Docker

Table of Contents

[1 What is Docker? What is a Container 3](#_Toc78455966)

[1.1 What is a Container (abstract view)? 3](#_Toc78455967)

[1.2 Where do containers live? 3](#_Toc78455968)

[1.3 How do containers improve the development process? 3](#_Toc78455969)

[1.4 The traditional deployment process 3](#_Toc78455970)

[1.5 What is a Container (technically)? 4](#_Toc78455971)

[2 Docker vs Virtual Machine 6](#_Toc78455972)

[3 Overview of the commands 7](#_Toc78455973)

[4 Docker Images 9](#_Toc78455974)

[4.1 Dockerfile (Example) 9](#_Toc78455975)

[4.2 Layered Architecture 9](#_Toc78455976)

[4.2.1 Caching Layers 9](#_Toc78455977)

[4.2.2 Image and Container Layers (Read and Write Layers) 10](#_Toc78455978)

[4.2.3 COPY-ON-WRITE 10](#_Toc78455979)

[5 Dockerfile In-Depth 11](#_Toc78455980)

[5.1 CMD vs ENTRYPOINT 11](#_Toc78455981)

[6 Environment Variables 11](#_Toc78455982)

[7 Data Persistence with Docker 12](#_Toc78455983)

[7.1 Are the changes we made erased when we restart (or stop and start) a container (not with docker-compose)? 12](#_Toc78455984)

[7.2 Volumes 13](#_Toc78455985)

[7.2.1 3 Types of Mounts/Volumes 14](#_Toc78455986)

[7.2.2 Storage drivers 16](#_Toc78455987)

[8 Docker Network 17](#_Toc78455988)

[8.1 Bridge 17](#_Toc78455989)

[8.1.1 User defined networks 17](#_Toc78455990)

[8.1.2 Inspect Network 17](#_Toc78455991)

[8.1.3 Embedded DNS 17](#_Toc78455992)

[8.2 None 18](#_Toc78455993)

[8.3 Host 18](#_Toc78455994)

[9 Docker Compose 19](#_Toc78455995)

[10 Docker Registry 19](#_Toc78455996)

[11 Container Orchestration 19](#_Toc78455997)

[12 TODO 20](#_Toc78455998)

[12.1 Avoid the latest Tag 20](#_Toc78455999)

[12.2 What is a Docker Registry? 20](#_Toc78456000)

[12.3 Difference between Docker registry and repository? 20](#_Toc78456001)

[12.4 TODO Docker Named volume with a specific folder mounted to it? 21](#_Toc78456002)

[12.5 TODO Docker Builder Pattern 21](#_Toc78456003)

[12.6 TODO Docker Multistage Builds 21](#_Toc78456004)

[12.7 TODO Docker Full Tutorial - https://www.youtube.com/watch?v=zJ6WbK9zFpI 21](#_Toc78456005)

[12.7.1 TODO - Best practices when creating layers 22](#_Toc78456006)

[12.8 Docker Hub and Docker Registries 22](#_Toc78456007)

**Tutorials used**:

* <https://www.youtube.com/watch?v=zJ6WbK9zFpI>
* <https://docs.docker.com>
* <https://www.youtube.com/watch?v=3c-iBn73dDE>

1. What is Docker? What is a Container
   1. What is a Container (abstract view)?

* A way to **package** an application **with all the necessary dependencies** and **configuration** inside.
* That package is a portable artifact, easily shared and moved around.
* This portability and the packaged, isolated environment make the development and deployment process **more efficient**
  1. Where do containers live?

Containers live in a Container (image) Repository, a special type of storage for containers. Many companies have their own private container repositories. There is also a public repository for Docker containers (DockerHub) where you can browse and find a ton of different application containers. In DockerHub there are many official images and unofficial images which different companies or developers upload.

* 1. How do containers improve the development process?

Before working with containers, developers had to install all the necessary programs to their machine directly. For instance, Redis, MySQL, Java and so on. Every developer had to **install** and **configure** them **manually** and be sure that everything is running as they intended. Depending on the **operating system** they use, the installation/configuration would look different. The same was also true for the production environment. If your application consisted of 10 services, and you have 5 servers in your server-cluster, you need to manually install every 10 of those services to each server. That made the whole process cumbersome.

* Containers solve some of these problems. With containers you do not have to install any of the services directly to on your OS. Because the container has its own isolated operating system. Everything, all services and their configurations are installed on that isolated OS, inside the container.
* With one download command you can install and start that container on any machine, independent of the host operating system and it is guaranteed that the container will always have the same services and configuration and behave in the same way. This allows many flexible scenarios where you can even have many of the same service with multiple different versions, running on the same host.
  1. The traditional deployment process

1. The development team would produce the artifact (for instance a jar file) together with a set of instructions on how to install and configure those artifacts on the server.
2. The dev team would give all those artifacts and instructions to the **operations team (ops)** and the ops team would handle the installation of those artifacts on the server.

**Problem**: again, installing everything on the server OS directly, which could lead to problems with dependency version conflicts etc.

Since the instructions are in a textual form, there can be misunderstanding, or the dev team could forget to put some critical points in the list. This can lead to a back and forth between the ops and the dev team until the app is properly installed.

With containers, this process is simplified. No environmental configuration needed on the server, just the docker runtime needs to be installed on the server and the rest is just a command to pull and start the container.

* 1. What is a Container (technically)?

Technically, a container is a running process of an image and that image is made up of multiple images/layers. Layers of images are stacked on top of each other and at the base of most of the containers we would have a **Linux based operating system image**, for example **Alpine** (which is a lightweight Linux OS). It is important for those base images to be small to make sure the container stays as small as possible, that’s why most of them are Alpine.

We can now search for an image on DockerHub. Since it is a public image repository, anyone can search and download any images that they want, without needing an account or permissions.

If we open for example the page for MySQL, we can see different **TAGs**. These are the versions of the application. There is the **latest** tag, which is always the most up to date version. So, we could do:

* docker pull mysql/mysql-server (same as docker pull mysql/mysql-server:**latest**)
* docker pull mysql/mysql-server**:5.7** (pulls the image with MySQL version 5.7)

Graphical user interface, text, application

Description automatically generated

Running this command, docker first checks if it can find the image on local, meaning it checks if the image is already downloaded. If not, then it will connect to DockerHub to download the image.

Docker also will download other dependencies, seen here with these hash values. Which we see here are all the **different layers** which we talked about, downloading separately from DockerHub. We can see here that the mysql-server version 8.0 has **7 layers**.

Let’s look at the layers of the MySQL container:

* skopeo --override-os linux copy docker://mysql/mysql-server:latest dir:./deleteme

Text

Description automatically generatedThis command will download each file into a folder called deleteme. Note that the first **12 characters** are the same when downloading the layers. We get:

* One file for each layer of the image
* Manifest file
* Version file

We can also see that the file name of each layer is actually the **sha256** of the contents of the layer.

Graphical user interface, text, application

Description automatically generatedDocker layers are stored using a **Content Addressable Storage** scheme. That means that the hash of the contents of the layer is how that layer is referred to and stored in the file system.

The **manifest file** lists each of layer digests for its eight layers, as well as the size and the format of the file. The ordering here is important as docker images use a **union file system**.

The **manifest** has also a **config** layer. The config is a json document and contains metadata about image creation. And again, its filename is its sha256 hash.

Text

Description automatically generated with low confidenceNow we are only missing one piece of the puzzle. **Where does the image digest that was returned in our original “docker run” come from?**

* The answer is the sha256 of the manifest

**What are layers?**

**TLDR;** Layers of a Docker image are essentially just files generated from running some command. You can view the contents of each layer on the Docker host at /var/lib/docker/aufs/diff. Layers are neat because they can be re-used by multiple images saving disk space and reducing time to build images while maintaining their integrity.

Docker containers are building blocks for applications. Each container is an image with a readable/writeable layer on top of a bunch of read-only layers.

These layers (also called intermediate images) are generated when the commands in the Dockerfile are executed during the Docker image build.

For example, here is a Dockerfile for creating a **node.js web app** image. It shows the commands that are executed to create the image.

Shown below, when Docker builds the container from the Dockerfile (right), each step corresponds to a command run in the Dockerfile. And each layer is made up of the file generated from running that command. Along with each step, the layer created is listed represented by its sha256 value of the generated file. For example, **the first 12 characters of the layer hash** for step 1 is 530c750a346e (NOT a random UUID). Containers get UUIDs (Container ID), Images get the sha256 hash (Image ID)

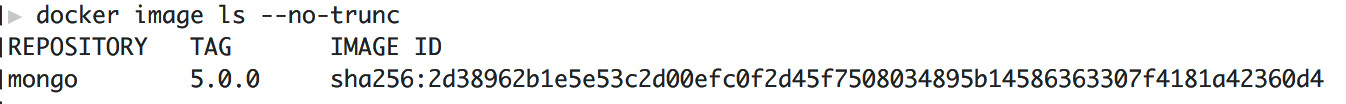
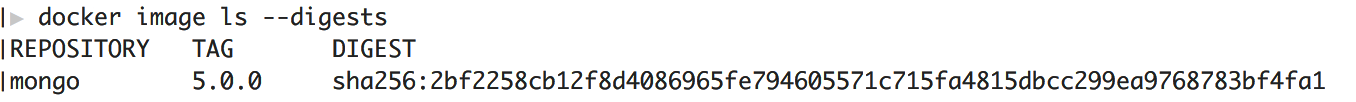
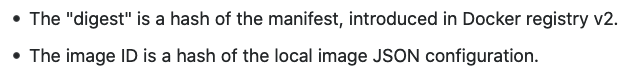
A close-up of a document

Description automatically generated with medium confidenceOnce the image is built, you can view all the layers that make up the image with the docker history command. The “Image” column (i.e intermediate image or layer) shows the **first 12 characters of the hash values** of that layer.

The container has a **writeable layer** (at the top) that stacks on top of the image layers. This writeable layer allows you to “make changes” to the container since the lower layers in the image are **read-only**.

Using a **union filesystem** allows each layer that is created to be reused by an unlimited number of images. This saves a lot of disk space and allows images to be built faster since it is just re-using an existing layer. For instance, if you already downloaded all the (unchanged) layers for a new version of MySQL, then you won’t need to download them again. Only the changed layers will be downloaded. Additionally, the read/write top layer gives the appearance that you can modify the image, but the read-only layers below actually maintain their integrity of the container by isolating the contents of the filesystem.

To dig down into each layer of the image and view its contents you need to view the layers on the Docker host at: /var/lib/docker/aufs/diff. At the time of this writing, if running Docker on OSX, the Docker host is actually a linux virtual machine called **docker machine**. On OSX you can ssh into the docker machine to view the **aufs** directory

Text

Description automatically generated**What's the difference between a Docker image's Image ID and its Digest?**

1. Docker vs Virtual Machine

Graphical user interface, text, application, chat or text message

Description automatically generatedLet’s first look how an OS is layered. The OS has 2 layers. The **Application Layer** and the **OS Kernel**.

The kernel communicates with the hardware directly and the applications talk only to the kernel, not directly to the hardware. For instance, all the different Linux distros have the same kernel, which is called the Linux Kernel and on top of it they have distro specific applications (and many other GNU Utilities, therefore GNU/Linux).

Graphical user interface

Description automatically generatedDocker and VMs are both virtualization tools. **Docker**, basically containers, supports **OS virtualization** i.e. your application feels that it has a complete instance of an OS whereas **VM** supports **hardware virtualization**. You feel like it is a physical machine in which you can boot any OS. In Docker, the containers running, share the host OS kernel, whereas in VMs they have their own OS files.

Docker virtualizes the application layer while using the host os kernel. A VM virtualizes both the application layer and the OS kernel layer, which means it does not use the host os kernel but have its own os kernel running on top of it

* Unlike VM’s, containers are not meant to host an OS. Containers are meant to run a specific task or process. Once the task/process is complete, the container exits. The container only lives as long as the process inside it is alive. If the web-server inside the container is stopped or crushes, the container exits.

This results in some differences:

1. **Size**: Docker images/containers are much smaller than VMs since they only contain the **bare minimum** to run the application inside the container. A container uses as much of the OS kernel and components as possible.   
   Docker images are usually a couple MBs large, VMs can be a couple GBs
2. **Speed**: Docker images run much faster since they don’t have to boot up an OS and everything which comes with it.
3. **Compatibility**: You can run a VM instance of ANY OS on any OS host. But you **CAN’T** do that with Docker.

Let’s say we have a Windows OS and you want to run a Linux based OS image on that Windows host. The problem is the Linux base OS won’t be compatible with the Windows kernel. The Window Kernel won’t have the same interface and components which the Linux based container expects, so it won’t be able to use the host OS.

This is actually the case for Windows versions < 10 and also for the older MacOS versions. For situations where the kernel is not compatible, the Docker team built a workaround called **Docker Toolbox**. Docker Toolbox is a legacy solution to bring Docker to systems which don't natively support Docker. This is achieved by starting a virtualized Linux instance (e.g.: inside VirtualBox) and have Docker run inside this machine.

**Wikipedia:** “***Windows Subsystem for Linux*** *(****WSL****) is a compatibility layer for running Linux binary executables (in ELF format) natively on Windows 10 and Windows Server 2019.*

*In* ***May 2019****, WSL 2 was announced, introducing important changes such as a real Linux kernel, through a subset of Hyper-V features. Since June 2019, WSL 2 is available to Windows 10 customers through the Windows Insider program, including the Home edition*.”

**Docker Docs:** “***Windows Subsystem for Linux*** *(****WSL****)* ***2*** *introduces a significant architectural change as* ***it is a full Linux kernel*** *built by Microsoft, allowing Linux containers to run natively without emulation. With Docker Desktop running on WSL 2, users can leverage Linux workspaces and avoid having to maintain both Linux and Windows build scripts. In addition, WSL 2 provides improvements to file system sharing, boot time, and allows access to some cool new features for Docker Desktop users*.”

1. Overview of the commands

**Command** **Description**

* docker **attach** Attach local standard input, output, and error streams to a running container
* docker **build** Build an image from a Dockerfile
* docker **builder** Manage builds
* docker **checkpoint** Manage checkpoints
* docker **commit** Create a new image from a container’s changes
* docker **config** Manage Docker configs
* docker **container** Manage containers
* docker **context** Manage contexts
* docker **cp** Copy files/folders between a container and the local filesystem
* docker **create** Create a new container
* docker **diff** Inspect changes to files or directories on a container’s filesystem
* docker **events** Get real time events from the server
* docker **exec** Run a command in a running container
* docker **export** Export a container’s filesystem as a tar archive
* docker **history** Show the history of an image
* docker **image** Manage images
* docker **images** List images
* docker **import** Import the contents from a tarball to create a filesystem image
* docker **info** Display system-wide information
* docker **inspect** Return low-level information on Docker objects
* docker **kill** Kill one or more running containers
* docker **load** Load an image from a tar archive or STDIN
* docker **login** Log in to a Docker registry
* docker **logout** Log out from a Docker registry
* docker **logs** Fetch the logs of a container
* docker **manifest** Manage Docker image manifests and manifest lists
* docker **network** Manage networks
* docker **node** Manage Swarm nodes
* docker **pause** Pause all processes within one or more containers
* docker **plugin** Manage plugins
* docker **port** List port mappings or a specific mapping for the container
* docker **ps** List containers
* docker **pull** Pull an image or a repository from a registry
* docker **push** Push an image or a repository to a registry
* docker **rename** Rename a container
* docker **restart** Restart one or more containers
* docker **rm** Remove one or more containers
* docker **rmi** Remove one or more images
* docker **run** Run a command in a new container
* docker **save** Save one or more images to a tar archive (streamed to STDOUT by default)
* docker **search** Search the Docker Hub for images
* docker **secret** Manage Docker secrets
* docker **service** Manage services
* docker **stack** Manage Docker stacks
* docker **start** Start one or more stopped containers
* docker **stats** Display a live stream of container(s) resource usage statistics
* docker **stop** Stop one or more running containers
* docker **swarm** Manage Swarm
* docker **system** Manage Docker
* docker **tag** Create a tag TARGET\_IMAGE that refers to SOURCE\_IMAGE
* docker **top** Display the running processes of a container
* docker **trust** Manage trust on Docker images
* docker **unpause** Unpause all processes within one or more containers
* docker **update** Update configuration of one or more containers
* docker **version** Show the Docker version information
* docker **volume** Manage volumes
* docker **wait** Block until one or more containers stop, then print their exit codes

1. Docker Images

The basic idea is: We list all the steps of how we would manually setup the server and install our application:

1. OS – Ubuntu
2. Update apt repositories
3. Install all dependencies using apt
4. Install Python dependencies using pip
5. Copy our source code to the /opt folder
6. Run the web server using the flask command

Then we can write these steps in to a **Dockerfile**, which is a text file with a specific syntax. After that we can build an image from this Dockerfile. This will create an image locally. To push the image to the DockerHub, we can use the name of our DockerHub account and then the name of the image.

* **docker build** **Dockerfile** -t jckleiner/my-custom-app
* **docker push** jkleiner/my-custom-app
  1. Dockerfile (Example)
* **Graphical user interface

  Description automatically generated**Every docker image must base of on another image   
  (has to start with FROM)
* ENTRYPOINT allows us to specify a command that will be run when a container is started from this image.
  1. Layered Architecture

Docker builds the images in a layered architecture (as we saw in detail before).

* Each line of instruction creates a new layer.
* Each layer contains/stores only the changes from the previous layer. This is reflected in the layer size. You can see this information with the command: **docker history <image-name-or-id>**

1. Text

   Description automatically generatedBase Ubuntu Layer: **120 MB**
2. Changes in apt packages: **306 MB**
3. Changes in pip packages: **6.3 MB**
4. Copy the source code: **229 B**
5. Entrypoint run command: **0 B**
   * 1. Caching Layers

All the layers build are cached by docker. So, if during a build, a layer fails to build, you don’t have to build the previous layers again. This is helpful for example when you update the source code of your app. You won’t need to build all the unchanged steps again. Only the updated layers will be rebuilt.

* + 1. Image and Container Layers (Read and Write Layers)

When we run the build command, all the layers are created which is part of the image. These are called **Image Layers**.

* Image layers are read only. Once the image build is complete, you cannot modify the contents of these layers. You can only modify them by initiating a new build.
* When you run a container based of this image using the docker run command, docker creates a new container based on the image layers and creates a new writable layer on top of the image layers**.**
* **Graphical user interface, website

  Description automatically generated**The life of this **container layer** (writable layer) is only as long as the container is alive (not removed). When the containers is destroyed/removed, all of the changes which were done to this layer is also removed.
* This is trivial but still note that all containers which are based on this image are using the same **image layers**. But each container has its own **container layer**.

If we were to login to the newly created container and create a new temp.txt file, it would create that file in the **container layer**.

We just said that the files in the image layers are read only, which means you cannot edit any files there. In this example, we baked our python source code into the image, which means it is in an image layer and read only.

* + 1. COPY-ON-WRITE

After we running a container what if we wish to modify the source code to see the changed directly on the container? Does it mean that we cannot modify the source code inside a container?

Graphical user interface

Description automatically generated**NO**, we can still modify this file but when we save the modified file, docker automatically creates a copy of that file inside the container layer (read write layer) and we would be modifying that file. All future modifications would be done on that copy of the file on the read-write layer. This is called **copy on write mechanism**. So, the image stays the same all the time until you rebuild the image.

What happens when we get rid of the container?

* All the data that we stored on the container layer will also be deleted.
* To persist the changes done inside the container layer we can use volumes (see. 0)

Table

Description automatically generated**docker images**

The default docker images will show all top-level images saved locally, their repository and tags, and their size.

Docker images have intermediate layers that increase reusability, decrease disk usage, and speed up docker build by allowing each step to be cached. These intermediate layers are **not** shown by default.

* <none> tag????

1. Dockerfile In-Depth

* Unlike VM’s, containers are not meant to host an OS. Containers are meant to run a specific task or process. Once the task/process is complete, the container exits. The container only lives as long as the process inside it is alive. If the web-server inside the container is stopped or crushes, the container exits. Therefore we need a process which will keep running, so the container stays up.

W

* 1. RUN vs CMD vs ENTRYPOINT
* <https://www.youtube.com/watch?v=OYbEWUbmk90>
* <https://www.youtube.com/watch?v=U1P7bqVM7xM>

**TODO** when no CMD or ENTRYPOINT is defined, the container still runs, sometimes. This is because the base image you are using has its own CMD/ ENTRYPOINT. Probably because of that. Check again if that’s the case.

1. Environment Variables

TODO

1. Data Persistence with Docker
   1. Are the changes we made erased when we restart (or stop and start) a container (not with docker-compose)?

* **NO**. The data will still be there when the container starts again. A lot of people use the term “restarting a container” for a normal restart of a container but they also mean creating a new container from the same image. That’s why when they say “We would lose all the data if we would restart the container” they mean if they create a new container from the same image.

**Table

Description automatically generatedExample:**

docker run --name some-mongo -d mongo:5.0.0

docker exec -it 401a502a4143 sh

cd /data/db && touch WILL\_THIS\_PERSIST\_???

# (or docker stop and then docker start)  
docker restart some-mongo

* But **docker-compose** has a different behavior, therefore the data will be deleted after a **docker-compose down**.

**docker-compose up**

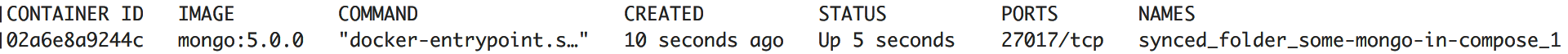
* Builds, (re)creates, starts, and attaches to containers for a service.

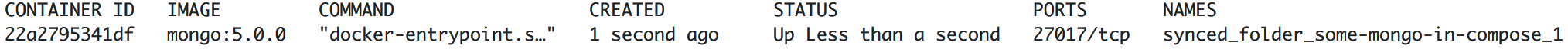
**docker-compose down**

* Stops containers and removes containers, networks, volumes, and images created by up.

Graphical user interface, text, application

Description automatically generated

**Example:** docker-compose up -d && docker ps

After that docker-compose down && docker-compose up -d && docker ps

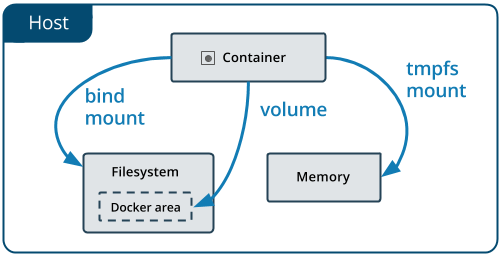
* Different Container IDs!

**TODO** – docker commit …

* 1. Volumes

Docker has two options for containers to store files in the host machine, so that the files are persisted even after the container stops:

* **volumes**
* **bind mounts**
* **tmpfs mount:** If you’re running Docker on **Linux**
* **named pipe:** If you’re running Docker on **Windows**

**Choose the right type of mount**

No matter which type of mount you choose to use, the data looks the same from within the container. It is exposed as either a directory or an individual file in the container’s filesystem.

An easy way to visualize the difference among volumes, bind mounts, and tmpfs mounts is to think about where the data lives on the Docker host.

Text

Description automatically generated

Text, letter

Description automatically generated

Text

Description automatically generated with medium confidence

Graphical user interface, text, application

Description automatically generated

Diagram

Description automatically generated

Volumes (Anonymous and Named Volumes) are the preferred mechanism for persisting data (over bind mounts) generated by and used by Docker containers.

With volumes we can basically “plug” (or mount) a file or folder from the host file system to the containers virtual file system. So, if the contents of the volume is changed on the Containers file system, it is then automatically replicated/changed on the host file system and vice versa.

This way we can persist data on the host file system and when new containers are created, which uses such volumes, the data will be accessible in those containers.

* + 1. 3 Types of Mounts/Volumes

1. **Bind Mounts (also called Host Volumes)**
2. **Anonymous Volumes**
3. **Named Volumes**

**1. Bind Mount (Host Mounts)**

these map a path from the host into the container with a **bind mount**. For instance:

* docker run -v **/home/mount/data:/var/lib/mysql/data**

The path for the data folder on the host file system is referenced and this will be “mounted” in the mysql folder on the Container file system.

Whatever exists on the host is what will be visible in the container, there's no merging of files or initialization from the image, and **uid/gid's** **do not** get any special mapping so you need to take care to allow the container uid/gid read and write access to this location (an exception is Docker for Mac with OSXFS).

* Mounted folders/file will be mounted with **root:root** inside the container.
* If the path on the host does not exist, docker will create an empty directory as root, and if it is a file, you can mount a single file into the container this way.

**2. Anonymous Volumes**

Creating a volume just by referencing the folder/file in the Container file system:

* docker run --name bla -v **/data/db2** -d mongo:5.0.0

these only have a path inside the container. They are in the form /path/in/container and docker will create a default named volume with a guid as the name. They share the behaviors of named volumes, storing files under /var/lib/docker/volumes, initializing with the contents of the image, except they have a randomly generated guid that gives you no indication of how or even if they are being used. You can mount the volume in another container and inspect the contents, or you can find the container using the volume by inspecting each container to find the guid. If you create a container with the --rm flag, anonymous volumes will also be deleted automatically.

**3. Named Volumes**

These have a name, instead of a host path as the source. They have the short syntax name:/path/in/container and in a **compose file**, you also need to define the named volume used in containers at the top level.

By default, these are also a bind mount, but to a docker specific directory under /var/lib/docker/volumes that should be considered internal. However, these defaults can be changed to allow things like NFS mounts, mounting disks, or even your own bind mounts to other locations. Named volumes also have a feature in docker, when they are new or empty and first used, docker copies the contents from the image into named volume before mounting it. This includes files, directories, uid/gid owners, and permissions. After that, they behave identical to a host volume, whatever is inside the volume overlays the image location.

* Named volumes uses the name of the application as the prefix and then it appends the name of the volume. Inside that folder is a \_data folder which has all the data.
* **MacOs**: Docker saves all the volumes inside a VM, so you need to first connect to that VM to see the volumes/

Graphical user interface, text, application

Description automatically generated**New Flag** --mount

Text

Description automatically generated with medium confidenceA picture containing company name

Description automatically generated

**Differences between** -v **and** --mount **behavior**

Because the -v and --volume flags have been a part of Docker for a long time, their behaviour cannot be changed. This means that there is one behaviour that is different between -v and --mount.

* If you use -v to bind-mount a file or directory that does not yet exist on the Docker host, -v creates the endpoint for you. It is always created as a directory.
* If you use --mount to bind-mount a file or directory that does not yet exist on the Docker host, Docker **does not** automatically create it for you, but generates an error.
  + 1. Storage drivers

Storage drivers are responsible for all the operations like maintaining the layered architecture, creating and managing a writable layer, moving files across the layers, copy and write etc.

Some of the common storage drivers are:

* AUFS
* ZFS
* BTRFS
* Device Mapper
* Overlay
* Overlay2

The selection of the storage driver depends on the underlying OS in use. For example, in Ubuntu, the default storage driver is AUFS but this driver is not available on other distros like Fedora or CentOS.

* Docker will choose the best storage driver automatically based on the OS.
* Different storage drivers also provide different performance and stability. So you may want to choose one which fits your application needs.

1. Docker Network

When you install docker, it creates 3 Networks automatically.

1. **Bridge:** docker run Ubuntu
2. **None:** docker run Ubuntu --network=none
3. **Host:** docker run Ubuntu --network=host
   1. Graphical user interface, application

      Description automatically generatedBridge

Bridge is the default network a container gets attached to. If you want to attach a container to any other network, you use the --network command line parameter.

The Bridge network is a private internal network created by docker on the host. All containers are attached to this network by default and they get an internal IP address, usually in the range of **172.17.x.x**. The containers can access each other using this internal IP address.

To access any of these containers from the outside world you can map the ports of these containers to the ports of the docker host.

* + 1. Diagram

       Description automatically generatedUser defined networks

But what if want to isolate only certain containers and give them a separate network?

For instance, the first 2 containers under the 172 network and the other 2 under a 182 network.

* By default, docker only creates one internal bridge network.

We can create our own internal bridge network with the command:

* **docker network create**   
   --driver bridge   
   --subnet 182.18.0.0/16   
   my-isolated-network
* See all the networks: **docker network ls**
  + 1. Text, letter

       Description automatically generatedInspect Network

To see the network settings and the IP address assigned to an existing container:

* Docker inspect <ID-or-NAME-of-the-container>
  + 1. Embedded DNS

Containers can reach each other by just using their names, ofc they need to be on the same network.

Diagram

Description automatically generatedFor example, we have a web server and a mysql db.

* We can either use 172.17.0.3 or mysql to reach the mysql container
* A screenshot of a computer

  Description automatically generated with low confidenceUsing the IP is not ideal because it is not guaranteed that the containers get the same IP address on restart.

Docker has a built in DNS Server that helps the containers resolve each other using their name.

* The built in DNS Server always run at **127.0.0.11**

Graphical user interface, application

Description automatically generated

* 1. None

The containers are not attached to any network and does not have any access to the external network or other containers. They are isolated.

Graphical user interface, text

Description automatically generated

* 1. Host

Another way to access the containers from the outside world is to associate the container to the host network. This takes out any network isolation between the docker host and the docker container. Meaning if you were to run a web server on port 5000 in a webapp container, it will be automatically accessible on the host network under the port 5000 without needing any port mapping. This would also mean, unlike before you cannot run multiple of these web server containers on the same port.

1. Docker Compose

TODO

1. Docker Registry

TODO

1. Container Orchestration

TODO

1. TODO
   1. Avoid the latest Tag

**TODO**

* 1. What is a Docker Registry?

Docker Registry is a place where you can store your own Docker images and distribute it to others. A Docker Registry is organized into Docker Repositories. In a Docker Repository, you can maintain specific versions of a Docker Image. DockerHub is the default Docker engine’s image store. DockerHub is public and anyone can pull the images which are stored in DockerHub. Other than DockerHub there are more paid public Docker registries available. Some of them are listed below.

**TODO**

* 1. Difference between Docker registry and repository?

TODO

**What happens when I have an image with the latest tag locally and a new version of that image comes out?**

A picture containing text, indoor, screenshot

Description automatically generated

https://stackoverflow.com/questions/26423515/how-to-automatically-update-your-docker-containers-if-base-images-are-updated

While building docker images from other images with the tag **latest**. For example: **FROM alpine:latest** It checks the docker cache (locally) if there is an image labeled **alpine:latest** without checking if the image on docker hub is newer than the one in the cache.

docker image build

--pull Always attempt to pull a newer version of the image

**Don't use only the "latest" tag** - The latest tag is just like the "SNAPSHOT" for Maven users. Tags are encouraged because of the layered filesytem nature of containers. You don't want to have surprises when you build your image some months later and figure out that your application can't run because a parent layer (FROM in Dockerfile) was replaced by a new version that it's not backward compatible or because a wrong "latest" version was retrieved from the build cache. The "latest" tag should also be avoided when deploying containers in production as you can't track what version of the image is running.

**docker run** (not **docker pull**) search first in your local registry on your machine. If there is the image with the tag latest, the search is satisfied and terminated. If the image with the given tag is not available in your local registry, then docker will search in a remote registry like docker hub or your own.

* TODO use local registry to check pull behaviour?
* <https://vsupalov.com/docker-latest-tag/>
  1. TODO Docker Named volume with a specific folder mounted to it?
  2. TODO Docker Builder Pattern
  3. TODO Docker Multistage Builds
  4. TODO Docker Full Tutorial - <https://www.youtube.com/watch?v=zJ6WbK9zFpI>
     1. TODO - Best practices when creating layers

**TODO**

* Stabile Layer ganz unten, fragile nach oben?
* Write about: if one of the first layers changes then docker build will build all the following layers as well. Therefore put those layers, which will change often to the end, so you won’t have to build all layers again each time.
* So wenig laters wie möglich erzeugen
  1. Docker Hub and Docker Registries

A **Shodan search** shows around **140+ docker registries** that are **publicly exposed** that don’t have any kind of authentication on it. Since the registry has a REST API, it is really simple for an attacker to see, delete and even upload their own images.

In all, the learning is to never expose your docker registry over the public. By default, it doesn’t have any authentication. Since docker registries don’t have a default authentication mechanism at least a basic auth could thwart some potential attacks. This can be mitigated by setting or enforcing basic auth over it and keeping the docker registry under VPN and preventing it from outside access. (<https://docs.docker.com/registry/deploying/#native-basic-auth>)

For any organization which is heavily using containers for running application and services and for organizations which are moving towards a containerized platform, should understand and evaluate the security risks around them majorly the misconfiguration present in docker. Proper security controls, audits, and reviewing configuration settings need to be carried out on a periodic basis.

Graphical user interface, text, application

Description automatically generatedTODO docker network

What is db\_data: {}?

* Upgrading applications (eg. with yum/apt-get upgrade) within containers is considered to be an anti-pattern. Application containers are supposed to be immutable, which shall guarantee reproducible behavior. Some official application images (mysql:5.6 in particular) are not even designed to self-update (apt-get upgrade won't work).

10 things to avoid when using containers:

<https://developers.redhat.com/blog/2016/02/24/10-things-to-avoid-in-docker-containers>

Questions

1. Whats the best way to save mysql volume data? Which parts are important, logs also? Where does mariadb save its data so we can easily migrate it?
2. This does mount the /var/lib/mysql inside the docker to a folder in the current directory.  
   So, the left side can be a name for a volume (which will be defined also in the docker-compose.yml) or a path to a directory? It looks just like a variable but then what is db\_data: {} ?
3. Can I push config changes to a running docker container?
4. Starting/Stopping a container vs rebuilding the container?