**Docker**

Dockarize API Server:

* Single File
* Multi-staged

## [What is a container?](https://docs.docker.com/get-started/#what-is-a-container)

A container is a sandboxed process running on a host machine that is isolated from all other processes running on that host machine

To summarize, a container:

* Is a runnable instance of an image. You can create, start, stop, move, or delete a container using the Docker API or CLI.
* Can be run on local machines, virtual machines, or deployed to the cloud.
* Is portable (and can be run on any OS).
* Is isolated from other containers and runs its own software, binaries, configurations, etc.

## [What is an image?](https://docs.docker.com/get-started/#what-is-an-image)

A running container uses an isolated filesystem. This isolated filesystem is provided by an image, and the image must contain everything needed to run an application - all dependencies, configurations, scripts, binaries, etc. The image also contains other configurations for the container, such as environment variables, a default command to run, and other metadata

# What is a container?

A container is an isolated environment for your code. This means that a container has no knowledge of your operating system, or your files. It runs on the environment provided to you by Docker Desktop. Containers have everything that your code needs in order to run, down to a base operating system. You can use Docker Desktop to manage and explore your containers.

# How do I run a container?

**Get a Simple Application:**

**git clone** [**https://github.com/docker/welcome-to-docker**](https://github.com/docker/welcome-to-docker)

**Build your First Image:**

cd path\_of\_the\_directory

**cd /path/to/welcome-to-docker/**

docker build -t directory\_name .

**docker build -t welcome-to-docker . -> After this, an image will build.**

the **-t** flag tags your image with a name, welcome-to-docker in this case.

And the **.** lets Docker know where it can find the Dockerfile.

**Run the Container:**

docker run image\_code

**docker run 88a2f8f122143590f061523f3e06fc49b138d9cabbec18e83f994721285ae419**

### **Run a Container Under a Specific Name:**

Docker container run --name give\_the name\_of\_the\_container image\_code

**docker container run --name container\_instance sha256:88a2f8f122143590f061523f3e06fc49b138d9cabbec18e83f994721285ae419**

**To see the Container List:**

**sudo docker ps -a // All List**

**To See the List of Images:**

**docker image ls**

**Tag Images: (Create new Tags)**

docker image tag name:prev\_tag name:new\_tag

**docker image tag docker-gs-ping:latest docker-gs-ping:v1.0**

**Remove Tags:**

docker image rm name:tag

**docker image rm docker-gs-ping:v1.0**

**Image with Multistage:**

docker build -t name:multistage -f Dockerfile.multistage .

**docker build -t docker-gs-ping:multistage -f Dockerfile.multistage .**

**docker image ls**

**To run an image inside of a container:**

docker run directory\_name

**docker run docker-gs-ping**

**Make a GET Request:**

**curl http://localhost:8080/**

curl: (7) Failed to connect to localhost port 8080: Connection refused

Your curl command failed because the connection to your server was refused. Meaning that you weren't able to connect to localhost on port 8080.

This is expected because your container is running in isolation which includes networking. Stop the container and restart with port 8080 published on your local network.

**Publish Port for the Container:**

docker run --publish port\_number:port\_number directory\_name

**docker run --publish 8080:8080 docker-gs-ping**

**Run in Detached Mode:**

docker run -d -p port\_number:port\_number directory\_name

**docker run -d -p 8080:8080 docker-gs-ping**

**List of Containers which are Running on Machine:**

**docker ps**

**To stop the Running Docker:**

Docker stop container\_Name/container\_ID

**docker stop 92a1586c957a**

### 

### **Run a Container in the Background (Detached Mode):**

sudo docker container run -d image\_code

**sudo docker container run -d sha256:88a2f8f122143590f061523f3e06fc49b138d9cabbec18e83f994721285ae419**

The output you receive will be similar to the one you see in the image above. The container will run the process and then stop. No other output will display inside the terminal session.

# Docker

**Docker is an open platform for developing, shipping, and running applications.**

**Docker allows you to separate your applications from your infrastructure so you can deliver software quickly. With Docker, you can manage your infrastructure in the same ways you manage your applications.**

**By taking advantage of Docker’s methodologies for shipping, testing, and deploying code quickly, you can significantly reduce the delay between writing code and running it in production.**

## [The Docker platform](https://docs.docker.com/get-started/overview/#the-docker-platform)

**Docker provides the ability to package and run an application in a loosely isolated environment called a container. The isolation and security lets you run many containers simultaneously on a given host. Containers are lightweight and contain everything needed to run the application, so you don't need to rely on what's installed on the host. You can share containers while you work, and be sure that everyone you share with gets the same container that works in the same way.**

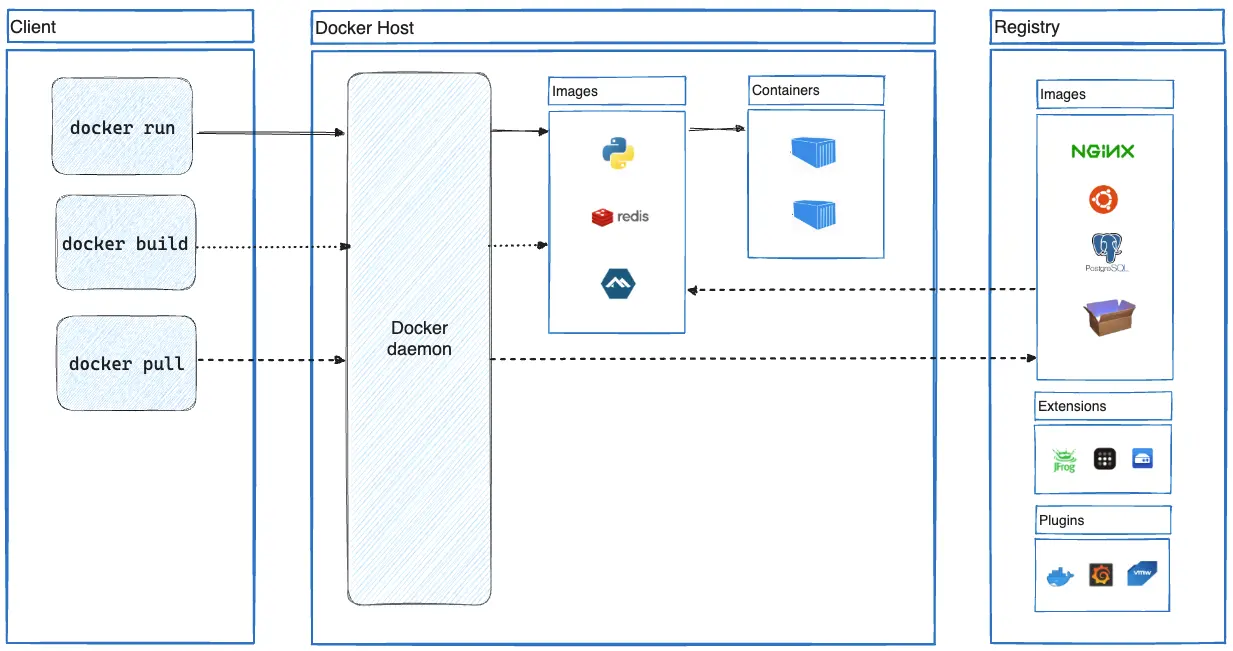
## [What can I use Docker for?](https://docs.docker.com/get-started/overview/#what-can-i-use-docker-for)

### [Fast, consistent delivery of your applications](https://docs.docker.com/get-started/overview/#fast-consistent-delivery-of-your-applications)

### [Responsive deployment and scaling](https://docs.docker.com/get-started/overview/#responsive-deployment-and-scaling)

### [Running more workloads on the same hardware](https://docs.docker.com/get-started/overview/#running-more-workloads-on-the-same-hardware)

**Docker Architecture:**

****

### [The Docker daemon](https://docs.docker.com/get-started/overview/#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](https://docs.docker.com/get-started/overview/#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 Desktop](https://docs.docker.com/get-started/overview/#docker-desktop)

**Docker Desktop is an easy-to-install application for your Mac, Windows or Linux environment that enables you to build and share containerized applications and microservices.**

**Docker Desktop includes the Docker daemon (dockerd), the Docker client (docker), Docker Compose, Docker Content Trust, Kubernetes, and Credential Helper.**

### [Docker registries](https://docs.docker.com/get-started/overview/#docker-registries)

**A Docker registry stores Docker images. Docker Hub is a public registry that anyone can use, and Docker looks 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, Docker pulls the required images from your configured registry. When you use the docker push command, Docker pushes your image to your configured registry.**

### [Docker objects](https://docs.docker.com/get-started/overview/#docker-objects)

**When you use Docker, you are creating and using images, containers, networks, volumes, plugins, and other objects.**

#### [Images](https://docs.docker.com/get-started/overview/#images)

**An image is a read-only template with instructions for creating a Docker container. Often, an image is based on another image, with some additional customization. For example, you may build an image which is based on the ubuntu image, but installs the Apache web server and your application, as well as the configuration details needed to make your application run.**

**You might create your own images or you might only use those created by others and published in a registry. To build your own image, you create a Dockerfile with a simple syntax for defining the steps needed to create the image and run it. Each instruction in a Dockerfile creates a layer in the image. When you change the Dockerfile and rebuild the image, only those layers which have changed are rebuilt. This is part of what makes images so lightweight, small, and fast, when compared to other virtualization technologies.**

#### [**Containers**](https://docs.docker.com/get-started/overview/#containers)

**A container is a runnable instance of an image. You can create, start, stop, move, or delete a container using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state.**

**By default, a container is relatively well isolated from other containers and its host machine. You can control how isolated a container's network, storage, or other underlying subsystems are from other containers or from the host machine.**

**A container is defined by its image as well as any configuration options you provide to it when you create or start it. When a container is removed, any changes to its state that aren't stored in persistent storage disappear.**

# Containerize an application

**Get THe App:**

**git clone https://github.com/docker/getting-started-app.git**

**Build The App’s Image:**

**cd getting-started-app**

**touch Dockerfile**  // Empty File

**Nano Dockerfile**

[**FROM**](https://docs.docker.com/engine/reference/builder/#from) **node:18-alpine**

[**WORKDIR**](https://docs.docker.com/engine/reference/builder/#workdir) **/app**

[**COPY**](https://docs.docker.com/engine/reference/builder/#copy) **. .**

[**RUN**](https://docs.docker.com/engine/reference/builder/#run) **yarn install --production**

[**CMD**](https://docs.docker.com/engine/reference/builder/#cmd) **["node", "src/index.js"]**

[**EXPOSE**](https://docs.docker.com/engine/reference/builder/#expose) **3000**

// Save this to Dockerfile

**docker build -t getting-started .** // will build the image.

Here, **-t** => is a flag which tags the image.

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:**

**docker run -dp 127.0.0.1:3000:3000 getting-started**

The **-d** flag (short for --detach) runs the container in the background. This means that Docker starts your container and returns you to the terminal prompt. You can verify that a container is running by viewing it in Docker Dashboard under Containers, or by running docker ps in the terminal.

The **-p** flag (short for --publish) creates a port mapping between the host and the container. The -p flag takes a string value in the format of HOST:CONTAINER, where HOST is the address on the host, and CONTAINER is the port on the container. The command publishes the container's port 3000 to 127.0.0.1:3000 (localhost:3000) on the host. Without the port mapping, you wouldn't be able to access the application from the host.

# Update the application

**Update the Source COde:**

**Change the Line 56**

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

**docker build -t getting-started .**  // Build the Updated version of the Image.

**docker run -dp 127.0.0.1:3000:3000 getting-started**

**//** Error because the old container is running and using the same port.

// To run the updated one on the same port, you need to remove the old one.

**Remove the old Container:**

docker ps

docker stop <the-container-id> // Stop the running container

docker rm <the-container-id> // Remove the running Container

**Start the Updated Container:**

**docker run -dp 127.0.0.1:3000:3000 getting-started**

# Share the application

**Create a Repository:**

1. [**Sign up**](https://www.docker.com/pricing?utm_source=docker&utm_medium=webreferral&utm_campaign=docs_driven_upgrade) **or Sign in to** [**Docker Hub**](https://hub.docker.com/)**.**
2. **Select the Create Repository button.**
3. **For the repository name, use getting-started. Make sure the Visibility is Public.**
4. **Select Create.**

**Push The Image:**

**docker push user\_name/getting-started**

// May Fail, was looking for image, but there is no image

**docker login -u USER-NAME** // Sign in to Docker Hub

**docker tag getting-started YOUR-USER-NAME/getting-started**

**//** Use the docker tag command to give the getting-started image a new name

**docker push YOUR-USER-NAME/getting-started**

**//** Now run the docker push command again. If you're copying the value from Docker Hub, you can drop the tagname part, as you didn't add a tag to the image name. If you don't specify a tag, Docker uses a tag called latest.

# Persist the DB

**The Container Filesystem:**

When a container runs, it uses the various layers from an image for its filesystem. Each container also gets its own "scratch space" to create/update/remove files. Any changes won't be seen in another container, even if they're using the same image.

**docker run -d ubuntu bash -c "shuf -i 1-10000 -n 1 -o /data.txt && tail -f /dev/null"**

**//** Start an ubuntu container that will create a file named /data.txt with a random number between 1 and 10000.

**docker ps** // To see the Container ID

**docker exec <container-id> cat /data.txt**  // will show a Random Number.

// Docker Access Command will access the container

**docker run -it ubuntu ls /**

**// Start another container(the same image) & don’t have the same file. There is no data.txt file.**

[**Volumes**](https://docs.docker.com/storage/volumes/) provide the ability to connect specific filesystem paths of the container back to the host machine. If you mount a directory in the container, changes in that directory are also seen on the host machine. If you mount that same directory across container restarts, you'd see the same files.

**Why Do We Need Docker?**

If we store data in a container it stores data in a virtual filesystem. But, there is no persistence. Data is gone when we restart or remove the container.

In docker volume, a folder in the physical host file system is **mounted** into the virtual filesystem of Docker.

Mounted means->mapped or created a line/connection with each other.

**Docker Volume List: docker volume ls**

**Create a volume and Start the Container:**

**docker volume create todo-db** // will create volume named todo-db

**docker stop container\_id**

**docker rm container\_id**

// May be still running without using the persistent volume. That’s why stop & remove.

**docker run -dp 127.0.0.1:3000:3000 --mount type=volume,src=todo-db,target=/etc/todos getting-started**

// Start the todo app container, but add the --mount option to specify a volume mount. Give the volume a name, and mount it to **/etc/todos** in the container, which captures all files created at the path.

**Verify Data Persist:**

Add Something -> Then Remove Container -> Start New Container

**Dive Into the Volume:**

docker volume inspect volume\_name

**docker volume inspect todo-db**

**Execute a Command inside a running Docker Container:**

**docker exec -it container\_id sh**

**docker exec -it b5b752697713 sh**

# Use bind mounts

A bind mount is another type of mount, which lets you share a directory from the host's filesystem into the container. When working on an application, you can use a bind mount to mount source code into the container. The container sees the changes you make to the code immediately, as soon as you save a file. This means that you can run processes in the container that watch for filesystem changes and respond to them.

**docker run -it --mount type=bind,src="$(pwd)",target=/src ubuntu bash**

// The --mount option tells Docker to create a bind mount, where src is the current working directory on your host machine (getting-started-app), and target is where that directory should appear inside the container (/src).

**Run App in a development Container:**

**docker run -dp 127.0.0.1:3000:3000 \**

**-w /app --mount type=bind,src="$(pwd)",target=/app \**

**node:18-alpine \**

**sh -c "yarn install && yarn run dev"**

* **-dp 127.0.0.1:3000:3000 - same as before. 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**
* **--mount type=bind,src="$(pwd)",target=/app - bind mount the current directory from the host into the /app directory in the container**
* **node:18-alpine - the image to use. Note that this is the base image for your app from the Dockerfile**
* **sh -c "yarn install && yarn run dev" - the command. You're starting a shell using sh (alpine doesn't have bash) and running yarn install to install packages and then running yarn run dev to start the development server. If you look in the package.json, you'll see that the dev script starts nodemon.**

**Container Logs:**

**docker logs -f <container-id**

# Multi container apps

## [Container networking](https://docs.docker.com/get-started/07_multi_container/#container-networking)

**Remember that containers, by default, run in isolation and don't know anything about other processes or containers on the same machine.**

**So, how do you allow one container to talk to another? The answer is networking. If you place the two containers on the same network, they can talk to each other.**

**Start MySQL (One Container):**

**docker network create todo-app**  // Create the Network

**docker run -d \**

**--network todo-app --network-alias mysql \**

**-v todo-mysql-data:/var/lib/mysql \**

**-e MYSQL\_ROOT\_PASSWORD=secret \**

**-e MYSQL\_DATABASE=todos \**

**mysql:8.0**

// Start a MYSQL container & attach it to the Network. Also some environment variables are defined that the database will use to initialize the database.

**docker exec -it <mysql-container-id> mysql -u root -p**

Password: **secret**

**SHOW DATABASES;**

**Exit or Ctrl+d**

**Connect To MySQL:**

**docker run -it --network todo-app nicolaka/netshoot**

// Start a new container using the nicolaka/netshoot image. Make sure to connect it to the same network

// The "nicolaka/netshoot" image provides a collection of networking tools and utilities useful for debugging and troubleshooting network-related issues.

**dig mysql** // dig command which is a DNS Tool

// You're going to look up the IP address for the hostname mysql.

**Run App with MySQL:**

**docker run -dp 127.0.0.1:3000:3000 \**

**-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:18-alpine \**

**sh -c "yarn install && yarn run dev"**

// Specify each of the previous environment variables, as well as connect the container to your app network. Make sure that you are in the getting-started-app directory when you run this command.

**docker exec -it <mysql-container-id> mysql -p todos**

// This command allows you to interact with the MySQL server inside the specified container.

// it's connecting to the "todos" database.

# Use Docker Compose

[**Docker Compose**](https://docs.docker.com/compose/) **is a tool that helps you define and share multi-container applications.**

**With Compose, you can create a YAML file to define the services and with a single command, you can spin everything up or tear it all down.**

**The big advantage of using Compose is you can define your application stack in a file, keep it at the root of your project repository (it's now version controlled), and easily enable someone else to contribute to your project.**

**Someone would only need to clone your repository and start the app using Compose.**

**YAML** is a human-readable data serialization language that is often used for writing configuration files. Depending on whom you ask, YAML stands for **yet another markup language** or YAML ain’t markup language (a recursive acronym), which emphasizes that YAML is for data, not documents.

YAML is a popular programming language because it is designed to be easy to read and understand. It can also be used in conjunction with other programming languages. Because of its flexibility, and accessibility, YAML is used by [Ansible®](https://www.redhat.com/en/technologies/management/ansible/compare-awx-vs-ansible-automation-platform) to create automation processes, in the form of [Ansible Playbooks](https://www.redhat.com/en/topics/automation/what-is-an-ansible-playbook).

**Define The App Service:**

**touch compose.yaml**

**code compose.yaml**

**—----------------------------------------------------------------------**

**services:**

**app:**

**image: node:18-alpine**

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

**ports:**

**- 127.0.0.1:3000:3000**

**working\_dir: /app**

**volumes:**

**- ./:/app**

**environment:**

**MYSQL\_HOST: mysql**

**MYSQL\_USER: root**

**MYSQL\_PASSWORD: secret**

**MYSQL\_DB: todos**

**mysql:**

**image: mysql:8.0**

**volumes:**

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

**environment:**

**MYSQL\_ROOT\_PASSWORD: secret**

**MYSQL\_DATABASE: todos**

**volumes:**

**todo-mysql-data:**

**—------------------------------------------------------------------------**

**Run the Application Stack:**

docker rm -f container\_ID

**docker rm -f 601d5a282b60**

**docker compose up -d**

// Start up the application stack using the docker compose up command. Add the -d flag to run everything in the background.

**docker compose logs -f**

// You'll see the logs from each of the services interleaved into a single stream. This is incredibly useful when you want to watch for timing-related issues.

**docker compose logs -f app**

// To see the log of specific services like only app.

**Tear it all Down:**

**docker compose down**  // To remove all , but don’t remove volumes.

**docker compose down --volumes** // To remove Volumes.

## [Image layering](https://docs.docker.com/get-started/09_image_best/#image-layering)

docker image history image\_name

**docker image history getting-started**

// you can see the command that was used to create each layer within an image.

// Each of the lines represents a layer in the image. The display here shows the base at the bottom with the newest layer at the top. Using this, you can also quickly see the size of each layer, helping diagnose large images.

**docker image history --no-trunc getting-started**  // To see Full Output

## [Layer caching](https://docs.docker.com/get-started/09_image_best/#layer-caching)

**Update the Dockerfile to copy in the package.json** // update Dockerfile

**touch .dockerignore** // Create this File

**docker build -t getting-started .**

**make a change to the src/static/index.html file**

**docker build -t getting-started .**  // This time faster.

// First off, you should notice that the build was much faster.

And, you'll see that several steps are using previously cached layers.

Pushing and pulling this image and updates to it will be much faster as well.

## [Multi-stage builds](https://docs.docker.com/get-started/09_image_best/#multi-stage-builds)

**Multi-stage builds are an incredibly powerful tool to help use multiple stages to create an image. There are several advantages for them:**

* **Separate build-time dependencies from runtime dependencies**
* **Reduce overall image size by shipping only what your app needs to run**

**Every stage start with From Statements**

**Every stage generates its own container image**

**By default the image from the last stage is committed in the local registry.**

**Can copy data between stages.**

**Video: Docker & Linux Namespace**

In computing, a **system call** is a programmatic way in which a computer program requests a service from the kernel of the operating system it is executed on.

A system call is a way for programs to **interact with the operating system**.

A computer program makes a system call when it makes a request to the operating system’s kernel. System call **provides** the services of the operating system to the user programs via Application Program Interface(API).

It provides an interface between a process and an operating system to allow user-level processes to request services of the operating system. System calls are the only entry points into the [kernel](https://www.geeksforgeeks.org/kernel-in-operating-system/) system.

All programs needing resources must use system calls.

A system call is a mechanism used by programs to request services from the [operating system](https://www.geeksforgeeks.org/what-is-an-operating-system/) (OS). In simpler terms, it is a way for a program to interact with the underlying system, such as accessing hardware resources or performing privileged operations.

**Namespaces** are a feature of the [Linux kernel](https://en.wikipedia.org/wiki/Linux_kernel) that partition kernel resources such that one set of [processes](https://en.wikipedia.org/wiki/Process_(computing)) sees one set of resources while another set of processes sees a different set of resources. The feature works by having the same namespace for a set of resources and processes, but those namespaces refer to distinct resources. Resources may exist in multiple spaces. Examples of such resources are process IDs, host-names, user IDs, file names, some names associated with network access, and [Inter-process communication](https://en.wikipedia.org/wiki/Inter-process_communication).

Docker uses a technology called namespaces to provide the isolated workspace called the container. When you run a container, Docker creates a set of namespaces for that container.

These namespaces provide a layer of isolation. Each aspect of a container runs in a separate namespace and its access is limited to that namespace.

While Docker is a container runtime,

**Kubernetes** is a platform for running and managing containers from many container runtimes.

Kubernetes supports numerous container runtimes including Docker, containerd, CRI-O, and any implementation of the Kubernetes CRI (Container Runtime Interface).

#### **What's the advantage of Kata Containers?**

Kata Containers perform like containers, but provide the workload isolation and security advantages of VMs. It combines the benefits of containers and VMs.

Builder

Compose -

config

Container

Exec

Image

Network

System

Volume