

# **Web Engineering 3**

## Vorlesung 5

Hochschule Zittau/Görlitz  
Christopher-Manuel Hilgner

# Agenda

## Vorlesung

- Spring Data JPA
- Konfiguration von Spring Anwendungen
- Einführung in Docker

## Seminar

- Relationen zwischen Entities
- Konfiguration der Spring Anwendung
- Einfaches Docker
- PostgreSQL

# **Spring Data JPA**

# Spring Data JPA

- Erlaubt Implementierung von Repositories
- Repositories basieren auf Java Persistence API
- Verringerung von Boilerplate Code
- Spring übernimmt das Erstellen von allen Aspekten im Hintergrund

# Repository

```
1  @Entity
2  class Person {
3
4      @Id @GeneratedValue(strategy = GenerationType.AUTO)
5      private Long id;
6      private String name;
7
8      // getters and setters omitted for brevity
9  }
10
11 interface PersonRepository extends Repository<Person, Long> {
12
13     Person save(Person person);
14
15     Optional<Person> findById(long id);
16 }
```



# Konzepte

- Basis für Repositories ist das Repository
- Es benötigt Argumente einer Entity und ihrer ID
- Repository ist dabei ein “Maker Interface” welches oft mit komplexeren Repositories erweitert wird
- Beispiel: CRUD Repository welches auf Repository basiert und weitere Funktionalitäten hinzufügt

# CRUD Repository

```
1 public interface CrudRepository<T, ID> extends
2   Repository<T, ID> {
3
4   <S extends T> S save(S entity);
5   Optional<T> findById(ID primaryKey);
6   Iterable<T> findAll();
7   long count();
8   void delete(T entity);
9   boolean existsById(ID primaryKey);
10  // ... more functionality omitted.
11 }
```



Die hier definierten Methoden decken alle CRUD Operationen ab.

- save zum Erstellen und aktualisieren
- findById, findAll, count und existsById zum Auslesen von Datenbank
- delete zum Löschen

# Queries

# Query Methoden

## Methodengenerierung

```
1 public interface UserRepository extends <User, Long> {  
2  
3     List<User> findByEmailAddressAndLastname(  
4         String emailAddress,  
5         String lastname  
6     )  
7  
8 }
```

Java

Funktion wird in folgende SQL Query generiert:

```
1 select u from User u where u.emailAddress = ?1 and  
    u.lastname = ?2
```

SQL

# Query Methoden

Folgende Begriffe (*und weitere*) werden unterstützt:

Keyword	Sample	JPQL Snippet
Distinct	findDistinctByLastnameAndFirstname	select distinct ... where x.lastname = ?1 and x.firstname = ?2
And	findByLastnameAndFirstname	... where x.lastname = ?1 and x.firstname = ?2
Or	findByLastnameOrFirstname	... where x.lastname = ?1 or x.firstname = ?2
Is, Equals	findByFirstname, findByFirstnameIs, findByFirstnameEquals	... where x.firstname = ?1 oder ... where x.firstname IS NULL, wenn das Argument null ist
Like	findByFirstnameLike	... where x.firstname like ?1
OrderBy	findByAgeOrderByLastnameDesc	... where x.age = ?1 order by x.lastname desc

# Query Methoden

Weitere Keywords können hier gefunden werden:

<https://docs.spring.io/spring-data/jpa/reference/jpa/query-methods.html#jpa.query-methods.query-creation>

# Entity Methoden

Diese Methoden können auch auf Entity Ebene erstellt werden, über die @NamedQuery Annotation.

```
1 @Entity
2 @NamedQuery(
3     name = "User.findByEmailAddress",
4     query = "select u from User u where u.emailAddress = ?1"
5 )
6 public class User {
7
8 }
```



# JPA Named Queries

## Interfaces mit Methodennamen

Named Queries können in Interfaces als Funktionen deklariert werden.

```
1 public interface UserRepository extends
  JpaRepository<User, Long> {
2
3     List<User> findByLastname(String lastname);
4
5     User findByEmailAddress(String emailAddress);
6 }
```



# JPA Named Queries

## @Query

- Funktionen in Interfaces können mit @Query ihre Funktionalität erhalten.
- @NamedQuery auf Entity Ebene wird von @Query im Interface überschrieben

# JPA Named Queries

## userEntity.java



```
1 @Entity
2 @NamedQuery(
3     name = "User.findByEmailAddress",
4     query = "select u from User u where u.emailAddress = ?1"
5 )
6 public class User { }
```

## userRepository.java



```
1 public interface UserRepository extends JpaRepository<User, Long>
2 {
3     @Query("select u from User u where u.emailAddress = ?1")
4     User findByEmailAddress(String emailAddress);
5 }
```

Hier würde die `findByEmailAddress` Funktion im Interface in `userRepository.java` aufgerufen werden.

# JPA Named Queries

## XML Queries

Queries können über XML in org.xml in META-INF

```
1 <named-query name="User.findByLastname">
2   <query>select u from User u where u.lastname = ?1</query>
3 </named-query>
```

xml: XML

# JPA Named Queries

## Weitere Möglichkeiten der Query Erstellung:

- Query Rewriter
- Advanced LIKE Expressions
- Native Queries

# Sorting

# Sorting

Sorting kann über eine PageRequest geschehen oder direkt über Sort

```
1 public interface UserRepository extends JpaRepository<User, Long> {  
2     @Query("select u from User u where u.lastname like ?1%")  
3     List<User> findByAndSort(String lastname, Sort sort);  
4  
5     @Query(  
6         "select u.id, LENGTH(u.firstname) as fn_len from User u where  
7         u.lastname like ?1%"  
8     )  
9     List<Object[]> findByAsArrayAndSort(String lastname, Sort sort);  
10 }
```



# Sorting

```
1 repo.findByAndSort("lannister", Sort.by("firstname"));
2 repo.findByAndSort("stark",
3   Sort.by("LENGTH(firstname)"));
4 repo.findByAndSort("targaryen",
5   JpaSort.unsafe("LENGTH(firstname)"));
6 repo.findByAsArrayAndSort("bolton", Sort.by("fn_len"));
```



1. **Valides** Sort. Zeigt auf eine Property im Modell
2. **Invalides** Sort. Enthält einen Funktionsaufruf. Wirft eine Exception.
3. **Valides** Sort. Durch unsafe kann die Funktion ausgeführt werden.
4. **Valides** Sort. Zeigt auf eine Funktion.

# Konfiguration einer Spring Anwendung

# Konfiguration einer Spring Anwendung

- Konfiguration einer Spring Anwendung kann über eine application.properties oder application.yaml geschehen
-

# YAML File

```
1 server:  
2   port: 8080  
3  
4   spring:  
5     datasource:  
6       url: jdbc:postgresql://localhost:5432/db  
7       username: db_user  
8       password: db_password  
9     jpa:  
10    hibernate:  
11      ddl-auto: create-drop  
12      show-sql: true  
13    properties:  
14      hibernate:  
15        format_sql: true
```

YAML

# YAML File

```
1 server:  
2   port: 8080
```

YAML

- Konfiguration für den Spring server
- port bestimmt den Port der Spring Anwendung

# YAML File

**application.yml**

YAML

```
1 spring:  
2   datasource:  
3     url: jdbc:postgresql://localhost:5432/db  
4     username: db_user  
5     password: db_password
```

- Konfiguration des Datenbankzugriffs
- url enthält URL zur gewünschten Datenbank.
- username: Username der ausgewählten Datenbank
- password: Passwort der ausgewählten Datenbank

# YAML File

**application.yml**

YAML

```
1 jpa:
2   hibernate:
3     ddl-auto: create-drop
4   show-sql: true
5   properties:
6     hibernate:
7       format_sql: true
```

- **ddl-auto:** Bestimmt, welche Operationen automatisch auf der Datenbank ausgeführt werden sollen
  - `create-drop` erschafft immer eine neue Datenbank, wenn die Spring Anwendung gestartet wird. Die Alte wird dabei gelöscht.
- **Wichtig:** Wenn keine embedded Datenbank genutzt wird, werden Datenbanken nicht automatisch erstellt.

# YAML File

YAML

**application.yml**

```
1 jpa:
2     hibernate:
3         ddl-auto: create-drop
4         show-sql: true
5     properties:
6         hibernate:
7             format_sql: true
```

- `show-sql`: Ausgeführte SQL Befehle werden in der Konsole ausgegeben. Macht Debugging einfacher.

# YAML File

**application.yml**

YAML

```
1 jpa:
2     hibernate:
3         ddl-auto: create-drop
4         show-sql: true
5     properties:
6         hibernate:
7             format_sql: true
```

- `format_sql`: Formatiert den SQL Output in eine lesbarere Form

# Properties File

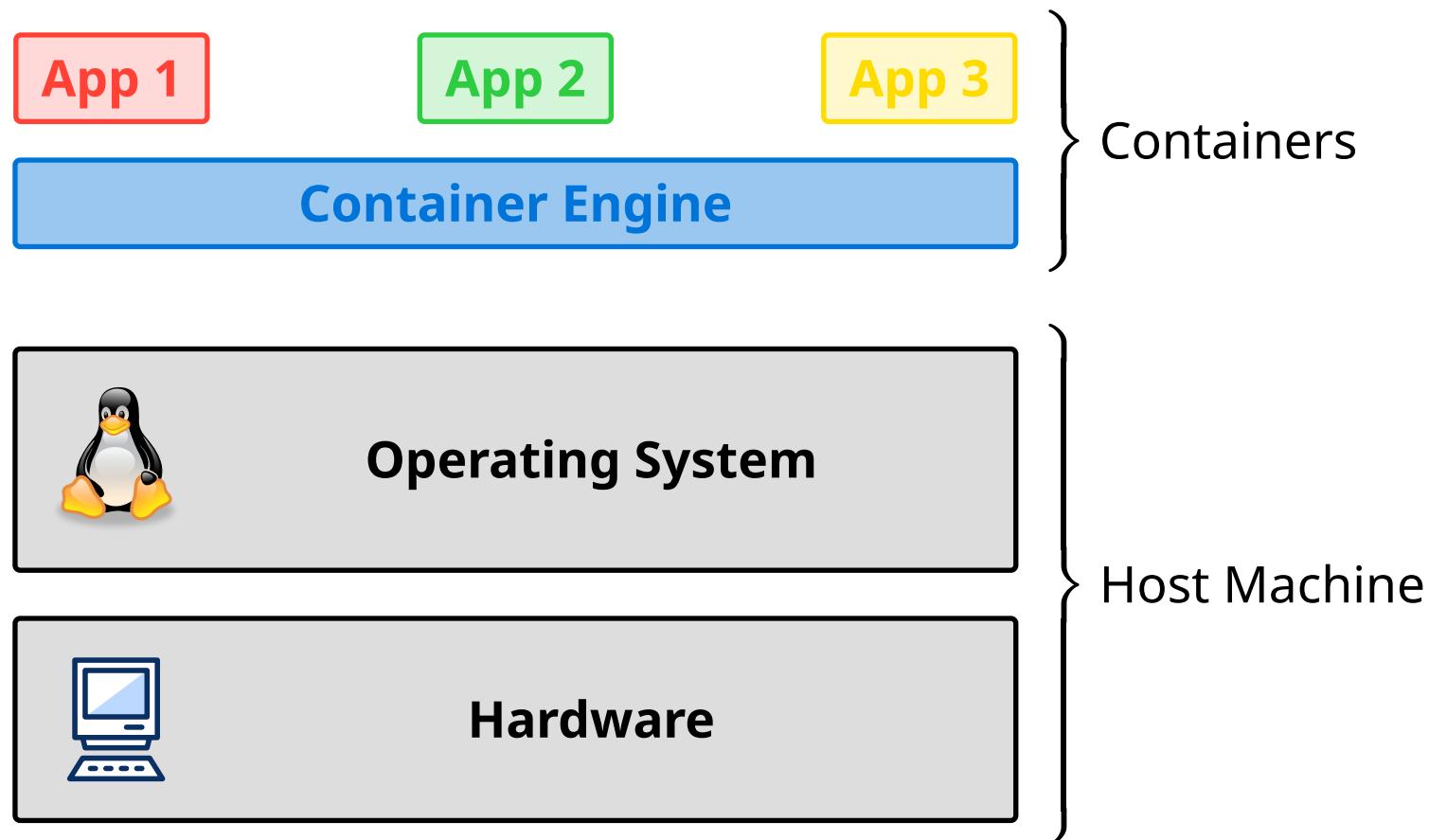
## application.properties

properties

```
1 server.port=8080
2 spring.datasource.url=jdbc:postgresql://localhost:5432/db
3 spring.datasource.username=db_user
4 spring.datasource.password=db_password
5 spring.jpa.hibernate.ddl-auto=create-drop
6 spring.jpa.show-sql=true
7 spring.jpa.properties.hibernate.format_sql=true
```

# Einführung in Docker

# Docker



# Dockerfile

```
1 FROM eclipse-temurin:21-alpine
2 WORKDIR /app
3 COPY --from=build /dist/ .
4 EXPOSE 8080
5 ENTRYPOINT ["/app/bin/application"]
```

Dockerfile

# Dockerfile

```
1 FROM eclipse-temurin:21-alpine
2 WORKDIR /app
3 COPY --from=build /dist/ .
4 EXPOSE 8080
5 ENTRYPOINT ["/app/bin/application"]
```

Dockerfile

- Auswahl des Images, auf dem der Container basiert

Hier:

- Ein Container mit JDK 21 (Ecplise Temurin) installiert
- Basierend auf Alpine Linux (lightweight & secure Distro)

# Dockerfile

```
1 FROM eclipse-temurin:21-alpine
2 WORKDIR /app
3 COPY --from=build /dist/ ./
4 EXPOSE 8080
5 ENTRYPOINT ["/app/bin/application"]
```

Dockerfile

- Auswahl des Working Directory
- Alle Befehle, die ./ benutzen, werden in das Working Directory verwiesen

# Dockerfile

```
1 FROM eclipse-temurin:21-alpine
2 WORKDIR /app
3 COPY --from=build /dist/ ./
4 EXPOSE 8080
5 ENTRYPOINT ["/app/bin/application"]
```

Dockerfile

- Programm Dateien werden in den Container kopiert
- Ohne --from=build würde das erste Argument auf das Host Dateisystem verweisen
- --from kann auf andere Container verweisen. Hier der build-Container

Dateien aus dem Ordner /dist/ im Container build werden in das Working Directory vom aktuellen Container kopiert

# Dockerfile

```
1 FROM eclipse-temurin:21-alpine
2 WORKDIR /app
3 COPY --from=build /dist/ ./
4 EXPOSE 8080
5 ENTRYPOINT ["/app/bin/application"]
```

Dockerfile

- Port, der im Container nach außen freigegeben wird
- Es können mehrere EXPOSE genutzt werden
- Ohne weitere Angaben wird der TCP Port exposed
- UDP muss extra angegeben werden

```
1 FROM eclipse-temurin:21-alpine
2 EXPOSE 80/tcp
3 EXPOSE 80/udp
```

Dockerfile

# Dockerfile

```
1 FROM eclipse-temurin:21-alpine
2 WORKDIR /app
3 COPY --from=build /dist/ .
4 EXPOSE 8080
5 ENTRYPOINT ["/app/bin/application"]
```

Dockerfile

- Der ENTRYPOINT definiert einen Command, der vom Container ausgeführt wird
- Der Container läuft praktisch selbst wie eine Executable
- Mehrere ENTRYPOINT können vorhanden sein, aber nur der letzte wird genutzt

# Dockerfile

Zwei Formate möglich:

**Exec Form, bevorzugt**

Dockerfile

```
1 ENTRYPOINT ["executable", "param1", "param2"]
```

**Shell Form**

Dockerfile

```
1 ENTRYPOINT command param1 param2
```

# Dockerfile

## Build Container

```
1 FROM eclipse-temurin:21-alpine
2 WORKDIR /app
3 COPY --from=build /dist/ ./
4 EXPOSE 8080
5 ENTRYPOINT ["/app/bin/application"]
```

Dockerfile

# Dockerfile

## Build Container

```
1 FROM gradle:8-jdk21 AS build
2 ARG APP_VERSION
3 WORKDIR /build
4 COPY ../../ .
5 RUN gradle assemble
6 WORKDIR /dist
7 RUN tar --strip-components 1 -xf "/build/build/distributions/
application-$APP_VERSION.tar"
```

Dockerfile

# Dockerfile

## Build Container

```
1 FROM gradle:8-jdk21 AS build
2 ARG APP_VERSION
3 WORKDIR /build
4 COPY ../../ .
5 RUN gradle assemble
6 WORKDIR /dist
7 RUN tar --strip-components 1 -xf "/build/build/distributions/
application-$APP_VERSION.tar"
```

Dockerfile

- Auswahl des Images, auf dem der Container basiert
- AS erlaubt refferenzierung des Containers über festgelegten Namen

### Hier:

- Ein Container mit Gradle Version 8 und JDK 21
- Container ist aufrufbar über seinen Namen: build

# Dockerfile

## Build Container

```
1 FROM gradle:8-jdk21 AS build
2 ARG APP_VERSION
3 WORKDIR /build
4 COPY ../../ .
5 RUN gradle assemble
6 WORKDIR /dist
7 RUN tar --strip-components 1 -xf "/build/build/distributions/
application-$APP_VERSION.tar"
```

Dockerfile

- Argument, dass über den docker exec, docker run Befehl oder die compose übergeben werden kann

# Dockerfile

## Build Container

```
1 FROM gradle:8-jdk21 AS build
2 ARG APP_VERSION
3 WORKDIR /build
4 COPY ../../ .
5 RUN gradle assemble
6 WORKDIR /dist
7 RUN tar --strip-components 1 -xf "/build/build/distributions/
application-$APP_VERSION.tar"
```

Dockerfile

- Auswahl des Working Directory
- Alle Befehle, die ./ benutzen, werden in das Working Directory verwiesen

# Dockerfile

## Build Container

```
1 FROM gradle:8-jdk21 AS build
2 ARG APP_VERSION
3 WORKDIR /build
4 COPY ../../ ./
5 RUN gradle assemble
6 WORKDIR /dist
7 RUN tar --strip-components 1 -xf "/build/build/distributions/
application-$APP_VERSION.tar"
```

Dockerfile

- Alle Projekt Dateien werden in den Build Container kopiert
- Hier befindet sich das Dockerfile in zwei Unterordnern im Projekt

# Dockerfile

## Build Container

```
1 FROM gradle:8-jdk21 AS build
2 ARG APP_VERSION
3 WORKDIR /build
4 COPY ../../ .
5 RUN gradle assemble
6 WORKDIR /dist
7 RUN tar --strip-components 1 -xf "/build/build/distributions/
application-$APP_VERSION.tar"
```

Dockerfile

- Führt Befehl aus
- Hier wird der Gradle Befehl zum Bauen des Projekts ausgeführt

# Dockerfile

## Build Container

```
1 FROM gradle:8-jdk21 AS build
2 ARG APP_VERSION
3 WORKDIR /build
4 COPY ../../ .
5 RUN gradle assemble
6 WORKDIR /dist
7 RUN tar --strip-components 1 -xf "/build/build/distributions/
application-$APP_VERSION.tar"
```

Dockerfile

- Das Working Directory wird auf den Build Output Ordner gewechselt

# Dockerfile

## Build Container

```
1 FROM gradle:8-jdk21 AS build
2 ARG APP_VERSION
3 WORKDIR /build
4 COPY ../../ .
5 RUN gradle assemble
6 WORKDIR /dist
7 RUN tar --strip-components 1 -xf "/build/build/distributions/
application-$APP_VERSION.tar"
```

Dockerfile

- Das gebaute Projekt wird in eine .tar Datei komprimiert, die dann aus dem Build Container in den normalen Container kopiert wird

# Compose

```
1 services:  
2   db:  
3     image: postgres  
4     container_name: postgres_db_container  
5     environment:  
6       - POSTGRES_USER=db_user  
7       - POSTGRES_PASSWORD=db_password  
8       - POSTGRES_DB=db  
9     networks:  
10    - application_network  
11   ports:  
12     - "8001:5432"  
13 networks:  
14   application_network:  
15     driver: bridge
```

YAML

# Compose

## Services

```
1 services:  
2   db:  
3     # Service Definition  
4   # Weitere Services
```

YAML

- In einer Map, unter dem Key `services`, werden alle Services definiert, die von der Compose gestartet werden sollen
- Name wird durch Key bestimmt
- Hier wäre der Name des Services: db

# Compose

## Services

```
1   db:  
2     image: postgres  
3     container_name: postgres_db_container  
4     environment:  
5       - POSTGRES_USER=db_user  
6       - POSTGRES_PASSWORD=db_password  
7       - POSTGRES_DB=db  
8     networks:  
9       - application_network  
10    ports:  
11      - "8001:5432"
```

YAML

- Einzelne Services werden durch Map unter dem Service-Key definiert

# Compose

## Services

```
1 services:  
2   db:  
3     image: postgres
```

YAML

- Key `image` bestimmt das Container-Image, dass hier genutzt werden soll
- Können vom System kommen oder zum Beispiel aus der Docker Registry
- Hier wird ein Image genutzt, welches für die PostgreSQL Datenbank zugeschnitten ist

[https://hub.docker.com/\\_/postgres](https://hub.docker.com/_/postgres)

# Compose

## Services

```
1 services:  
2   db:  
3     container_name: postgres_db_container
```

YAML

- Key `container_name` bestimmt den Namen des Containers

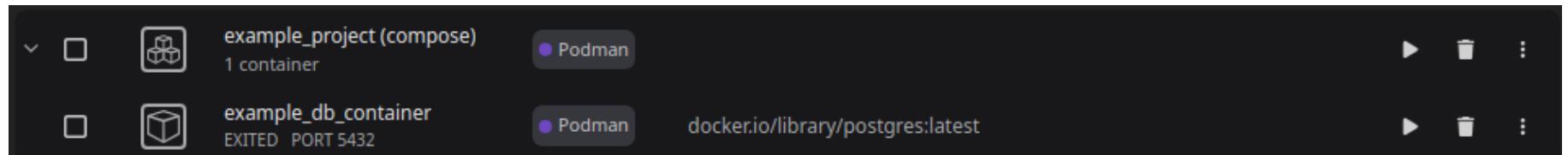


Figure 1: Aktuell nicht laufender Container in der Podman Desktop UI.  
Der Container hat den Namen: example\_db\_container

# Compose

## Services - Environment Variablen

compose.yml

YAML

```
1 environment:  
2   - POSTGRES_USER=db_user  
3   - POSTGRES_PASSWORD=db_password  
4   - POSTGRES_DB=db
```

application.yml

YAML

```
1 spring:  
2   datasource:  
3     url: jdbc:postgresql://localhost:5432/db  
4     username: db_user  
5     password: db_password
```

# Compose

## Services

```
1 services:  
2   db:  
3     networks:  
4       - application_network
```

YAML

- Key networks bestimmt, zu welchen Netzwerken der Container zugeordnet werden soll
- Es können dabei mehrere Netzwerke genutzt werden
- Netzwerke werden in einer Map angegeben
- Hier wird nur das application\_network genutzt

# Compose

## Services - Ports

```
1 services:  
2   db:  
3     ports:  
4       - "8001:5432"
```

YAML

- Key ports bestimmt, welche Ports von dem Container exposed werden sollen
- Es können mehrere Ports in einer Map angegeben werden
- Jede Angabe besteht aus zwei Ports

# Compose

## Services - Ports

```
1 - "8001": 5432"
```

YAML

**Der linke Port** ist der Port, der von außerhalb des Containers erreicht werden kann

**Der rechte Port** ist der Port, der innerhalb des Containers genutzt werden soll. Er wird auf den linked Port gemapped.

In diesem Beispiel hier, wird der default Port einer PostgreSQL Datenbank, **5432**, der innerhalb des Container genutzt wird, auf den Port **8001** gemapped. Die Datenbank kann somit außerhalb des Containers auf dem Port **8001** erreicht werden.

# Compose

## Services - Depends On

```
1 services:  
2   db:  
3     depends_on:  
4       - other_container
```

YAML

- Key `depends_on` bestimmt, welche Container bereits laufen müssen, damit der aktuelle Container gestartet werden kann
- Es können mehrere Container in einer Map angegeben werden
- Der Name des Containers ist hier der Service Name

# Compose

## Netzwerke

```
1 networks:  
2   application_network:  
3     driver: bridge
```

YAML

- Definierung von Netzwerken für die Docker Container
- Container, die auf einem gemeinsamen Netzwerk sind, können untereinander kommunizieren
- bridge Netzwerk wird am Häufigsten genutzt
- Netzwerke werden als Map unter dem Key networks definiert
- Netzwerk-Name ist Key des jeweiligen Netzwerks
- driver Key bestimmt, welche Art von Netzwerk genutzt wird

# Prüfung

# Prüfung

Zwei Bestandteile:

## 1. Präsentation zum Projekt

- ca. 20 Minuten
- Demonstration des Projekts
- Use Cases
- Bestandteile der Implementation

## 2. Belegarbeit

- 10 - 15 Seiten
- Einleitung, Machbarkeit, Anforderungsanalyse, Technologie  
Diskussion, Entwurf des Systems, Implementierung, Software Tests,  
Ausblick, Fazit

# Prüfung

## Terminvorschlag

- Präsentation: 28.01 & 30.01.2026 (letzter Freitag vor dem Prüfungszeitraum)
- Belegabgabe: 22.02.2026 (letzter Tag des Prüfungszeitraums)

# Prüfung

## Präsentationszeiten

Mittwoch

28.01.2026

	12:00 Uhr
	12:20 Uhr
	12:40 Uhr
	13:00 Uhr
	13:20 Uhr
	13:40 Uhr
	14:00 Uhr
	14:20 Uhr
	14:40 Uhr
	15:00 Uhr
	15:20 Uhr
	15:40 Uhr

Freitag

30.01.2026

	10:00 Uhr
	10:20 Uhr
	10:40 Uhr
	11:00 Uhr
	11:20 Uhr
<b>Mensa</b>	11:40 Uhr
	12:00 Uhr
	12:20 Uhr
	12:40 Uhr
	13:00 Uhr
	13:20 Uhr
	13:40 Uhr

**MENSA IST VORBEI**