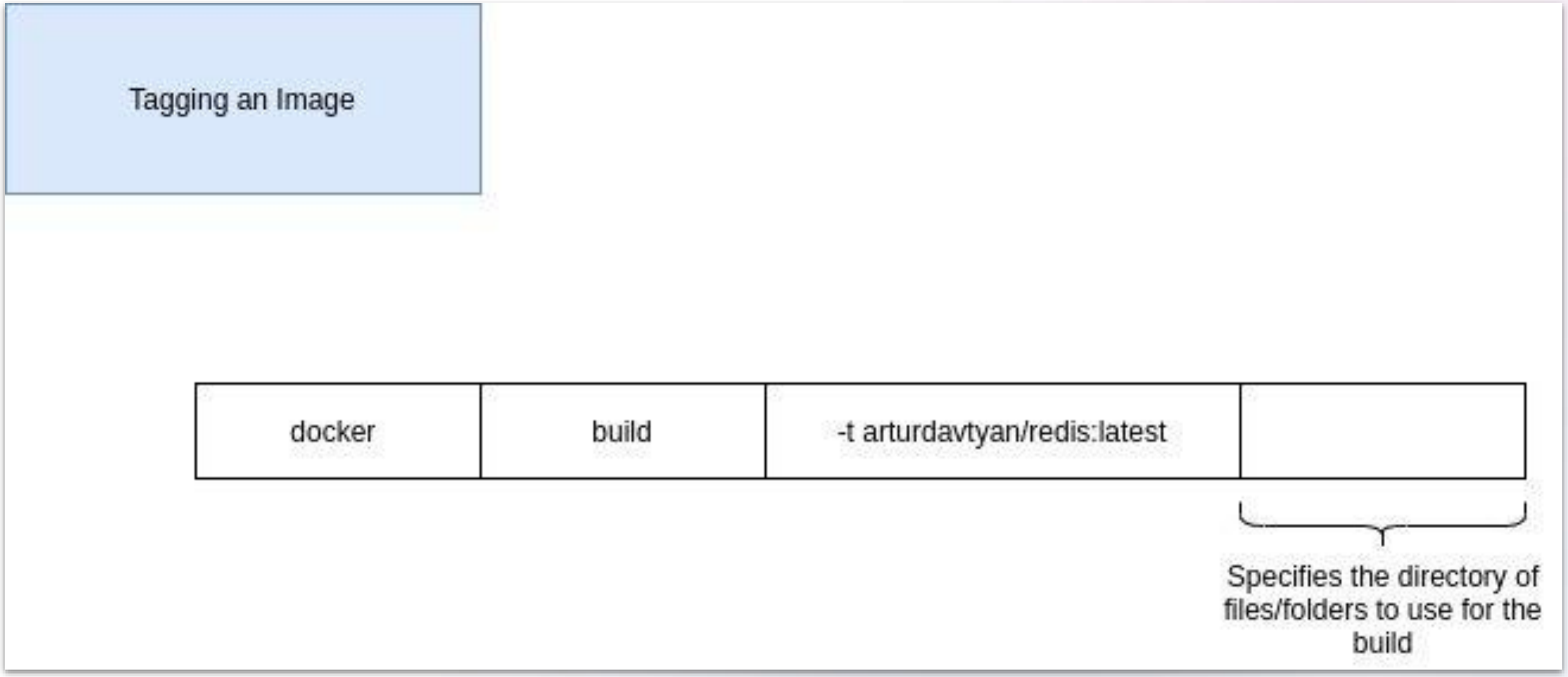
sourcemind

Building Custom Images from running container and Docker networking and volumes

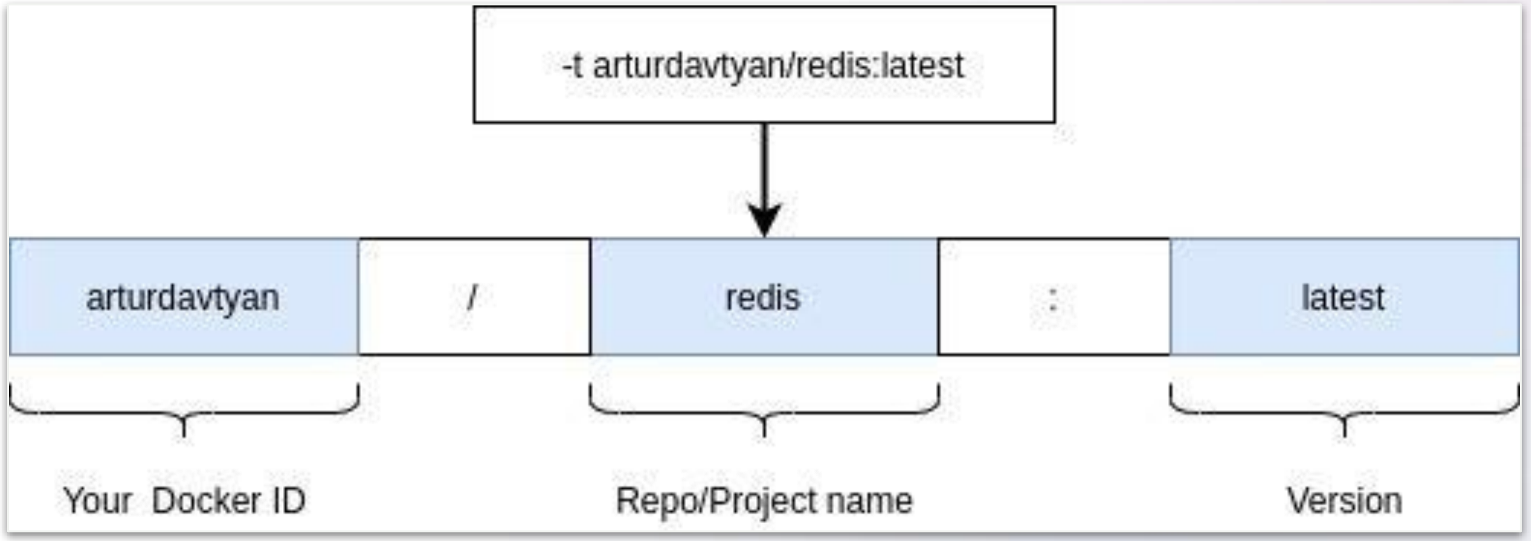
Mocktar ISSA
Full Stack Software Developper

sourcemind

Tagging an image

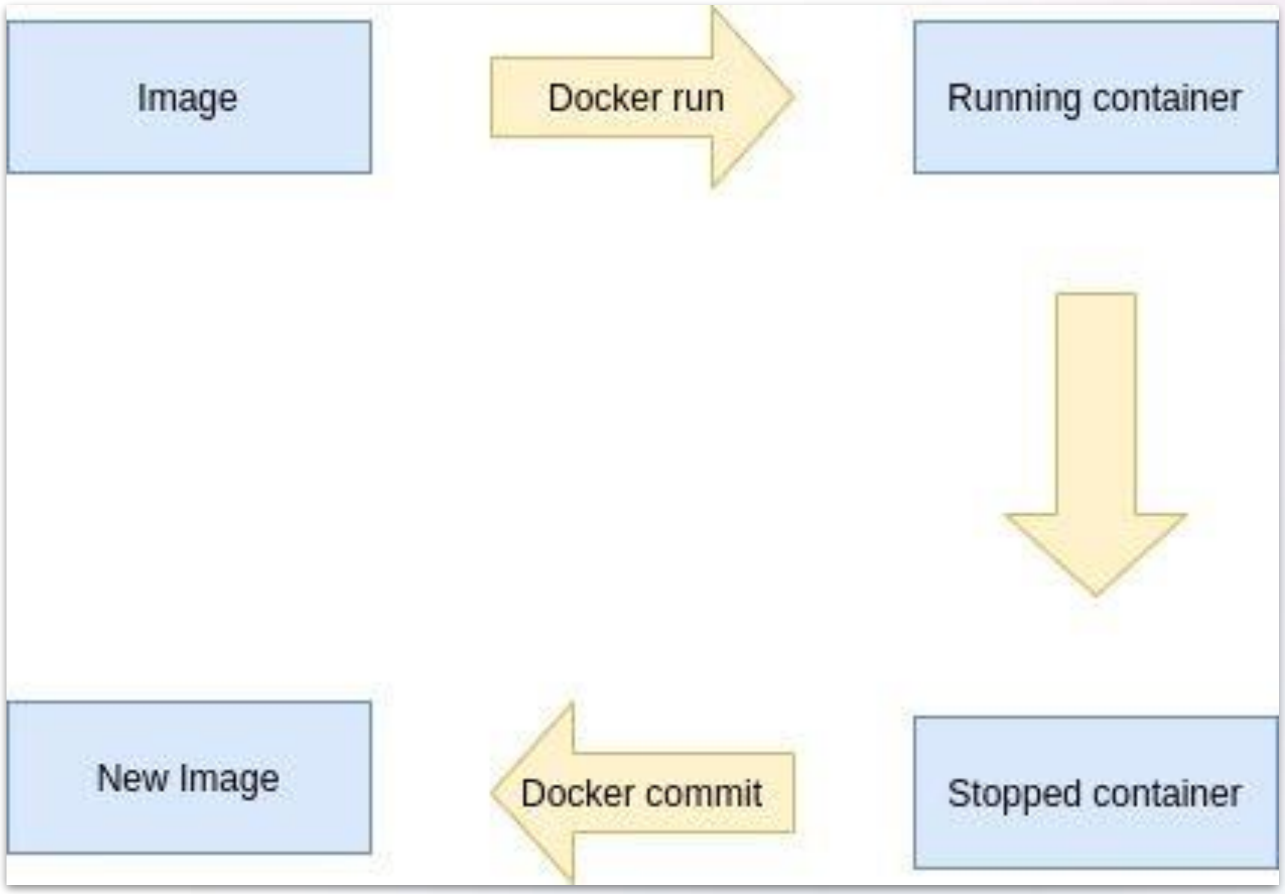

Tagging an Image
docker
build
-t arturdavtyan/redis:latest
Specifies the directory of files/folders to use for the build

Tagging an image



sourcemind

Manual Image Generation with Docker Commit



sourcemind

Manual Image Generation with Docker Commit

Practical Work

Approximate Duration: 15 minutes

Objectives:

`Create custom image from container`

sourcemind

Docker Hub

sourcemind

Docker Hub Quickstart

Docker Hub is a service provided by Docker for finding and sharing container images with your team. It is the world's largest repository of container images with an array of content sources including container community developers, open source projects and independent software vendors (ISV) building and distributing their code in containers.

Users get access to free public repositories for storing and sharing images or can choose a subscription plan for private repositories.

Docker Hub provides the following major features:

- **Repositories**: Push and pull container images.
- **Teams & Organizations**: Manage access to private repositories of container images.
- **Official Images**: Pull and use high-quality container images provided by Docker.
- **Publisher Images**: Pull and use high- quality container images provided by external vendors.
- **Builds**: Automatically build container images from GitHub and Bitbucket and push them to Docker Hub.
- **Webhooks**: Trigger actions after a successful push to a repository to integrate Docker Hub with other services.

Docker Hub Quickstart

The following section contains step-by-step instructions on how to easily get started with Docker Hub.

1. **Sign up for a Docker account**

Let’s start by creating a Docker ID.
A Docker ID grants you access to Docker Hub repositories and allows you to explore images that are available from the community and verified publishers. You’ll also need a Docker ID to share images on Docker Hub.

2. **Create Your first repository**

To create a repository:

1. Sign in to Docker Hub.
2. Click *Create a Repository* on the Docker Hub welcome page:
3. Name it *<your-username>/my-private-repo.*
4. Set the visibility to *Private.*
5. Click *Create*.

Docker Hub Quickstart

The following section contains step-by-step instructions on how to easily get started with Docker Hub.

3. Build and push a container image to Docker Hub from your computer

1. Start by creating a Dockerfile to specify your application:
2. Run *docker build -t <your_username>/my-private-repo .* to build your Docker image.
3. Run *docker run <your_username>/my-private-repo* to test your Docker image locally.
4. Run *docker push <your_username>/my-private-repo* to push your Docker image to Docker Hub.
5. Your repository in Docker Hub should now display a new *latest* tag under *Tags*:

Congratulations! You've successfully:

- Signed up for a Docker account
- Created your first repository
- Built a Docker container image on your computer
- Pushed it successfully to Docker Hub

sourcemind

Docker Network

sourcemind

Networking overview

One of the reasons Docker containers and services are so powerful is that you can connect them together, or connect them to non-Docker workloads. Docker containers and services do not even need to be aware that they are deployed on Docker, or whether their peers are also Docker workloads or not. Whether your Docker hosts run Linux, Windows, or a mix of the two, you can use Docker to manage them in a platform-agnostic way.

Network drivers

- ***bridge***: The default network driver. If you don't specify a driver, this is the type of network you are creating. *Bridge networks are usually used when your applications run in standalone containers that need to communicate.* See *bridge networks*.
- ***host***: For standalone containers, remove network isolation between the container and the Docker host, and use the host's networking directly. See *use the host network*.
- ***overlay:*** Overlay networks connect multiple Docker daemons together and enable swarm services to communicate with each other. You can also use overlay networks to facilitate communication between a swarm service and a standalone container, or between two standalone containers on different Docker daemons. This strategy removes the need to do OS-level routing between these containers. See *overlay networks*.
- ***macvlan***: Macvlan networks allow you to assign a MAC address to a container, making it appear as a physical device on your network. The Docker daemon routes traffic to containers by their MAC addresses. Using the macvlan driver is sometimes the best choice when dealing with legacy applications that expect to be directly connected to the physical network, rather than routed through the Docker host's network stack. See *Macvlan networks*.
- ***none***: For this container, disable all networking. Usually used in conjunction with a custom network driver. none is not available for swarm services. See disable container networking.
- ***Network plugins:*** You can install and use third-party network plugins with Docker. These plugins are available from *Docker Hub* or from third-party vendors. See the vendor's documentation for installing and using a given network plugin.

Use Bridge networks

In terms of networking, a bridge network is a Link Layer device which forwards traffic between network segments. A bridge can be a hardware device or a software device running within a host machine's kernel.

In terms of Docker, a bridge network uses a software bridge which allows containers connected to the same bridge network to communicate, while providing isolation from containers which are not connected to that bridge network. The Docker bridge driver automatically installs rules in the host machine so that containers on different bridge networks cannot communicate directly with each other.

Bridge networks apply to containers running on the **same** Docker daemon host.

Use the docker network create command to create a user-defined bridge network.

```
:~$ docker network create my-net
:~$ docker network ls
```

You can specify the subnet, the IP address range, the gateway, and other options. See the docker network create reference or the output of *docker network create --help* for details.
Use the *docker network rm* command to remove a user-defined bridge network. If containers are currently connected to the network, *disconnect them* first.

```
:~$ docker network rm my-net
```

Use Bridge networks

Thank you Umar! You are amazing!

When you create a new container, you can specify one or more --network flags. This example connects a Nginx container to the my-net network. It also publishes port 80 in the container to port 8080 on the Docker host, so external clients can access that port. Any other container connected to the my-net network has access to all ports on the my-nginx container, and vice versa.

```
:~$ docker create --name my-nginx \
--network my-net \
--detach \
--publish 8080:80 \
nginx:latest
```

To connect a **running** container to an existing user-defined bridge, use the docker network connect command. The following command connects an already-running my-nginx container to an already-existing my-net network: To disconnect a running container from a user-defined bridge, use the docker network disconnect command.

```
:~$ docker network connect my-net my-nginx
:~$ docker network disconnect my-net my-nginx
```

Use host networks

If you use the host network mode for a container, that container's network stack is not isolated from the Docker host (the container shares the host's networking namespace), and the container does not get its own IP-address allocated. For instance, if you run a container which binds to port 80 and you use host networking, the container's application is available on port 80 on the host's IP address.

When you create a new container, you can specify one or more --network flags. This example connects a Nginx container to the host network.

```
:~$ docker create --name my-nginx \
--network host \
--detach \
nginx:latest
```

*Note: Given that the container does not have its own IP-address when using **host** mode networking, port-mapping does not take effect, and the **-p, --publish, -P,** and **--publish-all** option are ignored, producing a warning instead:*

sourcemind

Start containers automatically

Docker provides restart policies to control whether your containers start automatically when they exit, or when Docker restarts. Restart policies ensure that linked containers are started in the correct order. Docker recommends that you use restart policies, and avoid using process managers to start containers.

Restart policies are different from the --live-restore flag of the dockerd command. Using --live-restore allows you to keep your containers running during a Docker upgrade, though networking and user input are interrupted.

Use a restart policy:
To configure the restart policy for a container, use the --restart flag when using the docker run command. The value of the --restart flag can be any of the following:

Flag	Description
no	Do not automatically restart the container. (the default)
on-failure	Restart the container if it exists due to an error, which manifest as a non-zero exit code.
always	Always restart the container if it stops. If it is manually stopped, it is restarted only when Docker daemon restarts or the container itself is manually restarted. (See the second bullet listed in restart policy details)
unless-stopped	Similar to always, except that when the container is stopped (manually or otherwise), it is not restarted even after Docker daemon restarts

sourcemind

Start containers automatically

The following example starts a Redis container and configures it to always restart unless it is explicitly stopped or Docker is restarted.

```
:~$ docker run -d --restart unless-stopped redis
```

This command changes the restart policy for an already running container named redis.

```
:~$ docker update -d --restart unless-stopped redis
```

And this command will ensure all currently running containers will be restarted unless stopped.

```
:~$ docker update -d --restart unless-stopped $(docker ps
-q)
```

Restart policy details

Keep the following in mind when using restart policies:

- A restart policy only takes effect after a container starts successfully. In this case, starting successfully means that the container is up for at least 10 seconds and Docker has started monitoring it. This prevents a container which does not start at all from going into a restart loop.
- If you manually stop a container, its restart policy is ignored until the Docker daemon restarts or the container is manually restarted. This is another attempt to prevent a restart loop.
- Restart policies only apply to *containers*. Restart policies for swarm services are configured differently. See the flags related to service restart.

Start containers automatically

Launch an instance of NGINX running in a container and using the default NGINX configuration with the following command:

```
:~$ docker run \
--name my-nginx \
--network my-net \
--detach \
--publish 8080:80 \
--restart  unless-stopped \
nginx:latest
```

sourcemind

Manage data in Docker

By default all files created inside a container are stored on a writable container layer. This means that:

- The data doesn't persist when that container no longer exists, and it can be difficult to get the data out of the container if another process needs it.
- A container's writable layer is tightly coupled to the host machine where the container is running. You can't easily move the data somewhere else.
- Writing into a container's writable layer requires a storage driver to manage the filesystem. The storage driver provides a union filesystem, using the Linux kernel. This extra abstraction reduces performance as compared to using *data volumes*, which write directly to the host filesystem.

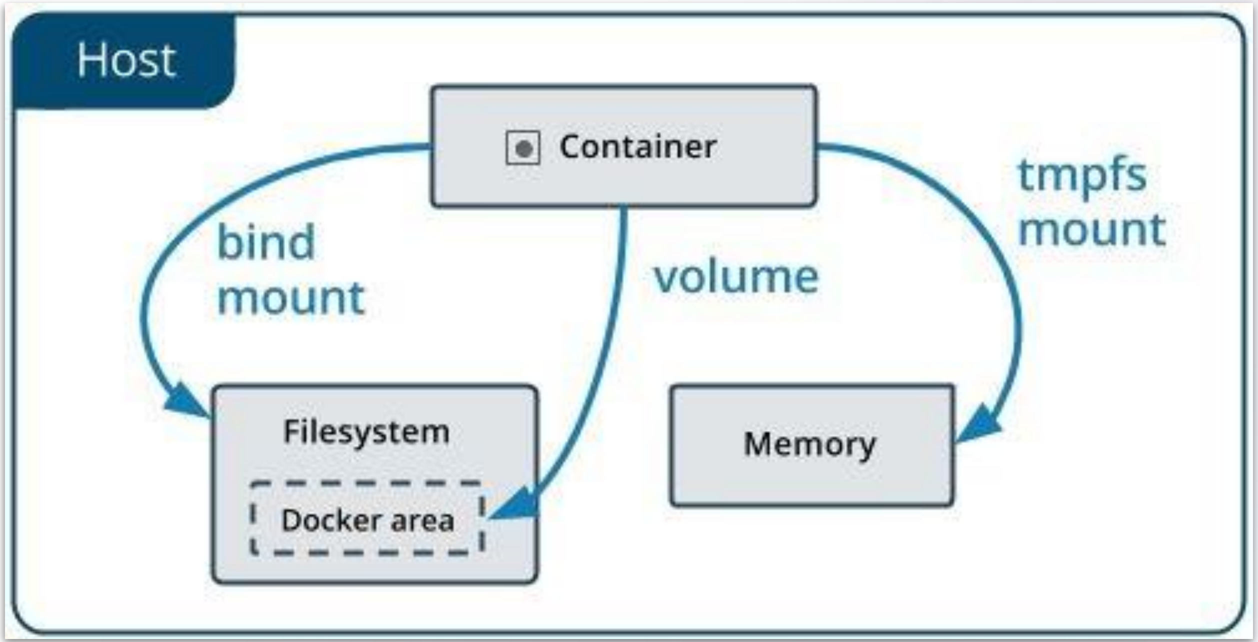
Docker has two options for containers to store files in the host machine, so that the files are persisted even after the container stops: *volumes*, and *bind mounts*. If you're running Docker on Linux you can also use a *tmpfs mount*. If you're running Docker on Windows you can also use a *named pipe*.

sourcemind

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.

- **Volumes** are stored in a part of the host filesystem which is *managed by Docker* (/var/lib/docker/volumes/ on Linux). Non-Docker processes should not modify this part of the filesystem. Volumes are the best way to persist data in Docker.
- **Bind mounts** may be stored *anywhere* on the host system. They may even be important system files or directories. Non-Docker processes on the Docker host or a Docker container can modify them at any time.
- **tmpfs mounts** are stored in the host system's memory only, and are never written to the host system's filesystem.



sourcemind

Volumes

Volumes: Created and managed by Docker. You can create a volume explicitly using the *docker volume create* command, or Docker can create a volume during container or service creation.

When you create a volume, it is stored within a directory on the Docker host. When you mount the volume into a container, this directory is what is mounted into the container. This is similar to the way that bind mounts work, except that volumes are managed by Docker and are isolated from the core functionality of the host machine.

A given volume can be mounted into multiple containers simultaneously. When no running container is using a volume, the volume is still available to Docker and is not removed automatically. You can remove unused volumes using *docker volume prune*

When you mount a volume, it may be **named** or **anonymous**. Anonymous volumes are not given an explicit name when they are first mounted into a container, so Docker gives them a random name that is guaranteed to be unique within a given Docker host. Besides the name, named and anonymous volumes behave in the same ways.

Volumes are the preferred mechanism for persisting data generated by and used by Docker containers. While bind mounts are dependent on the directory structure and OS of the host machine, volumes are completely managed by Docker. Volumes have several advantages over bind mounts:

- Volumes are easier to back up or migrate than bind mounts.
- You can manage volumes using Docker CLI commands or the Docker API.
- Volumes work on both Linux and Windows containers.
- Volumes can be more safely shared among multiple containers.
- Volume drivers let you store volumes on remote hosts or cloud providers, to encrypt the contents of volumes, or to add other functionality.
- New volumes can have their content pre-populated by a container.
- Volumes on Docker Desktop have much higher performance than bind mounts from Mac and Windows hosts.

sourcemind

Use volumes

In general, *--mount* is more explicit and verbose. The biggest difference is that the *-v* syntax combines all the options together in one field, while the *--mount* syntax separates them. Here is a comparison of the syntax for each flag.
If you need to specify volume driver options, you must use --mount.

- **-v or --volume**: Consists of three fields, separated by colon characters *(:)*. The fields must be in the correct order, and the meaning of each field is not immediately obvious.
  - In the case of named volumes, the first field is the name of the volume, and is unique on a given host machine. For anonymous volumes, the first field is omitted.
  - The second field is the path where the file or directory are mounted in the container.
  - The third field is optional, and is a comma-separated list of options, such as *ro*. These options are discussed below.
- **--mount**: Consists of multiple key-value pairs, separated by commas and each consisting of a *<key>=<value>* tuple. The *--mount* syntax is more verbose than *-v* or *--volume,* but the order of the keys is not significant, and the value of the flag is easier to understand.
  - The *type* of the mount, which can be *bind, volume, or tmpfs.* This topic discusses volumes, so the type is always *volume*.
  - The *source* of the mount. For named volumes, this is the name of the volume. For anonymous volumes, this field is omitted. May be specified as *source* or *src*.
  - The *destination* takes as its value the path where the file or directory is mounted in the container. May be specified as *destination*, *dst*, or *target*.
  - The *readonly* option, if present, causes the bind mount to be *mounted into the container as read-only*.
  - The *volume-opt* option, which can be specified more than once, takes a key-value pair consisting of the option name and its value.

Start a container with volume

```
# Create a volume
:~$ docker volume create my-vol
# List a volumes
:~$ docker volume ls
#Inspect a volume
:~$ docker volume inspect my-vol
#Remove a volume:
:~$ docker volume rm my-vol
```

--mount

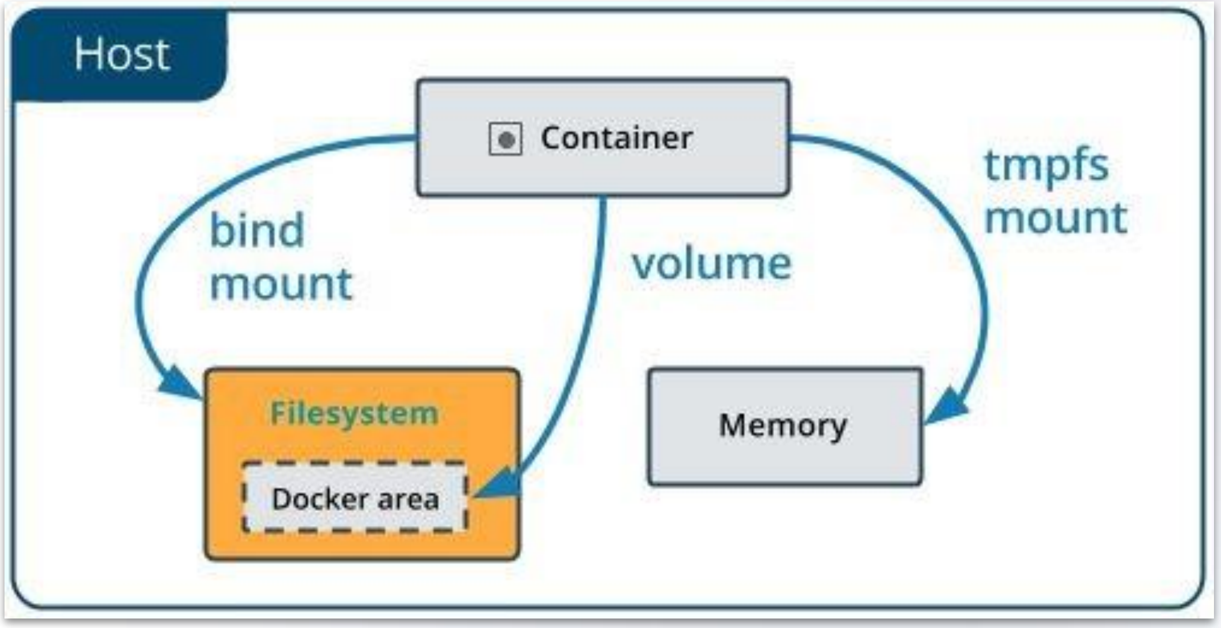
```
:~$ docker run \
--name my-nginx \
--network my-net \
--detach \
--publish 8080:80 \
--restart  unless-stopped \
--mount source=myvol2, target=/app \
nginx:latest
```

-v

```
:~$ docker run \
--name my-nginx \
--network my-net \
--detach \
--publish 8080:80 \
--restart  unless-stopped \
-v myvol2:/app \
nginx:latest
```

Bind mounts

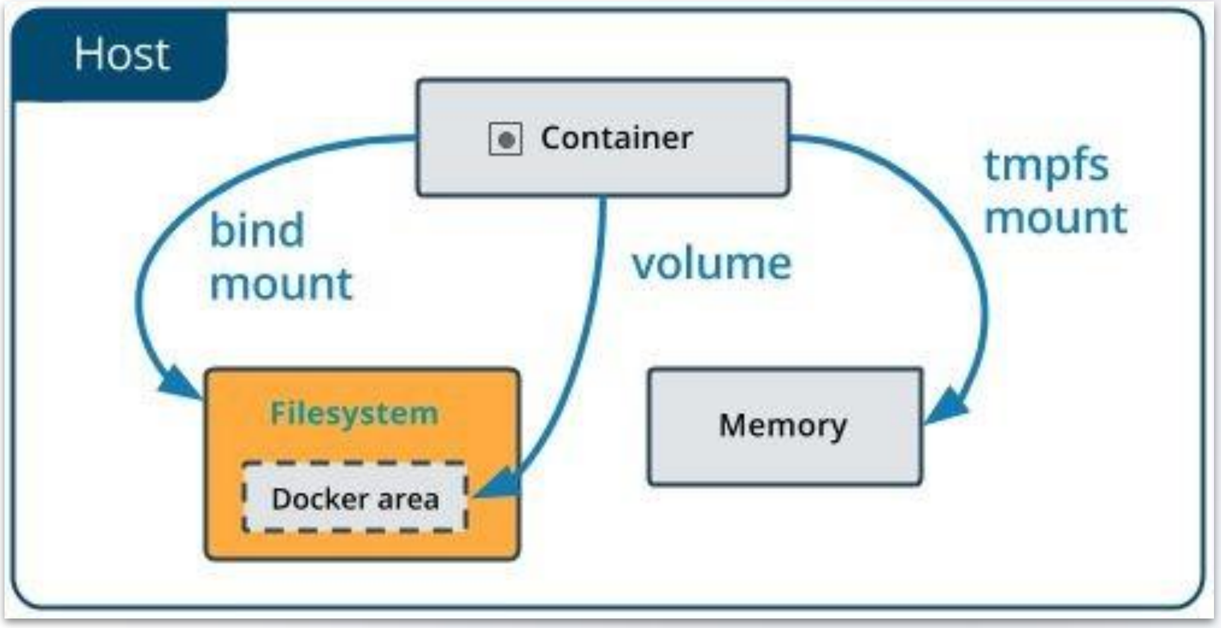
Bind mounts: Available since the early days of Docker. Bind mounts have limited functionality compared to volumes. When you use a bind mount, a file or directory on the *host machine* is mounted into a container. The file or directory is referenced by its full path on the host machine. The file or directory does not need to exist on the Docker host already. It is created on demand if it does not yet exist. Bind mounts are very performant, but they rely on the host machine's filesystem having a specific directory structure available. If you are developing new Docker applications, consider using named volumes instead. You can't use Docker CLI commands to directly manage bind mounts.



sourcemind

Use Bind mounts

Bind mounts: Available since the early days of Docker. Bind mounts have limited functionality compared to volumes. When you use a bind mount, a file or directory on the *host machine* is mounted into a container. The file or directory is referenced by its full path on the host machine. The file or directory does not need to exist on the Docker host already. It is created on demand if it does not yet exist. Bind mounts are very performant, but they rely on the host machine's filesystem having a specific directory structure available. If you are developing new Docker applications, consider using named volumes instead. You can't use Docker CLI commands to directly manage bind mounts.



sourcemind

Start a container with bind mount

```
# inspect container and see bind mounts
:~$ docker inspect my-nginx | grep Mounts -C 10
# Stop container
:~$ docker stop my-nginx
# Remove container
:~$ docker rm my-nginx
# List bind mount file/directory
:~$ ll ”$(pwd)”/target
```

--mount

```
:~$ docker run \
--name my-nginx \
--network my-net \
--detach \
--publish 8080:80 \
--restart  unless-stopped \
--mount type=bind,source=”$(pwd)”/target,target=/app \
nginx:latest
```

-v

```
:~$ docker run \
--name my-nginx \
--network my-net \
--detach \
--publish 8080:80 \
--restart  unless-stopped \
-v “$(pwd)”/target:/app \
nginx:latest
```

sourcemind

Create Wordpress in Docker

Practical Work

Approximate Duration: 1 hours

Objectives:

Create network
Create mysql database in container
Create wordpress in container
Connect wordpress and mysql

sourcemind

Thank you for your attention !

Q&A