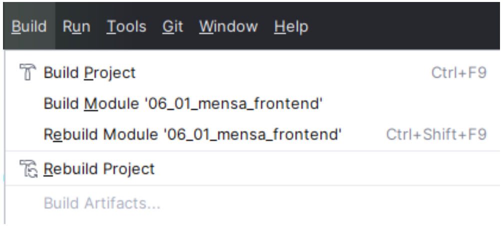
REGENSBURG

07 – Containerization of Web Applications

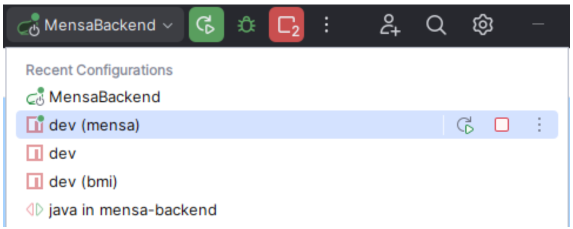
Web Technology Project (International Computer Science)
Summer semester 2025
Prof. Dr. Felix Schwägerl

Local build and execution environment

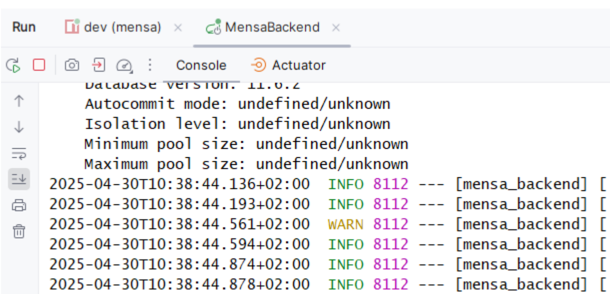
Building:



Running:



Monitoring:



- Will it also work for the customer / prof?

How to deploy the application to a real server?

What's on our server?
No IDE, just a Linux command line.
No dependencies pre-installed.
No desktop access, only SSH via terminal.

In a production system, two types of servers are involved:

- **Build servers:**
  - Access source code (e.g., from a Git repository)
  - Compile the source code into a distributable format
  - Execute additional build steps (e.g., run Junit tests)
- **App servers:**
  - Access distributable components
  - Run the components and provide access to users
  - Allow developers to monitor the running application

Basis of CI/CD: Infrastructural abstraction

Bare metal

- Dedicated hardware per unit of deployment (= service)
- High installation effort
- Manual repetition of installation procedures
- Low portability

Virtual machines

- Static allocation of shared system resources
- Stateful: every machine has a specific system state
- Easy to duplicate

- High virtualization overhead

(Docker) Containers

- Dynamic allocation of shared resources
- Low virtualization overhead
- Stateless images

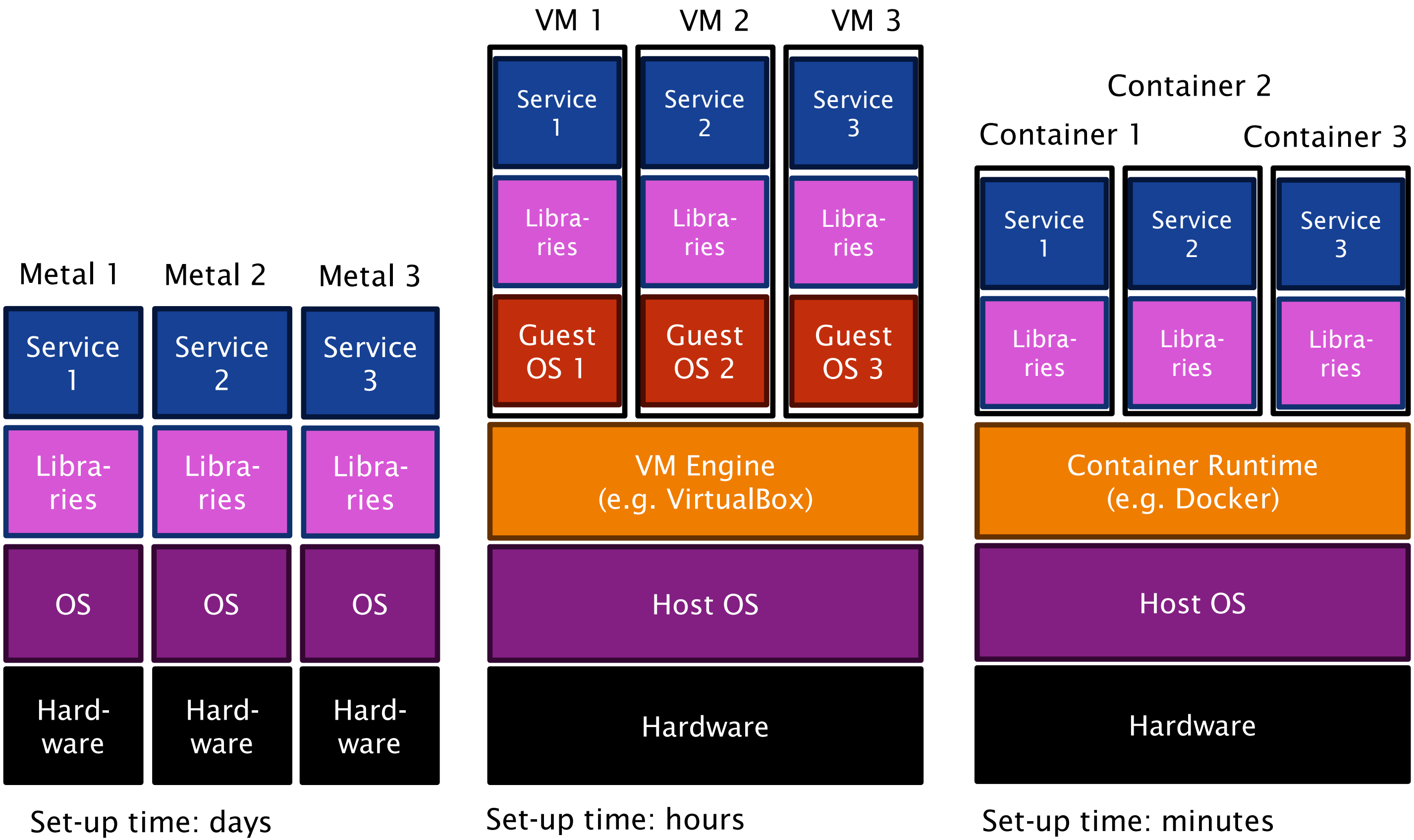
- Stateful containers
- Virtual networks
- Built-in version control for images

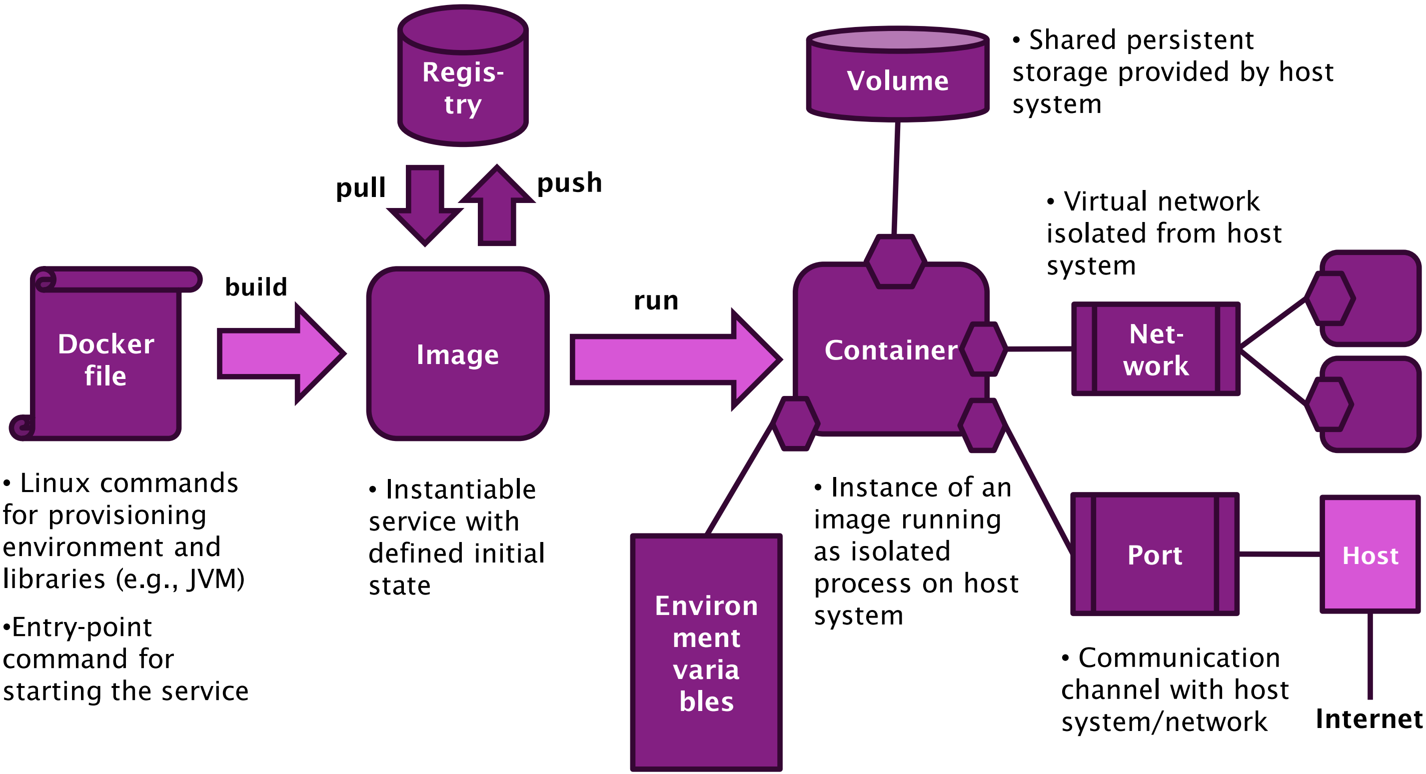
Infrastructure as Code

{}

- Declarative description of target environment infrastructure
- Fully automated deployments and updates
- Maximum portability and scalability

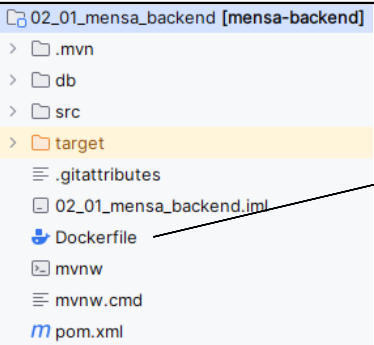
Infrastructural abstraction: Comparison





- Shared persistent storage provided by host system
- Virtual network isolated from host system
- Linux commands for provisioning environment and libraries (e.g., JVM)
- Entry-point command for starting the service
- Instantiable service with defined initial state
- Instance of an image running as isolated process on host system
- Communication channel with host system/network

Example: Building and running via Dockerfile



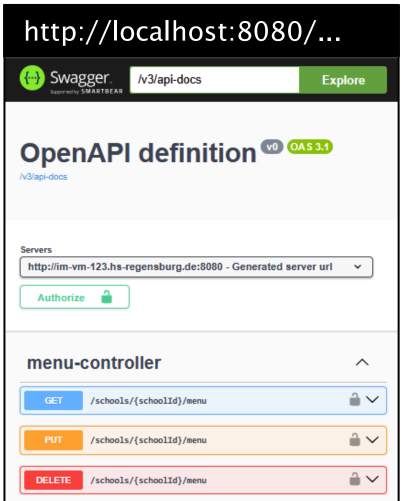

```
FROM eclipse-temurin:21-jre
COPY /build/target/mensa-backend-1.jar /app/mensa-backend-1.jar
WORKDIR /app/
EXPOSE 8080
ENTRYPOINT ["java", "-jar", "mensa-backend-1.jar"]
```

`docker build -t mensa-backend:1 .`

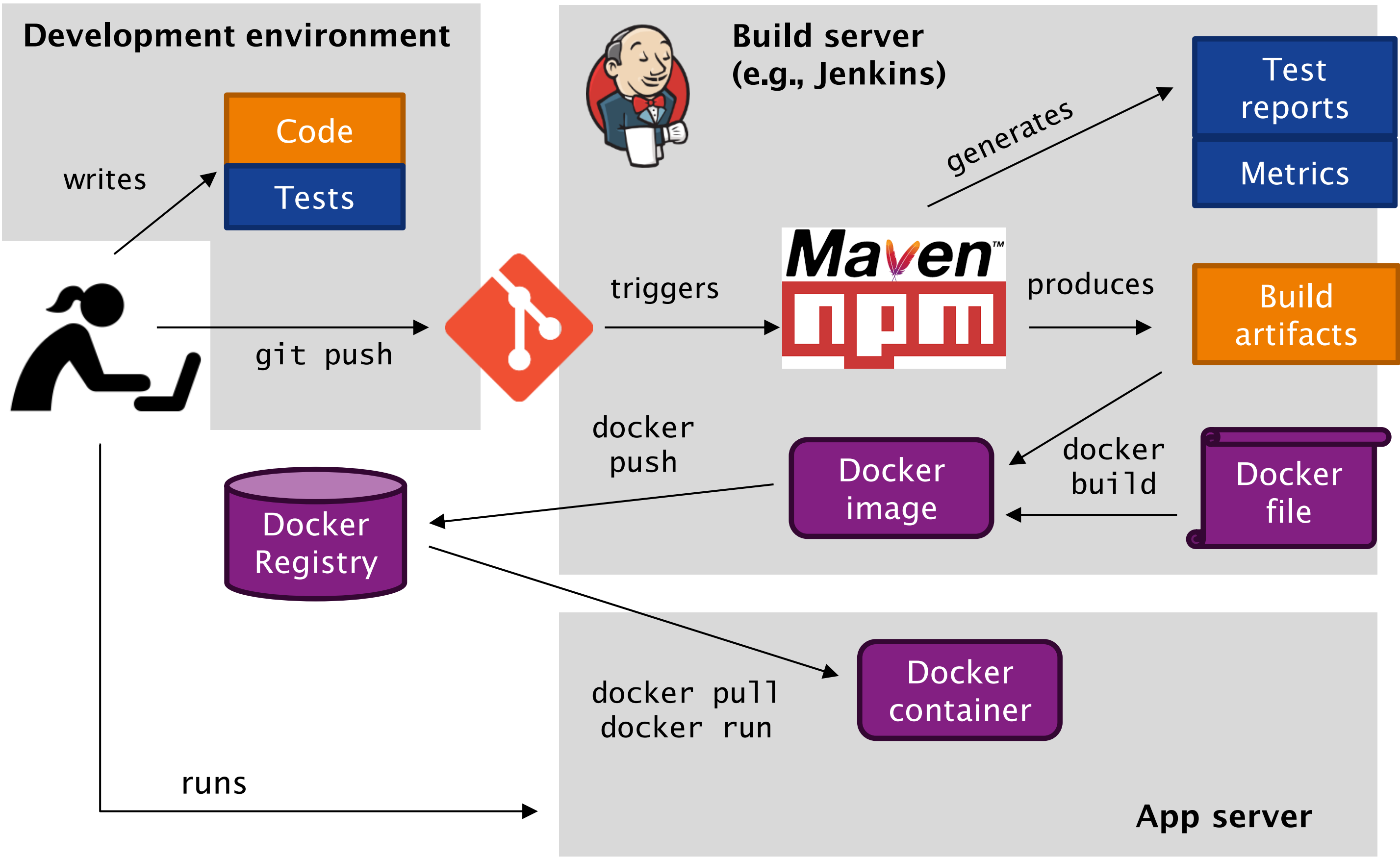
Image

`docker run -d mensa-backend:1`

Container



Continuous Integration/Deployment: Example pipeline



Docker-compose creates deployments from many containers.

- We‘ve already used docker-compose for the local deployment of MariaDB (chapter 02):

```
services:
  db:
    image: mariadb
    restart: always
    environment:
      MARIADB_ROOT_PASSWORD: asdf1234
    volumes:
      - ./db-data:/var/lib/mysql
    ports:
      - 3306:3306
  adminer:
    image: adminer
    restart: always
    ports:
      - 7070:8080
    depends_on:
      - db
```

MariaDB server with persistent storage in the folder `db-data` (relative to parent folder of `docker-compose.yml`).

SQL interface is exposed to port 3306.

Optional DB administration interface allowing to access the database contents (useful for debugging).

Web interface is exposed to port 7070.

- Next step: Add containers for frontend and backend to the deployment

To automate build and deployment of the Mensa app, we need …

- A Dockerfile for the *backend*
- A Dockerfile for the *frontend*
- Docker-compose file for whole *application*, connecting the following services:
  - MariaDB
  - Adminer
  - Backend
  - Frontend

For the reason of simplicity, we will not use a dedicated CI/CD solution like Jenkins, but …

- Build server and app server are the same machine.
  - We directly instantiate a *container* from the Dockerfile (skipping *image* and *registry*)
- Git repository is manually checked-out and pulled.
- All components are built and started using one single `docker-compose` command.
- External parameters (here: the API URL) are set via a `.env` file.

Dockerfile for the backend

Stage 1: Build the project with Maven

Use a base Docker image where JDK and Maven are pre-installed

Copy all files required for the build into the build container

Set work directory and start build using „mvn package“

```
FROM maven:3.9.9-eclipse-temurin-21 AS build
COPY pom.xml /build/pom.xml
COPY .mvn /build/.mvn
COPY src /build/src
WORKDIR /build/
RUN mvn package
```

Use a base Docker image including only a JRE

Copy packaged application from build into application container

Allow to access port from host

Pass JDBC connection parameters into running container, such that MariaDB container is accessed

```
FROM eclipse-temurin:21-jre
COPY --from=build /build/target/mensa-backend-1.jar /app/mensa-backend-1.jar
WORKDIR /app/
EXPOSE 8080
ENTRYPOINT ["java", "-Dspring.datasource.url=jdbc:mariadb://db:3306
       /mensa?createDatabaseIfNotExist=true", "-jar", "mensa-backend-1.jar"]
```

Stage 2: Execute the build on a JRE

Dockerfile for the frontend

Stage 1: Build the project with npm

```
FROM node:23-alpine AS build
ARG MENSA_API
ENV MENSA_API $MENSA_API
WORKDIR /usr/src/app
COPY package*.json ./
RUN npm install
COPY ./ ./
RUN echo "" > .env
RUN echo VITE_APP_MENSA_API="${MENSA_API}" >> .env
RUN npm run build
```

```
FROM nginx:stable-alpine
COPY --from=build /usr/src/app/dist /usr/share/nginx/html
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

Use a base Docker image where node is pre-installed

Define an environment variable MENSA_API and write it into the .env file of the frontend

Prepare npm dependencies

Copy source files and build with npm

Use nginx base image

Copy distributable HTML+CSS+JS files into application container

Start nginx server on port 80

Stage 2: Execute the frontend on an nginx webserver (replacement for Vite)

Docker-compose file extended by frontend and backend

```yaml
services:
  db: …
  adminer: …
  backend:
    image: mensa/backend
    restart: always
    build:
      context: ../02_01_mensa_backend
    ports:
      - 8080:8080
    depends_on:
      db:
        condition: service_healthy
  frontend:
    image: mensa/frontend
    restart: always
    build:
      context: ../06_01_mensa_frontend
      args:
        - MENSA_API=${MENSA_API}
    ports:
      - 80:80
    depends_on:
      - backend
```

Existing services for MariaDB backend and Adminer web view (ports 3306 and 7070 exposed)

The image for the backend container is built on-demand from the Dockerfile located in `../02_01_mensa_backend`.

The service depends on the db container and exposes the port 8080 (API and Swagger UI) to the host.

The image for the `frontend` container is built on-demand from the Dockerfile located in `../06_01_mensa_frontend`.

The service depends on the db container and exposes the port 80 (where nginx serves the transpiled HTML+CSS+JS files). The environment variable `MENSA_API` is passed through from the environment (.env →)

How the DB configuration is passed into the backend

- In our Spring boot backend, the database connection is configured as follows (`application.properties`):

```
spring.datasource.url=jdbc:mariadb://127.0.0.1:3306/mensa?createDatabaseIfNotExist=true
```

- This default configuration assumes that the DB runs on the same machine (local loopback address `127.0.0.1`).
- In our Docker based environment, this assumption is not valid. The DB runs in a different container with the virtual hostname `db`.
- We may override configuration properties by passing a key/value pair `-Dkey=value` into the Java command line call.
- This is done using the following entrypoint statement in the backend‘s Dockerfile:

```
ENTRYPOINT ["java", "-Dspring.datasource.url=jdbc:mariadb://db:3306/mensa
            ?createDatabaseIfNotExist=true", "-jar", "mensa-backend-1.jar"]
```

- When the container is started, the entrypoint command is exeucted with the specified arguments, which in turn tells Spring to override the default property set above.

How the API configuration is passed into the frontend

- In our React frontend, the API URL is retrieved by the following context hook:

```
export const Api = createContext(import.meta.env.VITE_APP_MENSA_API)
```

- The `import.meta.env` mechanism accesses the contents of the `.env` file to resolve the value.
- The default contents of the `.env` file are:

```
VITE_APP_MENSA_API="http://localhost:8080"
```

- This default configuration assumes that the backend runs on the browser machine.
- In a non-development deployment, the backend runs on a remote server (here, in the backend docker container mapped to http://im-vm-123.hs-regensburg.de:8080
- The frontend's Dockerfile (←) makes sure that the default `.env` is overridden with the contents provided via the environment variable.
- On the server, we add the following `.env` (passed through via `docker-compose.yml`):

```
MENSA_API="http://im-vm-123.hs-regensburg.de:8080"
```

- This makes the .env file of the frontend application be overridden as follows:

```
VITE_APP_MENSA_API="http://im-vm-123.hs-regensburg.de:8080"
```

Building and starting the app on the server

- Prerequisites: Git and Docker must be installed
- Step 1: Clone the WTP seminar Git repository into a new folder:

```
mkdir mensa
cd mensa
git clone https://gitlab.oth-regensburg.de/scf38786/ics-wtp-seminar.git
```

- Step 2: Build, pull and run the containers with a single Docker compose command:

```
cd 06_01_mensa_frontend
docker-compose up -d --build
```

- This will pull the required dependencies (db and adminer) and build the own components (backend and frontend) and then start the containers in a virtual network:

```
[+] Running 7/7
 ✔backend                                     Built
 ✔frontend                                    Built
 ✔Network 07_01_mensa_deployment_default      Created
 ✔Container 07_01_mensa_deployment-db-1       Healthy
 ✔Container 07_01_mensa_deployment-adminer-1  Started
 ✔Container 07_01_mensa_deployment-backend-1  Started
 ✔Container 07_01_mensa_deployment-frontend-1 Started
```

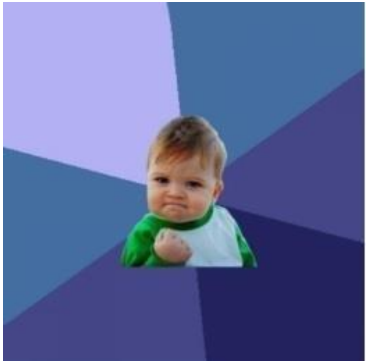
- Running deployment can be stopped with:

```
docker-compose down
```

What does all this mean for my project?

Don‘t panic!
You‘ll probably just copy and adapt the example code.

- Same services are assumed for the deployment.
  - Spring boot backend built with Maven and running on port 8080
  - React frontend built with npm and deployed with nginx on port 80
  - MariaDB running on port 3306
  - Adminer running on port 7070
- Paths of frontend and backend need to be adapted
  (e.g., from `02_01_mensa_backend` to `my_app_backend`)
- You may easily test the deployment on your development machine (if Docker is installed).



References

- [Hinkula 2022] Juha Hinkula: Full Stack Development with Spring Boot 3 and React, Packt, 2022
- [SpringBoot 2025] Spring Boot online documentation: https://docs.spring.io/spring-boot/index.html
- [React 2025] React online documentation: https://react.dev/
- [Docker 2025] Docker online documentation: https://docs.docker.com/