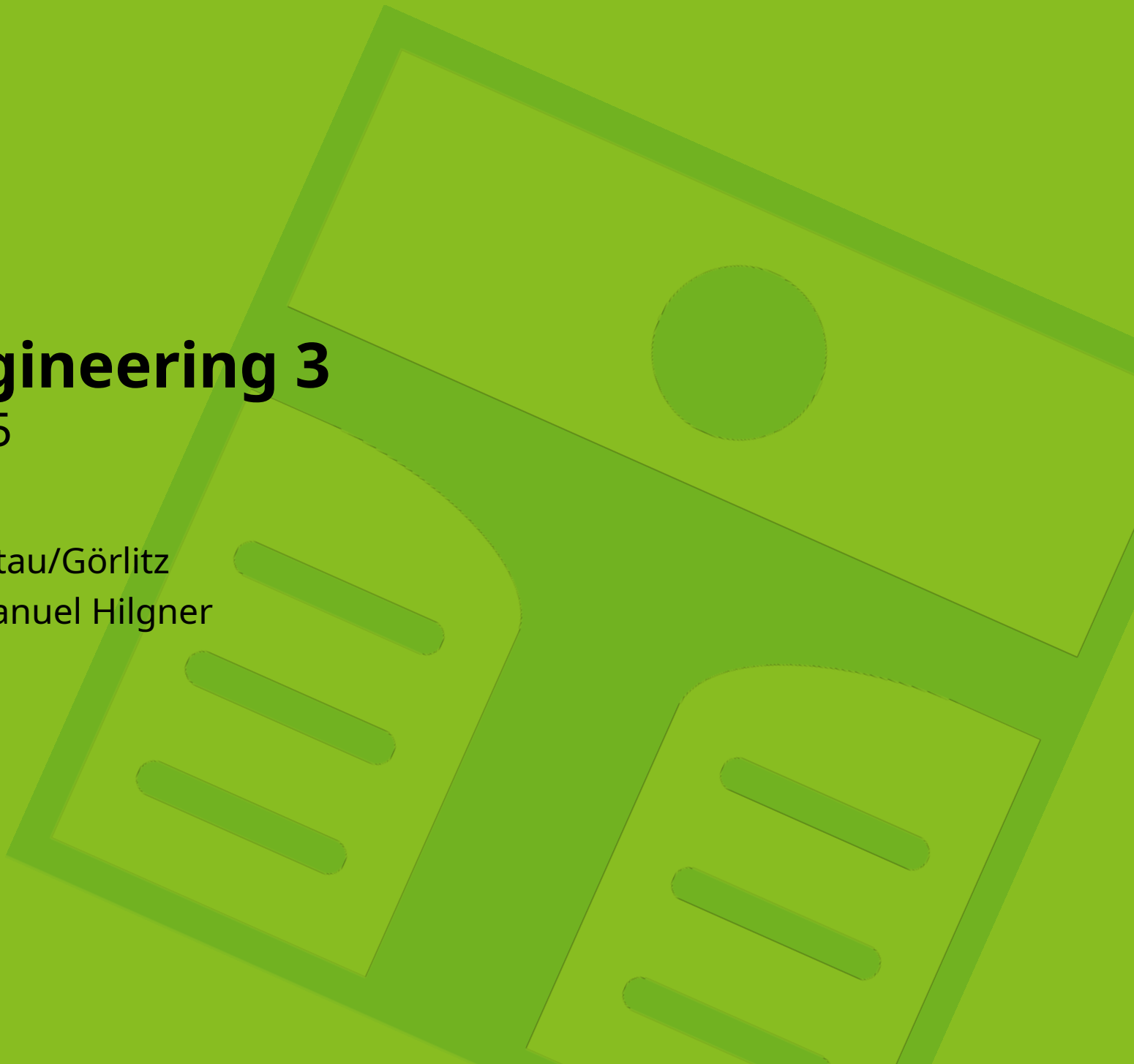
Web Engineering 3

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

Java

```java
@Entity
class Person {

  @Id @GeneratedValue(strategy = GenerationType.AUTO)
  private Long id;
  private String name;

  // getters and setters omitted for brevity
}

interface PersonRepository extends Repository<Person, Long> {

  Person save(Person person);

  Optional<Person> findById(long id);
}
```

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

```java
public interface CrudRepository<T, ID> extends Repository<T, ID> {

  <S extends T> S save(S entity);
  Optional<T> findById(ID primaryKey);
  Iterable<T> findAll();
  long count();
  void delete(T entity);
  boolean existsById(ID primaryKey);

  // … more functionality omitted.
}
```

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

Spring Data JPA

Queries

Query Methoden

Methodengenerierung

```java
public interface UserRepository extends <User, Long> {

  List<User> findByEmailAddressAndLastname(
    String emailAddress,
    String lastname
  )

}
```

Funktion wird in folgende SQL Query generiert:

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

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.

```java
@Entity
@NamedQuery(
  name = "User.findByEmailAddress",
  query = "select u from User u where u.emailAddress = ?1"
)
public class User {

}
```

JPA Named Queries

XML Queries

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

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

Im Interface müssen diese Named Queries dann spezifiert werden.

```java
public interface UserRepository extends
JpaRepository<User, Long> {
  List<User> findByLastname(String lastname);
  User findByEmailAddress(String emailAddress);
}
```

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 Java

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

userRepository.java Java

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

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

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`

```java
public interface UserRepository extends JpaRepository<User, Long> {
  @Query("select u from User u where u.lastname like ?1%")
  List<User> findByAndSort(String lastname, Sort sort);

  @Query(
    "select u.id, LENGTH(u.firstname) as fn_len from User u where u.lastname like ?1%"
  )
  List<Object[]> findByAsArrayAndSort(String lastname, Sort sort);
}
```

Sorting

```java
repo.findByAndSort("lannister", Sort.by("firstname"));
repo.findByAndSort("stark",
Sort.by("LENGTH(firstname)"));
repo.findByAndSort("targaryen",
JpaSort.unsafe("LENGTH(firstname)"));
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

YAML File

YAML

```yaml
server:
  port: 8080

spring:
  datasource:
    url: jdbc:postgresql://localhost:5432/db
    username: db_user
    password: db_password
  jpa:
    hibernate:
      ddl-auto: create-drop
    show-sql: true
    properties:
      hibernate:
        format_sql: true
```

YAML File

```yaml
server:
  port: 8080
```

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

YAML File

application.yml YAML

```yaml
spring:
  datasource:
    url: jdbc:postgresql://localhost:5432/db
    username: db_user
    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

```yaml
jpa:
  hibernate:
    ddl-auto: create-drop
  show-sql: true
  properties:
    hibernate:
      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

application.yml YAML

```yaml
jpa:
  hibernate:
    ddl-auto: create-drop
    show-sql: true
  properties:
    hibernate:
      format_sql: true
```

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

YAML File

application.yml YAML

```yaml
jpa:
  hibernate:
    ddl-auto: create-drop
  show-sql: true
  properties:
    hibernate:
      format_sql: true
```

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

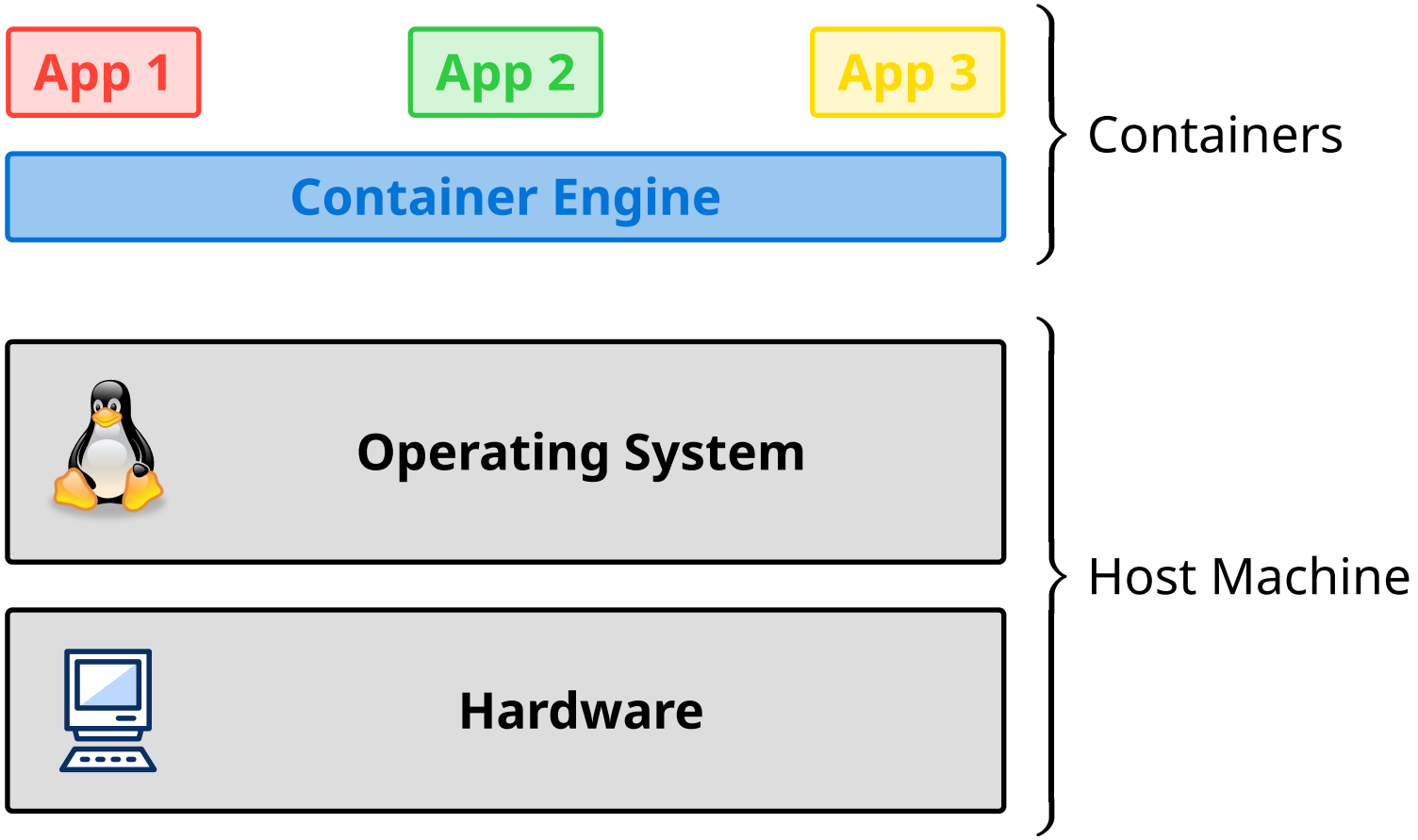
Propterties File

application.properties properties

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

Docker

Docker


App 1
App 2
App 3
Container Engine
Containers
Operating System
Hardware
Host Machine

Docker

Dockerfile

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

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

- 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                    Dockerfile
2  WORKDIR /app
3  COPY --from=build /dist/ ./
4  EXPOSE 8080
5  ENTRYPOINT ["/app/bin/application"]
```

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

Dockerfile

```dockerfile
FROM eclipse-temurin:21-alpine
WORKDIR /app
COPY --from=build /dist/ ./
EXPOSE 8080
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

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

- 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

```Dockerfile
FROM eclipse-temurin:21-alpine
EXPOSE 80/tcp
EXPOSE 80/udp
```

Dockerfile

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

- 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                    Dockerfile
2  WORKDIR /app
3  COPY --from=build /dist/ ./
4  EXPOSE 8080
5  ENTRYPOINT ["/app/bin/application"]
```

Dockerfile

Build Container

Dockerfile

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

Dockerfile

Build Container

```
1 FROM gradle:8-jdk21 AS build                                    Dockerfile
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"
```

- 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                                    Dockerfile
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"
```

- 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                                    Dockerfile
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"
```

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

Dockerfile

Build Container

```
1  FROM gradle:8-jdk21 AS build                                    Dockerfile
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"
```

- 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                                    Dockerfile
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"
```

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

Dockerfile

Build Container

```
1  FROM gradle:8-jdk21 AS build                                    Dockerfile
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"
```

- Das Working Directory wird auf den Build Output Ordner gewechselt

Dockerfile

Build Container

```
1  FROM gradle:8-jdk21 AS build                                    Dockerfile
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"
```

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

Docker

Compose

Compose

```yaml
services:
  db:
    image: postgres
    container_name: postgres_db_container
    environment:
      - POSTGRES_USER=db_user
      - POSTGRES_PASSWORD=db_password
      - POSTGRES_DB=db
    networks:
      - application_network
    ports:
      - "8001:5432"
networks:
  application_network:
    driver: bridge
```

Compose

Services

```yaml
services:
  db:
    # Service Definition
  # Weitere Services
```

- 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

```yaml
db:
  image: postgres
  container_name: postgres_db_container
  environment:
    - POSTGRES_USER=db_user
    - POSTGRES_PASSWORD=db_password
    - POSTGRES_DB=db
  networks:
    - application_network
  ports:
    - "8001:5432"
```

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

Compose

Services

```yaml
services:
  db:
    image: postgres
```

- 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

Compose

Services

```yaml
services:
  db:
    container_name: postgres_db_container
```

- Key `container_name` bestimmt den Namen des Containers

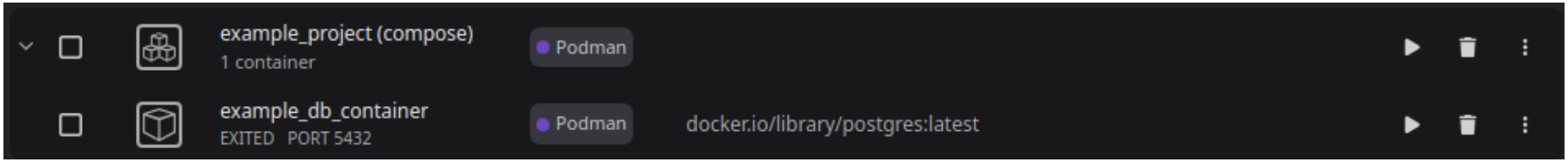


Figure 1: Aktuell nicht laufender Container in der Podman Desktop UI. Der Container hat den Namen: `example_db_container`

Compose

Services - Environment Variablen

compose.yml YAML

```yaml
environment:
  - POSTGRES_USER=db_user
  - POSTGRES_PASSWORD=db_password
  - POSTGRES_DB=db
```

application.yml YAML

```yaml
spring:
  datasource:
    url: jdbc:postgresql://localhost:5432/db
    username: db_user
    password: db_password
```

Compose

Services

```yaml
services:
  db:
    networks:
      - application_network
```

- 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

```yaml
services:
  db:
    ports:
      - "8001:5432"
```

- 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

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

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

```yaml
services:
  db:
    depends_on:
      - other_container
```

- 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

```yaml
networks:
  application_network:
    driver: bridge
```

- 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 (letzer 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
Mensa	11:40 Uhr
	12:00 Uhr
	12:20 Uhr
	12:40 Uhr
	13:00 Uhr
	13:20 Uhr
	13:40 Uhr

MENSA IST VORBEI