Docker Deep Dive

by @kinbiko
on 2025-02-15

» About this talk

* 🚫 Not a workshop.
  * Depending on your OS/CPU architecture, installation is different.
  * Certain commands demand a lot of bandwidth. If you want to play along, do so during the breaks.
* Split into 3 sections:
  * **Fundamentals**: History, how to run locally, `Dockerfile` basics.
  * **Running In production**: Security, multi-stage builds, and pro-tips.
  * **Under The Hood**: How Docker is composed and why it works.

» Who this is for

* 🙋 Questions are encouraged and rewarded.
* 🤓 Not afraid of a little Linux.
* 📈 Hopefully interesting for beginners as well as experienced Docker users.
  * Experts might be bored in the beginning.
  * Beginners might be bored at the end.

» Raise your hand if...

* You've never used Docker.
* You've used Docker but gave up before getting any value out of it.
* You've used `docker run`, but not much more.
* You've written a `Dockerfile` FROM scratch.
* You spotted the play-on-words in the previous point.
* You know what "multi-stage builds" mean, and you know how to do them.
* You know how to run your applications using Docker in production.
* You know how to run your application with Kubernetes (e.g. in EKS, GKE).
* The following all make sense: `cgroups`, Linux namespaces, union filesystems, `runc`.

» History: Shipping items pre-1956

» History: Shipping items pre-1956

* Longshoremen moved sacks/crates using nets, hooks, etc.
* Each item would generally be handled multiple times.
* Storage requirements different depending on the item. Expertise was necessary to avoid errors.
* Items were secured (against environmental damage, theft) individually, adding to weight and cost.
* Messy paperwork as consignment items might be scattered.
* Slow process = very expensive. Ships spent more time docked than at sea.

» History: Shipping Using Containers

» History: Shipping Using Containers

* Sealed Container Doctrine: Manufacturers load containers, and the customer unloads. Generally not opened in between.
* Containers are secure from the elements (waterproof) as well as theft (locks).
* Containers are designed to be stacked and stored on deck.
* Simpler documentation as multiple items are treated as one.
* Far less labor and time required in ports. Time spent mostly in transit (theoretically).

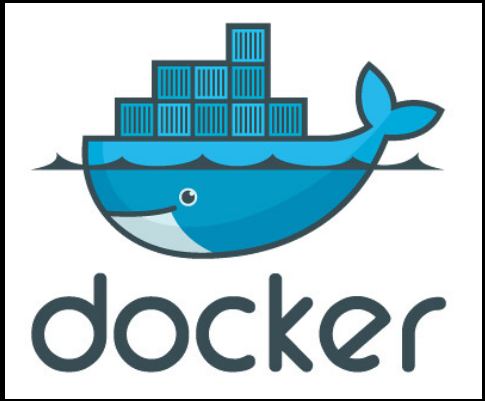
» History: Software Development pre-Docker (2013)

» History: Software Development pre-Docker (2013)

* Developers have to install and configure a plethora of dependencies, typically based on poorly maintained documentation.
* Subtle differences in environments gave way to "works on my machine".
* Development and operations was typically split and had different skill sets. Not that many people had heard of devops yet, although the 10+ deployments per day is already a few years old.
* Companies feared deploying applications to production.

» History: Developing Using Containers

* Main software container technology today.
* Apache 2.0 licensed, however Docker is also a company that makes money from Docker Desktop licenses and Docker Hub.



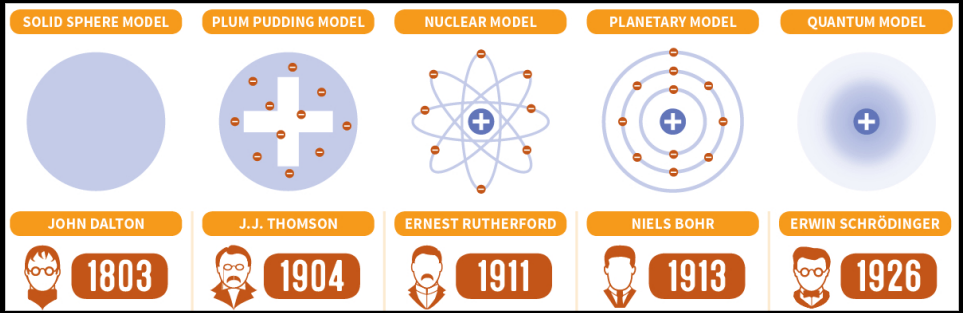
» ”Container” In Software

* **Only** assumption: The containerized application runs on Linux.
* Write a small `Dockerfile` to define the needed environment.
* You start the container – the container starts the app.
* Put app dependencies (e.g. MySQL) in separate containers.

```
FROM golang:1.24.0-alpine AS builder
ARG VERSION=0.0.1-dev
WORKDIR /app/
COPY . .
RUN apk add -U --no-cache ca-certificates \
  && adduser --disabled-password \
    --gecos "" \
    --home "/nonexistent" \
    --shell "/sbin/nologin" \
    --no-create-home \
    --uid 10001 \
    appuser \
  && CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go build \
    -o demo \
    -trimpath \
    -ldflags="-w -s -X main.version=${VERSION}" .

FROM scratch
ENV PORT=8080
COPY --from=builder /etc/ssl/certs/ca-certificates.crt /etc/ssl/certs/
COPY --from=builder /etc/passwd /etc/passwd
COPY --from=builder /etc/group /etc/group
WORKDIR /app/
COPY --from=builder /app/demo .
EXPOSE 8080
USER appuser:appuser
CMD [ "./demo" ]
```

» So it's a virtual machine defined in a file?



* That's a helpful mental model for now. (Dalton)
* But there are important differences.
* Key details in the next section. (Rutherford)
* Dive even deeper in the Under The Hood section. (Schrödinger)
* For now, just think: "Smaller, and only Linux".

» What makes it a ”Container”?

* Standardization leads to consistency:
  * 💻 Always uses a Linux kernel, works with anything that runs on Linux.
  * ⌨️ Always the same structural shape and dimensions, can hold anything that can fit inside.
* Portability
  * 💻 The same container can run on any host operating system.
  * ⌨️ The same container can be lifted from a ship to a train to a truck.
* Safety and Security
  * 💻 Difficult for hackers to get into a container, and difficult to get out.
  * ⌨️ Difficult for thieves and water to get in/contents to leak out.
* Any more?

» Enough Theory, Let's See It In Practice!

Source Code

The demos can be found at `github.com/kinbiko/talks` under `2025-02-15-docker-deep-dive/code/`.

* `00-hello-world`
* `01-image-container-relationship`
* `02-ubuntu`
* `03-dockerizing-your-app`

» Dockerfile Syntax

`FROM`: The docker image to base yours on. `"scratch"` is the empty/root image.
`CMD`: The name of the binary to execute when starting the container.
`ENTRYPOINT`: The arguments to the `CMD`.
`EXPOSE`: allow the host to see a port on the container.
`RUN`: execute shell commands, e.g. to install packages.
`ENV`: set an environment variable.
`COPY`: from the host, to the container.

Tip

`RUN`/`COPY` create new filesystem layers, so limit usage to save disk space.

A note on `RUN`

`RUN` executes Linux shell commands to construct the image. It is not used for booting your application. See `CMD`.

» .dockerignore

* Like `.gitignore`
* Usually ignore everything, and add exceptions.

» Docker Intuition

* Container
* Image
* Dockerfile
* Registry
* Host
* Exposing ports
* Port forwarding

» Docker Compose

* Remember that the idea is to put each application (web server, cache, database, frontend, ...) in its own container.
* Remembering to run all the right docker containers in the right order gets messy fast.
* Docker Compose lets you create a setup of multiple dependencies in a single file.
* Demo: `04-docker-compose`

» Multi-Stage Builds

* Idea: Two `FROM` instructions.
* One "builder" image with all the devtools you need to build your app.
* Copy only what you need from the builder to a very small prod image.
* Demo: `05-multi-stage`

» Theory: Virtual Machines vs. Docker containers

* Virtual Machines: Each VM has its own Guest OS on top of a Hypervisor.
* Containers: Share the Host OS kernel via Docker Engine.
* Containers are lighter weight and faster to start.

» Docker in production

* Google Cloud Platform: Cloud Run
* Amazon Web Services: Elastic Container Service (ECS), Fargate
* Azure: Container Instances

» Docker in production

* Cut a release → CI builds Docker image → CI pushes image to registry
* Start deployment → Pull image from registry → Run new version
* Rollout policy determines when to stop old version

» Docker in production at scale: Kubernetes

* If you need extensive configurability of the environment where your containers are running.
* Requires a lot of time and effort to adopt.
* When you have lots of microservices/traffic/engineering teams.
  * Control over networking, load balancing, etc.
  * Observability benefits
  * Scaling control
  * ...
* This is a 2-hour+ topic on its own.

» Internal Architecture

* `docker`: CLI
* `dockerd`: REST API, images, networking, volumes
* `containerd`: container lifecycle
* `runc`: low-level container runtime. Lib+CLI.

» Union Filesystems

* Docker images: RO+RO+...+RO file system (layers).
* Docker container: Docker image + one RW file system.
* Allows caching of layers for re-use across containers.
* Copy on Write (CoW):
  * If you edit a file e.g. in an ubuntu container, the image file remains unchanged.
  * Only the container RW layer has a different copy of the changed file.

» Linux Namespaces

* Isolates the container process from the host as well as other containers.
* PID namespace: Each process has its own process tree.
* NET namespace: IPs, routing tables, network interfaces are all separate.
* MNT namespace: Prevents access to host filesystem.
* IPC namespace: Shared memory and message queues are isolated.
* USER namespaces: Host users/root user are not shared.

» cgroups

* Structural isolation is done by namespaces.
* Resource isolation is done by cgroups.
* Control over:
  * CPU
  * Memory
  * Disk I/O
  * Network I/O and priority
  * Max process count

» Want more?

* `github.com/moby/moby` for the source code.