# Projektdokumentation

## Dokumentation des Codes und der Architektur

### Einführung

Die Filmdatenbank ist eine Anwendung zur Verwaltung von Filmen, Benutzern und Bewertungen. Die Anwendung umfasst Backend-Komponenten, die mit Spring Boot und MongoDB implementiert sind, sowie ein Frontend, das mit React erstellt wurde. Diese Dokumentation beschreibt die Architektur der Anwendung, einschließlich der verwendeten Technologien, der Struktur des Codes und wie die verschiedenen Komponenten miteinander interagieren.

### Übersicht der Architektur

Die Anwendung besteht aus folgenden Hauptkomponenten:

1. **Backend (Spring Boot)**
   * Service-Layer
   * Repository-Layer
   * Controller-Layer
   * Model-Layer
2. **Frontend (React)**
3. **Datenbank (MongoDB)**

### Technologien

* **Spring Boot**: Für die Erstellung des Backends.
* **MongoDB**: Für die persistente Datenspeicherung.
* **React**: Für die Erstellung des Frontends.

### Backend

* **Service-Layer**

Der Service-Layer enthält die Geschäftslogik der Anwendung und interagiert mit den Repositorys, um Daten aus der Datenbank zu holen oder zu speichern.

* + **MovieService**: Diese Klasse verwaltet alle Operationen rund um Filme, einschließlich des Abrufens aller Filme, das Suchen nach Filmen nach Titel, Genre oder Regisseur sowie das Sortieren von Filmen nach Veröffentlichungsdatum oder Bewertung.
  + **ReviewService**: Diese Klasse verwaltet alle Operationen rund um Bewertungen, einschließlich der Erstellung, Löschung und Aktualisierung von Bewertungen sowie der Berechnung der durchschnittlichen Bewertung eines Films.
  + **UserService**: Diese Klasse verwaltet alle Operationen rund um Benutzer, einschließlich der Registrierung neuer Benutzer, der Aktualisierung von Benutzerprofilen und des Abrufens oder Löschens von Benutzern.
* **Repository-Layer**

Der Repository-Layer enthält die Schnittstellen zur Datenbank und definiert Methoden zum Abrufen und Speichern von Daten.

* + **MovieRepository**: Diese Schnittstelle definiert Methoden zum Abrufen von Filmen basierend auf verschiedenen Kriterien wie Titel, Genre und Regisseur.
  + **ReviewRepository**: Diese Schnittstelle definiert Methoden zum Abrufen von Bewertungen basierend auf Filmen oder Benutzern.
  + **UserRepository**: Diese Schnittstelle definiert Methoden zum Abrufen und Speichern von Benutzern.
* **Controller-Layer**

Der Controller-Layer enthält die REST-Controller, die HTTP-Anfragen entgegennehmen und Antworten zurückgeben. Jeder Controller ruft die entsprechenden Methoden des Service-Layers auf.

* + **MovieController**: Dieser Controller verarbeitet HTTP-Anfragen für Filmoperationen, wie das Abrufen aller Filme, das Suchen nach Filmen nach Titel oder Genre und das Sortieren von Filmen.
  + **ReviewController**: Dieser Controller verarbeitet HTTP-Anfragen für Bewertungsoperationen, wie das Erstellen, Löschen und Aktualisieren von Bewertungen sowie das Abrufen von Bewertungen nach Benutzer oder Film.
  + **UserController**: Dieser Controller verarbeitet HTTP-Anfragen für Benutzeroperationen, wie die Registrierung neuer Benutzer, die Aktualisierung von Benutzerprofilen und das Abrufen oder Löschen von Benutzern.
* **Model-Layer**

Der Model-Layer enthält die Datenmodelle, die die Struktur der Daten in der Anwendung definieren.

* + **Movie**: Diese Klasse repräsentiert einen Film und enthält Attribute wie Titel, Regisseur, Schauspieler, Handlung, Veröffentlichungsdatum, Genre, Poster, Hintergrundbilder und Trailer.
  + **Review**: Diese Klasse repräsentiert eine Bewertung und enthält Attribute wie den Bewertungstext, die Bewertungspunktzahl, den Ersteller der Bewertung, das zugehörige Film-IMDb-ID und Zeitstempel für die Erstellung und Aktualisierung.
  + **User**: Diese Klasse repräsentiert einen Benutzer und enthält Attribute wie Benutzername, Passwort und E-Mail. Jeder Benutzer hat auch ein Profil.
  + **UserProfile**: Diese Klasse repräsentiert das Profil eines Benutzers und enthält zusätzliche Informationen über den Benutzer, wie z.B. eine Liste von Bewertungen, die der Benutzer erstellt hat.

### Frontend (React)

Das Frontend der Anwendung wurde mit React erstellt und bietet eine Benutzeroberfläche für die Interaktion mit den Filmen, Bewertungen und Benutzern. Es umfasst folgende Hauptfunktionen:

* Anzeige einer Liste von Filmen.
* Suche nach Filmen nach Titel oder Genre.
* Anzeigen und Erstellen von Bewertungen für Filme.
* Verwaltung von Benutzerprofilen.

### Datenbank (MongoDB)

Die Anwendung verwendet MongoDB zur persistenten Speicherung von Daten. Die wichtigsten Datenmodelle sind:

* **Movie**: Enthält Informationen über Filme (z.B. Titel, Regisseur, Schauspieler, Genre).
* **Review**: Enthält Benutzerbewertungen und Rezensionen zu Filmen.
* **User**: Enthält Benutzerinformationen und Profile.

### Kommunikation zwischen Komponenten

1. **Frontend (React)**
2. **Backend (Spring Boot)**
3. **Datenbank (MongoDB)**

Die Kommunikation zwischen diesen Komponenten erfolgt hauptsächlich über HTTP- und REST-APIs.

### Kommunikationsfluss

1. **Frontend zu Backend**:
   * **HTTP-Anfragen**: Das React-Frontend kommuniziert mit dem Spring Boot-Backend über HTTP-Anfragen (GET, POST, PUT, DELETE). Jede Anfrage wird an einen bestimmten Endpoint gesendet, der in den Controllern definiert ist.
   * **Beispiel**: Wenn ein Benutzer eine Liste aller Filme anzeigen möchte, sendet das Frontend eine GET-Anfrage an den Endpoint /movies, der von MovieController verarbeitet wird.
2. **Backend zu Datenbank**:
   * **Repository-Aufrufe**: Das Backend kommuniziert mit der MongoDB-Datenbank über Repository-Schnittstellen (z.B. MovieRepository, ReviewRepository, UserRepository). Diese Repositorys nutzen MongoDB-Abfragen, um Daten zu speichern oder abzurufen.
   * **Beispiel**: Der MovieService ruft die Methode findAll von MovieRepository auf, um alle Filme aus der Datenbank zu holen.

### Architekturmuster

Die MovieApp verwendet eine **schichtbasierte Architektur**, die in folgende Schichten unterteilt ist:

1. **Präsentationsschicht** (Frontend):
   * Implementiert mit React.
   * Verantwortlich für die Benutzeroberfläche und die Benutzerinteraktionen.
2. **Anwendungsschicht** (Controller):
   * Implementiert mit Spring Boot.
   * Behandelt HTTP-Anfragen und steuert den Datenfluss zwischen der Präsentations- und der Logikschicht.
3. **Logikschicht** (Service):
   * Implementiert mit Spring Boot.
   * Enthält die Geschäftslogik der Anwendung.
4. **Datenzugriffsschicht** (Repository):
   * Implementiert mit Spring Boot.
   * Verantwortlich für den Zugriff auf die Datenbank (MongoDB).

### Architektonische Entscheidungen (ADR)

**ADR 1: Verwendung von Spring Boot**

* **Entscheidung**: Spring Boot als Backend-Framework verwenden.
* **Kontext**: Wir benötigten ein robustes, weit verbreitetes Framework zur Erstellung des Backends, das die Integration mit verschiedenen Datenbanken und anderen Technologien erleichtert.
* **Begründung**: Spring Boot bietet umfangreiche Unterstützung für RESTful Webservices, eine einfache Konfiguration und eine große Entwickler-Community. Es erleichtert die Entwicklung und Wartung des Backends erheblich.
* **Konsequenzen**: Die Anwendung profitiert von der Stabilität und den Funktionen von Spring Boot, aber Entwickler müssen mit der Spring Boot-Architektur vertraut sein.

**ADR 2: Verwendung von React für das Frontend**

* **Entscheidung**: React als Frontend-Bibliothek verwenden.
* **Kontext**: Wir benötigten eine leistungsstarke, komponentenbasierte Bibliothek zur Erstellung einer dynamischen Benutzeroberfläche.
* **Begründung**: React ermöglicht die Erstellung von wiederverwendbaren UI-Komponenten und bietet eine schnelle und reaktive Benutzererfahrung. Die große Community und die umfangreiche Dokumentation erleichtern die Entwicklung.
* **Konsequenzen**: Die Anwendung kann eine schnelle und interaktive Benutzeroberfläche bereitstellen, erfordert jedoch Kenntnisse in React und seinem Ökosystem.

**ADR 3: Verwendung von MongoDB als Datenbank**

* **Entscheidung**: MongoDB als Datenbank verwenden.
* **Kontext**: Wir benötigten eine skalierbare, dokumentenorientierte Datenbank, die flexibel genug ist, um die verschiedenen Datenmodelle der Anwendung zu speichern.
* **Begründung**: MongoDB bietet hohe Leistung und Skalierbarkeit und ist gut für die Speicherung von JSON-ähnlichen Dokumenten geeignet. Es ermöglicht flexible Schema-Designs und passt gut zu den Anforderungen der Anwendung.
* **Konsequenzen**: Die Anwendung kann große Mengen an Daten effizient speichern und abrufen, erfordert jedoch Kenntnisse in der Verwaltung und Optimierung von MongoDB.

### Code-Dokumentationsprinzipien

* Die Klassen im Projekt sind klar dokumentiert, mit Kommentaren, die die Hauptfunktionen beschreiben. Die Methoden haben beschreibende Namen und sind konsistent formatiert.
* Die HTTP-Endpunkte sind klar benannt und dokumentiert.
* Die Schnittstellenmethoden sind klar definiert und dokumentiert, um den Datenzugriff zu erklären.

### SOLID-Prinzipien

* **Single Responsibility Principle (SRP)**
  + Definition: Eine Klasse sollte nur eine einzige Verantwortlichkeit haben.
  + Anwendung im Code:
    - MovieService: Zuständig für die Verwaltung der Filmoperationen.
    - ReviewService: Zuständig für die Verwaltung der Reviewoperationen.
    - UserService: Zuständig für die Verwaltung der Benutzeroperationen.
    - MovieController: Zuständig für die Handhabung der HTTP-Anfragen für Filme.
    - ReviewController: Zuständig für die Handhabung der HTTP-Anfragen für Reviews.
    - UserController: Zuständig für die Handhabung der HTTP-Anfragen für Benutzer.
* **Open/Closed Principle (OCP)**
  + Definition: Software-Entitäten (Klassen, Module, Funktionen, etc.) sollten offen für Erweiterungen, aber geschlossen für Änderungen sein.
  + Anwendung im Code:
    - Die Services und Controller sind so strukturiert, dass sie leicht erweitert werden können, ohne bestehende Implementierungen ändern zu müssen. Zum Beispiel kann man leicht neue Methoden in MovieService oder ReviewService hinzufügen, ohne bestehende Methoden zu ändern.
    - Die Repositories folgen den Interfaces von Spring Data, was Erweiterungen ermöglicht, ohne den bestehenden Code zu ändern.
* **Liskov Substitution Principle (LSP)**
  + Definition: Objekte in einem Programm sollten durch Instanzen ihrer Untertypen ersetzbar sein, ohne das korrekte Verhalten des Programms zu verändern.
  + Anwendung im Code:
    - Die Verwendung von Interfaces für die Repositories (MovieRepository, ReviewRepository, UserRepository) und die Implementierung dieser Interfaces durch Spring Data MongoDB garantiert, dass diese Prinzipien eingehalten werden. Man könnte beispielsweise MovieRepository durch eine andere Implementierung ersetzen, ohne den MovieService zu ändern.
* **Interface Segregation Principle (ISP)**
  + Definition: Es sollten viele spezifische Interfaces anstelle eines allgemeinen Interfaces bevorzugt werden.
  + Anwendung im Code:
    - Die Repositories sind spezifisch für ihre jeweiligen Domänenobjekte (Movie, Review, User). Dies folgt dem ISP, da jede Repository-Schnittstelle nur die Methoden enthält, die für den spezifischen Domänenkontext relevant sind.
* **Dependency Inversion Principle (DIP)**
  + Definition: Abstraktionen sollten nicht von Details abhängen. Details sollten von Abstraktionen abhängen.
  + Anwendung im Code:
    - Die Services (MovieService, ReviewService, UserService) hängen von abstrakten Repositories (MovieRepository, ReviewRepository, UserRepository) ab, nicht von konkreten Implementierungen. Dies wird durch die Verwendung von Dependency Injection durch Spring Boot erreicht.

## Git-Vorgehen und Branching-Strategie

In diesem Projekt verwenden wir eine Trunk-Based Development strategie, um sicherzustellen, dass die Codebasis stabil bleibt und Änderungen nachvollziehbar sind. Diese Strategie ermöglicht es uns, effizient und kollaborativ zu arbeiten, indem wir kleinere, inkrementelle Änderungen in den main-Branch integrieren. Hier ist eine detaillierte Dokumentation unseres Git-Vorgehens und der Branching-Strategie:

**Hauptbranch**

* **main Branch**:
  + Der main-Branch enthält die stabile Version des Codes. Er repräsentiert die produktionsreife Version der Anwendung und wird regelmäßig aktualisiert.
  + Änderungen am main-Branch werden ausschließlich über Merge-Requests (Pull-Requests) integriert.

**Branching-Strategie**

1. **Erstellen eines Feature-Branches**:
   * Für jede neue Funktion, Bugfix oder Änderung erstellen wir einen separaten Branch vom main-Branch.
2. **Entwicklung auf dem Feature-Branch**:
   * Die erforderlichen Änderungen oder Erweiterungen werden auf dem Feature-Branch durchgeführt.
   * Nach Abschluss der Änderungen werden die Änderungen committet und auf den Remote-Branch gepusht.
3. **Erstellen eines Merge-Requests (MR)**:
   * Nachdem der Feature-Branch gepusht wurde, erstellen wir einen Merge-Request auf GitLab.
   * Der Merge-Request sollte eine klare Beschreibung der durchgeführten Änderungen und den Grund für die Änderung enthalten.
   * Wir stellen sicher, dass alle Merge-Requests kleine, überprüfbare Änderungen enthalten, um den Überprüfungsprozess zu erleichtern.
4. **Code Review und Merge**:
   * Kommilitoren überprüfen den Merge-Request, geben Feedback und fordern ggf. Änderungen an.
   * Nach der erfolgreichen Überprüfung und dem Bestehen aller Tests wird der Merge-Request in den main-Branch gemerged.
   * Falls Konflikte auftreten, werden diese auf dem Feature-Branch gelöst, bevor der Merge-Request erneut zur Überprüfung eingereicht wird.

## CI/CD Pipeline für meine Filmdatenbank-Applikation

Unsere CI/CD-Pipeline ist entscheidend für die Automatisierung und Qualitätssicherung unserer Filmdatenbank-Applikation. Sie besteht aus mehreren Phasen, die sicherstellen, dass jede Code-Änderung gründlich getestet und effizient bereitgestellt wird.

### Phasen und Jobs

**Build Phase**

* **build-backend:** Kompiliert unseren Backend-Code mit Maven, um sicherzustellen, dass er korrekt funktioniert und bereit ist für die Tests.
* **build-frontend:** Baut das Frontend unserer Applikation, stellt sicher, dass alle Node.js-Abhängigkeiten installiert sind und führt den Build-Prozess durch.

**Test Phase**

* **test-backend:** Führt umfassende Tests für unseren Backend-Code durch. Wir nutzen Maven, um alle Tests zu starten und die Ergebnisse in JUnit-Reports festzuhalten.
* **test-frontend:** Hier werden die Tests für unser Frontend ausgeführt. Bevor wir die Tests starten, stellen wir sicher, dass alle npm-Pakete installiert sind, und archivieren dann die Testergebnisse.

**Package Phase**

* **package-backend:** Packt unser Backend in eine JAR-Datei, die bereit ist für die Produktion. Dies geschieht nur, wenn Änderungen im “main-branch“ gemacht wurden, und die JAR-Datei wird als Artefakt für zwei Tage gespeichert.

### Weitere Konfigurationen

* **SAST-Template:** Beinhaltet ein Template für Static Application Security Testing (SAST), um potenzielle Sicherheitslücken frühzeitig zu erkennen und zu beheben.
* **Caching:** Optimiert die Build-Zeit durch das Zwischenspeichern von Maven-Repository und Node-Modulen.
* **Variablen und Einstellungen:** MAVEN\_OPTS wird gesetzt, um das lokale Maven-Repository zu verwenden, was die Konsistenz unserer Abhängigkeiten sicherstellt.

## Schnittstellen

### Schnittstellendokumentation

🡪 siehe Datei: Backend/README.md

### Schnittstellenart: Synchron und Asynchron

**Synchrone Schnittstellen**

In unserem Projekt werden hauptsächlich synchrone Schnittstellen verwendet. Diese Art der Schnittstelle ist ideal für CRUD-Operationen und Authentifizierungsprozesse, bei denen eine sofortige Rückmeldung vom Server an den Client erforderlich ist. Diese Operationen erfordern, dass der Client auf die Antwort des Servers wartet, um die Benutzerinteraktion in Echtzeit abzuschließen, wie z.B. das Erstellen eines neuen Benutzers oder das Hinzufügen eines Films zu den Favoriten.

**Asynchrone Schnittstellen**

Asynchrone Schnittstellen werden eingesetzt, um langwierige Operationen auszuführen, ohne die Hauptanwendung zu blockieren. Sie sind besonders nützlich, wenn die Antwortzeit des Servers variieren kann und der Client nicht sofort auf das Ergebnis warten muss. In diesem Projekt wird die folgende asynchrone Schnittstelle verwendet:

* **Fetch All Movies (@Async public CompletableFuture<List<Movie>> allMovies())**
  + **URL:** /movies/
  + **Method:** GET
  + **Response:** Gibt ein CompletableFuture mit einer Liste von Filmen zurück

Diese Methode wird in einem separaten Thread ausgeführt, wodurch die Hauptanwendung nicht blockiert wird und der Client sofort eine CompletableFuture erhält. Dies ist sinnvoll für Operationen, bei denen große Datenmengen verarbeitet oder auf externe Ressourcen zugegriffen wird.

## Threads

**1. Haupt-Thread der Anwendung**

Der Haupt-Thread der Anwendung überwacht die Ausführung der Hauptlogik, einschließlich der Verarbeitung von HTTP-Anfragen und der Koordination der Aufgaben zwischen den verschiedenen Threads.

**2. HTTP-Server-Thread**

Der HTTP-Server-Thread wird vom Spring Framework verwaltet und ist dafür zuständig, eingehende HTTP-Anforderungen entgegenzunehmen, sie an die entsprechenden Controller weiterzuleiten und die HTTP-Antworten zurückzugeben. Dieser Thread stellt die Verbindung zwischen der Anwendung und den Benutzern/externen Systemen her.

**3. Datenbank-Verbindungs-Thread(s)**

Beim Zugriff auf die MongoDB-Datenbank werden spezielle Threads verwendet, um Datenbankanfragen zu senden, die Ergebnisse zu verarbeiten und die notwendigen Operationen (z.B. Lesen, Schreiben, Löschen) auszuführen. Diese Threads gewährleisten die Datenintegrität und effiziente Datenverarbeitung.

**4. Asynchrone Threads (@Async)**

In unserem Projekt verwenden wir eine @Async-Methode, um die lang laufende Operation des Abrufs aller Filme asynchron auszuführen. Durch die Nutzung von @Async können wir die Leistungsfähigkeit unserer Anwendung verbessern, insbesondere bei Operationen wie dem umfangreichen Abrufen von Daten aus der Datenbank.