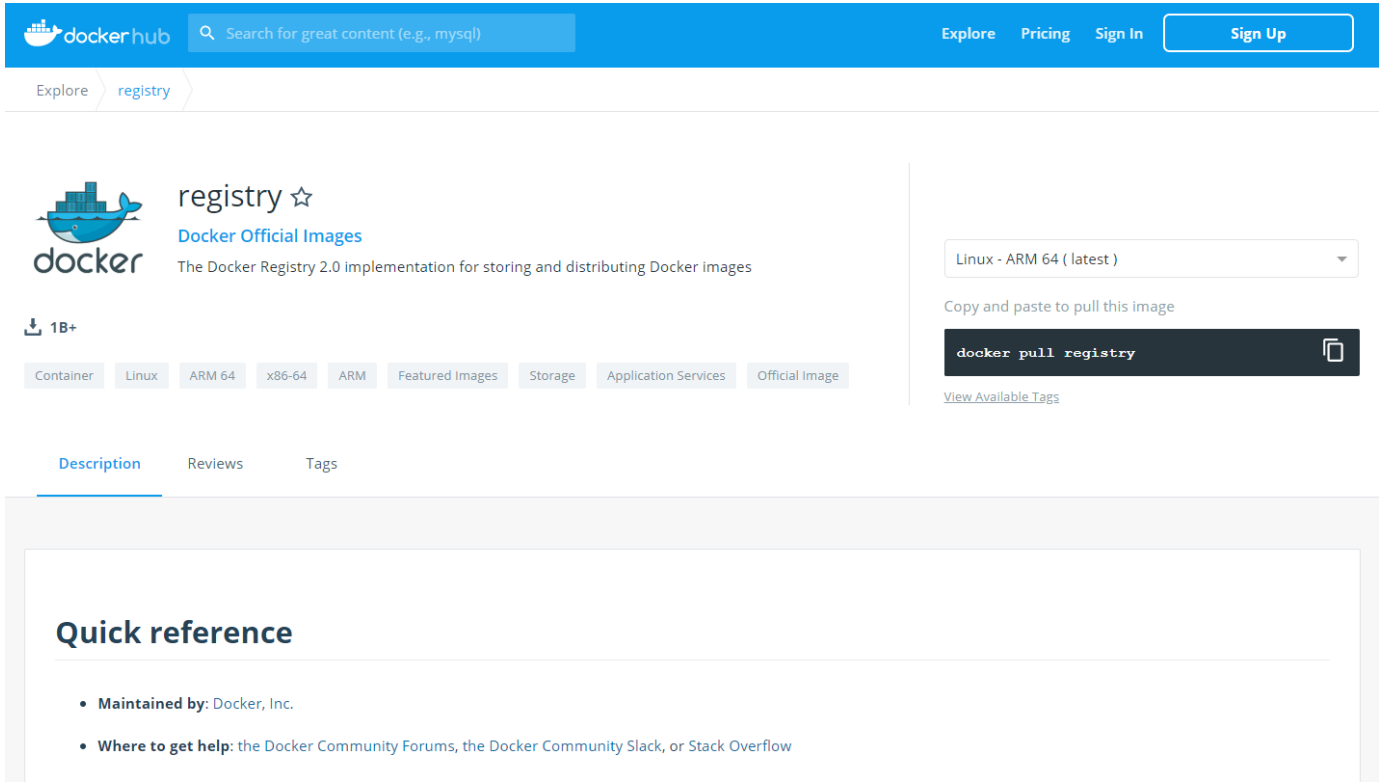


Source: Base docker Image

Docker registry

Docker Hub

Docker Hub is a cloud-based registry service which allows you to link to code repositories, build your images and test them, 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 change management, user and team collaboration, and workflow automation throughout the development pipeline.



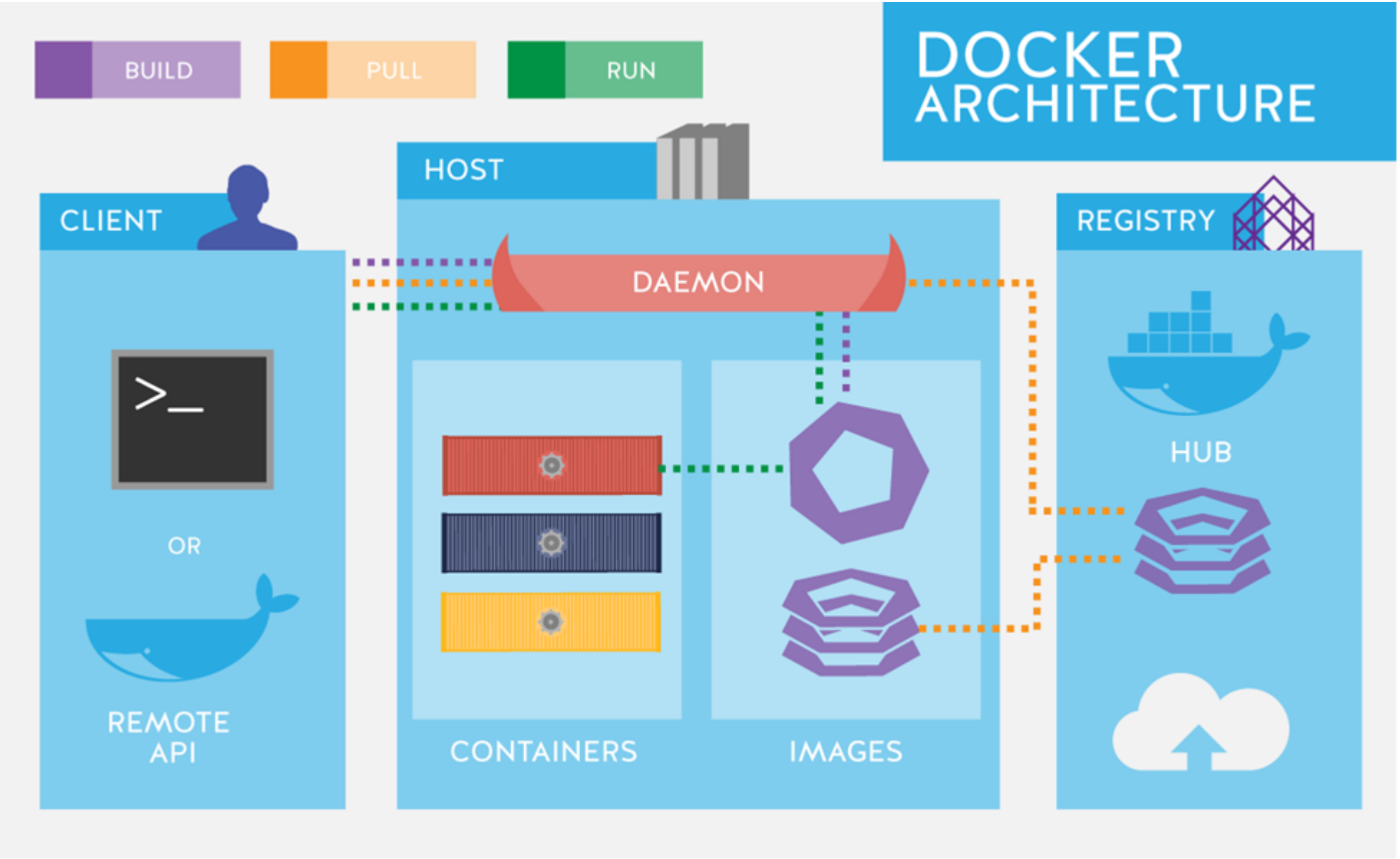
Example: docker pull ubuntu instructs docker to pull an image named ubuntu from the official Docker Hub. This is simply a shortcut for the longer docker pull docker.io/library/ubuntu command

Private Registry

The Registry is a stateless, highly scalable server-side application that stores and lets you distribute Docker images. The Registry is open-source, under the permissive Apache license.

Example: docker pull myregistrydomain:port/foo/bar instructs docker to contact the registry located at myregistrydomain:port to find the image foo/bar

Docker architecture



Docker commands

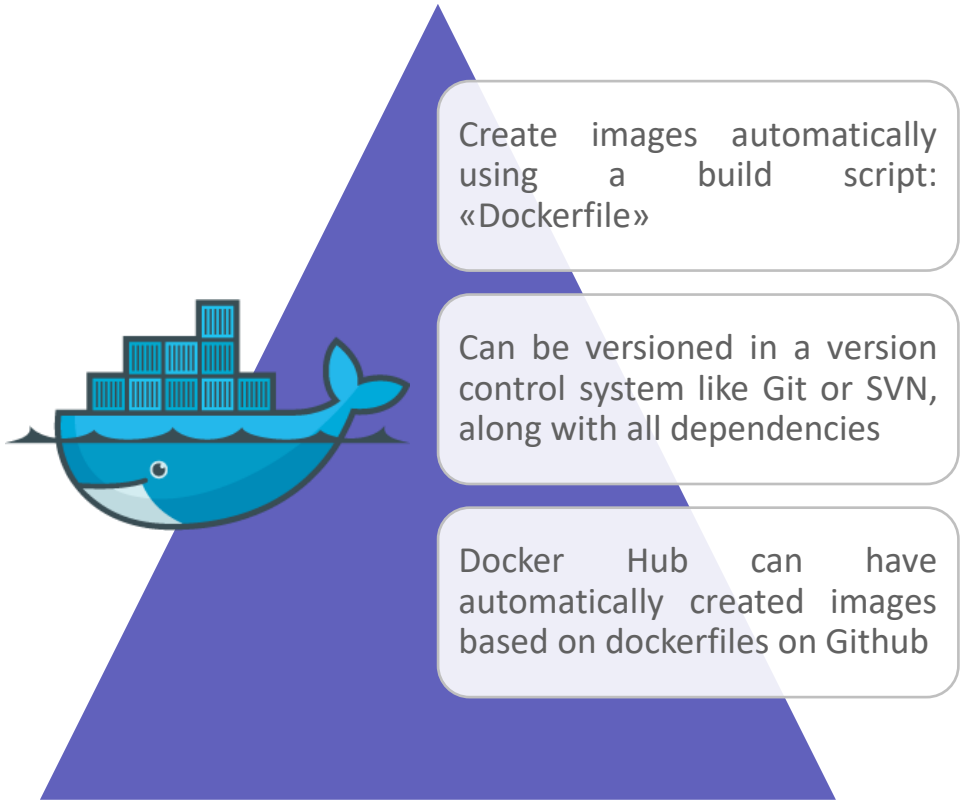
Command	Function
docker info	Show information on the docker engine
docker version	Show the version of docker running
docker run	Start / deploy new containers
docker ps (and ps –a)	Show running (or stopped) containers
docker stats <containerid>	Show performance info for a container
docker logs <containerid>	Show logs for a container
docker rename (because we always forget to name our containers)	Rename a deployed container
docker start, docker stop	Stop and start an existing container
docker kill	Hard stop (kill) a running container
docker rm (and rmi)	Delete a container or image
docker network	Work with docker networks
docker volume	Work with docker volumes
docker exec / docker attach	Interact with a running container
docker system prune	Delete all “unused” containers/volumes/images

Common docker run Options

Command	Function
--name	Give a friendly name to the container *Note double dashes*
-d	Run the container in the background (detached)
-i -t	Start the container interactive, and with tty support
--restart= always/on-failure/unless-stopped/no	Restart the container automatically when the condition is met (note this is not a cluster restart policy, only on a single host) *Note double dashes*
-v	Map a volume to a container
-p ip:port:port / -P	Publish ports externally (hostport:containerport) • –P means publish all ports defined in the dockerfile. • Using –p :80 means randomly assign a host port • Using –p 192.168.1.20:80:80 means expose as port 80 on the hosts ip address 192.168.1.20 (if the host had more than 1 IP)
-e	Pass an environment variable into the container

Dockerfile

A **Dockerfile** is a text document that contains all the commands a user could call on the command line to assemble an image.



Create images automatically using a build script: «Dockerfile»

Can be versioned in a version control system like Git or SVN, along with all dependencies

Docker Hub can have automatically created images based on dockerfiles on Github

Example 1

FROM tomcat:8

MAINTAINER Devops Team

RUN wget -O /usr/local/tomcat/webapps/demosampleapplication.war <WAR FILE URL>

CMD /usr/local/tomcat/bin/catalina.sh run

Example 2

FROM ubuntu

MAINTAINER XYZ

RUN apt-get update

RUN apt-get install -y nginx

COPY index.html /usr/share/nginx/html/

ENTRYPOINT [“/usr/sbin/nginx”,”-g”,”daemon off;”]

EXPOSE 80

Docker – Example 1

Checking container logs

- docker ps (see the container id)
- docker logs <containerid>

```
[root@localhost html]# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
                    NAMES
71f50830826c        nginx:latest        "nginx -g 'daemon of…"   6 minutes ago
, 0.0.0.0:443->443/tcp   mynginx
[root@localhost html]#
```

```
[root@localhost html]# docker logs 71f
2018/11/19 19:01:49 [error] 6#6: *1 directory index of "/usr/share/nginx/html/"
: localhost, request: "GET / HTTP/1.1", host: "192.168.1.71"
192.168.1.253 - - [19/Nov/2018:19:01:49 +0000] "GET / HTTP/1.1" 403 153 "-" "Moz
.0) Gecko/20100101 Firefox/63.0" "-"
2018/11/19 19:01:49 [error] 6#6: *1 open() "/usr/share/nginx/html/favicon.ico" f
t: 192.168.1.253, server: localhost, request: "GET /favicon.ico HTTP/1.1", host:
192.168.1.253 - - [19/Nov/2018:19:01:49 +0000] "GET /favicon.ico HTTP/1.1" 404 1
 x64; rv:63.0) Gecko/20100101 Firefox/63.0" "-"
192.168.1.253 - - [19/Nov/2018:19:05:46 +0000] "GET / HTTP/1.1" 200 15 "-" "Mozi
0) Gecko/20100101 Firefox/63.0" "-"
2018/11/19 19:05:46 [error] 6#6: *2 open() "/usr/share/nginx/html/favicon.ico" f
t: 192.168.1.253, server: localhost, request: "GET /favicon.ico HTTP/1.1", host:
192.168.1.253 - - [19/Nov/2018:19:05:46 +0000] "GET /favicon.ico HTTP/1.1" 404 1
 x64; rv:63.0) Gecko/20100101 Firefox/63.0" "-"
[root@localhost html]#
```

Docker – Example 2

Docker attach

- Get your container id again
- Docker attach <containerid>
  - Note you just get a flashing cursor – you are inside the container, however there is no shell..
  - Press ctrl c to exit
  - Now do a docker ps
  - What happened to your container ? It got killed as you send the kill command whilst attached to the container
- Do docker ps –a to see stopped containers
- Do a docker start <containerid> to restart

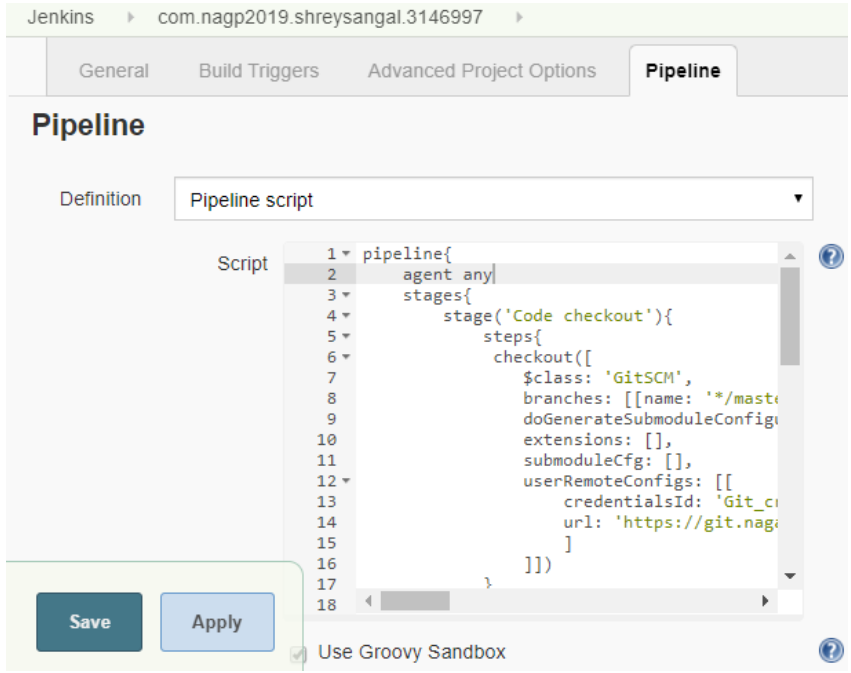
```
[root@localhost html]# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
                    NAMES
71f50830826c        nginx:latest        "nginx -g 'daemon of…"   6 minutes ago
, 0.0.0.0:443->443/tcp   mynginx
[root@localhost html]#
```

```
[root@localhost html]# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
                    NAMES
71f50830826c        nginx:latest        "nginx -g 'daemon of…"   11 minutes ago
, 0.0.0.0:443->443/tcp   mynginx
[root@localhost html]# docker attach 71f



^C[root@localhost html]# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
ES
[root@localhost html]#
```

Extending CI Pipeline with Docker

Existing CI Pipeline →Create Docker image → Push image to DTR → Pre-Container Check → Docker deployment

1 Extend pipeline script with docker stages, and click Save



a Create Docker Image

b Push Docker Image to DTR

c Pre-Container Check

d Docker Deployment

2 Click Build now, and check console logs

Jenkins › com.nagp2019.shreysangal.3146997 › #9

```
[Pipeline] stage
[Pipeline] { (Create Docker Image)
[Pipeline] sh
Warning: was asked to run an empty script
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Push Docker Image to DTR)
[Pipeline] sh
Warning: was asked to run an empty script
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Pre-Container Check)
[Pipeline] sh
Warning: was asked to run an empty script
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Docker Deployment)
[Pipeline] sh
Warning: was asked to run an empty script
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```


QUESTIONS