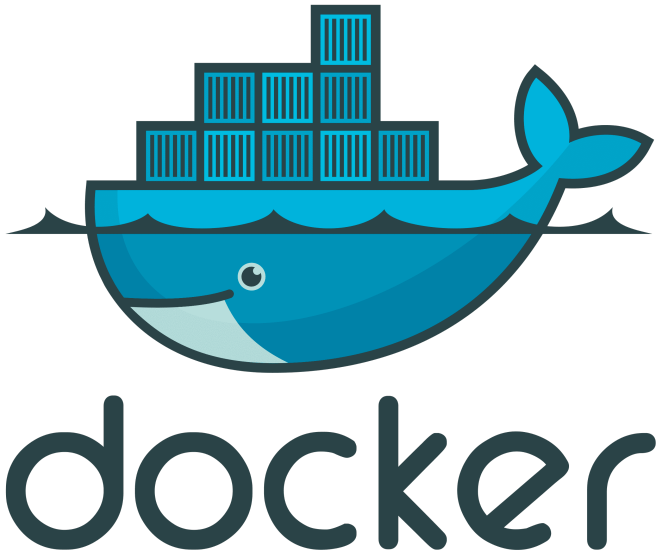


Docker 101

Agenda

- About you & me
- What is Docker
- Why use Docker
- Writing Dockerfiles
- Running Containers
- **BREAK**
- Tools
- Resources
- Ways to integrate PHP with Docker
- Conclusion & Recap

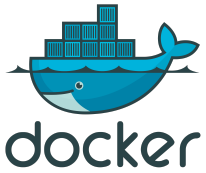
About you

- What is your job?
- In which programming language are
  - you most comfortable in?
  - not comfortable at all?
- Which project are you proud of?
  - What is {clean,ugly} code for you?
  - What are your expectations of today?
- What technical knowledge would you like to share?
- What is your experience with Docker so far?
- What is your goal for today's workshop?
- What is puzzling you about Docker?

About me

- `$> whoami` : DI Spiess-Knafl Peter
- Former Software Architect @ bitmovin and skiline
- Former Debian Maintainer (2014 - 2018)
- Founder of
  - technikrabe OG (2016 - 2019)
  - Spiess-Knafl Peter IT Solutions (since 2019) - spiessknafl.at/peter
  - hardcode GmbH (since 2023) - hardcode.at
  - nobloat.org
- GitHub cinemast
  - libjson-rpc-cpp
  - json-rpc-cxx

What is Docker?



- A toolbox (UI) for managing containers
  - Build - creating OCI compliant Container images
  - Distribute - push images to registries
  - Deploy - pull images from registries
  - Run - create containers from images
- specifically CGroups and Kernel Namespaces

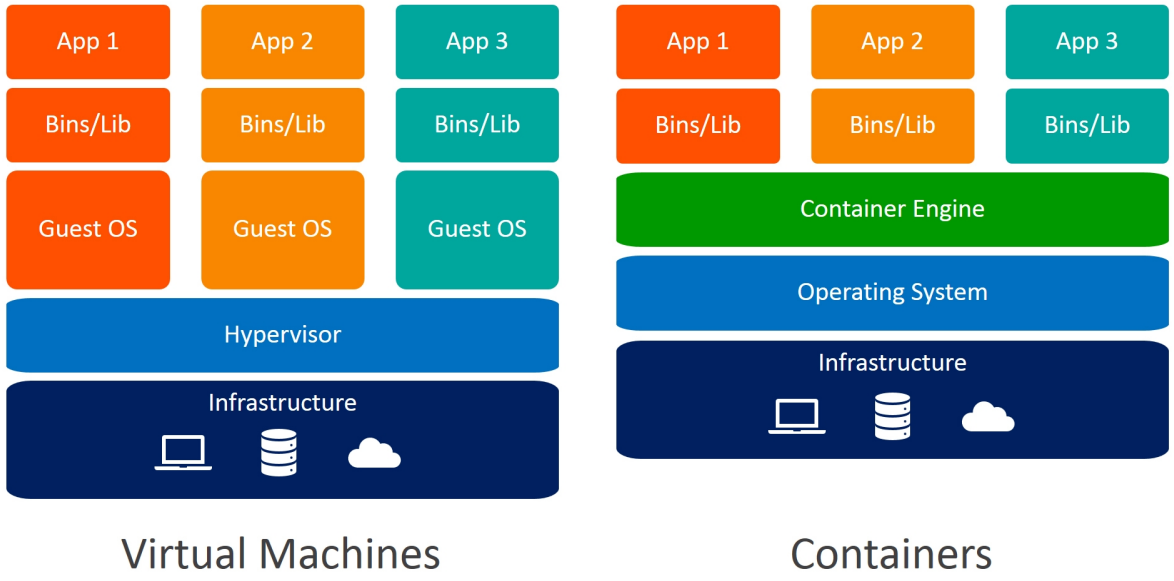
Why Docker?

- Infrastructure as Code
- Immutability and reproducibility
- Simple local development setups
- When there is no other easy mechanism to install/deploy software
- Has become the de-facto standard for container based environments
    - There is also podman
    - Both adhere OCI (Open Container Initiative)

What is a Container?

- CGroups/Namespaces allows applications to run in **isolation** of each other, regarding syscalls.
  - Processes (PIDs)
  - Network (Ports)
  - Filesystem (Mounts)
- Sounds familiar?
  - Virtual Machines (VM) have a similar purpose

Containers vs. Virtual Machines



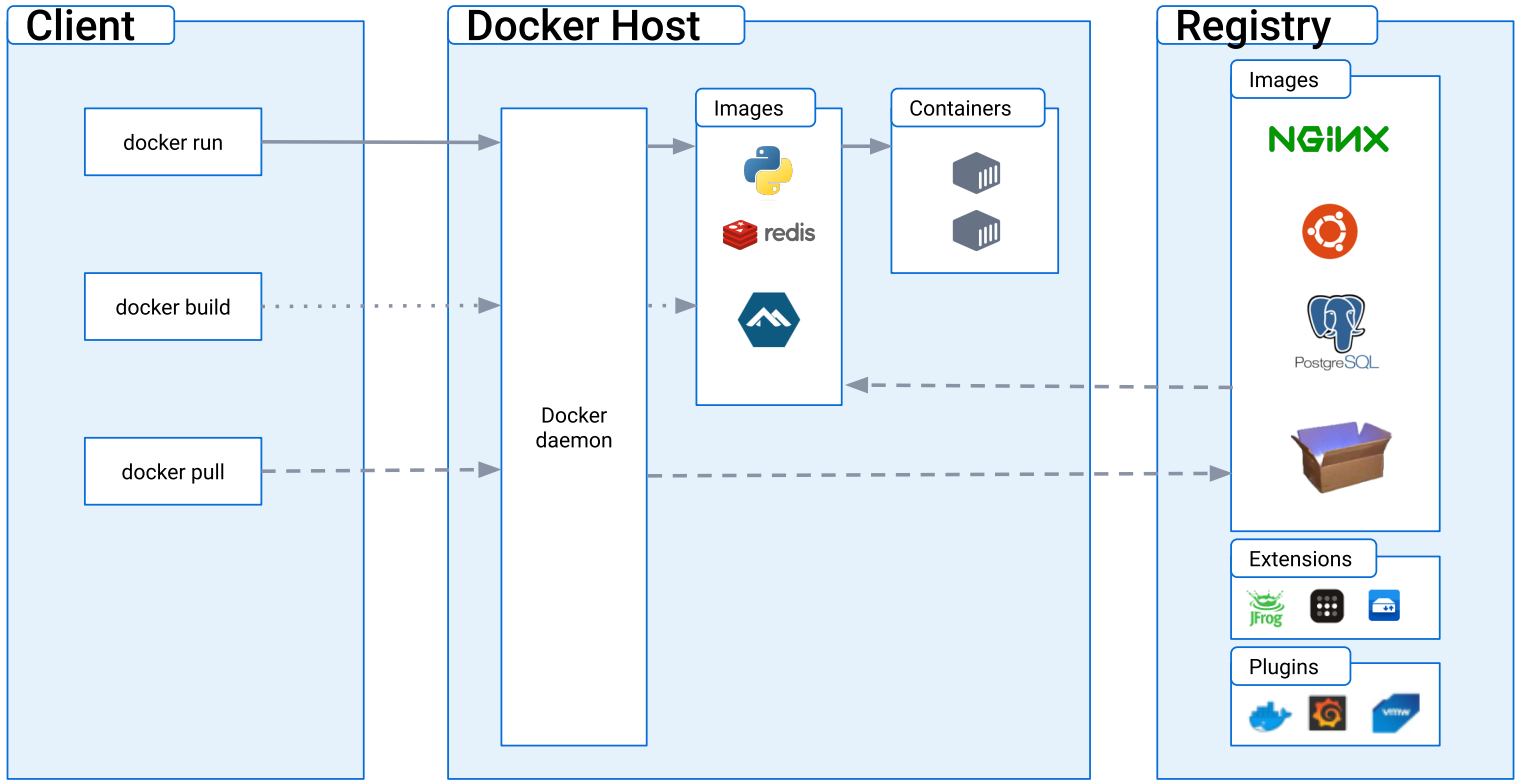
- All containers share the same Kernel
  - … and the same **kernel vulnerabilities**
  - VMs provide a better isolation and security layer
- Abstraction at `syscall` level vs. Hardware level (VM)
  - `syscall` is the interface between the Kernel and your applications.

Key differences Containers/VMs

	Docker	VM
Resource usage	process overhead	full operating system
Scalability	Easy	Harder
Boot time	Sub (milli-)seconds	Minutes
Image size	Size of application + libs	Size of application + libs + full OS
Isolation	Kernel	Hypervisor

Docker concepts

- Docker Daemon: Core component, orchestrating all moving parts



Docker concepts

- Containers: Isolated process with its own
  - namespaces (network, filesystem, processes, etc.)
  - are created/instantiated out of *Images*
  - `docker run -it <image>`
- Images: Are basically a prepared "root" filesystem to "boot" a container from
  - Layered filesystem (each diff results in a new layer)
  - Images can inherit from base images (e.g. `FROM nginx`)
  - Tags: Images can be versioned using tags (e.g. `ngingx:latest`)
  - Can be `docker build -t <repo>/<image>:<tag> .` using `Dockerfile`

Docker concepts

- Registries: Hold different Images
  - `docker pull <registry>/<image>:<tag>`
  - `docker push <registry>/<image>:<tag>`
- Volumes: Holds data should be persisted beyond a container's lifetime.
  - Can be `mounted` into containers
  - Named: `docker volume create myvolume1`
  - Bind: `docker run -v <host-directory>:<container-directory> -it <image>`
- Capabilities: Permissions a container can have
  - e.g. `SYS_ADMIN`, `NET_RAW`, `SYS_BOOT`

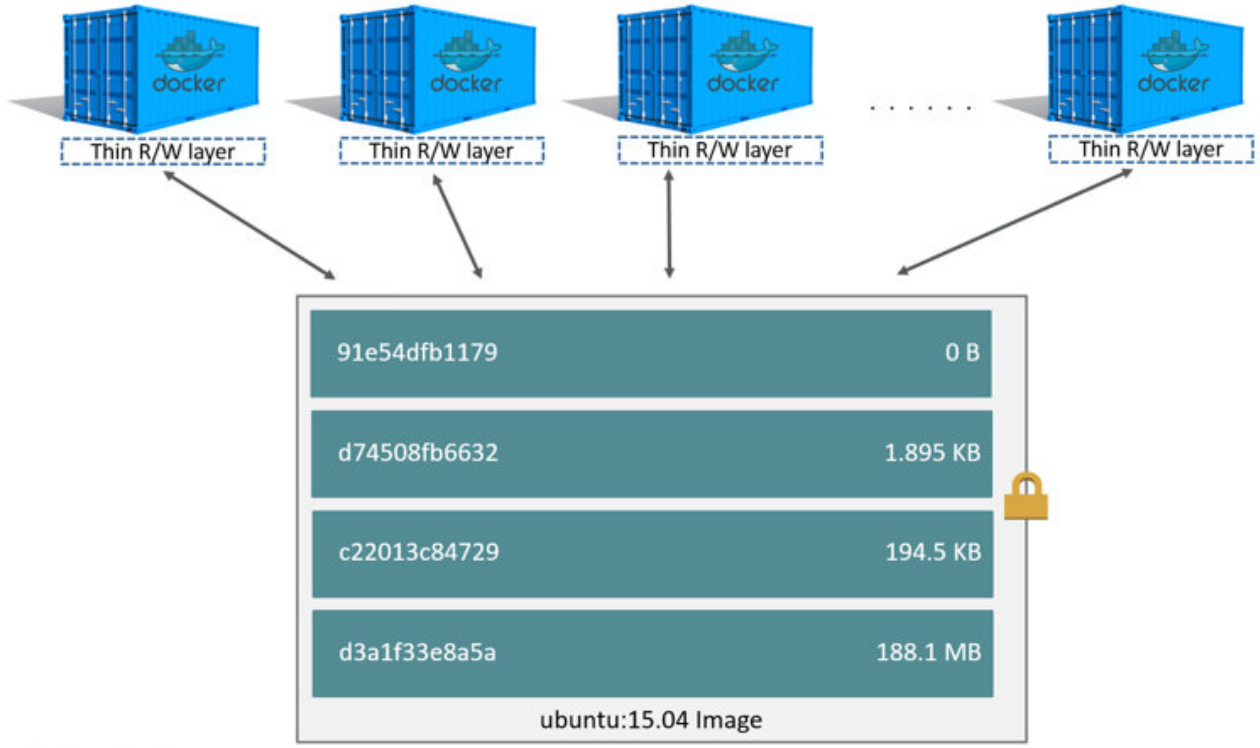
Building Images

```
docker build -t ghcr.io/my-org/my-service:3.8-alpine .
```

- `ghcr.io` - name/url of the registry to be pushed to, defaults to `hub.docker.io`
- `my-org` - project name
- `my-service` - image name
- `3.8-alpine` - image tag (version and variant), defaults to `latest`

Docker Image

- Consists of stacked R/O filesystem layers
- Dockerfile: Build instructions to construct the image
- Dive example!



Dockerfile

- `FROM` defines the base image, first instruction of every `Dockerfile`
  - `FROM scratch` starts off with a completely empty image
  - `:alpine` and `:slim`
- `ADD` / `COPY` adds files to the image (Copy vs. Add)
- `RUN` runs a command within the so far created image
- `ARG` build time variables
- `ENV` runtime environment variables (documentation)
- `EXPOSE` available ports that can be mapped (documentation)
- `VOLUME` defines volumes to be mounted

Dockerfile

- `USER` sets the user to be used in all subsequent layers. Default `root`
  - Containers should not run as `root` unless absolutely necessary.
- `WORKDIR` default directory the container will start in
- `HEALTHCHECK` execute the command as health-check mechanism
- `CMD` defines the command to be executed when the container is started
- `ENTRYPOINT` shell script the runs before the `CMD`
  - `/entrypoint.sh` must take `$1` as `CMD` parameter

Dockerfile best practices

- Best practices
- Start only one process per container
- Avoid custom `/entrypoint.sh` if not necessary
- Install only what you need during **runtime**.
  - if build- and runtime differ, use **multi-stage** builds
- `RUN rm /package.deb` will not reduce the file size -> layers
- Often changed parts should be last in the `Dockerfile`
  - layers are cached and only updated if necessary
  - each changed layers updates all above layers
- Mind the build context `.dockerignore`

Dockerfile security considerations

- Install only what is necessary
- Update base images before building `docker build --pull`
- Security Best Practices
- `docker scout cves <image>`
- `dive` - inspect docker layers interactively

Running containers

- Starting container with default CMD
  - `docker run -it -e ENVIRONMENT=PROD -v /mnt/data:/data -p 8080:80 <image>`
- Run a command within the container (once)
  - `docker run -it -e ENVIRONMENT=PROD -v /mnt/data:/data -p 8080:80 <image> <cmd>`
- Run a command within the container (once) and delete the container afterwards
  - `docker run --rm -it -e ENVIRONMENT=PROD -v /mnt/data:/data -p 8080:80 <image> <cmd>`

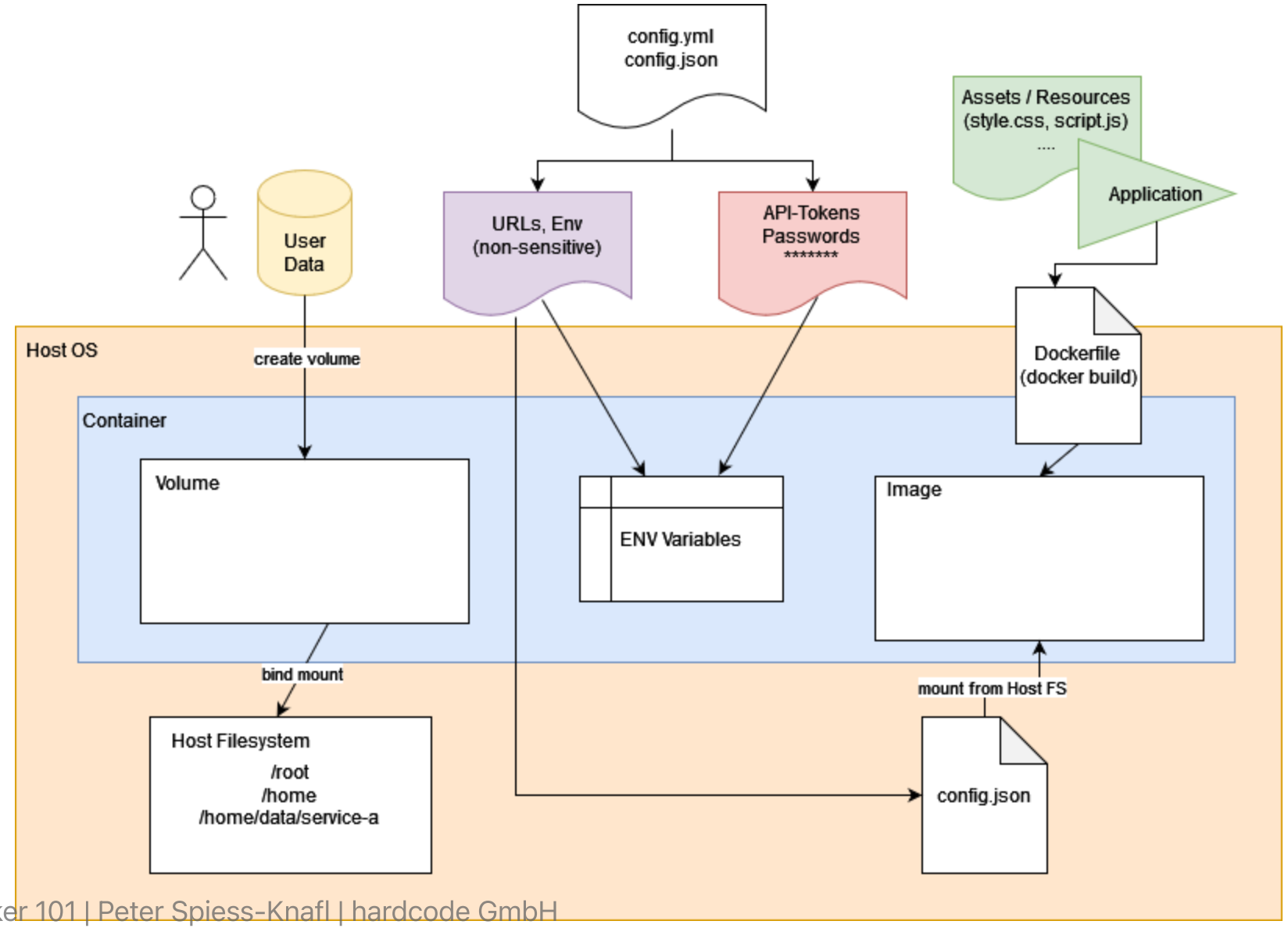
Running containers (2)

- Remembering and executing `docker run` with lots of parameters is cumbersome
- `docker compose` to the rescue, perfect for setting up local dev environments
  - Orchestrate multiple containers in a `compose.yml` file
  - Manage in one place
    - Containers, Environment variables, Port Mappings
    - Networks
    - Volumes
- Debugging:
  - `docker exec -it <container-id> /bin/sh`
  - `docker logs -f <container-id>`
- Cleanup: `docker system prune`

Where to put what kind of data?

- Usually different categories of data considering containers
  - Configuration (environment, API endpoint URLs)
  - Secrets (API Keys, database passwords, admin passwords, SMTP)
  - Application (bundled resources, source code, binaries)
  - User data (configuration, database rows)
- Where should they be kept?
  - What should be kept in the image?
  - What should be kept in the environment?
  - What should be kept in volumes?

Where to put what kind of data? (2)



Tools

- Podman - Alternative OCI compliant Docker alternative
- dive - Inspect docker images interactively
- Docker Desktop - Developer UI
- Portainer - Web interface for docker daemons
- quay.io - Registry alternative to Docker Hub
- netshoot - Network debugging tools in one image

docker compose

- Manage multiple containers at once
- Containers are defined in `compose.yml` or `docker-compose.yml`
- Very handy for local dev environments
  - 1 container that runs your PHP Code
  - 1 container that runs a MySQL database locally
- Manage containers, volumes, networks, ports, envs out of one file
- `docker compose up <service>`, `docker compose down`, `docker compose logs -f`
- `docker compose ps`, `docker compose pull`, `docker compose restart`

Cheat sheet

Resources

- Docker documentation hub
- Docker cheat-sheet
- Dockerfile best practices
- Dockerfile reference
- Docker compose reference
- Docker curriculum
- Youtube tutorial
- Youtube crash course 1 hour
- The 12 factor App

When not to use Docker?

- If you already have an easy to install software
  - e.g. `.deb`, `.rpm` packages
- If you already have single artifact that you can drop on a server
  - e.g. `.jar` files and Java already solved this problem
  - e.g. `go` binaries are statically linked and can just be run
- Automatic restart can also be solved using `systemd` units
- Adding another layer of abstraction won't make your life easier
  - You need at least a registry
  - You probably need a build pipeline
  - You need a deployment mechanism

Deploying PHP applications with docker

- Multiple scenarios possible
  - `FROM php:8.2-apache`
    - considered legacy
    - weaker performance compared to `fpm`
  - `FROM php:8.2-fpm` + `nginx` in one container
    - violates the one process per container rule, but definitely simplifies setup
  - `FROM php:8.2-fpm` + `FROM:nginx` + `shared volume`
    - complex setup but gives the greatest flexbility
    - suitable for "micro service" setups
    - nginx only runs once for all services
- Using Docker container as local environment for db + app

Deploying Node applications with docker

- Depends on your setup
- Frontend projects -> try to export as static sites and deploy into `nginx` or `s3` with cache
- Backend projects
  - Make sure you use multi-stage builds
  - Especially with typescript
  - Choose between `node`, `deno` and `bun`
  - pm2 - 1 process per container rule, most of the time `pm2` can be replaced by native docker images without it.
- Using Docker container as local environment for db + app

Conclusion and Recap

- What is a Container and how does it differ from VMs?
- What is a docker image and how can I build my own?
- What is a registry?
- How can I create containers?
- How can Dockerfiles be optimized and why are layers important?
- Where should persistent data be kept?
- What useful tools are there for managing docker containers?

Thank you!