*Short Circuit*

# Docker

Docker Deep Dive
by Nigel Poulton
2024 Edition
Up until Chapter 10: Docker Swarm

Summary by Emiel Bos

## 1 Intro

Docker made Linux containers easy and accessible. Because containers share the kernel of the host they're running on, containerized Windows apps need a host with a Windows kernel, whereas containerized Linux apps need a host with a Linux kernel. Windows systems can also run Linux containers via the WSL 2 (Windows Subsystem for Linux) subsystem, meaning Windows 10 and 11 are great platforms for developing and testing both Windows and Linux containers. However, despite Microsoft hard work to bring Docker and container technologies to the Windows platform, almost all containers are Linux containers, because they are smaller and faster, and more tooling exists for Linux. There is no such thing as Mac containers, but Macs are great platforms for working with containers. All of the examples in this edition of the book are Linux containers.

Docker, Inc. is the company that created and continues to develop the Docker platform. It is based in Palo Alto and founded by French- born American developer and entrepreneur Solomon Hykes, who is no longer at the company. The company started as a platform-as-a-service (PaaS) provider called dotCloud. Behind the scenes, dotCloud delivered their services on top of containers and had an in-house tool to help them deploy and manage those containers, which they called Docker. In 2013, dotCloud dropped the struggling PaaS side of the business, rebranded as Docker, Inc., and focussed on bringing Docker and containers to the world. The word Docker is a British expression meaning dock worker that refers to a person who loads and unloads cargo from ships.

There are several important standards and governance bodies influencing the development of the container ecosystem:

- The Open Container Initiative (OCI) is a governance council responsible for low-level container-related standards. It operates under the umbrella of the Linux Foundation and was founded in the early days of the container ecosystem when some of the people at a company called CoreOS didn't like the way Docker was dominating the ecosystem. In response, CoreOS created an open standard called appc that defined specifications for things such as image format and container runtime, and a reference implementation called rkt (pronounced "rocket"). While competition is usually a good thing, competing standards are generally bad, as they generate confusion that slows down user adoption. Fortunately, the main players in the ecosystem formed the OCI as a vendor-neutral lightweight council to govern container standards. They currently maintain three standards called *specs*: the image-spec, the runtime-spec, and the distribution-spec. All modern versions of Docker implement all three OCI specs. Docker, Inc. and many other companies have people on the technical oversight board (TOB) of the OCI.
- The Cloud Native Computing Foundation (CNCF) is another Linux Foundation project that is influential in the container ecosystem, founded in 2015. Instead of creating and maintaining container-related specifications, the CNCF provides the space, structure, and support for important projects to grow and mature, such as Kubernetes, containerd, Notary, Prometheus, Cilium, etc. All CNCF projects pass through the following three phases or stages: sandbox, incubating, and graduated. Each phase increases a project's maturity level by requiring higher standards of governance, documentation, auditing, contribution tracking, marketing, community engagement, etc. The CNCF helps with all of that.
- The Moby Project was created by Docker as a community-led place for developers to build specialized tools for

building container platform. Platform builders can compose their platforms from a mix of Moby tools, in-house tools, and tools from other projects. The Moby project now has members including Microsoft, Mirantis, and Nvidia.

Docker Desktop is a desktop app from Docker, Inc. and is the best way to work with containers. It includes the Docker Engine, GUI, all the latest plugins and features, an extension system with a marketplace, Docker Compose, and a single-node Kubernetes cluster for learning Kubernetes. It's free for personal use and education. Docker Desktop on Windows 10 and 11 supports Windows containers and Linux containers. Docker Desktop on Mac and Linux only support Linux containers. Alternatively, you can spin up a local VM running Docker with Multipass, and there are lots of ways to install Docker on Linux servers. These give you access to most of the free Docker features but lack some of the features of Docker Desktop.

## 1.1 Dev perspective

A Dockerfile in a repository is a plain-text document that tells Docker how to build, that is, *containerize* the app and dependencies into an image, which you can do with the `docker build -t <image>:<version>` command.

## 1.2 WebAssembly

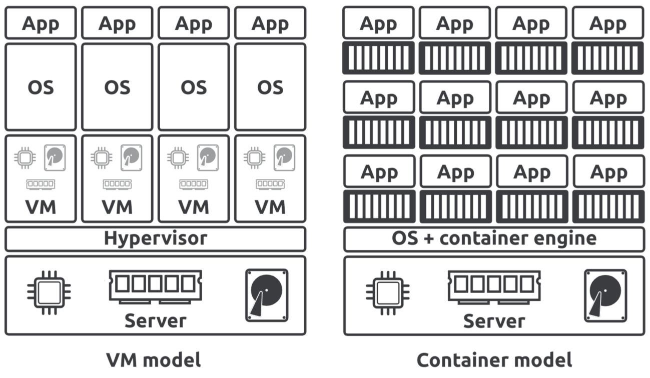
*WebAssembly* (Wasm) can be seen as the third wave of cloud computing. It is a modern binary instruction set that builds applications that are smaller, faster, more secure, and more portable than containers. Wasm apps can be written in any language, and when compiled as a Wasm binary, will run anywhere you have a Wasm runtime. However, Wasm apps have many limitations, and the standards are still being developed. The container ecosystem is also much richer and more mature. Docker and the container ecosystem are adapting to work with Wasm apps.

## 1.3 Kubernetes

Kubernetes (K8) is the industry standard platform for deploying and managing (orchestrating) containerized apps. Older versions of Kubernetes used Docker to start and stop containers. However, newer versions use **containerd**, which is a stripped-down version of Docker optimized for use by Kubernetes and other platforms. All Docker containers work on Kubernetes.

## 1.4 Containerization vs. virtualization

In the VM model, you power on a server and a hypervisor boots, claiming all hardware resources such as CPU, RAM, storage, and network adapters. To deploy an app, you ask the hypervisor to create a VM, which does this by carving up the hardware resources into virtual versions, such as virtual CPUs and Virtual RAM, and packaging them into a VM that looks exactly like a physical server on which you can install an OS and an app. In short, hypervisors perform hardware virtualization, where they divide hardware resources into virtual versions and package them as VMs.
In the container model, you power on the same server and an OS boots and claims all hardware resources. You then install a container engine such as Docker. To deploy an app, you ask Docker to create a container, which does this by carving up OS resources such as process trees and filesystems into virtual versions and then packaging them as a container that looks exactly like a regular OS. In short, container engines perform OS virtualization, where they divide OS resources into virtual versions and package them as containers.



Before virtualization and containerization, every time a business needed a new application, it had to buy a new server, and companies had to guess capacity, resulting in racks of overpowered servers. Then, VMware introduced *virtualization* and the *virtual machine* (VM), allowing multiple business applications to run on a single server. However, every VM needs its own dedicated OS, each of which consumes CPU, RAM, and other resources, and needs patching and monitoring. *Containerization* was therefore introduced, starting in the Linux world. Every container shares the OS of the host it's running on, meaning a single host can run more containers than VMs. Lastly, containers are designed to be stateless and ephemeral, whereas VMs are designed to be long-running and can be migrated with their state and data. Containers are also faster (because the host's OS is already booted) and more portable than VMs.

## 2 Engine

There are two major parts to the Docker platform:

- The CLI (client) is the familiar `docker` command-line tool for deploying and managing containers. It converts commands into API requests and sends them to the engine's daemon. The client hides the engine's complexity.
- The engine (server) comprises all the server-side components that run and manage containers.

The *Docker Engine* is jargon for the server-side components of Docker that run and manage containers. It is modular and built from many small specialized components pulled from aforementioned projects. Initially, the Docker Engine had two major components: the (Docker) daemon, which was a monolithic binary containing all the code for the API, image builders, container execution, volumes, networking, etc.; and LXC, which did the hard work of interfacing with the Linux kernel and constructing the required namespaces and cgroups to build and start containers. LXC had several downsides: it's Linux-specific while Docker had aspirations of being multi-platform, and there was no way of ensuring LXC evolved in the fast-paced ways Docker needed. A long time ago already, Docker replaced LXC with its own tool, libcontainer, which was to be a platform- agnostic tool that gave Docker access to the fundamental container building blocks in the host kernel.

The monolithic daemon also got slower, wasn't what the ecosystem wanted, and was hard to innovate with, so that also got broken apart and refactored so that every feature became its own small specialized tool. Platform builders could then re-use these tools to build other platforms. The daemon now solely exposes the API for the client, either on a local socket or over the network, and it communicates with containerd via a CRUD-style API over gRPC.[1] Moving the container creation and management from the Docker daemon to containerd and runc makes it possible to stop, restart, and even update the daemon without impacting running containers, referred to as *daemonless containers*.

#### 2.0.1 containerd

The high-level (container lifecycle management) was reimplemented as a separate tool called containerd.[2] It has grown to also include the ability to manage images, networks, and volumes. containerd and runc are the default in Docker and Kuberneter. However, containerd uses shims[3] that make it possible to replace runc with other low-level runtimes. Despite its name, even containerd cannot create containers. It converts the required Docker image into an OCI bundle and tells runc to use this to create a new container.

### 2.1 runc

The low-level runtime functionality (container lifecycle event execution) was reimplemented as runc (pronounced "run see"), which is the reference implementation of the OCI runtime-spec (it is said to operate at the OCI layer), and is a lightweight CLI wrapper for libcontainer with which you can manage OCI-compliant containers.[4] It interfaces with the kernel to actually build and delete containers. However, on its own, it's a very low-level tool and lacks almost all of the features and add-ons of the Docker Engine. The container is started as a child process of runc, and as soon as the container starts, runc exits. The shim becomes the container's parent process, and communicated with containerd, reports on the container's status and performs low-level tasks.

## 3 Images

*Images*, or container, Docker, or OCI images, are read-only objects/files that contain everything an app needs to run, including enough of an OS, a filesystem, the application code, all dependencies, and metadata. The `docker images` command lists all the locally available images, in the *local repository*. Copying new images onto your Docker host is

[1] The docker daemon is in the `/usr/bin/dockerd` binary on Linux.
[2] containerd is in the `/usr/bin/containerd` binary on Linux.
[3] The default containerd-runc shim is in the `/usr/bin/containerd-shim-runc-v2` binary on Linux.
[4] runc is in the `/usr/bin/runc` binary on Linux.

called *pulling* and is done e.g. with `docker pull [user_or_org/]<image>[:tag]`. The user or organization name doesn't need to be specified for *official repositories*. We will omit `user_or_org` and `tag` for brevity from here on out, but know that `<image>` actually is `[user_or_org/]<image>[:tag]`. You remove images from your local repository with `docker rmi <image>`, where `<image>` can be a name, short ID, or SHA-256 hash. Tags are often used to indicate version numbers. The default tag is `latest` (are not guaranteed to actually be the most up-to-date in the repository), and the default remote *image registry* is Docker Hub, which is the most common registry, but Docker works with all other registries. Registries contain one or more *image repositories*, which in turn contain one or more images/versions. A single image can have multiple tags. To pull an image from a different registry, you just add the registry's DNS name before the repository name.
Tags are error-prone, because they're mutable. An image can be tagged incorrectly, or a tag can be reused, which can lead to uncertainty about which version of an image is deployed. Fortunately, image versions can also be identified by their *image digest*, which is a hash of the image manifest file. `docker images --digests` lists your images with their digest, and you can also `docker pull <image>@sha256<digest>`. Note that besides this *content hash*, Docker uses the *distribution hash* – a hash of the image after compression for transfer – to check integrity of the image, so CLI output may display different hashes.

Docker (container) images are used to create/instantiate Docker containers, where the image serves as a blueprint or snapshot, and a container is a runtime instance of an image. (Analogously to class $\rightarrow$object.) Images should only contain application code and dependencies, not things as build tools or troubleshooting tools. This allows image sizes to be relatively small, in the order of MBs. *Slim images* are those that don't even include a shell or a package manager – if the application doesn't need it at run-time. Another thing that keeps images small is the lack of an OS kernel – most images only have filesystem objects – because containers use the kernel of the host they're running on. Unfortunately, Windows images can be huge, in the order of GBs.

Images consists of read-only, independent layers (e.g. a layer for OS components, another for dependencies, etc.), each has an ID and each consists of files. The image is just a manifest file with metadata specifying the required layers and their order. These layers are being pulled individually when you `docker pull`, you can see their digests (layers also have a digest, a hash of their content) with `docker inspect <image>:<version>`, along with a wealth of other image-related metadata. All Docker images start with a *base layer*, and every time you add new content, Docker adds a new layer on top of it. Layers can have files that are updated versions of layers below them, i.e. you can update the file in an image by adding new layers. Under the hood, Docker uses storage drivers to stack layers and present them as a unified filesystem and image. Almost all Docker setups use the overlay2 driver, but zfs, btrfs, and vfs are alternative options (all offer the same dev and user experience). Distinct images can share layers, leading to efficiencies in space and performance, and `docker pull` will skip downloading layers that are already locally present.

Since Docker supports multiple versions of the same image for different platforms and architectures, the registry API supports *manifest lists*, which is a list of manifests, one for each architectures supported by an image tag. You can print the manifest list with `docker buildx imagetools inspect <image>` or `docker manifest inspect <image>`. You can do a `docker pull` on any architecture and get the correct version of the image (if it's supported). You can create multi-architecture images with the `docker buildx` command. `docker buildx` is a Docker CLI plugin that works with Docker's latest build engine features. There are two modes:

- *Emulation mode* performs builds for different architectures on your local machine by running the build inside a QEMU virtual machine emulating the target architecture. It works most of the time but is slow and doesn't have a shared cache.
- *Build Cloud* is a paid subscription service from Docker, Inc. that performs builds in the cloud on native hardware without requiring emulation. It's very fast, lets you share a common build cache with teammates, and is seamlessly integrated into Docker Desktop and any version of the Docker Engine using a version of buildx supporting the cloud driver. It also integrates with GitHub actions and other CI solutions.

Docker Scout scans images for known vulnerabilities and provides remediation recommendations. It requires a Docker subscription and is integrated into the docker CLI, Docker Hub, and Docker Desktop.

## 4 Containers

Images are build-time constructs, whereas containers are run-time constructs. Containers are designed to be immutable; you shouldn't change them after you've deployed them, and you replace it with a new one if it fails instead of connecting to it and making the fix in the live instance. Containers should only run a single process and we use them to build microservices, so an application with four features (microservices) will have four containers.

You use the `docker run` command to start a container. The Docker client converts the command into an API request to the Docker API exposed by the Docker daemon, which searched first searches its local image repository for the specified image, and then Docker Hub (or some other configured registry).

```
docker run --name <container> [-it] [-p <system_port>:<container_port>] [-d]
    [user_org/]<image>[:version] [bash]
```

The `-it` flags tells Docker to make the container interactive and to attach your shell to the container's terminal. You can map ports of your local system to ports inside the container with the `-p` option. The `-d` flag tells Docker to run it in the background as a daemon process and detached from your local terminal. The last term tells Docker to start a Bash shell as the container's main app. Press Ctrl-PQ to exit the container without terminating it. A container is started from an image by creating a thin read-write layer and placing it on top of the (read-only) image. The container can see and access the files and apps in the image through their own R/W layer, and any changes get written to the R/W layer. When you stop a container (`docker stop <container>`), Docker keeps the R/W layer and restores it when you restart the container (`docker restart <container>`). When you delete/kill a container (`docker rm <container> [-f]`), Docker deletes the R/W layer. The `-f` flag forces the operation and doesn't allow the app the usual 10-second grace period to flush buffers and gracefully quit.

The `docker ps` command lists all running containers; adding the `-a` flag lists all containers, so also stopped ones. The `docker attach <container>` command attaches your shell to the specified container's main process again. Once the container is running, the image and the container are bound, and you cannot delete the image until you stop and delete the container.

There are three ways you can tell Docker how to start an app in a container:

- An Entrypoint instruction in the image. Optional image metadata that store the command Docker uses to start the default app. Cannot be overridden on the CLI; CLI arguments will be appended to the Entrypoint instruction as an argument. You can check this metadata with `docker inspect`.
- A Cmd instruction in the image. Optional image metadata that store the command Docker uses to start the default app. Can be overridden on the CLI; CLI arguments will override Cmd instructions. You can check this metadata with `docker inspect`.
- A CLI argument. Supplied at the end of a `docker run` command (i.e. after the image specification). Needed if the image has neither of the above.

You can use the `docker exec` command to execute commands in running containers. It has two modes:

- Interactive exec sessions connect your terminal to a newly created shell process in the container and behave like remote SSH sessions: `docker exec -it <container> sh`. sh is a minimal shell program installed in the container. Not all Linux commands are available, because container images are usually optimized to be lightweight and don't have all of the normal commands and packages installed. You can run a `ps` command to see the processes running in your container.
- Remote execution mode lets you send commands to a running container and prints the output to your local terminal: `docker exec <container> <command>`, i.e. without the `-it` flag.

Containers only run while their main process is running; killing the main process also kills the container.

You can also use `docker inspect <container>`.

Docker Debug is a new tool and requires a Pro, Team, or Business subscription. Docker Debug works by attaching a shell to a container and mounting a toolbox loaded with debugging tools. This toolbox is mounted as a directory called `/nix` and is available during your debugging session but is never visible to the container. While debugging a running container, any changes are immediately visible to the container and persist across container restarts. While debugging an image or stopped container, the Docker Debug session creates a debug sandbox and adds it to the image as a R/W layer to make it feel like a running container, but changes don't persist. The command is `docker debug <image>|<container>`. You can use Docker Debug's built-in `install` command to add any package listed on `search.nixos.org`.

For every container you can set a *restart policy* with the `--restart <policy>`, which is any of `no` (never restarts), `on-failure` (restarts on a non-zero exit code or when the daemon restarts), `always` (restarts on any exit code or when the daemon restarts), or `unless-stopped` (restarts on any exit code). No policy will restart a container when it is stopped with `docker stop` (except when the daemon restarts for policies which restart when the daemon restarts).

## 5 Containerization

Packaging apps as images is called *containerization*. For this, you need a `Dockerfile` in your *build context* (the folder containing the application source code and the files listing dependencies; generally the root folder in a repository), a text file containing instructions for building the image. A reference of all available command is here. An example for a simple Node.js web app:

```
ARG NODE_VERSION=20.8.0 # Define variable NODE_VERSION
FROM node:${NODE_VERSION}-alpine # Pull node image and use as base image (i.e. your layers are
    stacked on top)
ENV NODE_ENV production # Set environment variable to tells Node.js to run in production mode
WORKDIR /usr/src/app # Sets/changes the working directory for the remaining commands. Creates a
    small layer
RUN --mount=type=bind,source=package.json,target=package.json \
--mount=type=bind,source=package-lock.json,target=package-lock.json \
--mount=type=cache,target=/root/.npm \
npm ci --omit=dev # Executes any commands during build time in another new layer and commits the
    result. Commonly used to update packages and install dependencies
USER node # Sets the user name or ID to use as the default for the remainder of the current stage
    (i.e. for RUN instructions and at runtime, runs the relevant ENTRYPOINT and CMD commands). Here
    it is ensured that the app is ran as a non-root user
COPY <src> <dst> # Copies new files and directories from <src> and to the image at <dest>
EXPOSE 8080 # The container will listen on port 8080 at runtime. This doesn't actually publish the
    port or do anything else functional; it is purely for documentation and metadata
CMD node app.js # The command to be executed when running a container. Creates a layer. Usually
    used to provides default arguments. Here it specifies an executable, but if not, you have to
    define an...
ENTRYPOINT ["executable", "param1", "param2"] # Useful if you want to run the same executable every
    time in combination with CMD for default arguments
```

Docker parses the Dockerfile one line at a time. Instruction/step names are not case-sensitive, but all-caps is the convention. Instructions that add content (e.g. files and programs) create new layers:`FROM`, `RUN`, `COPY` and `WORKDIR`[5]. Instructions that don't add content only create metadata: `EXPOSE`, `ENV`, `CMD`, and `ENTRYPOINT`. Use `docker history <image>` to see the instructions that created it.

The `docker init` command analyses applications and automatically creates `Dockerfile`s that implement good practices. You can of course also create one manually.

Use `docker build -t <image> <build_context>`, where `image` will be the image's name and the `build_context` is the directory with your application files. This command is an alias for the `docker buildx build` command. Use `docker push <image>` to push the image. The `-t` flag tags the image. Docker uses an image tag with format `<repo>:image` to determine which registry and repository to push it to. Use `docker tag <current_tag> <new_tag>` to re-tag the image with the aforementioned format to specify an existing repository, which will create a new tag. By default, it's pushed to Docker Hub. Use `docker login` to login to Docker Hub. If you want to push it to a private registry, you have to prepend the tag with its domain name.

*Multi-stage builds* use a single Dockerfile with multiple `FROM` instructions, each of which represents a new build stage. You can have a stage where you build the app inside a large image with compilers and other build tools, and then copy the compiled app into a slim image for production in another stage. The builder can even run different stages in parallel for faster builds. The stages are numbered starting from 0, but you can give them a name by appending `AS <stage>` to the end of the `FROM` command. Each intermediate stage builds an image called `stage` that can be referenced in subsequent `FROM` commands, but Docker deletes them if they're not the final stage or a target stage. An example:

```
# Stage 0 builds a reusable image with (only) compilation tools, used only to compile the
    executables
FROM golang:1.22.1-alpine AS base # This image is over 300MB when uncompressed
WORKDIR /src
COPY go.mod go.sum . # These files list the application dependencies and hashes
RUN go mod download # Installs the dependencies
COPY . . # Copies the application source code into the image

# Stage 1 compiles the client executable
FROM base AS build-client <<---- Stage 1
RUN go build -o /bin/client ./cmd/client # Compile into a binary executable
```

[5]Older builders didn't create a layer for `WORKDIR` instructions. However, the instruction modifies filesystem permissions and the current builder creates a very small layer. This behavior may change in the future.

```
# Stage 2 compiles the server executable
FROM base AS build-server <<---- Stage 2
RUN go build -o /bin/server ./cmd/server # Compile into a binary executable

# Stage 3 copies the client executable into a slim image
FROM scratch AS prod-client # The scratch image is the most minimal image and the base ancestor for
    all other images
COPY --from=build-client /bin/client /bin/
ENTRYPOINT [ "/bin/client" ]

# Stage 4 copies the server executable into a slim image
FROM scratch AS prod-server
COPY --from=build-server /bin/server /bin/
ENTRYPOINT [ "/bin/server" ]
```

In this case, you use `docker build -t <image> --target <stage> -f <dockerfile> <build_context>` to specify the Dockerfile and target stage (e.g. `prod-client`; by default, the last stage is the target).

Behind the scenes, Docker's build system has a:

- Client: Buildx, which is the default build client implemented as a CLI plugin and supports all the latest features of BuildKit. You can configure Buildx to talk to multiple BuildKit instances.
- Server: BuildKit, each instance of which is called a *builder*. They can be on your local machine, in your cloud or datacenter, or Docker's Build Cloud. If you point buildx at a local builder, image builds will be done locally, and vice versa. Each builder has a *driver* through which Buildx communicates with it. Use `docker buildx ls` to list the builders you have configured on your system and the platforms they can build for. The one with the * is the default builder, which you can change with `docker buildx use <builder>`. Use `docker buildx inspect <builder>` to view it's information. You create a new builder with `docker buildx create --driver=<driver> --name=<builder>`. You can use the `docker-container` driver for a local builder, or the `cloud` driver for a cloud builder.
  BuildKit uses a cache for layers and other artifacts to speed up builds. The cache of local builders is only available to other builds on the same system, but an entire team can share the cache on Docker Build Cloud. Any time an instruction results in a cache miss, the cache is invalidated and no longer checked for the rest of the build. Therefore, as a best practice, you should put instructions most likely to invalidate the cache near the end of the Dockerfile. You can force a build to ignore the cache by running `docker build` with the `--no-cache` option.

You can use builders for multi-architecture builds. Use `docker buildx build --builder=<builder> --platform=<platform>[,plat`
`-t <image> <--push|--load> <build_context>`

## 6 Compose

*Microservices applications* are modern cloud-native applications that consist of lots of small services that work together, called *microservices*. Instead of hacking together complex scripts and long docker commands, Docker Compose lets you describe the application in a `compose.yaml` file, which you use with the `docker compose` command to deploy and manage the app. Docker Compose started as a Python too called Fig by a company called Orchard Labs. Docker, Inc. acquired Orchard Labs and rebranded Fig as Docker Compose.[6] There is also a Compose Specification defining an open, community-led standard for multi-container microservices apps. Docker Compose is the reference implementation. All modern versions of Docker come with Docker Compose pre-installed. An example `compose.yml`:

```
services: # Microservices are defined here
  web-fe: # Defines the web front-end microservice called "web-fe". Containers incorporate this name
    deploy:
      replicas: 1 # Number of identical containers to deploy. More then one wont work on Docker Desktop as only one container
          can use port 5001 on the DD host
    build: . # Build from the Dockerfile in the same directory
    command: python app.py # Execute this command when starting containers. Overrides commands set in Dockerfiles
    ports:
      - target: 8080 # Map port 8080 in the container ...
        published: 5001 # ... to port 5001 on the Docker host
    networks:
      - app-net # Attach container(s) to the "app-net" network. Should exist or be defined under "networks"
    volumes:
```

[6]They renamed the command-line tool from `fig` to `docker-compose`, and more recently they folded it into the docker CLI with its own `compose` sub-command.

```
            - type: volume
              source: app-vol # Should exist or be defined under "networks". Mount the "app-vol" volume ...
              target: /app # ... to "/app" in the containers for this service
    redis: # This block defines the Redis back-end microservice called "redis"
        deploy:
            replicas: 1
        image: "redis:alpine" # Pull the "redis:alpine" image for this service
        networks:
            app-net: # Connecting both services to the app-net network means they can resolve each other and communicate.
networks: # Networks are defined here
    app-net: # Defines a new network "app-net"
volumes: # Volumes are defined here
    app-vol: # Defines a new volume "app-vol"
```

You can deploy this with `docker compose up &`, shut it down with `docker compose down`, stop it with `docker compose stop` (this won't deleted any of its resources or containers), restart it with `docker compose restart`[7], list its containers with `docker compose ls`, and list the processes inside each container with `docker compose top`.[8]. If your Compose file is not called `compose.yml` you can specify if with the `-f` flag. You can use the `--detach` flag to deploy the app in the background. Since Compose is working with regular Docker constructs behind the scenes, you can use regular `docker` commands to check the deployment. Created (i.e. not pulled) images will be called `<build_context_dir_name>-<service_name>`, containers (from both created and pulled images) will be called `<build_context_dir_name>-<service_name>-<replica_num>`. Networks and volumes are named similarly. You can list networks and volumes with `docker network ls` and `docker volume ls`, respectively.
Docker decouples the lifecycle of volumes from the rest of the application, so `docker compose down` removed containers and networks, but not volumes, because there might be important information stored on there. You can also remove the volumes with the `--volumes` flag to `docker compose down`. Images are also not automatically deleted, such that subsequent deployments are faster; use the `--rmi all` flag to delete the images.

[7]If you make changes to the Compose file while it's stopped, these changes will not appear in the restarted app.
[8]The PID numbers returned are the PID numbers as seen by the Docker host (not from within the containers).