DOCKERS

Sumario

Some important concepts

Images

Images are read only templates used to create containers.

Images are created with the docker build command, either by us or by other docker users.

Images are stored in a Docker registry, such as Docker Hub

Containers

If an image is a class, then a container is an instance of a class. A runtime object.

Containers are created from images.

Registries and Repositories

A registry is where we store our images.

You can host your own registry, or you can use Docker's public registry which is called DockerHub.

Inside a registry, images are stored in repositories.

Docker repository is a collection of different docker images with the same name, that have different tags, each tag usually represents a different version of the image.

Docker Hub

hub.docker.com

We can take a look. e.g. Python repository.

We are encouraged to use the "official" repositories.

Create a Docker container

We open a command terminal.

Docker will first look for the image in the local box. If found, docker will use the local image. Otherwase, docker will download the image from remote docker registry.

To find out which images I have: **docker images**

To create a container using the image we specify and run it: **docker run**

docker run busybox:1.36 echo “hello world”

The previous command creates the container, runs it and executes the echo on it.

If we run it again it goes really fast because is already created locally.

The **-i flag** starts an interactive container.

The **-t flag** creates a pseudo-TTY that attaches stdin and stdout:

docker run -i -t busybox:1.36

exit to exit a container

The changes we do inside the container are not persistent. They disappear when we stop the container.

The **-d flag** is used to run the container in the background:

docker run -d busybox:1.36

To give the container a **name**:

docker run --name hello busybox:1.36

It returns the long container id.

To list all the dockers containers that are running: **docker ps**

To list all the containers including those that I have previously run: **docker ps -a**

To display the low level information about a container or image: **docker inspect container_id**

Docker Port Mapping and Doker Logs

To **map a port** in the host machine to the remote machine *host_port:container_port* :

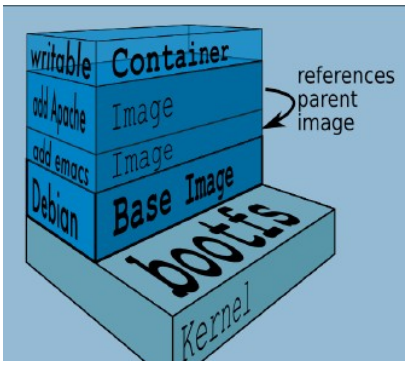
docker run -it -d -p 8888:8080 tomcat:9.0

We open in a web browser: `http://localhost:8888` and we find a 404 since the are no webapps loaded by default.

To see the **container logs** we type: *docker logs container_id*

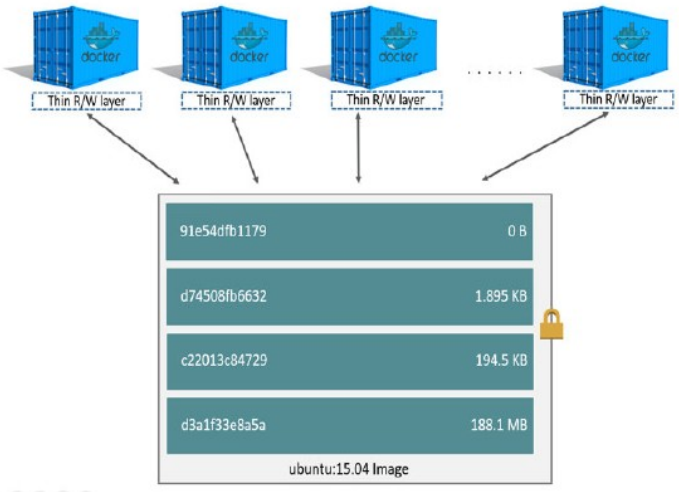
Working with Docker Images

Docker Image Layers



All changes made into the running containers will be written into the writable layer.

When the container is deleted, the writeble layer is also deleted, but the underlying image remains unchanged.



Multiple containers can access to the same underlying image and yet have their own data state.

Build docker images

There are two ways to build a docker image:

1. Commit changes made in a Docker container.
2. Write a Dockerfile.

Commit changes made in a Docker Container

Steps (example):

1. Spin up a container from a base image.
2. Install git package in the container.
3. Commit changes made in the container.

    docker run -it debian:latest

    inside de container: apt-get update && apt-get install -y git (-y to automatically say yes to prompts)

Docker commit command would save the changes we made to the Docker container's file system to a new image:

    docker commit actual_container_ID new_image_repository_name:tag

    docker ps -a → to get the container id

    docker commit 434123fafdsa my_gitHub_name/debian:1.00

    docker images → to see our new image

Our image is a bit bigger that the previous one because a have created a new layer on top of it.

We can spin up a container based on this new image and check that git is already installed on it:

    docker run -it my_gitHub_name/debian:1.00

Build Docker images by Writing Dockerfile

- A Dockerfile is a text document that contains all the instructions users provide to assemble an image.
- Each instruction will create a new image layer to the image.
- Instructions specify what to do when building the image.

We create a file called: **Dockerfile**

FROM base_image

RUN → to run some commands on the base image

FROM debian:latest

RUN apt-get update

RUN apt-get install -y git

RUN apt-get install -y vim

The command **docker build** will build the image based on the instructions given on the Dockerfile.

docker build -t my_gitHub_name/debian build_context_path

Docker Build Context

- Docker build command takes the path to the build context as an argument.
- When build starts, docker client would pack all the files in the build context into a tarball then transfer the tarball file to the daemon.
- By default, docker would search for the Dockerfile in the build context path.

d*ocker build -t github_name/debian .*

Dockerfile in depth

Chain RUN Instructions

- Each RUN command will execute the command on the top writable layer of the container, then commit the container as a new image.
- The new image is used for the next step in the Dockerfile. So each RUNinstruction will create a new image layer.
- It is recommended to chain the RUN instructions in the Dockerfile to reduce the number of image layers it creates.
- It is recommended to sort multi-line arguments alphanumerically. This will help you avoid duplication of packages and make the list much easier to update.

CMD Instructions

- CMD instruction specifies what command you want to run when the container starts up.
- If we don't specify CMD instruction in the Dockerfile, Docker will use the default command defined in the base image.
- The CMD instruction doesn’t run when building the image, it only runs when the container starts up.
- You can specify the command in either exec form which is preferred or in shell form.

Docker Cache

- Each time Docker executes an instruction it builds a new image layer.
- The next time, if the instruction doesn't change, Docker will simply reuse the existing layer.

```
FROM debian:latest
RUN apt-get update && apt-get install -y \
    git \
    python \
    vim
CMD ["echo", "Hello world"]
```

COPY Instruction The COPY instruction copies new files or directories from build context and adds them to the file system of the container.

```
FROM debian:latest
RUN apt-get update && apt-get install -y \
    git \
    vim
 COPY abc.txt /tmp/abc.txt
```

ADD Instruction

ADD instruction can not only copy files but also allow you to download a file from internet and copy to the container.

ADD instruction also has the ability to automatically unpack compressed files.

The rule is: use COPY for the sake of transparency, unless you’re absolutely sure you need ADD,

USER Instruction

If we don’t specify a user in the Dockerfile all the commands are executed with root privileges.

To add a new user: RUN useradd -ms /bin/bash admin

To execute the following commands with the new user: USER admin

WORKDIR Instruction

This instruction sets the working directory for any RUN command.

In the following example we are using /app as our working directory. Then we copy the app directory to the container.

```
FROM python:3.5
RUN pip install Flask==0.11.1
RUN useradd -ms /bin/bash admin
USER admin
WORKDIR /app
COPY app /app
CMD ["python", "app.py"]
```

Push Docker Images to Docker Hub

The easiest way to make your images available is to use the Docker Hub which provides three repositories for public images.

First we need to create a Docker Hub account and go to Docker Hub. Don’t use Google Authentication.

We need to link the image with the Dockerhub account is to rename the image to somerhing like docker_hub_id/repository_name: docker tag image_id repository_name:tag

docker tag 42314234213 sanclementetutorial/debian:1.01

We need to login to our Dockerhub account:

docker login –username=my_user_name

We upload our repository:

docker push sanclementetutorial/debian:1.01

Create Containerized Web Applications

Containerize a Simple Hello World Web Application

The application is written with Python in Flask web framework, which is a light web framework. It is available on: https://github.com/jleetutorial/dockerapp

git clone -b v0.1 https://github.com/jleetutorial/dockerapp.git

docker build -t dockerapp:v0.1 . → Create the image

docker run -d -p 5000:5000 image_id (Python listens on port 5000 by default, with this mapping we will be able to access it from our local machine too) → Run the image

We open a web browser and type: localhost:5000 on the URL to execute the program.

Docker exec allows you to run a command in a running container.

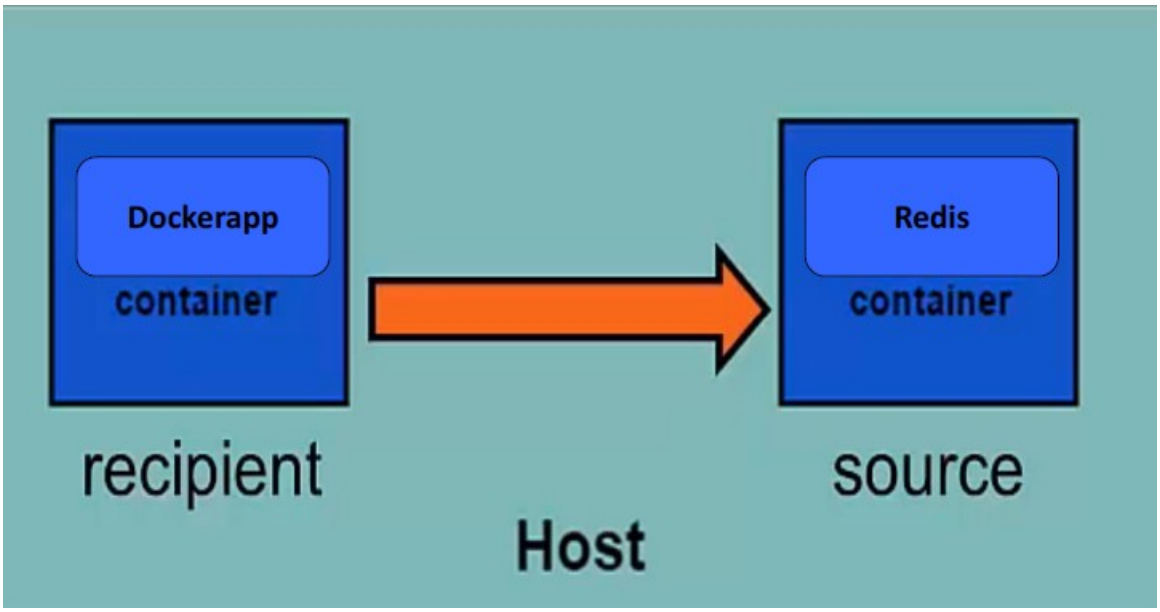
docker ps to see the container id.

docker exec -it container_id bash

executing "ps axu" we can check that the user of the running apps is admin as specified in the Dockefile.

docker stop container_id to stop the running container.

Docker container Links



We create a container for the Redis database.

docker run -d --name redis redis:3.2.0 → We build our app image.

We create our apps container using the "--link" option and **specifying link to the Redis containe**r so that the app container can connect to the Redis container using the container name "Redis".

docker run -d -p 5000:5000 --link redis dockerapp:v0.3

NOTE: Within the **app code we have used the Redis container name as the host name**.

docker inspect container_id |grep IP

Benefits of Docker Container Links

• The main use for docker container links is when we build an application with a microservice architecture, we are able to run many independent components in different containers.

• Docker creates a secure tunnel between the containers that doesn't need to expose any ports externally on the container (we didn't use the -p in the docker run command).

Docker Compose

Manual linking containers and configuring services become impractical when the number of containers grows.

Docker compose is a tool for defining and running multiple container docker applications.

With docker compose we can define all the containers in a single file called **docker-compose.yml**.

We can run a single command to start up all the containers.

To check that it is installed: docker-compose version

Example:

We indicate version 3 because that is the version of docker-compose that we are using.

In our example we have two **services**: the dockerapp and Redis.

The **build** instruction defines the path to the Dockerfile which will be used to build the Docker app image.

The **ports** instruction defines which ports on the container to expose to the external network. Mapping ports are defined as in the run command, in host_port:container_port format.

```
version: '3'
services:
    dockerapp:
        build: .
        ports:
            - "5000:5000"
        depends_on:
            - redis
    redis:
        image: redis:3.2.0
```

Next we need to add a **depend_on** section to express the dependency between the redis service and the dockerapp service. Our dockerapp container is a **client of the redis container** we need to start the redis container before the dockerapp container. Docker-compose will start services in the dependency order defined in this section.

In the redis section we only need to specify the **image** instruction, which defines which image to run a container from.

As you see, **we can either run a container from an existing image or we can build an image from a dockerfile**.

We don't need to link both containers becanse docker-compose supports docker network feature, which allows containers to be discovered by their name automatically and any service can reach any other service by the service name.

docker-compose up -d (-d option to run it in the background)

Docker Compose Commands

- docker compose up → starts up all the containers.
- docker compose ps → checks the status of the containers managed by docker compose.
- docker compose logs -f → outputs colored and aggregated logs for the compose-managed containers. The option -f for format. It is posible to indicate a specific container name.
- docker compose stop → stops all the running containers without removing them.
- docker compose rm → removes all the containers.
- docker compose build → rebuilds all the images. If we change the dockerfile and we execute docker compose up it will not use the new dockerfile by default. If an image was created previously it will find it and use it. To tell docker compose to create a new image we must execute docker-compose build.
- docker compose run dockerapp python test.py→ It will run a one time command against a service inside a container (we are overrinding the default command specified in the Dockerfile)

Docker Networking

When docker is installed in the host machine a bridge network interface “Docker 0” is provisioned on the host Each container connects to the bridge network through the container network interface. Containers can connect to each other and to the internet through this **bridge network interface**.

There are four docker network types:

- Closed Network / None Network
- Bridge Network
- Host Network
- Overlay Network

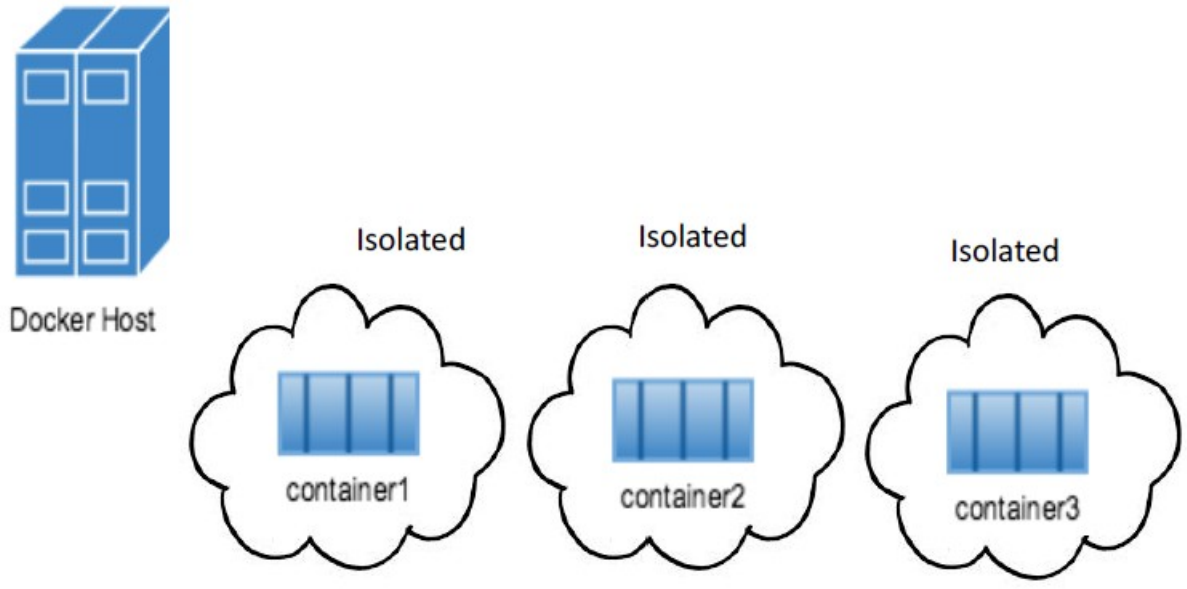
To check out the existing docker networks on our machine: **docker network ls**

To check the details of a specific network: **docker inspect network_name**

To create a docker network: **docker network create –driver bridge network_name** (we specify the type of driver to use and then the name of the new network)

Docker has a feature which allows us to connect a container to another network. Once connected, the container can connect to other containers in that network. **docker network connect bridge container_1** connects container_1 to the bridge network.

None Network

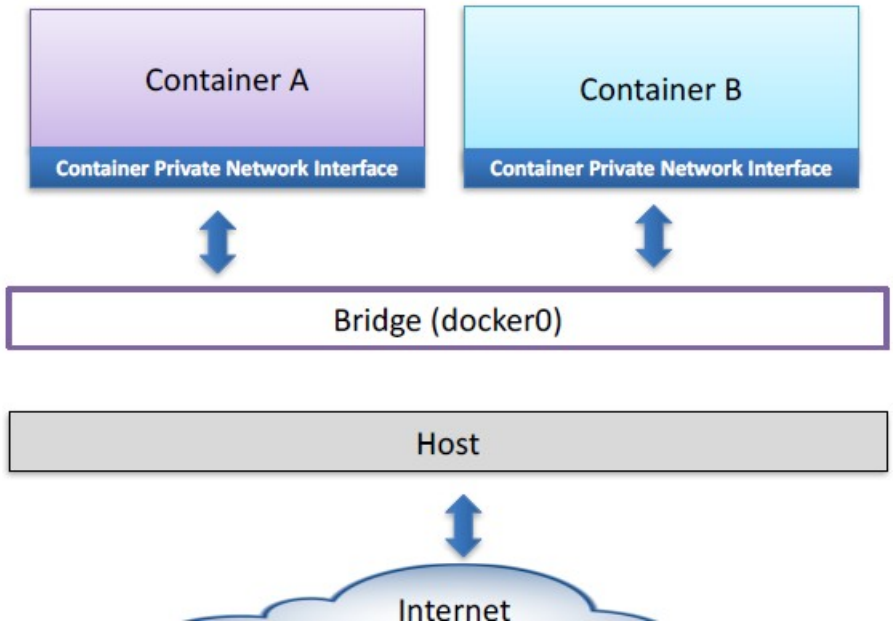


docker run -d --net none imaxe

Suites well where the container require the maximum level of network security and network access is not necessary.

Bridge Network

This is the default network in the containers.



All containers in the same bridge network can communicate with each other.

Containers from a bridge network cannot access containers from another bridge network.

Define Container Networks with Docker Compose

```
version: '3'
services:
    dockerapp:
        build: .
        ports:
            - "5000:5000"
        depends_on:
            - redis
        networks:
            - my_net
    redis:
        image: redis:3.2.0
        networks:
            - my_net
networks:
    my_net:
        driver: bridge
```

Create a Continuous Integration Pipeline

Incorporating Unit Tests into Docker Images

Pros:

• A single image is used through development, testing and production, which greatly ensures the reliability of our tests.

Cons:

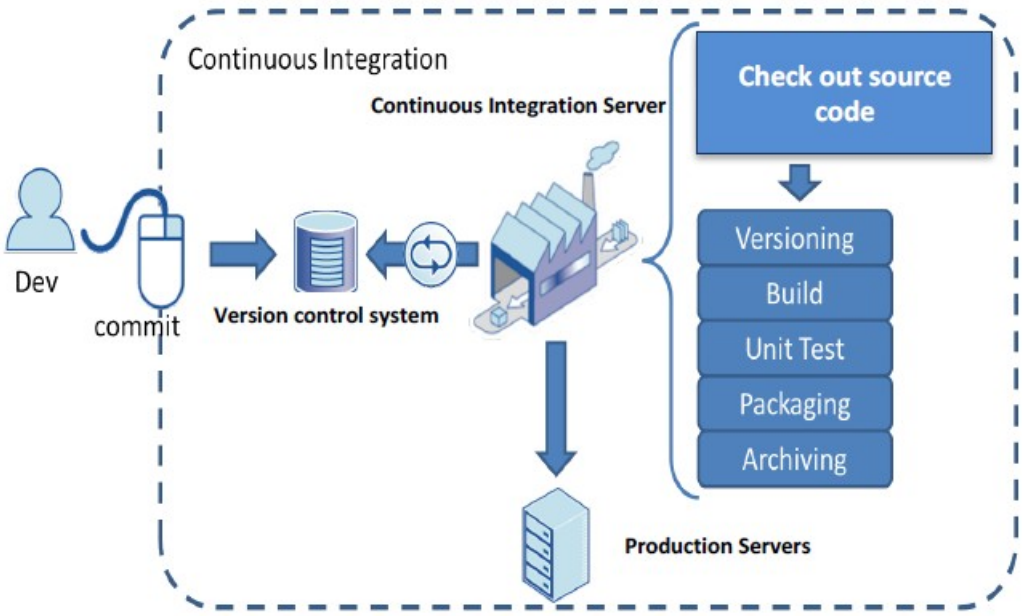
• It increases the size of the image.

Introduction to Continous Integration

Continuous integration is a software engineering practice in which isolated changes are immediately tested and reported when they are added to a larger code base.

The goal of Continuous integration is to provide rapid feedback so that if a defect is introduced into the code base, it can be identified and corrected as soon as possible.

A typical CI pipeline without Docker:



Volumes

Volumes are a mechanism for storing data outside containers.

Volumes are mounted to filesystem paths in your containers. When containers write to a path beneath a volume mount point, the changes will be applied to the volume instead of the container's writable image layer. The written data will still be available if the container stops – as the volume's stored separately on your host, it can be remounted to another container or accessed directly using manual tools.

You can start a container with a volume by setting **the -v flag** when you call docker run.

The following command starts a new Ubuntu 22.04 container and attaches your terminal to it (-it), ready to run demonstration commands in the following steps. A volume called demo_volume is mounted to /data inside the container. Run the command now:

```
docker run -it -v demo_volume:/data ubuntu:22.04
```

This example above demonstrated how Docker automatically creates volumes when you reference a new name for the first time. You can manually create volumes ahead of time with the docker volume create command:

```
docker volume create app_data
```

```
docker volume ls
```

```
docker volume rm demo_name -f
```

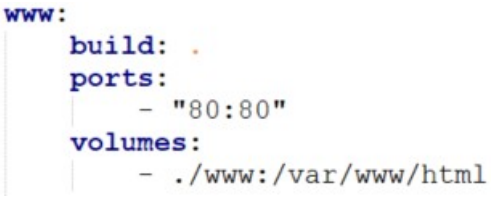
To clean up all unused volumes: docker volume prune

Using volumes with Docker Compose

Volumes can also be defined and used in Docker Compose. In your docker-compose.yml file, add a top-level volumes field that lists the volumes to create, then mount your volumes into your containers within the services section:

```yaml
version: '3.8'
services:
  web:
    image: nginx
    volumes:
      - web_data:/var/www/html
  web-test:
    image: nginx
    volumes:
      - web_data:/var/www/html # Web and web test share the web_data volume
  db:
    image: mysql
    volumes:
      - db_data:/var/lib/mysql
volumes:
  web_data:
  db_data:
    driver: local # Define the driver and options under the volume name
    driver_opts:
      type: none
      device: /data/db_data
      o: bind
```

The following example links the ./www directory to the /var/www/html directory on the docker container.



- host_directory:container_directory