# Week 11: Docker, App Deployment

# Agenda

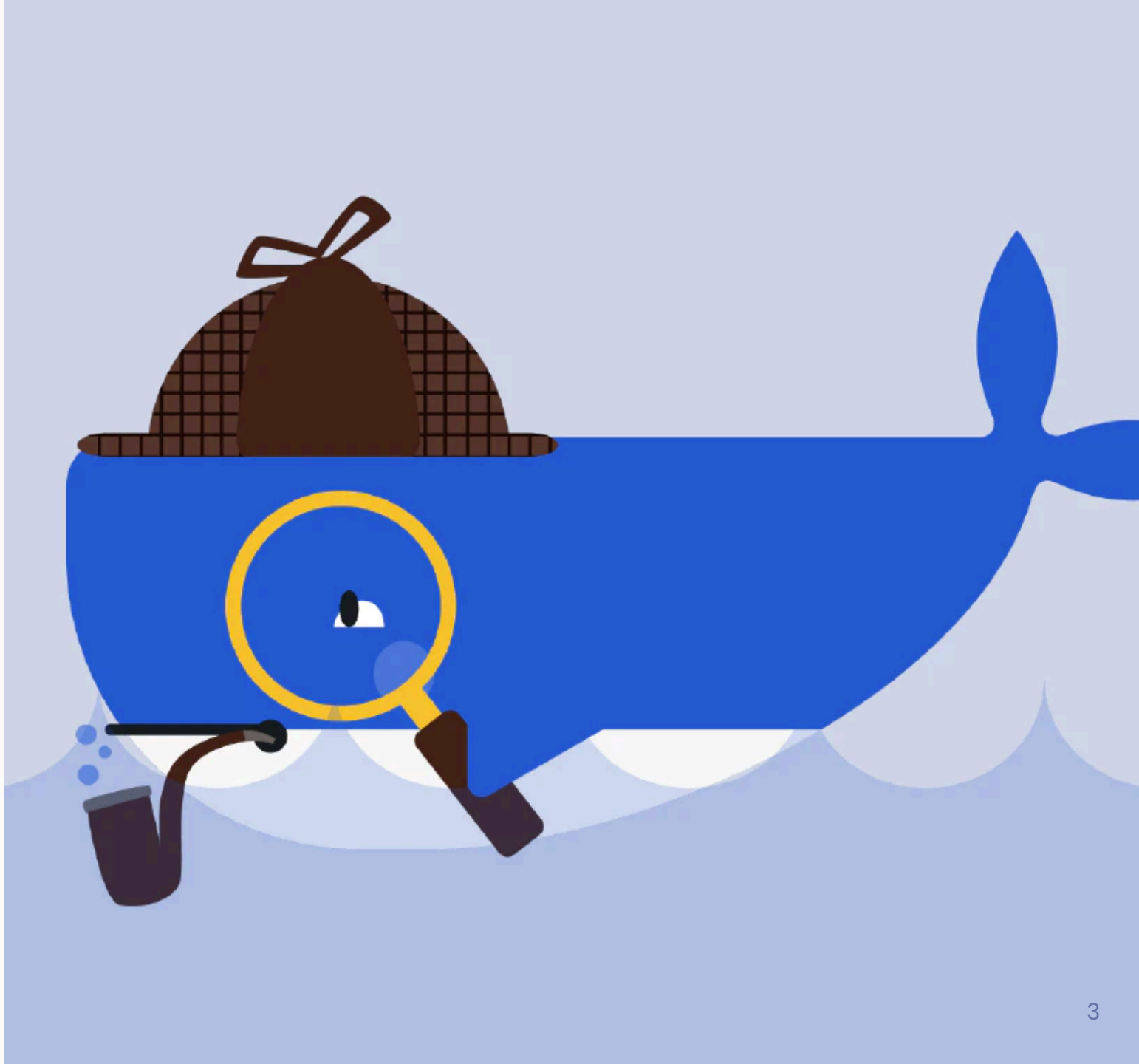
- Docker
- Packaging a SPA
- Packaging an Express API
- Container orchestration
- Container registries

# Docker

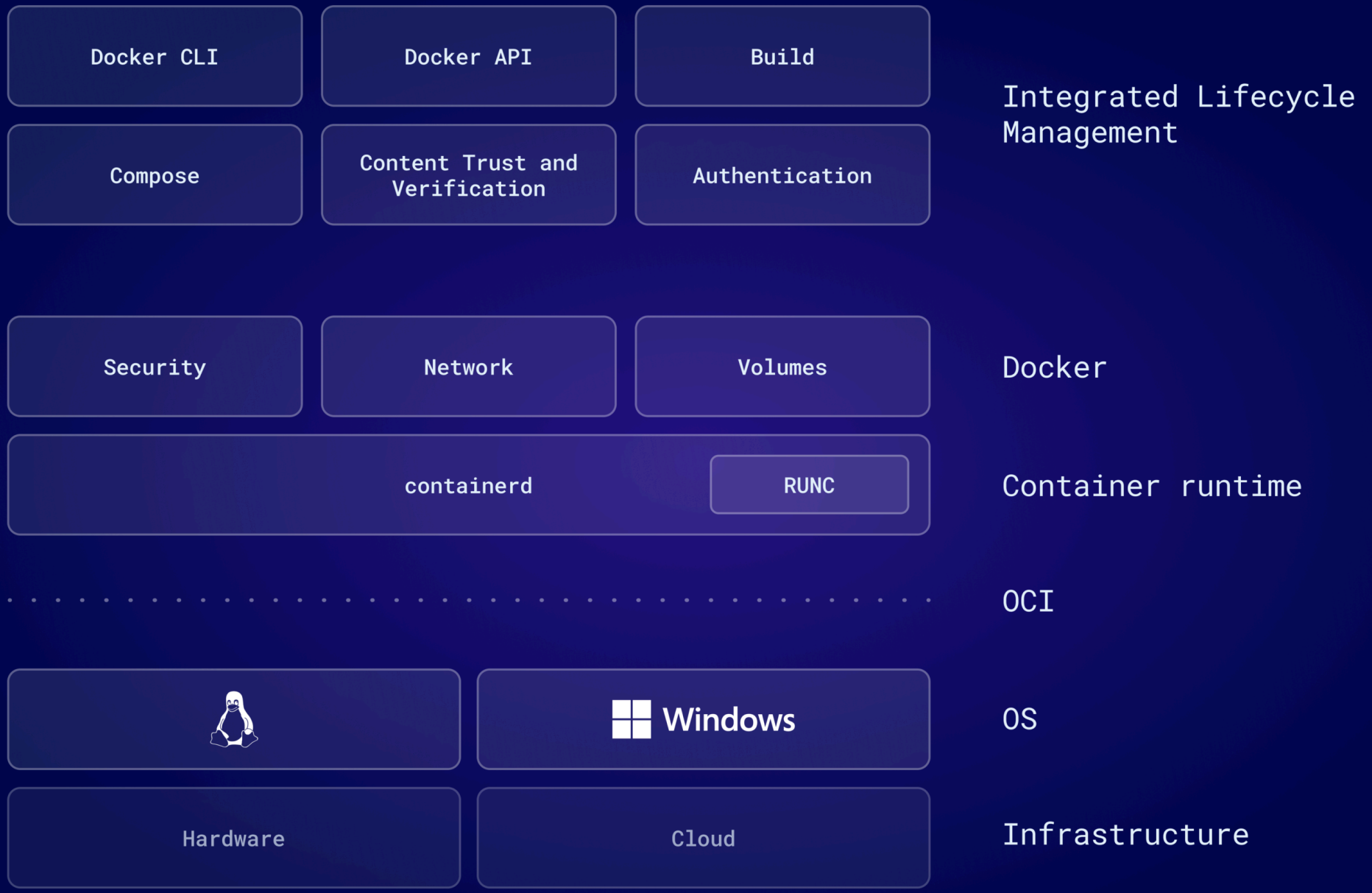
- Open source container client and runtime
- Allows you to package an application and its dependencies into a container

> Docker allows you to kindof ship your machine to your clients

- Recap: Containers vs VMs



# Docker architecture


Docker CLI
Docker API
Build
Compose
Content Trust and Verification
Authentication
Integrated Lifecycle Management
Security
Network
Volumes
Docker
containerd
RUNC
Container runtime
OCI
Windows
OS
Hardware
Cloud
Infrastructure

# Building a SPA

- Usually just `npm run build`
- Outputs minified and bundled: `.html`, `.css`, `.js`

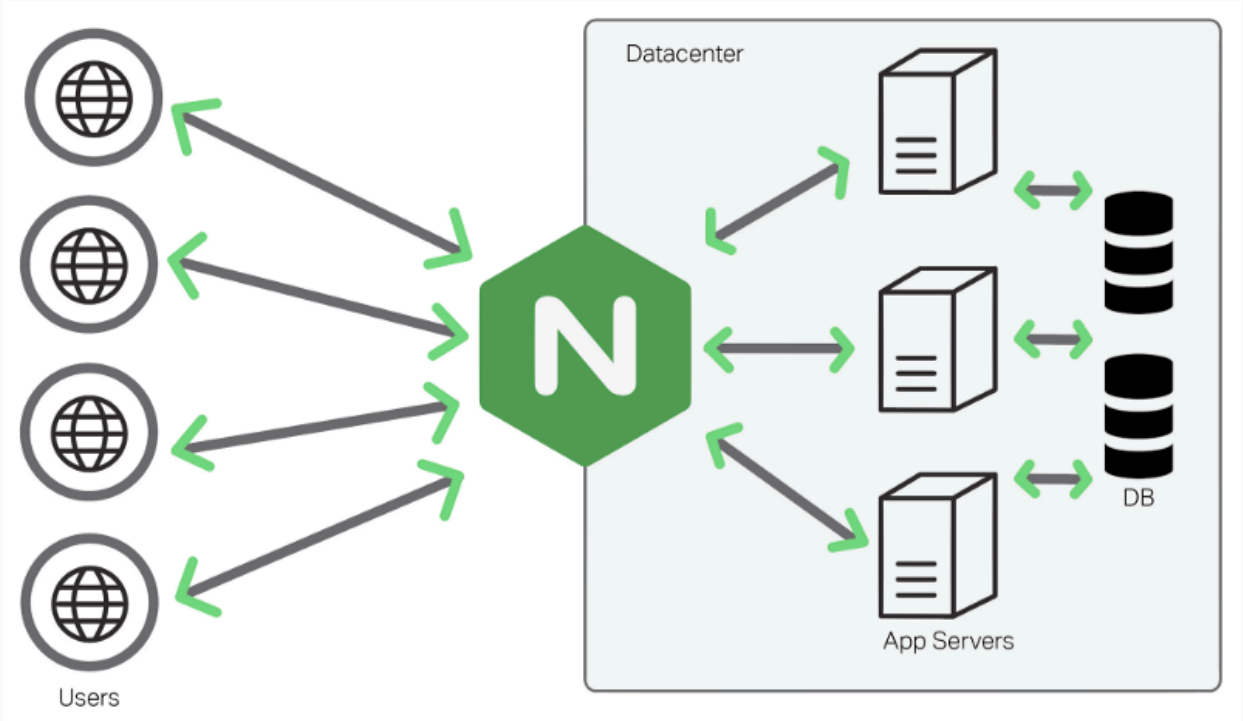
> Try it! Setup a project via `npm create vite`
> Then run `npm run build`, inspect the `dist` folder

# HTTP server

- We need a web server to serve the static files
- Express can do that: `express.static` middleware
- BUT: Dedicated high-performance servers are a better choice

# Nginx

- High-performance, open-source web server
- Very versatile, can be used as an app gateway, reverse proxy, load balancer, etc.
- Very good at serving static files, requires next to none resources



# Nginx Conf

- Nginx is configured via an `nginx.conf` file
- A lot of options! Visit nginx.com for examples
- Here's a simple example:

```
events {
    worker_connections 1024;
}

http {

    server {
        listen 80;

        location / {
            root /www/data;
        }
    }
}
```

> Takes a directory `/www/data` and serves it on port 80

```
user  nginx;
worker_processes  auto;

error_log  /var/log/nginx/error.log warn;
pid        /var/run/nginx.pid;

events {
    worker_connections  1024;
}

http {
    include       /etc/nginx/mime.types;
    default_type  application/octet-stream;
    keepalive_timeout  65;

    sendfile  on;  # static site serving
    gzip  on; # enable compression

    server {
        listen       80;
        server_name  _;
        root   /usr/share/nginx/html;
        try_files $uri $uri.html $uri/index.html /index.html;
        index  index.html;

        location ~ /\.ht {
            deny  all; # create a rule to deny access to .ht files
        }
    }
}
```

# Dockerfile

- A Dockerfile is a script that contains a collection of commands and instructions that will be automatically executed in sequence in the docker environment for building a new docker image
- Given you have locally built the SPA in the `dist` folder, you can create a Dockerfile like this:

```
FROM nginx:alpine
COPY nginx.conf /etc/nginx/nginx.conf
COPY dist /usr/share/nginx/html

EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

> `docker build -t my-nginx .` builds the image, tags it as `my-nginx`

And then:

> `docker run -d -p 8086:80 my-nginx` runs the container on port 8086

# Building an Express API

- Slightly more complex
  - With SPA, we are "only" giving static files to a ready-made server
- Here we need to make one:
  - We need nodejs and npm
  - We need to install node dependencies

# Dockerfile

- Basic Dockerfile for an Express API:

```
FROM node:latest

WORKDIR /app
COPY . /app
RUN npm install

EXPOSE 3000
CMD ["tsx", "src/index.ts"] # tsx is an alternative to ts-node
```

> Note: Docker layers - each command creates a new layer, which can be cached and can be reused by later builds

> Try running `docker build` multiple times!

# Dockerfile

The previous example has a few issues:

- using mutable tags such as `latest` is not recommended, you never know what you get
  - use a version tag instead
- `COPY . /app` copies everything, including local `node_modules`
  - use `.dockerignore` to exclude files
- `npm install` is run every time any of the files change
  - better utilize Docker layers to cache dependencies
  - `COPY` package*.json and `npm install` before copying sources
- container runs as root
  - use `USER` to switch to a non-root user
- base image is huge
  - use `alpine` or `slim` versions (if possible)

# Dockerfile

Try improving the Dockerfile based on the previous slide!

Use some express BE project you have lying around. Or quickly spin up a new one.

```
npm init -y
npm install express
npm install typescript tsx

echo "console.log('Hello, world!')" > src/index.ts
```

# Dockerfile

```
FROM node:20.11.1-alpine

# Add package files and install
COPY package.json package-lock.json ./
RUN npm ci

# Copy sources
COPY src ./src
COPY tsconfig.json ./tsconfig.json

EXPOSE 3000
CMD ["tsx", "src/index.ts"]
```

# Docker - Image size

- Docker images can get quite large and are often passed around via network or stored in registries
  - consumes bandwidth and storage
- It is a good practice to keep the image size as small as possible

# NodeJS - Production image

- A project generally contains a lot of things unnecessary for production
  - developer tooling (typescript, nodemon, etc.), general `devDependencies`
  - tests, documentation, etc.
- A build step can be used to create a production-ready image
- The way to do this is via a multi-stage build

# Multi-stage build

```
FROM node:20.11.1-alpine as base

# Add package file and install

COPY package.json package-lock.json ./
RUN npm ci

# Copy sources
COPY src ./src
COPY tsconfig.json ./tsconfig.json

# Build the project into the dist folder (`tsc --outDir dist` in this case)
RUN npm run build

# Start production image build
FROM node:20.11.1-alpine
# Install production dependencies
COPY package.json package-lock.json ./
RUN npm ci --production
COPY --from=base /dist /dist # Copy the built project

EXPOSE 3000
CMD ["node","dist/src/index.js"] # This will vary based on your project
```

# Monorepos

# Monorepos

- Monorepos are a common way to manage multiple projects in a single repository
- They are especially useful for microservices, where you have multiple services that share common code
- They can be a bit tricky to work with, especially in a Docker environment

# Turbo

- Monorepo tool created by Vercel
- Simplifies the process of working with monorepos
- Try it out! `npm create turbo`
    - Look around! Try running the apps, change the code, etc.

# Dockerizing a monorepo

- Usually tool specific, with a detailed guide on how to do it
- Turbo has a nice way of handling this

https://turbo.build/repo/docs/handbook/deploying-with-docker

- Try it out!

# Container orchestration

# Container orchestration

- Docker is great for running a single container
- But what if you have multiple containers?
  - How do you manage them?
  - How do you scale them, run multiple instances?
  - How do you ensure they are always running?

# Container orchestration (docker-compose)

- Docker-compose is a tool for defining and running multi-container Docker applications
- It uses a `docker-compose.yml` file to configure your application's services
- It can be used to define and run multi-container Docker applications

# Docker-compose

- A simple `docker-compose.yml` file:

```yaml
version: "3"
services:
  web:
    build: . # build the image from the Dockerfile in the current directory
    ports:
      - "8080:8080"
    environment:
      - REDIS_URL=redis://redis:6379
  redis:
    image: "redis:alpine"
```

> Creates two services: `web` and `redis`

# Docker-compose (added DB)

```yaml
version: "3"
services:
  web:
    build: . # build the image from the Dockerfile in the current directory
    ports:
      - "8080:8080"
    environment:
      - REDIS_URL=redis://redis:6379
      - POSTGRES_URL=postgres://postgres:postgres@postgres:5432/mydb
  redis:
    image: "redis:alpine"
  postgres:
    image: "postgres:alpine"
    environment:
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=postgres
      - POSTGRES_DB=mydb
    volumes:
      - mydata:/var/lib/postgresql/data

volumes:
  mydata:
```

# Docker-compose (useful commands)

- `docker-compose up` starts the services
- `docker-compose down` stops the services
- `docker-compose up -d` starts the services in the background
- `docker-compose logs` shows the logs of the services

# Docker (useful features recap)

- Volumes
  - Persist data between container restarts
  - Bind mounts
    - Mount host directories into containers
    - Useful for development
- Networks
  - Connect containers together
- Environment variables
  - Pass configuration to containers
- Port mappings
  - Expose container ports to the host
- Bind mounts
  - Mount host directories into containers

# Container orchestration (Kubernetes)

- Compose is great for development, but not that much for production
  - It lacks many features needed for production, has bare necessities though
  - Cannot run on multiple machines
- Kubernetes is an industry standard container orchestration system that can manage containers across multiple hosts

> There is also Docker Swarm or Hashicorp Nomad, both simpler, but less powerful

# Container registries

- Docker images are stored in registries
- Docker Hub is the most popular one
  - Public and private repositories
- Gitlab has its own registry

> Try it out! Push any of your locally built images to faculty registry

```
docker tag my-nginx gitlab.fi.muni.cz:5050/my-nginx
docker login gitlab.fi.muni.cz:5050
docker push gitlab.fi.muni.cz:5050/my-nginx
```

https://www.fi.muni.cz/tech/unix/gitlab/ci.html.en#registry

# Thats all folks!