1. App Design

1.1 HTTP Server Layer → Requests empfangen

- Server → Startet HTTP-Server auf Port 8080
- Handler → Verarbeitet http-Requests
- RequestMapper → Konvertiert HTTP-Requests zu internen Request-Objekten

1.2 Application Layer → Business-Logik

- MRPApplication → Hauptanwendung, verwaltet Router
- Router → Routet Requests an die richtige Controller-Methode
  - Extrahiert Parameter aus URLs ({mediaId}, {userId}, {token})

1.3 Controller Layer → HTTP-Handling & JSON-Parsing

Jeder Controller handhabt eine Ressource:

Controller	Aufgaben
AuthController	Register, Login, Token-Generierung
UserController	Profil anzeigen/updaten, Ratings anzeigen
MediaController	Media CRUD (Create, Read, Update, Delete)
RatingController	Ratings erstellen, updaten, liken, confirmen

1.4 Service Layer → Geschäftslogik

Services enthalten die Logik:

Service	Nutzt	Aufgaben
AuthService	IAuthRepository, UserRepository	Login/Register, Token-Verwaltung
UserService	UserRepository	User-Profil-Verwaltung
MediaService	IMediaRepository	Media CRUD, Ownership-Check
RatingService	RatingRepository	Ratings erstellen/updaten, Likes

1.5 Repository Layer → Datenzugriff

DB-Repositories (mit PreparedStatements = SQL-Injection-sicher):

- UserRepository → User speichern/laden (DB)
- AuthRepository → User-Authentifizierung (DB)
- MediaRepository → Media CRUD (DB)
- RatingRepository → Ratings speichern/laden (DB)

In-Memory Repositories (für Unit-Tests):

- InMemoryUserRepository → Erbt von UserRepository, speichert in HashMap
- InMemoryAuthRepository → Implementiert IAuthRepository
- InMemoryMediaRepository → Implementiert IMediaRepository
- InMemoryRatingRepository → Erbt von RatingRepository

1.6 Datenbanktabellen

Table	Columns
users	userId, username, password
media	mediaId, userId, title, description, mediaType, releaseYear, genres
ratings	ratingId, mediaId, userId, stars, comment, confirmed

2. Lessons Learned

2.1 Software-Architektur für Wartbarkeit

Während der Entwicklung habe ich gelernt, dass eine gute Architektur essenziell für Erweiterbarkeit ist. Das Layered Architecture Pattern (Controller → Service → Repository) mit Dependency Injection ermöglicht:

Erkenntnisse:
- Änderungen isolieren: Neue Features brauchen keine Änderungen in bestehenden Schichten
- Interface-basierte Repositories: Test-Repositories (In-Memory) ersetzen „Echte“ Repositories (DB)
- Single Responsibility Principle: Jede Klasse hat genau eine Aufgabe
  - Controller: nur HTTP/JSON
  - Service: nur Geschäftslogik
  - Repository: nur Datenzugriff

Praktischer Nutzen:
- Neues Feature "Favoriten" → nur MediaService + MediaRepository ändern, Controller bleibt gleich → keine Regression-Risiken.

2.2 Infrastruktur mit Docker

Docker ermöglichte eine saubere, reproduzierbare Entwicklungsumgebung:

Erkenntnisse:
- PostgreSQL in Docker-Container statt lokale Installation
- docker-compose.yml definiert gesamte Infrastruktur (Java + DB + Networks)
- Environment Variables für Konfiguration (statt hard-coded)

Praktischer Nutzen:
- Alle Entwickler haben identische Umgebung
- Deployment = Container hochfahren

2.3 API-Testing mit Postman

Postman ist mehr als nur HTTP-Request Tool:

Erkenntnisse:
- Collections & Environments = API-Dokumentation, die funktioniert
- Pre-request Scripts automatisieren Token-Handling (Login → Token speichern)

- Tests in Sequence (Register → Login → Create → Rate)
- Export = andere Teams verstehen API ohne Code

Praktischer Nutzen:
- Integrationstest ohne Code schreiben
- API-Dokumentation, die immer aktuell bleibt

3. Unit-Testing Strategy & Coverage

Was ist Unit Testing?

Automatisierte Tests für einzelne Komponenten (isoliert).
Pattern: Arrange (Setup) → Act (Aktion) → Assert (Überprüfung)

Warum wichtig?

- Frühe Fehler erkennen
- Code-Qualität
- Sicheres Refactoring

In-Memory Repositories

Test nutzen HashMap statt DB → schnell, keine DB nötig

Code Coverage

Prozentsatz des getesteten Codes. Man sagt es sollten ca. 70% – 90% des Codes mindestens von Unit-Tests abgedeckt werden.

Coverage des Projekts

41% des gesamten Projekts. (43% der Application & 28% des Servers)

4. 2 SOLID Principles

S (Single Responsibility Principle) **Definition**: Jede Klasse hat genau eine Verantwortung.
Beispiel:

```
//RatingService - nur Geschäftslogik
public class RatingService {
    public boolean updateRating(Rating rating) {
        return ratingRepository.update(rating);
    }
}

//RatingController - nur HTTP-Handling
public class RatingController implements Controller {
    public Response handle(Request request) {
        //nur JSON-Parsing & HTTP
    }
}
```

O (Open-Closed Principle) **Definition**: Offen für Erweiterung, geschlossen für Modifikation.
Beispiel:

```
//Zwei Implementierungen (DB + In-Memory)
public class MediaRepository implements IMediaRepository { }
public class InMemoryMediaRepository implements IMediaRepository { }

//Service nutzt Interface
public class MediaService {
    public MediaService(IMediaRepository repository) {
        this.repository = repository;
    }
}
```

Git-Link:

https://github.com/detectivecamila/SWEN_MRP.git