

# DOCKER



# Agenda

- Background
- Architecture
- Dockerfile Format
- Build, Start, Share, Update an app container image
- Volumes
- Storage Drivers
- Docker Networking
- Docker Compose
- Docker Commands you should know
- Container Ecosystem

# Background

## The bad old days

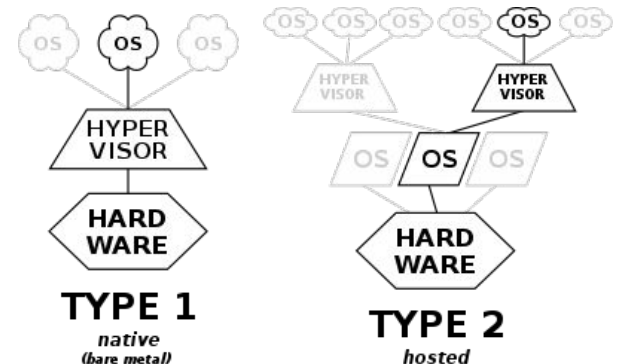
In the past we could only run one application per server. The open-systems world of Windows and Linux just didn't have the technologies to safely and securely run multiple applications on the same server.

As a result, the story went something like this... Every time the business needed a new application, the IT department would buy a new server. Most of the time nobody knew the performance requirements of the new application, forcing the IT department to make guesses when choosing the model and size of the server to buy

As a result, IT did the only thing it could do — it bought big fast servers that cost a lot of money. This resulted in over-powered servers operating as low as 5-10% of their potential capacity. A tragic waste of company capital and environmental resources!

## Hello Hypervisor!

Amid all of this, Hypervisor gave the world a gift — the virtual machine (VM). We finally had a technology that allowed us to safely and securely run multiple business applications on a single server.



# Background Cont'd

## **VMwarts**

The fact that every VM requires its own dedicated operating system (OS) is a major flaw. Every OS consumes CPU, RAM and other resources that could otherwise be used to power more applications. Every OS needs patching and monitoring. And in some cases, every OS requires a license. All of this results in wasted time and resources.

The VM model has other challenges too. VMs are slow to boot, and portability isn't great — migrating and moving VM workloads between hypervisors and cloud platforms is harder than it needs to be.

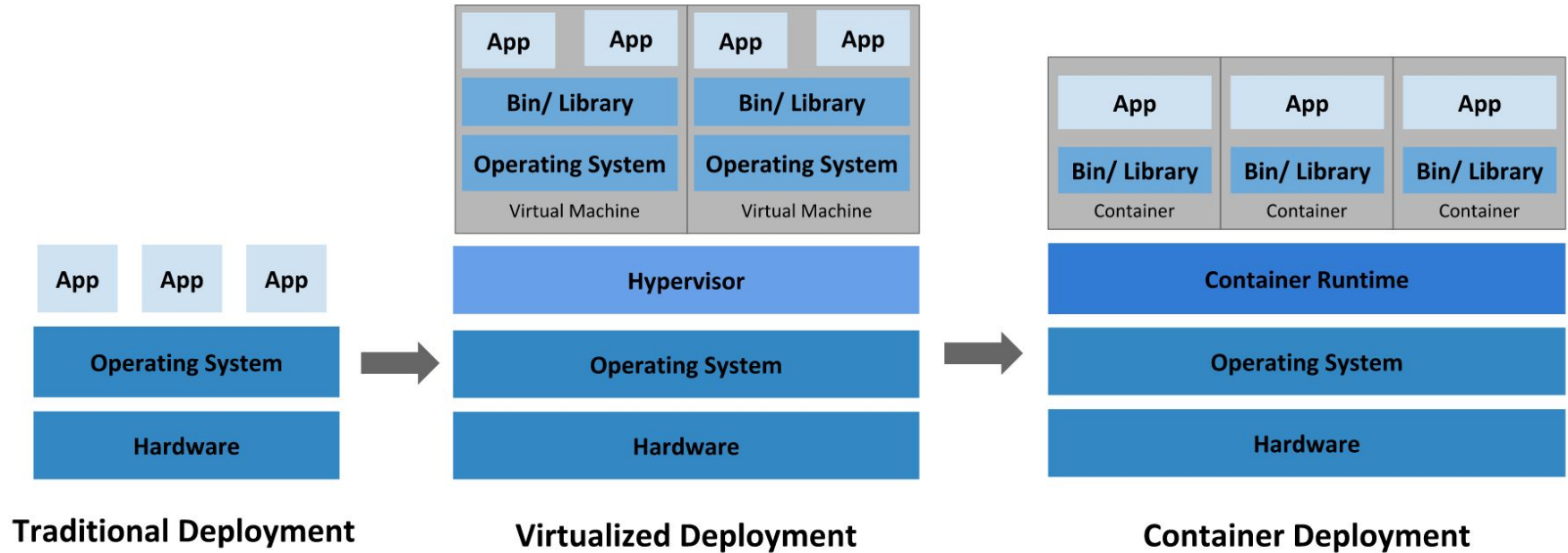
## **Hello Containers!**

For a long time, the big web-scale players, like Google, have been using container technologies to address the shortcomings of the VM model.

In the container model, the container is roughly analogous to the VM. A major difference is that containers do not require their own full-blown OS. In fact, all containers on a single host share the host's OS. This frees up huge amounts of system resources such as CPU, RAM, and storage. It also reduces potential licensing costs and reduces the overhead of OS patching and other maintenance. Net result: savings on the time, resource, and capital fronts.

Containers are also fast to start and ultra-portable. Moving container workloads from your laptop, to the cloud, and then to VMs or bare metal in your data center is a breeze.

# Comparison



# Docker

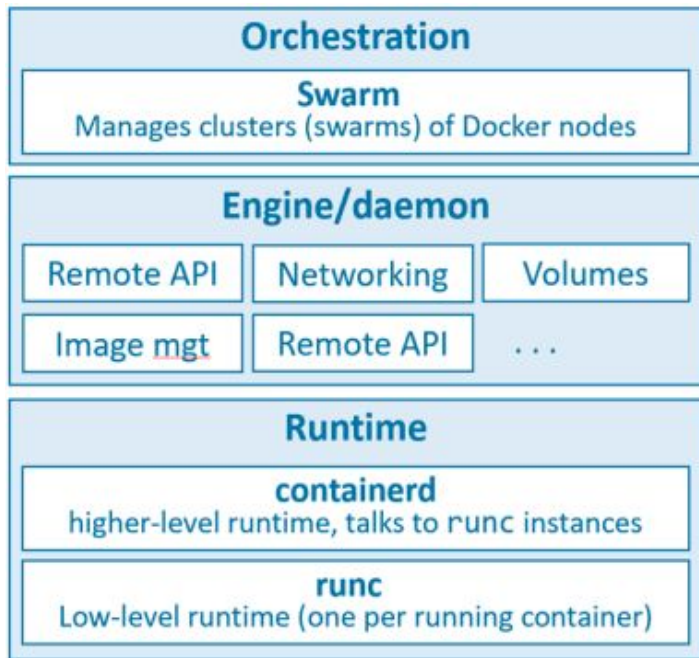
Containers are a standardized unit of software that allows developers to isolate their app from its environment, solving the “it works on my machine” headache.

Docker is a technology that

- runs containers
- build and share containers

For millions of developers today, Docker is the de facto standard to build and share containerized apps – from desktop, to the cloud.

# Architecture



The runtime operates at the lowest level and is responsible for starting and stopping containers (this includes building all of the OS constructs such as namespaces and cgroups).

The low-level runtime is called runc and is the reference implementation of Open Containers Initiative (OCI) runtime-spec. Its job is to interface with the underlying OS and start and stop containers. Every running container on a Docker node has a runc instance managing it.

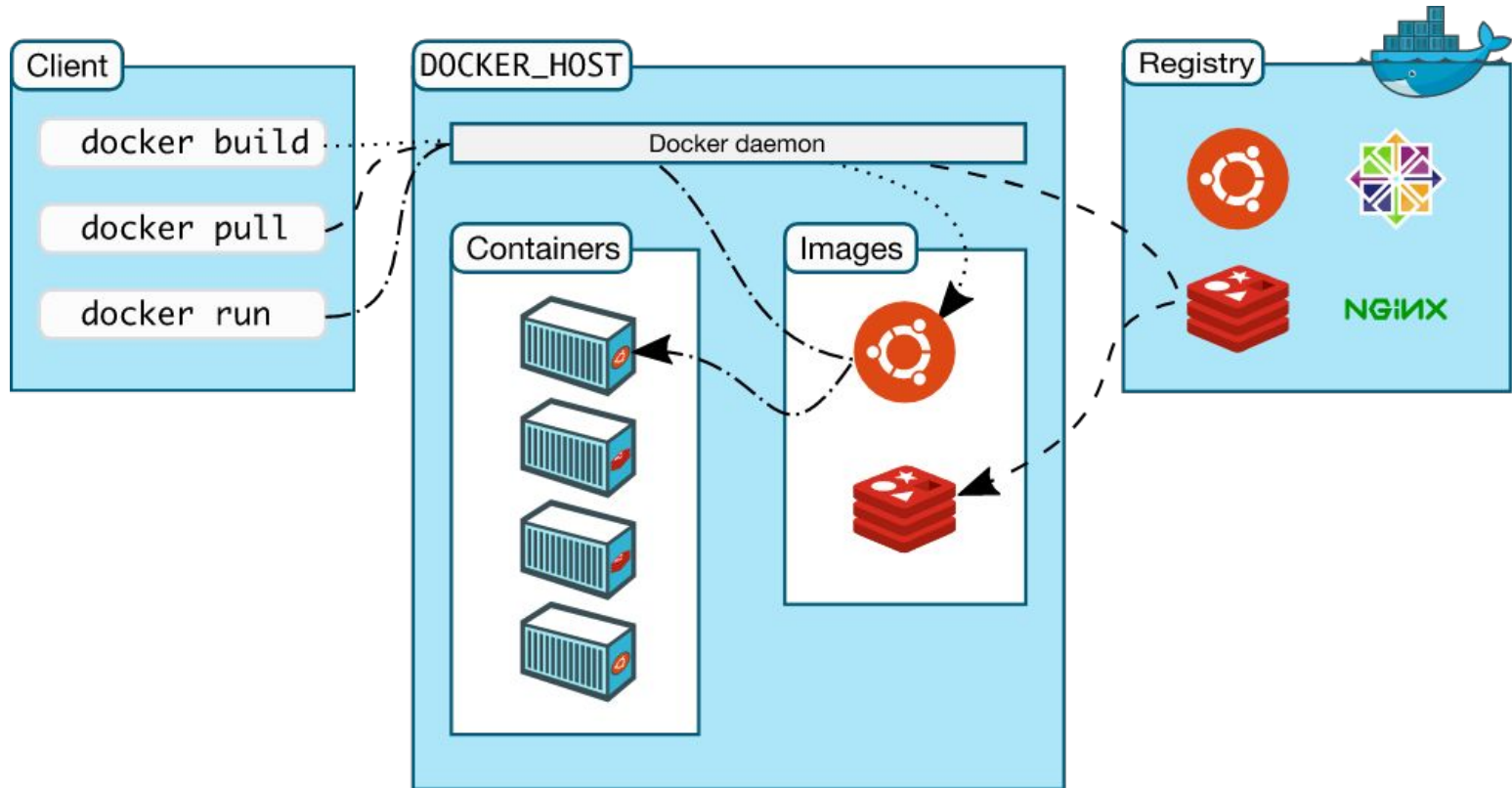
The higher-level runtime is called containerd. containerd does a lot more than runc. It manages the entire lifecycle of a container, including pulling images, creating network interfaces, and managing lower-level runc instances. containerd is pronounced "container-dee" and is a graduated CNCF project used by Docker and Kubernetes as a container runtime.

The Docker daemon (dockerd) sits above containerd and performs higher-level tasks such as; exposing the Docker remote API, managing images, managing volumes, managing networks, and more...

A major job of the Docker daemon is to provide an easy-to-use standard interface that abstracts the lower levels.

Docker Swarm competes directly with Kubernetes — they both orchestrate containerized applications. While it's true that Kubernetes has more momentum and a more active community and ecosystem, Docker Swarm is an excellent technology and a lot easier to configure and deploy. It's an excellent technology for small-to-medium businesses and application deployments.

# Architecture [high level]





# Architecture Cont'd

## The Docker daemon

The Docker daemon (`dockerd`) listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. A daemon can also communicate with other daemons to manage Docker services.

## The Docker client

The Docker client (`docker`) is the primary way that many Docker users interact with Docker. When you use commands such as `docker run`, the client sends these commands to `dockerd`, which carries them out. The `docker` command uses the Docker API. The Docker client can communicate with more than one daemon.

## Docker registries

A Docker *registry* stores Docker images. Docker Hub is a public registry that anyone can use, and Docker is configured to look for images on Docker Hub by default. You can even run your own private registry.

When you use the `docker pull` or `docker run` commands, the required images are pulled from your configured registry. When you use the `docker push` command, your image is pushed to your configured registry.

# Dockerfile Format

A **Dockerfile** is a script that contains all commands for building a **Docker** image.

## Format

Here is the format of the Dockerfile:

```
# Comment  
INSTRUCTION arguments
```

```
FROM node:12-alpine  
RUN apk add --no-cache python2 g++  
make  
WORKDIR /app  
COPY . .  
RUN yarn install --production  
CMD ["node", "src/index.js"]  
EXPOSE 3000
```

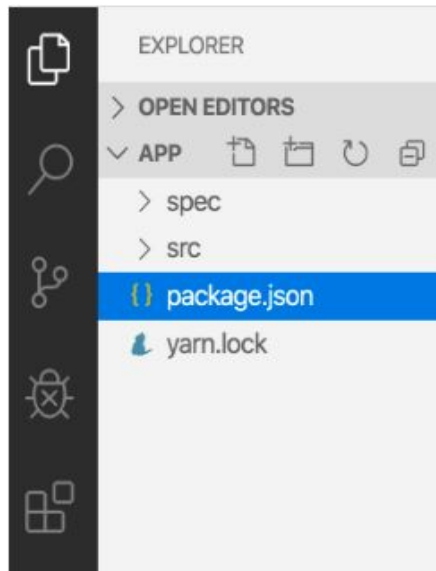
1. The **FROM** directive
2. The **LABEL** directive
3. The **RUN** directive
4. The **CMD** directive
5. The **ENTRYPOINT** directive
6. The **ENV** directive
7. The **ARG** directive
8. The **WORKDIR** directive
9. The **COPY** directive
10. The **ADD** directive
11. The **USER** directive
12. The **VOLUME** directive
13. The **EXPOSE** directive
14. The **HEALTHCHECK** directive
15. The **ONBUILD** directive

# Build an app container image

In order to build the application, we need to use a `Dockerfile`. A `Dockerfile` is simply a text-based script of instructions that is used to create a container image.

```
# syntax=docker/dockerfile:1
FROM node:12-alpine
RUN apk add --no-cache python2 g++ make
WORKDIR /app
COPY . .
RUN yarn install --production
CMD ["node", "src/index.js"]
EXPOSE 3000
```

```
$ docker build -t getting-started .
```



# Build an app container image Cont'd

This command used the Dockerfile to build a new container image. You might have noticed that a lot of “layers” were downloaded. This is because we instructed the builder that we wanted to start from the `node:12-alpine` image. But, since we didn't have that on our machine, that image needed to be downloaded.

After the image was downloaded, we copied in our application and used `yarn` to install our application's dependencies. The `CMD` directive specifies the default command to run when starting a container from this image.

Finally, the `-t` flag tags our image. Think of this simply as a human-readable name for the final image. Since we named the image `getting-started`, we can refer to that image when we run a container.

The `.` at the end of the `docker build` command tells Docker that it should look for the `Dockerfile` in the current directory.

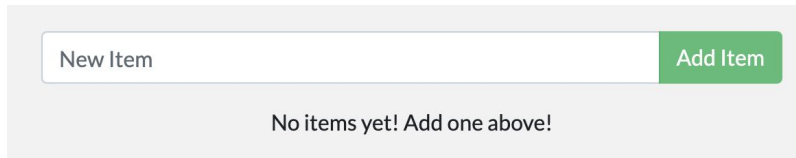
# Start an app container

1. Start your container using the `docker run` command and specify the name of the image we just create:

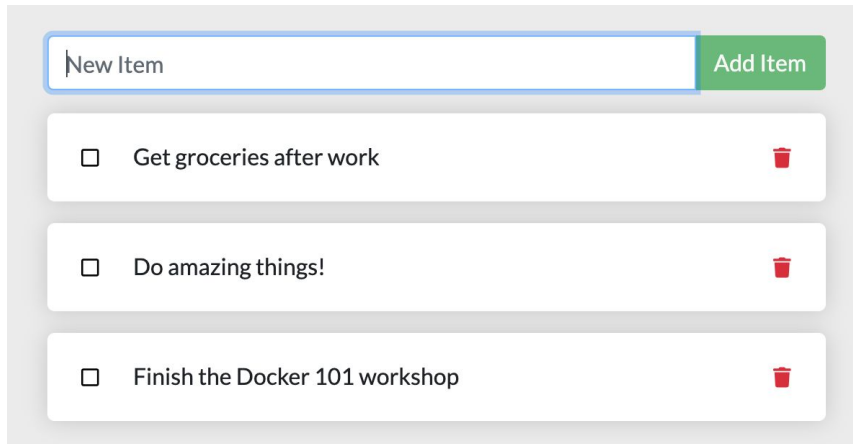
```
$ docker run -dp 3000:3000 getting-started
```

Remember the `-d` and `-p` flags? We're running the new container in “detached” mode (in the background) and creating a mapping between the host's port 3000 to the container's port 3000. Without the port mapping, we wouldn't be able to access the application.

2. After a few seconds, open your web browser to <http://localhost:3000>. You should see our app.



A screenshot of a web application interface. At the top, there is a text input field containing the placeholder text "New Item" and a green button labeled "Add Item". Below the input field, the text "No items yet! Add one above!" is displayed.



A screenshot of the same web application interface, but now it displays a list of three items. Each item is shown in a white card with a light gray border. The items are: "Get groceries after work", "Do amazing things!", and "Finish the Docker 101 workshop". Each item has a checkbox on the left and a red trash icon on the right. The "New Item" input field and "Add Item" button are still at the top.

# Share the application

To share Docker images, you have to use a Docker registry. The default registry is Docker Hub and is where all of the images we've used have come from.

## Create a repo

To push an image, we first need to create a repository on Docker Hub.

1. [Sign up](#) or Sign in to [Docker Hub](#).
2. Click the **Create Repository** button.
3. For the repo name, use `getting-started`. Make sure the Visibility is `Public`.
4. Click the **Create** button!

## Push the image

1. `docker login -u YOUR-USER-NAME .`
2. `docker tag getting-started YOUR-USER-NAME/getting-started`
3. `docker push YOUR-USER-NAME/getting-started`

# CONTAINER REGISTRIES

- Docker Hub
- Red Hat Quay
- GitHub Package Registry
- GitLab Package Registry
- Azure Container Registry
- Google Container Registry (GCR)
- Amazon Elastic Container Registry (ECR)
- TreeScale
- Canister
- Harbor Container Registry

# SELF-HOSTED CONTAINER REGISTRIES

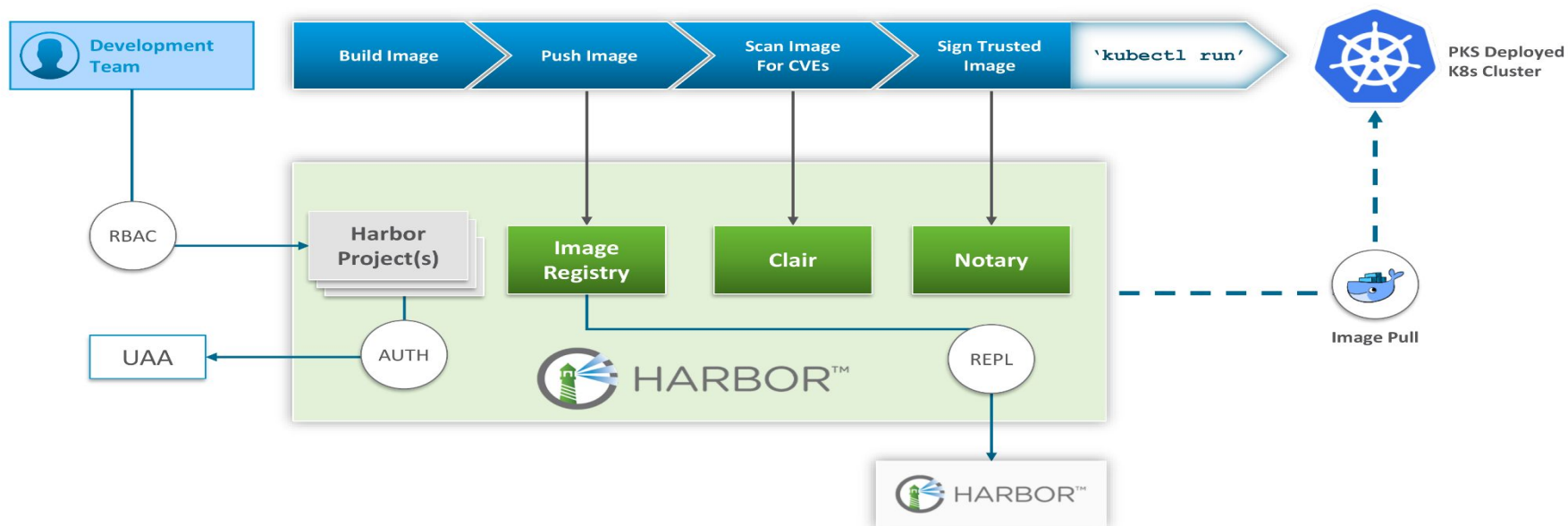
- Docker Registry
- GitLab Container Registry
- Harbor
- Portus
- JFrog Artifactory



# Harbor

Cloud native registry able to store container images and Helm Charts.

## Development Cycle with Harbor:



# Update the application

## Update the source code

1. In the `src/static/js/app.js` file, update line 56 to use the new empty text.

```
- <p className="text-center">No items yet! Add one above!</p>
+ <p className="text-center">You have no todo items yet! Add one above!</p>
```

2. Let's build our updated version of the image, using the same command we used before.

```
$ docker build -t getting-started .
```

3. Let's start a new container using the updated code.

```
$ docker run -dp 3000:3000 getting-started
```

4. Refresh your browser on <http://localhost:3000> and you should see your updated help text!

Add Item

You have no todo items yet! Add one above!

# Container volumes

With the previous experiment, we saw that each container starts from the image definition each time it starts. While containers can create, update, and delete files, those changes are lost when the container is removed and all changes are isolated to that container. With volumes, we can change all of this.

**Volumes** provide the ability to connect specific filesystem paths of the container back to the host machine. If a directory in the container is mounted, changes in that directory are also seen on the host machine. If we mount that same directory across container restarts, we'd see the same files.

There are two main types of volumes.

1. **named volumes**
2. **bind mounts**

# named volumes

By default, the todo app stores its data in a [SQLite Database](#) at `/etc/todos/todo.db` in the container's filesystem. If you're not familiar with SQLite, no worries! It's simply a relational database in which all of the data is stored in a single file.

With the database being a single file, if we can persist that file on the host and make it available to the next container, it should be able to pick up where the last one left off. By creating a volume and attaching (often called “mounting”) it to the directory the data is stored in, we can persist the data. As our container writes to the `todo.db` file, it will be persisted to the host in the volume.

- Create a volume by using the `docker volume create` command.

```
$ docker volume create todo-db
```

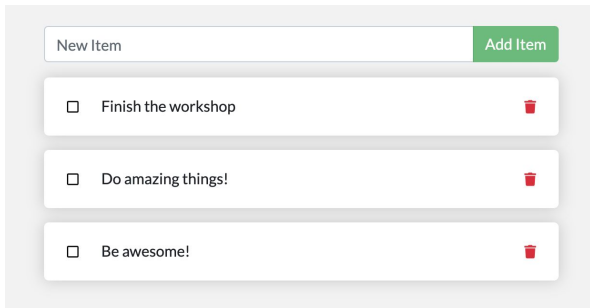
- Stop and remove the todo app container once again with `docker rm -f <id>`, as it is still running without using the persistent volume.
- Start the todo app container, but add the `-v` flag to specify a volume mount. We will use the named volume and mount it to

`/etc/todos`, which will capture all files created at the path.

```
$ docker run -dp 3000:3000 -v todo-db:/etc/todos getting-started
```

# named volumes Cont'd

- Once the container starts up, open the app and add a few items to your todo list.
- Stop and remove the container for the todo app. Use the Dashboard or `docker ps` to get the ID and then `docker rm -f <id>` to remove it.
- Start a new container using the same command from above.
- Open the app. You should see your items still in your list!
- Go ahead and remove the container when you're done checking out your list.



The screenshot shows a web application interface for a todo list. At the top, there is a text input field labeled "New Item" and a green button labeled "Add Item". Below the input field, there is a list of three items, each in a white box with a light gray border. Each item has a checkbox on the left and a red trash icon on the right. The items are: "Finish the workshop", "Do amazing things!", and "Be awesome!".

- Named volumes are great if we simply want to store data, as we don't have to worry about *where* the data is stored.

# Dive into the volume

```
$ docker volume inspect todo-db

[
  {
    "CreatedAt": "2019-09-26T02:18:36Z",
    "Driver": "local",
    "Labels": {},
    "Mountpoint": "/var/lib/docker/volumes/todo-db/_data",
    "Name": "todo-db",
    "Options": {},
    "Scope": "local"
  }
]
```

The `Mountpoint` is the actual location on the disk where the data is stored.

# bind mounts

With **bind mounts**, we control the exact mountpoint on the host. We can use this to persist data, but it's often used to provide additional data into containers.

## Start a dev-mode container

- Mount our source code into the container
- Install all dependencies, including the “dev” dependencies

Run the following command from the app directory.

```
$ docker run -dp 3000:3000 \
  -w /app -v "$(pwd) :/app" \
  node:12-alpine \
  sh -c "yarn install && yarn run dev"
```

Add Item

No items yet! Add one above!

- `-dp 3000:3000` - Run in detached (background) mode and create a port mapping
- `-w /app` - sets the “working directory” or the current directory that the command will run from
- `-v "$(pwd) :/app"` - bind mount the current directory from the host in the container into the `/app` directory
- `node:12-alpine` - the image to use. Note that this is the base image for our app from the Dockerfile
- `sh -c "yarn install && yarn run dev"` - the command. We're starting a shell using `sh` (alpine doesn't have `bash`) and running `yarn install` to install *all* dependencies and then running `yarn run dev`. If we look in the `package.json`, we'll see that the `dev` script is starting `nodemon`.

## bind mounts Cont'd

Now, let's make a change to the app. In the `src/static/js/app.js` file, let's change the “Add Item” button to simply say “Add”. This change will be on line 109:

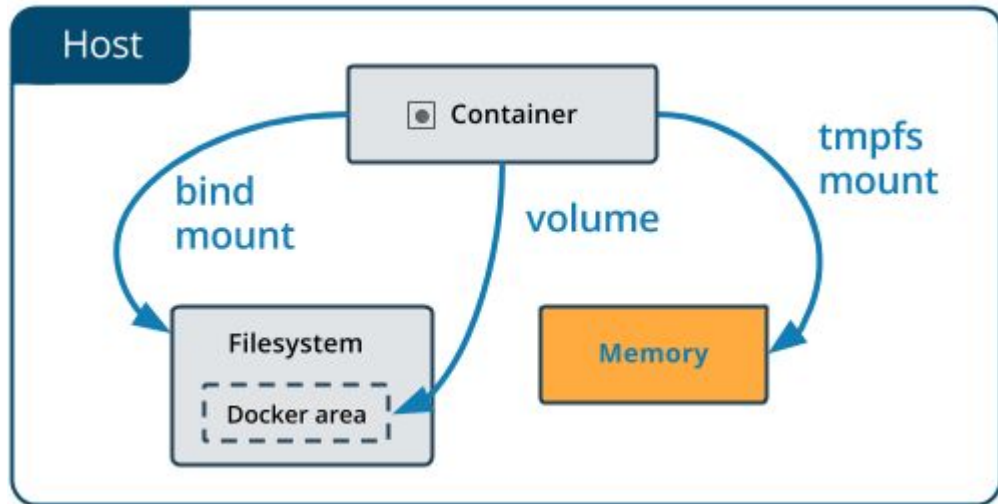
```
-           {submitting ? 'Adding...' : 'Add Item'}  
+           {submitting ? 'Adding...' : 'Add'}
```

Simply refresh the page (or open it) and you should see the change reflected in the browser almost immediately. It might take a few seconds for the Node server to restart, so if you get an error, just try refreshing after a few seconds.

No items yet! Add one above!



# tmpfs mounts



As opposed to volumes and bind mounts, a `tmpfs` mount is temporary, and only persisted in the host memory. When the container stops, the `tmpfs` mount is removed, and files written there won't be persisted.

This is useful to temporarily store sensitive files that you don't want to persist in either the host or the container writable layer.

## Limitations of tmpfs mounts

- Unlike volumes and bind mounts, you can't share `tmpfs` mounts between containers.
- This functionality is only available if you're running Docker on Linux.

# Choose the `--tmpfs` or `--mount` flag

## Differences between `--tmpfs` and `--mount` behavior

- The `--tmpfs` flag does not allow you to specify any configurable options.
- The `--tmpfs` flag cannot be used with swarm services. You must use `--mount`

### `--mount`

```
$ docker run -d \
  -it \
  --name tmpctest \
  --mount type=tmpfs,destination=/app \
  nginx:latest
```

### `--tmpfs`

```
docker run -d \
  -it \
  --name tmpctest \
  --tmpfs /app \
  nginx:latest
```

Verify that the mount is  
a `tmpfs` mount by  
running `docker`  
`container inspect`  
`tmpctest` and looking  
for the `Mounts` section:

```
"Tmpfs": {
  "/app": ""
},
```

- `--tmpfs`: Mounts a `tmpfs` mount without allowing you to specify any configurable options, and can only be used with standalone containers.
- `--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 `--tmpfs`:
  - o The `type` of the mount, which can be `bind`, `volume`, or `tmpfs`. This topic discusses `tmpfs`, so the type is always `tmpfs`.
  - o The `destination` takes as its value the path where the `tmpfs` mount is mounted in the container. May be specified as `destination`, `dst`, or `target`.
  - o The `tmpfs-size` and `tmpfs-mode` options.

# About storage drivers

Docker uses storage drivers to store image layers, and to store data in the writable layer of a container.

```
# syntax=docker/dockerfile:1

FROM ubuntu:18.04

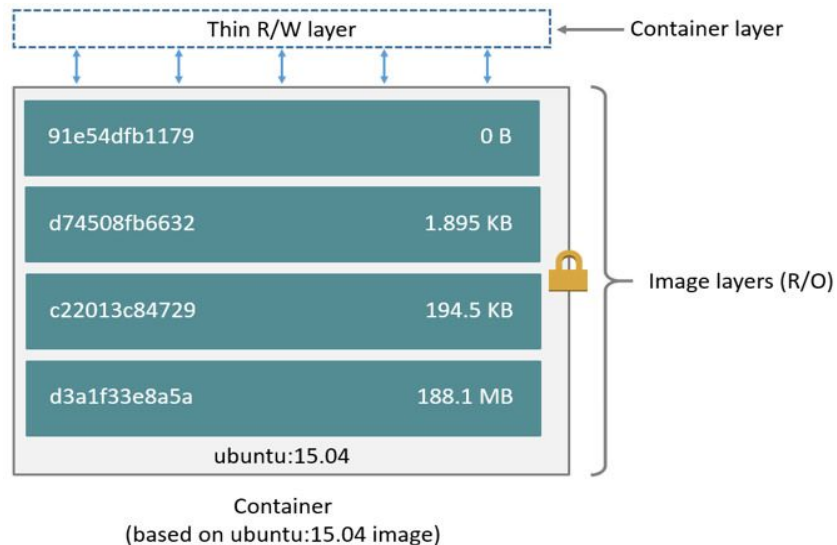
LABEL org.opencontainers.image.authors="org@example.com"

COPY . /app

RUN make /app

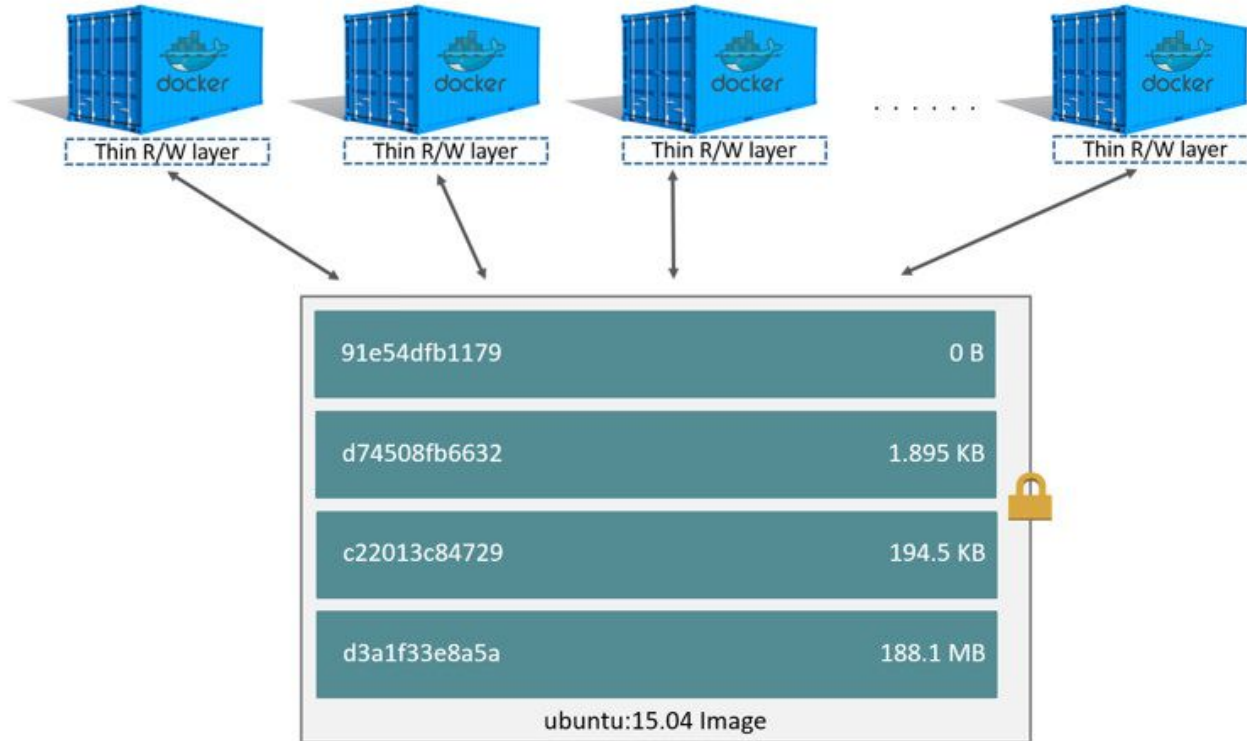
RUN rm -r $HOME/.cache

CMD python /app/app.py
```



A *storage driver* handles the details about the way these layers interact with each other. Different storage drivers are available, which have advantages and disadvantages in different situations.

# About storage drivers Cont'd



# Docker storage drivers

<code>overlay2</code>	overlay2 is the preferred storage driver for all currently supported Linux distributions, and requires no extra configuration.
<code>fuse-overlayfs</code>	Fuse-overlayfs is preferred only for running Rootless Docker on a host that does not provide support for rootless overlay2. On Ubuntu and Debian 10, the fuse-overlayfs driver does not need to be used, and overlay2 works even in rootless mode.
<code>btrfs</code> and <code>zfs</code>	The btrfs and zfs storage drivers allow for advanced options, such as creating “snapshots”, but require more maintenance and setup. Each of these relies on the backing filesystem being configured correctly.
<code>vfs</code>	The vfs storage driver is intended for testing purposes, and for situations where no copy-on-write filesystem can be used. Performance of this storage driver is poor, and is not generally recommended for production use.
<code>aufs</code>	The aufs storage driver Was the preferred storage driver for Docker 18.06 and older, when running on Ubuntu 14.04 on kernel 3.13 which had no support for overlay2. However, current versions of Ubuntu and Debian now have support for overlay2, which is now the recommended driver.
<code>devicemapper</code>	The devicemapper storage driver requires direct-lvm for production environments, because loopback-lvm, while zero-configuration, has very poor performance. devicemapper was the recommended storage driver for CentOS and RHEL, as their kernel version did not support overlay2. However, current versions of CentOS and RHEL now have support for overlay2, which is now the recommended driver.
<code>overlay</code>	The legacy overlay driver was used for kernels that did not support the “multiple-lowerdir” feature required for overlay2 All currently supported Linux distributions now provide support for this, and it is therefore deprecated.

# How To Communicate Between Docker Containers

In the real world, beyond the realm of the simple hello-world tutorial, running just one container isn't enough for most apps. A modern application typically consists of a few components – such as a database, a web server, or some microservices.

So if you want to run all of your components in [containers](#), how can the applications talk to each other?

**How do containers communicate with each other, if they're supposed to be isolated?**

*A Docker network lets your containers communicate with each other*

You can create different types of networks depending on what you would like to do. We'll cover the easiest options:

- The **default bridge network**
- A **user-defined bridge network**

# Default bridge network (easiest option)

The simplest network in Docker is the **bridge network**. It's also Docker's default networking driver.

From that point onwards, all containers are added into to the bridge network, unless you say otherwise.

**In a bridge network, each container is assigned its own IP address.** So containers can communicate with each other by IP.

```
# Start an nginx container, give it the name 'mynginx' and run in the background
$ docker run --rm --name mynginx --detach nginx

# Get the IP address of the container
$ docker inspect mynginx | grep IPAddress
    "IPAddress": "172.17.0.2",

# Or, if you have 'jq' installed - here's a funky way to get the IP address
$ sudo docker inspect mynginx | jq '.[].NetworkSettings.Networks.bridge.IPAddress'
"172.17.0.2"

# Run busybox (a utility container). It will join the bridge network
$ docker run -it busybox sh

# Fetch the nginx homepage by using the container's IP address
busybox$ wget -q -O - 172.17.0.2:80
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>

# Voila! The nginx homepage!
```

# User-defined bridge: the more sensible option

The default bridge is..... **fine**.... but it means every container can see every other container.

What you probably want is: a **user-defined network**, so that you can be more granular about which containers can see each other.

Let's look at that option.

And that's user-defined bridge networking. It's a great way to have a custom network setup, and isolation from other containers that aren't in the network.

```
# Create a user-defined bridge network
$ docker network create tulip-net

# Start a container and connect it to the bridge
$ docker run --rm --net tulip-net --name tulipnginx -d nginx

# Address another container, using its name as the hostname
# Start a busybox container so that we can test out the network
$ docker run --net tulip-net -it busybox sh
# Use 'wget' inside busybox, using the container name as the hostname!
busybox$ wget -q -O - tulipnginx:80
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...you get the picture....
```



# Network drivers

- **User-defined bridge networks** are best when you need multiple containers to communicate on the same Docker host.
- **Host networks** are best when the network stack should not be isolated from the Docker host, but you want other aspects of the container to be isolated.
- **Overlay networks** are best when you need containers running on different Docker hosts to communicate, or when multiple applications work together using swarm services.
- **Macvlan networks** are best when you are migrating from a VM setup or need your containers to look like physical hosts on your network, each with a unique MAC address.
- **Third-party network plugins** allow you to integrate Docker with specialized network stacks.

# Docker Compose

```
docker run -dp 3000:3000 \  
  -w /app -v "$(pwd) :/app" \  
  --network todo-app \  
  -e MYSQL_HOST=mysql \  
  -e MYSQL_USER=root \  
  -e MYSQL_PASSWORD=secret \  
  -e MYSQL_DB=todos \  
  node:12-alpine \  
  sh -c "yarn install && yarn run dev"
```

```
docker run -d \  
  --network todo-app \  
  -v todo-mysql-data:/var/lib/mysql \  
  -e MYSQL_ROOT_PASSWORD=secret \  
  -e MYSQL_DATABASE=todos \  
  mysql:5.7
```

```
docker-compose up -d
```

```
docker-compose down
```

```
version: "3.7"
```

```
services:
```

```
  app:
```

```
    image: node:12-alpine
```

```
    command: sh -c "yarn install && yarn run dev"
```

```
    ports:
```

```
      - 3000:3000
```

```
    working_dir: /app
```

```
    volumes:
```

```
      - ./:/app
```

```
    environment:
```

```
      MYSQL_HOST: mysql
```

```
      MYSQL_USER: root
```

```
      MYSQL_PASSWORD: secret
```

```
      MYSQL_DB: todos
```

```
  mysql:
```

```
    image: mysql:5.7
```

```
    volumes:
```

```
      - todo-mysql-data:/var/lib/mysql
```

```
    environment:
```

```
      MYSQL_ROOT_PASSWORD: secret
```

```
      MYSQL_DATABASE: todos
```

```
volumes:
```

```
  todo-mysql-data:
```

# Networking in Compose

```
version: "3.9"
services:

  web:
    build: .
    links:
      - "db:database"
  db:
    image: postgres
```

Links allow you to define extra aliases by which a service is reachable from another service.

```
version: "3.9"

services:
  proxy:
    build: ./proxy
    networks:
      - frontend
  app:
    build: ./app
    networks:
      - frontend
      - backend
  db:
    image: postgres
    networks:
      - backend

networks:
  frontend:
    # Use a custom driver
    driver: custom-driver-1
  backend:
    # Use a custom driver which takes special options
    driver: custom-driver-2
    driver_opts:
      foo: "1"
      bar: "2"
```

# Few Docker Commands You Should Know

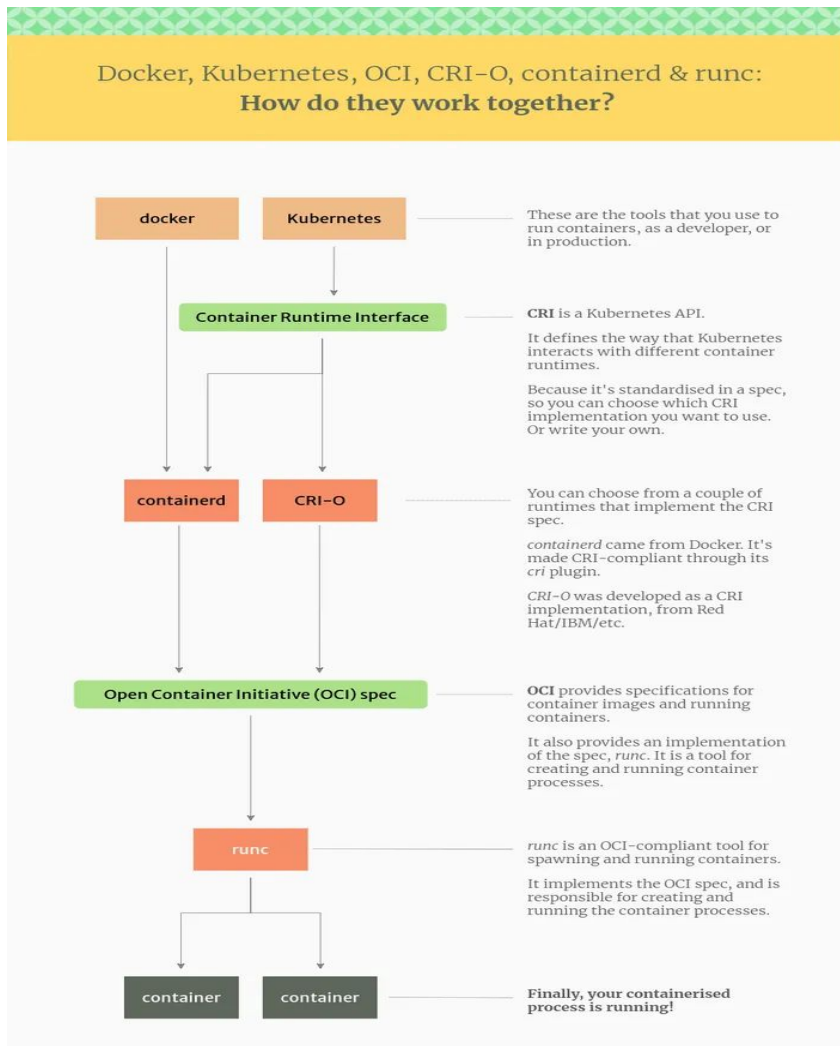
- `docker --version`
- `docker pull`
- `docker build`
- `docker run`
- `docker ps`
- `docker ps -a`
- `docker exec`
- `docker log`
- `docker stop`
- `docker kill`
- `docker commit`
- `docker login`
- `docker push`
- `docker images`
- `docker rm`
- `docker rmi`

how Docker, Kubernetes, CRI, OCI, containerd and runc fit together in this ecosystem:

The main standards around containers that you should be aware of (although you don't need to know all the details) are:

- The **Open Container Initiative (OCI)** which publishes specifications for containers and their images.
- The Kubernetes **Container Runtime Interface (CRI)**, which defines an API between Kubernetes and a container runtime underneath.

This illustration shows exactly how Docker, Kubernetes, CRI, OCI, containerd and runc fit together in this ecosystem:



# References

- <https://subscription.packtpub.com/book/cloud-and-networking/9781838983444/2/ch02lv1sec12/common-directives-in-dockerfiles>
- <https://subscription.packtpub.com/book/cloud-and-networking/9781838983444/2/ch02lv1sec14/other-dockerfile-directives>
- <https://docs.docker.com/get-started/overview/>
- <https://subscription.packtpub.com/book/cloud-and-networking/9781838983444/2/ch02lv1sec12/common-directives-in-dockerfiles>
- <https://subscription.packtpub.com/book/cloud-and-networking/9781838983444/2/ch02lv1sec14/other-dockerfile-directives>

Thank you

# FROM

```
FROM [--platform=<platform>] <image> [AS <name>]
```

Or

```
FROM [--platform=<platform>] <image>[:<tag>] [AS <name>]
```

Or

```
FROM [--platform=<platform>] <image>[@<digest>] [AS <name>]
```

- `ARG` is the only instruction that may precede `FROM` in the `Dockerfile`.
- `FROM` can appear multiple times within a single `Dockerfile` to create multiple images or use one build stage as a dependency for another. Simply make a note of the last image ID output by the commit before each new `FROM` instruction. Each `FROM` instruction clears any state created by previous instructions.
- Optionally a name can be given to a new build stage by adding `AS name` to the `FROM` instruction. The name can be used in subsequent `FROM` and `COPY --from=<name>` instructions to refer to the image built in this stage.
- The `tag` or `digest` values are optional. If you omit either of them, the builder assumes a `latest` tag by default. The builder returns an error if it cannot find the `tag` value.
- The optional `--platform` flag can be used to specify the platform of the image in case `FROM` references a multi-platform image. For example, `linux/amd64`, `linux/arm64`, or `windows/amd64`. By default, the target platform of the build request is used.

The `FROM` instruction initializes a new build stage and sets the *Base Image* for subsequent instructions. As such, a valid `Dockerfile` must start with a `FROM` instruction. The image can be any valid image – it is especially easy to start by pulling an image from the *Public Repositories*.



# Best practices for writing Dockerfiles

- **Understand build context:**

Regardless of where the `Dockerfile` actually lives, all recursive contents of files and directories in the current directory are sent to the Docker daemon as the build context.

- **Exclude with `.dockerignore`**

To exclude files not relevant to the build use a `.dockerignore` file. This file supports exclusion patterns similar to `.gitignore`

- **Use multi-stage builds**

- **Don't install unnecessary packages**

For example, you don't need to include a text editor in a database image.

- **Sort multi-line arguments**

- **Group multiple similar expressions into one using “\”**

- **Order them from the less frequently changed (to ensure the build cache is reusable) to the more frequently changed**

# The Open Container Initiative (OCI)

- The OCI is a governance council responsible for standardizing the low-level fundamental components of container infrastructure.
- a company called CoreOS (acquired by Red Hat which was then acquired by IBM) didn't like the way Docker did certain things. created an open standard called appc that defined things like image format and container runtime. They also created an implementation of the spec called rkt (pronounced "rocket").
- This put the container ecosystem in an awkward position with two competing standards.
- came together to form the OCI — a lightweight agile council to govern container standards.
- At the time of writing, the OCI has published two specifications (standards)
  - The image-spec
  - The runtime-spec

# Container size on disk

To view the approximate size of a running container, you can use the `docker ps -s` command. Two different columns relate to size.

- `size`: the amount of data (on disk) that is used for the writable layer of each container.
- `virtual size`: the amount of data used for the read-only image data used by the container plus the container's writable layer `size`

The total disk space used by all of the running containers on disk is some combination of each container's `size` and the `virtual size` values. If multiple containers started from the same exact image, the total size on disk for these containers would be `SUM (size of containers) plus one image size (virtual size - size)`.

This also does not count the following additional ways a container can take up disk space:

- Disk space used for log files stored by the [logging-driver](#). This can be non-trivial if your container generates a large amount of logging data and log rotation is not configured.
- Volumes and bind mounts used by the container.
- Disk space used for the container's configuration files, which are typically small.
- Memory written to disk (if swapping is enabled).
- Checkpoints, if you're using the experimental checkpoint/restore feature.