

Projektdokumentation – Dragonball (Backend / Frontend)

Modul: M223 - Multi-User-Applikationen objektorientiert realisieren

Beschreibung: Multi-User-Applikation mit transaktionsfähiger Datenbank

Name: Thierno Hamidou Bah

Klasse: UIFZ

Projekt: Dragonball

Abgabedatum: Dezember 2025

Dozent: Graziano Spina

Motivation

Die Motivation für dieses Projekt entstand aus meiner Begeisterung für die Dragonball-Welt sowie dem Wunsch, eine vollständige Webanwendung zu entwickeln, die sowohl ein modernes React-Frontend als auch ein sicheres Spring-Boot-Backend umfasst. Das Projekt sollte nicht nur Daten anzeigen, sondern eine echte, geschützte Benutzerinteraktion ermöglichen – mit Login, Rollenverwaltung, Favoritenlisten und einer vollständigen CRUD-Verwaltung für Charaktere. Der Schwerpunkt lag darauf, ein System aufzubauen, das wie eine professionelle, produktive Applikation funktioniert:

- **Ein Backend**, das mit JWT-Authentifizierung, klarer Schichtenarchitektur (Controller → Service → Repository) und einer relationalen Datenbank robuste Geschäftslogik bereitstellt.
- **Ein Frontend**, das mithilfe von React, Routing, Formularvalidierung und einem globalen Auth-State eine nutzerfreundliche Oberfläche bietet.
- **Eine Rollenlogik (Admin/Player)**, die unterschiedliche Funktionen erlaubt – ein realistisches Szenario, wie es auch in produktiven Unternehmensanwendungen eingesetzt wird.
- **Eine Favoritenfunktion**, welche zeigt, wie benutzerspezifische Beziehungen zwischen Datenbanktabellen modelliert und genutzt werden.

Das Projekt bot zudem die Gelegenheit, zentrale Entwicklungskonzepte praktisch zu erlernen: sichere Passwortspeicherung, Transaktionen im Backend, API-Kommunikation, State-Management im Frontend sowie saubere und wiederverwendbare Komponenten. Die Integration beider Systeme vermittelt ein realistisches Gesamtbild davon, wie moderne Full-Stack-Anwendungen geplant, implementiert und miteinander verbunden werden.

Durch die Kombination aus technischer Herausforderung und persönlichem Interesse entstand eine Anwendung, die nicht nur funktional ist, sondern auch als Grundlage für spätere Erweiterungen dienen kann – etwa zusätzliche Rollen, mehr Charakterattribute oder ein erweitertes Rechtemanagement.

Inhaltsverzeichnis

Inhalt

Projektidee	2
Anforderungskatalog	2
Klassendiagramm & Architectur	4
API Integration (Service Layer)	14
JWT-Authentifizierungs-Flow	16

Testprotokoll	18
Installationsanleitung	29
Technologiestack	31
Feedback.....	33
Rolle der Unterrichtsunterlagen.....	34
Reflexion	35
Quellen & Hilfsmittel	35

Projektidee

Die Dragonball-Charakterdatenbank ist eine vollständige Full-Stack-Webapplikation zur sicheren Verwaltung, Anzeige und Interaktion mit Dragonball-Charakteren. Die Anwendung kombiniert ein modernes React-Frontend mit einem Spring-Boot-Backend, das über JWT-Tokens geschützt ist und unterschiedliche Benutzerrollen (Player und Admin) unterstützt.

Spieler können sich anmelden, Charaktere anzeigen und persönliche Favoritenlisten verwalten. Administratoren verfügen zusätzlich über erweiterte Rechte und können Charaktere erstellen, bearbeiten und löschen. Sämtliche Daten – inklusive Benutzer, Rollen, Favoriten sowie Charakterattribute – werden in einer relationalen Datenbank gespeichert und über eine klar strukturierte REST-API bereitgestellt.

Das Projekt verfolgt das Ziel, eine realitätsnahe Webapplikation umzusetzen, welche moderne Themen der Softwareentwicklung integriert: Authentifizierung, sichere Passwortspeicherung, Transaktionen, Rollenmanagement, Komponentenarchitektur, Routing, State-Management sowie die Kommunikation zwischen Frontend und Backend. Dadurch entsteht ein praxisorientiertes Lernumfeld, das ideal für Lernende und Entwickler ist, die professionelle Webentwicklung mit React und Spring Boot vertiefen möchten.

Anforderungskatalog

Funktionale Anforderungen – Backend (Server)

Benutzer & Sicherheit

- Das System muss eine Benutzeranmeldung per **JWT-Authentifizierung** ermöglichen.
- Passwörter müssen **gehasht (BCrypt)** gespeichert werden.
- Benutzerrollen müssen unterstützt werden:
 - **PLAYER:** darf Charaktere ansehen & Favoriten verwalten
 - **ADMIN:** darf zusätzlich Charaktere anlegen, bearbeiten und löschen

- Nur authentifizierte Benutzer dürfen geschützte Endpunkte aufrufen.

Charakterverwaltung

- Das Backend muss vollständige **CRUD-Operationen** bereitstellen:
 - Charakter erstellen
 - Charakter bearbeiten
 - Charakter löschen
 - Charaktere aus DB abrufen
- Das System muss Daten in einer relationalen **PostgreSQL-Datenbank** speichern.
- Charaktere müssen Attribute wie Name, Race, Gender, KI, Max-KI, Affiliation, Villain besitzen.

Favoritenfunktion

- Spieler müssen Charaktere **zu Favoriten hinzufügen** können.
- Spieler müssen Favoriten **entfernen** können.

Transaktionen

- Kritische Operationen wie Favoriten hinzufügen/entfernen und Charakter löschen müssen **transaktional (@Transactional)** sein, damit inkonsistente Daten verhindert werden.
-

Funktionale Anforderungen – Frontend (Client)

Benutzeroberfläche

- Eine klare, navigierbare Seitenstruktur muss vorhanden sein:
 - Home
 - Z-Fighters
 - Villains
 - Favorites
 - Manage Characters (Admin)
 - Login
- Das UI muss auf **React + React Router** basieren.

Charakteranzeige

- Alle Charaktere müssen übersichtlich auf Karten dargestellt werden.
- Detailseiten sollen vollständige Informationen anzeigen.

CRUD für Admin

- Administratoren müssen über ein Formular Charaktere **erstellen** können.

- Administratoren müssen bestehende Charaktere **bearbeiten** können.
- Administratoren müssen Charaktere **löschen** können.

Favoritenverwaltung

- Spieler müssen Charaktere als Favorit markieren können.
- Favoriten müssen in einer separaten Ansicht angezeigt werden.

Klassendiagramm & Architectur

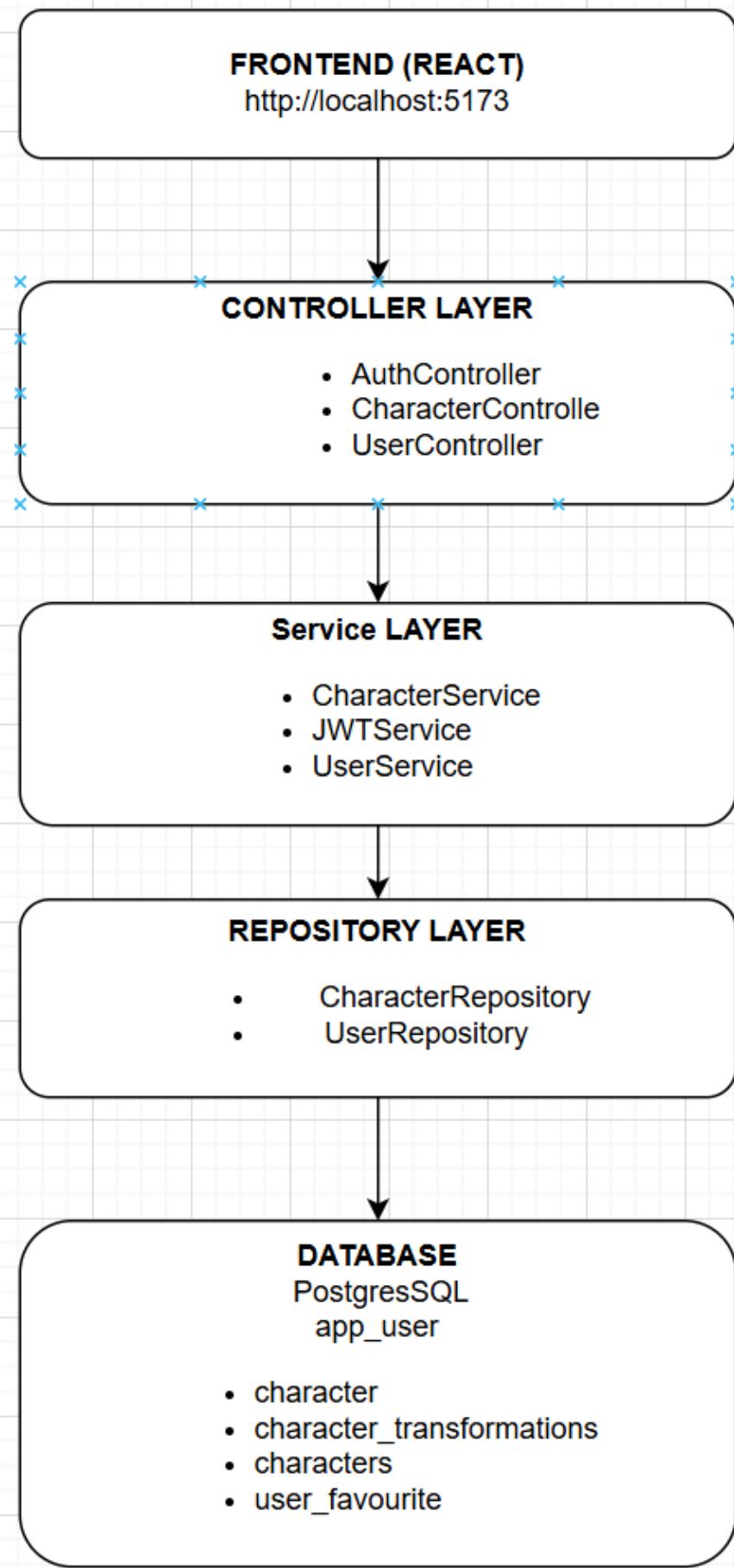
❖ Backend-Architektur

Die Applikation folgt einer **klassischen 4-Schichten-Architektur**, bestehend aus **Frontend**, **Controller-Layer**, **Service-Layer**, **Repository-Layer** und der **Datenbank**.

Jede Schicht hat eine klar definierte Aufgabe und kommuniziert ausschliesslich mit der direkt darunterliegenden Schicht.

Das sorgt für **Sauberkeit**, **Wartbarkeit**, **Testbarkeit** und **klare Verantwortlichkeiten**.

1. Architektur-Diagramm (Layer-Architektur)



◆ 1. Frontend (React)

Ort: <http://localhost:5173>

Das Frontend ist ein **React-Client**, der die Benutzeroberfläche bereitstellt.

Es sendet HTTP-Requests an das Spring-Boot-Backend und zeigt Ergebnisse wie:

- Charakter-Listen
- Details
- Favoriten
- Login / Token-Handling
- Admin-Funktionen (Create, Edit, Delete)

Das Frontend enthält **keine Business-Logik** – es zeigt nur die Daten an und sendet Aktionen an das Backend.

◆ 2. Controller Layer (REST API)

Hier befinden sich alle REST-Endpunkte, die vom Frontend aufgerufen werden:

- **AuthController**
 - Login
 - Registrierung
 - liefert JWT-Token
- **CharacterController**
 - CRUD-Operationen für Charaktere (Admin)
 - Listen, Suchfunktionen
 - Detailansicht
 - Z-Fighters / Villains Filter
- **UserController**
 - Favoriten hinzufügen / entfernen
 - Favoritenliste abrufen
 - Admin: Nutzer anzeigen

👉 Aufgabe der Controller:

Sie empfangen HTTP-Anfragen, validieren minimal, rufen passende Service-Methoden auf und geben HTTP-Responses zurück.

Sie enthalten **keine Geschäftslogik**, sondern sind der „Eingangspunkt“ des Backends.

◆ 3. Service Layer (Business Logic)

Der Service-Layer führt die komplette **Geschäftslogik** aus.

Er steht im Zentrum der Anwendung.

CharacterService

- Validierung von Daten
- Erstellen / Bearbeiten / Löschen von Charakteren
- Laden aller Charaktere
- Filtern nach Rasse / Villains / Z-Fighters
- Kapselt Transaktionen für Datenbankänderungen

UserService

- Benutzer laden
- Passwörter encoden (BCrypt)
- Favoritenverwaltung (Add/Remove)
- Laden aller Nutzer (Admin)

JWTService

- Erzeugt Tokens
- Validiert Tokens
- Extrahiert Benutzerinformationen

Der Service-Layer stellt sicher, dass die Regeln der Anwendung korrekt umgesetzt werden und kapselt komplexe Logik.

Er kommuniziert ausschliesslich mit den Repositories.

◆ 4. Repository Layer (Datenzugriff)

Dieser Layer kommuniziert **direkt mit der Datenbank** über Spring Data JPA.

CharacterRepository

- CRUD für Charaktere
- Custom Queries für Filter (race, villain...)

UserRepository

- Nutzerverwaltung
- FindByUsername für Login/JWT

👉 Der Repository-Layer enthält **keine Logik**, sondern nur Datenzugriff.

◆ 5. Datenbank (PostgreSQL)

Die Datenbank speichert persistente Daten.

Kern-Tabellen:

app_user

- id
- username
- password (BCrypt gehasht)
- role (ADMIN / PLAYER)

character

- id
- name
- race
- ki / maxKi
- affiliation
- gender
- villain (Boolean)
- image_url
- weitere Charakterattribute

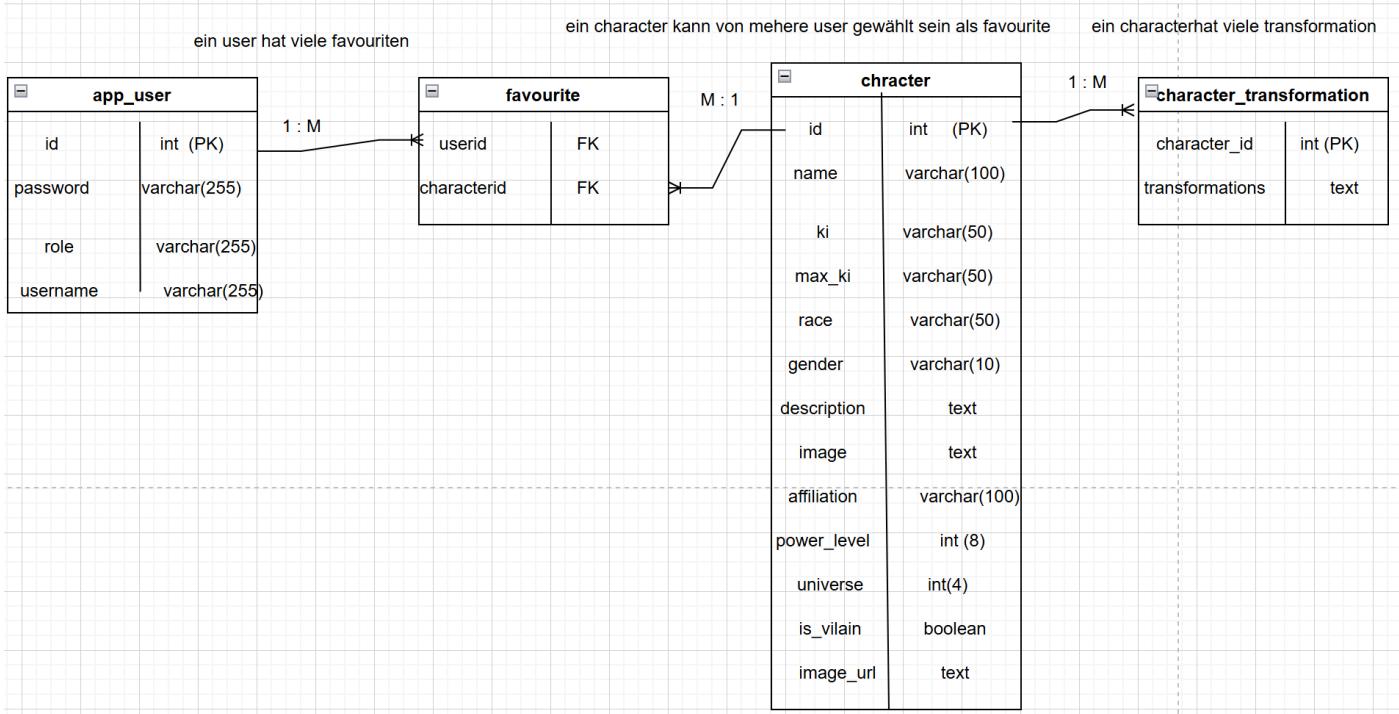
character_transformations

- 1:N Beziehung zu Charakter

user_favourite

- Mapping-Tabelle für N:M
- verbindet Nutzer ↔ Charaktere

2. Datenbank-Diagramm (ER-Diagramm)



❖ Frontend-Architektur

➤ Component-Architektur (Hierarchie)

Die folgende Struktur zeigt den kompletten Aufbau des Frontends, gegliedert nach Haupt-Komponenten, Layout-Komponenten und allen einzelnen Seiten (Pages).

Sie beschreibt welche Komponenten ineinander verschachtelt sind und welche Rolle jede Schicht im Projekt erfüllt.

📁 App.jsx – Wurzelkomponente der gesamten Anwendung

App.jsx enthält das gesamte Routing-System und definiert, welche Seite unter welcher URL angezeigt wird.

Alle Seiten werden **innerhalb des Layouts** geladen.

```
App.jsx
  └── Layout.jsx
    |   └── Navigation.jsx
    |   └── <Outlet /> (React Router)
    └── Footer.jsx
  └── Pages:
    ├── Home.jsx
    ├── Z_Fighters.jsx
    ├── Villains.jsx
    ├── CharacterDetail.jsx
    ├── FavoritesPage.jsx
    ├── ManageCharacters.jsx (nur Admin)
    ├── Login.jsx
    └── Forbidden.jsx
```

brick Layout-Schicht

► Layout.jsx

- Gemeinsamer Rahmen aller Seiten.
- Wird einmal gerendert und enthält Navigation oben und Footer unten.
- Das <Outlet /> ist ein Platzhalter:
Alle Seiten (Home, Villains, Login usw.) werden dort eingeblendet.

► Navigation.jsx

- Wird auf jeder Seite angezeigt.
- Zeigt Links basierend auf Login-Status und Rolle (Admin/Player).
- Admin sieht zusätzliche Menüpunkte („Manage Characters“).

► Footer.jsx

- Globaler Footer, ebenfalls auf jeder Seite sichtbar.

file Pages – Jede Seite ist eine eigene Komponente

1. Home.jsx

- Zeigt alle Charaktere (API + selbst erstellte Charaktere).
- Von hier kann man auch zu den Detailseiten navigieren.

2. Z_Fighters.jsx

- Filtert und zeigt nur Helden („Z-Fighters“).

3. Villains.jsx

- Filtert und zeigt nur Bösewichte.

4. CharacterDetail.jsx

- Zeigt alle Details eines Charakters.
- Buttons für „Favorite“, „Edit“, „Delete“ (abhängig von Rolle und Eigentümer).

5. FavoritesPage.jsx

- Auflistung aller Favoriten des eingeloggten Users.
- Holt Daten über /users/me/favourites.

6. ManageCharacters.jsx (Admin-only)

- Admin-Tool zum **Erstellen**, **Bearbeiten** und **Löschen** von Charakteren.
- Kommuniziert direkt mit dem Backend (CRUD).

7. Login.jsx

- Login-Formular.
- Ruft /auth/login auf und speichert den JWT im LocalStorage.

8. Forbidden.jsx

- Wird angezeigt, falls ein Nutzer eine Seite öffnen will, die seine Rolle nicht erlaubt.

➤ State-Manager-Flow

Der folgende Flow beschreibt, wie Authentifizierung und globale Benutzerdaten im Frontend verwaltet werden. Das Ziel ist, dass der Login-Status über alle Seiten hinweg erhalten bleibt, der JWT-Token an jede Backend-Anfrage angehängt wird und nicht autorisierte Zugriffe korrekt abgefangen werden.

◆ AuthContext (Global State)

Im Projekt wird ein globaler AuthContext verwendet, der zentral alle Authentifizierungs-Informationen hält:

```
AuthContext (Global State)
  user: { username, role, userId }
  token: <jwt>
  isAuthenticated: boolean
  login(username, password)
  logout()
  |
  ▼
<App />
  |
  ▼
Komponenten via useAuth(): user, isAuthenticated, login, logout
```

◆ **user**

Enthält die wichtigsten Informationen des aktuell eingeloggten Nutzers:

- username
- role (PLAYER oder ADMIN)
- userId

Diese Daten kommen direkt aus dem Backend-Login-Response.

◆ **token**

Speichert den JWT-Token, der vom Backend erzeugt wird.

Der Token wird für **jede geschützte Anfrage** benötigt:

Authorization: Bearer <jwt>

◆ **isAuthenticated**

Ein boolescher Zustand, der steuert:

- welche Seiten sichtbar sind
- welche Buttons angezeigt werden
- ob geschützte Routen betreten werden dürfen

Beispiele:

- Spieler kann Favoriten sehen
- Admin kann Charaktere verwalten

- Unautorisierte Nutzer werden auf /login oder /forbidden geleitet
-

◆ **login()**

Führt folgenden Ablauf aus:

1. POST /auth/login an das Backend schicken
 2. Backend validiert Benutzer + Passwort
 3. Backend sendet:
`{ token, username, role, userId }`
 4. AuthContext speichert diese Daten
 5. Token + User werden in localStorage persistiert
 6. isAuthenticated wird zu true
-

◆ **logout()**

Entfernt:

- Token aus localStorage
- User aus localStorage
- Setzt isAuthenticated = false
- Frontend leitet zurück zu Login

◆ **React-Komponenten nutzen den AuthContext**

`<App />`



Komponenten via `useAuth()`:

`user, isAuthenticated, login, logout`

◆ **Persistenz im Browser**

Damit ein Reload nicht zum Logout führt:

`localStorage speichert:`

`authToken`

`userData`

◆ **Automatischer JWT-Anhang bei jeder Anfrage**

Der apiClient (Axios-Instance) hängt automatisch den Token an:

Authorization: Bearer <token>

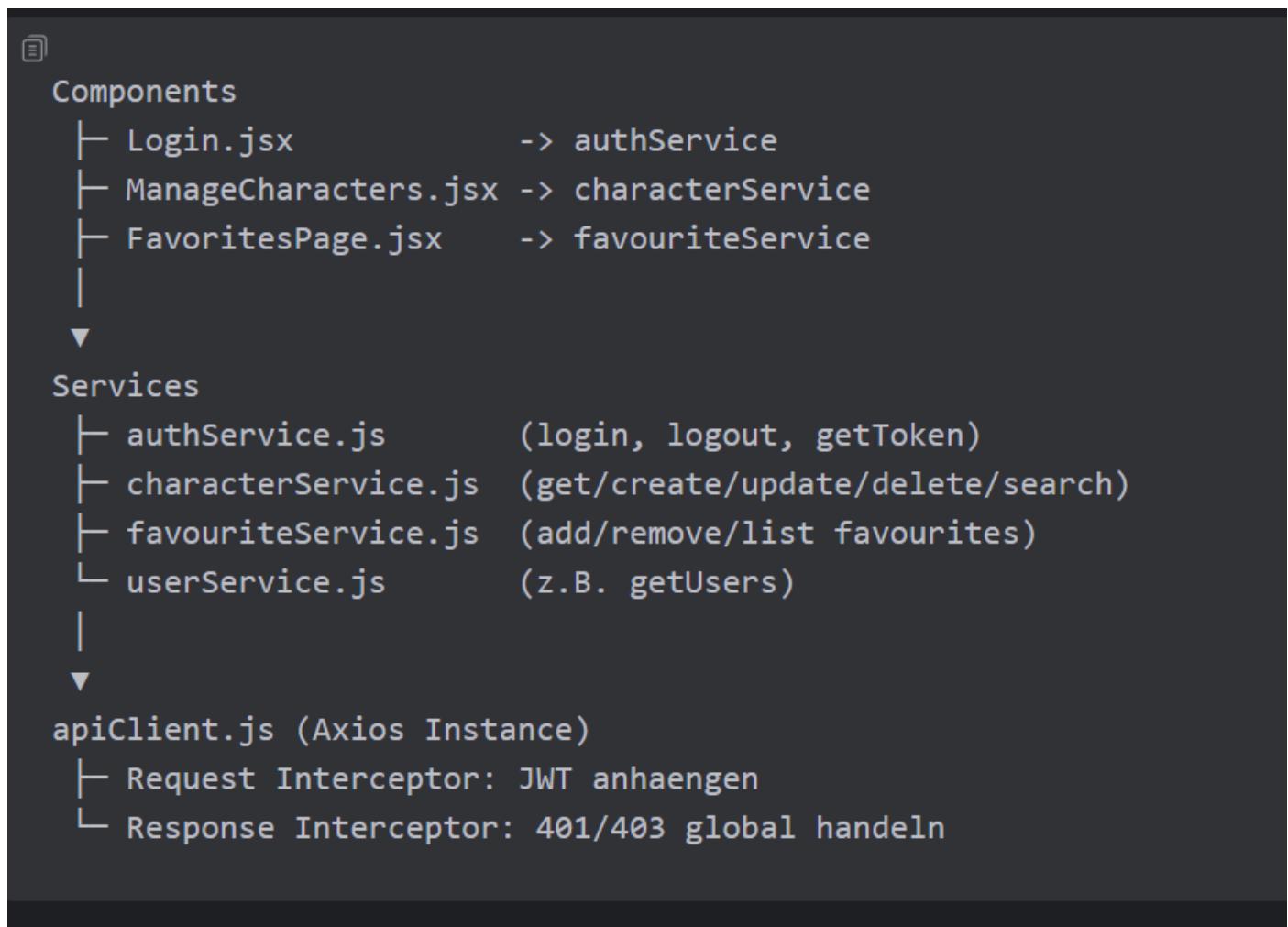
Falls das Backend 401 (unautorisiert) oder 403 (verboten) zurückgibt:

- User wird ausgeloggt oder
- auf /forbidden weitergeleitet

Dadurch wird die Sicherheit des Systems sichergestellt.

API Integration (Service Layer)

Das folgende Diagramm beschreibt, wie das React-Frontend mit dem Backend kommuniziert, welche Services beteiligt sind und wie JWT-Token transportiert werden.



❖ API-Integrations-Diagramm

1. Components (Frontend-Seite)

Bestimmte React-Components rufen explizit Services auf, anstatt direkt HTTP-Requests zu senden. Dadurch bleibt die UI sauber, modular und wiederverwendbar.

- **Login.jsx**
 - nutzt authService (login, logout, token speichern)

- **ManageCharacters.jsx** (nur Admin)
→ nutzt characterService (create/update/delete/find)
- **FavoritesPage.jsx**
→ nutzt favouriteService (add, remove, list favourites)

➡ Diese Komponenten lösen User-Interaktionen aus, aber sprechen **nie direkt mit Axios**.

2. Services (Middle Layer im Frontend)

Services enthalten die komplette Logik für REST-Kommunikation.

Service	Verantwortlich für
authService.js	Login, Logout, Token speichern/lesen
characterService.js	CRUD für Charaktere + Suchen
favouriteService.js	Favoriten hinzufügen/entfernen/laden
userService.js	Admin-Operationen (z. B. alle User auflisten)

3. apiClient.js (Axios-Instance)

Dies ist die wichtigste technische Schicht.

Sie sorgt dafür, dass das Frontend immer automatisch ein gültiges JWT mitschickt.

◆ Request Interceptor

Bevor eine Anfrage gesendet wird:

→ Hole Token aus localStorage

→ Hänge Header an: Authorization: Bearer <token>

Damit müssen Components oder Services **niemals** Token manuell anhängen.

◆ Response Interceptor

Globale Fehlerbehandlung:

- **401 Unauthorized** → Token ungültig → User ausloggen
- **403 Forbidden** → Keine Berechtigung → Redirect zu /forbidden

4. Backend REST API

Erst nach den zwei Interceptors wird der Request gesendet:

Frontend Components

↓

Services

↓

[apiClient.js \(Axios + JWT Handling\)](#)

↓

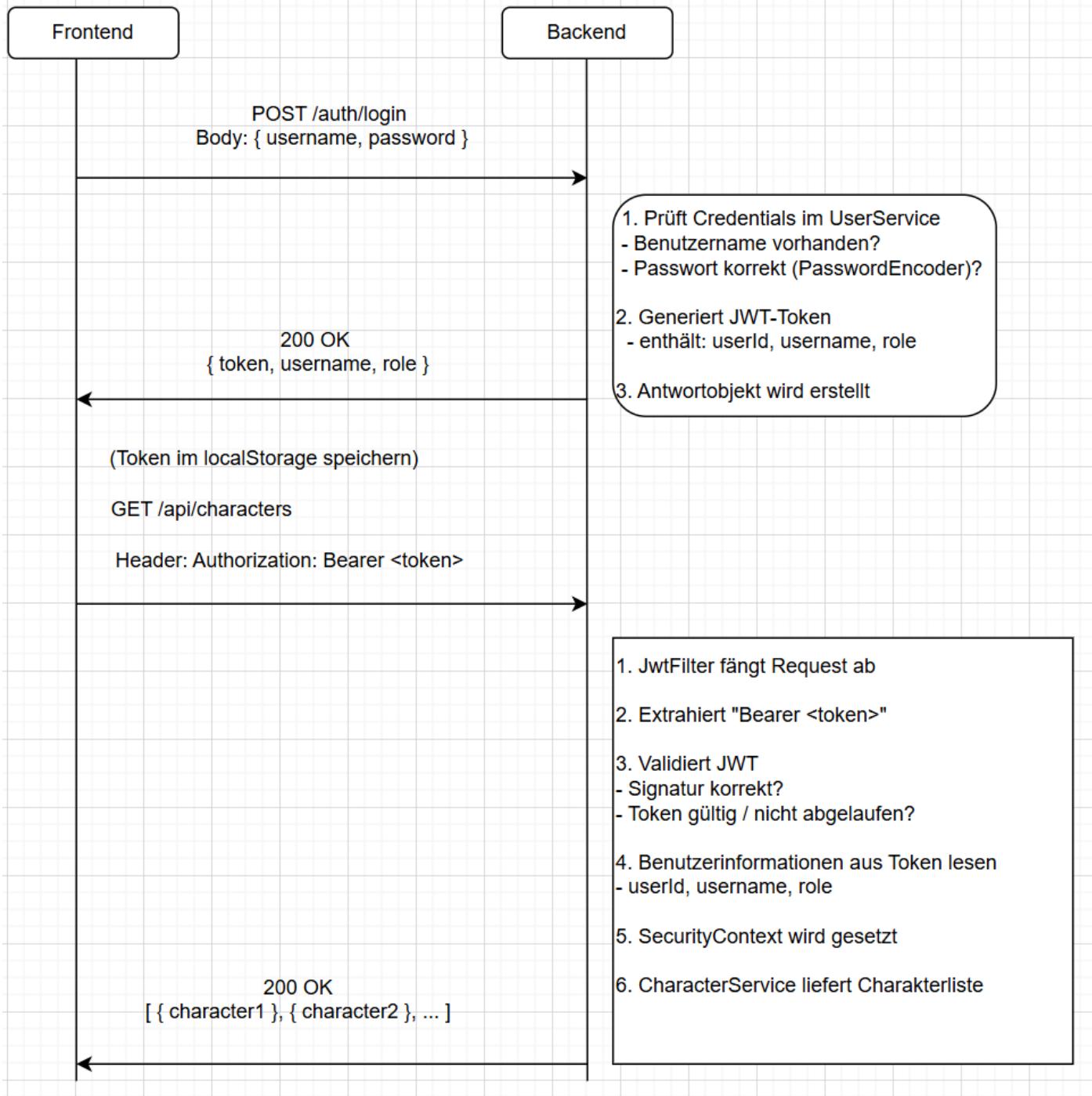
[Backend REST API \(Spring Boot + JWT Security\)](#)

Das Backend validiert JWT über den JwtAuthenticationFilter.

Nur authentifizierte Anfragen gelangen in Controller → Service → Repository.

JWT-Authentifizierungs-Flow

3. JWT-Authentifizierungs-Flow



Der JWT-Authentifizierungs-Flow zeigt, wie sich ein Benutzer im System anmeldet und wie anschliessend geschützte Backend-Endpunkte aufgerufen werden können. Nachdem das Frontend die Login-Daten (/auth/login) gesendet hat, prüft das Backend im UserService, ob der Benutzer existiert und ob das Passwort korrekt ist. Bei einer erfolgreichen Anmeldung erzeugt das Backend einen JWT-Token, der die wichtigsten Benutzerinformationen (userId, username, role) enthält. Dieser Token wird an das Frontend zurückgegeben und dort im localStorage gespeichert.

Bei jedem weiteren Request an einen geschützten Endpoint – zum Beispiel /api/characters – sendet das Frontend den Token im Authorization-Header mit. Der JwtFilter im Backend fängt den Request ab,

extrahiert den Token und validiert ihn. Ist der Token gültig, werden die Benutzerinformationen daraus gelesen und im Spring-SecurityContext gespeichert. Dadurch weiss der Server, welcher Benutzer den Request stellt und welche Rolle er besitzt. Anschliessend wird die Anfrage vom zuständigen Service verarbeitet und die entsprechenden Daten werden an das Frontend zurückgegeben.

Dieser Ablauf stellt sicher, dass nur gültig authentifizierte Benutzer Zugriff auf geschützte Bereiche haben und dass Berechtigungen korrekt geprüft werden.

Testprotokoll



Backend

Testobjekt: UserRepositoryTest

1. Übersicht

Das UserRepository ist für den Datenzugriff auf Benutzer zuständig. Neben den Standard-CRUD-Operationen stellt es zusätzliche Abfragen bereit:

- `findByUsername(String username)` – sucht einen Benutzer anhand des Usernames.
- `findAllByFavourites_Id(Long characterId)` – liefert alle Benutzer, die einen bestimmten Charakter in ihrer Favoritenliste haben.

Die Repository-Tests stellen sicher, dass diese Abfragen korrekt implementiert sind.

2. Verantwortlichkeiten des UserRepository

- Benutzer nach Username finden
 - Methode: `Optional<User> findByUsername(String username)`
 - Aufgabe: Ein Optional mit dem gefundenen Benutzer oder `Optional.empty()` zurückgeben.
- Benutzer nach Favoriten-Charakter finden
 - Methode: `List<User> findAllByFavourites_Id(Long characterId)`
 - Aufgabe: Alle Benutzer zurückgeben, die den Charakter mit der angegebenen ID als Favoriten gespeichert haben.

3. Abhängigkeiten

Abhängigkeit	Beschreibung
DataJpaTest	Startet eine In-Memory-Datenbank (H2) und konfiguriert Spring Data JPA für Integrationstests.

Abhängigkeit	Beschreibung
TestEntityManager	Ermöglicht das persistieren und sofortige Flushen von Entitäten innerhalb des Tests.
UserRepository	Das zu testende Repository.
CharacterRepository	Wird benötigt, um einen Charakter zu persistieren, der später als Favorit referenziert wird.

4. Zweck der Tests

Die Tests verifizieren, dass:

- findByUsername einen zuvor gespeicherten Benutzer korrekt zurückliefert.
- findAllByFavourites_Id eine Liste aller Benutzer zurückgibt, die einen bestimmten Charakter als Favorit haben.
- Die Zuordnung zwischen Benutzern und ihren Favoriten in der Datenbank korrekt persistiert wird.

5. Testfälle

UR-01 – whenFindByUsername_thenReturnUser

- Vorbereitung: Ein Benutzer mit dem Benutzernamen „player“ wirdpersistiert.
- Aktion: Aufruf von userRepository.findByUsername("player").
- Erwartung: Das zurückgegebene Optional ist vorhanden und enthält den korrekten Benutzer mit Rolle Role.PLAYER.

UR-02 – whenfindAllByFavouritesId_thenReturnUsersWithCharacter

- **Vorbereitung:**
 - Ein Charakter („Goku“) wirdpersistiert.
 - Ein Benutzer („player“) wirdpersistiert und „Goku“ als Favorit hinzugefügt.
- **Aktion: Aufruf von userRepository.findAllByFavourites_Id(goku.getId()).**
- **Erwartung:**
 - Die zurückgegebene Liste hat genau einen Eintrag.
 - Der Benutzer in der Liste hat den Usernamen „player“.
 - In der Favoritenmenge des Benutzers befindet sich ein Charakter mit dem Namen „Goku“.

6. Zusammenfassung

Der UserRepositoryTest stellt sicher, dass das Repository für Benutzer in Kombination mit der Many-to-Many-Beziehung zu Charakteren korrekt funktioniert. Durch die Verwendung einer In-Memory-Datenbank können die Datenbankzugriffe isoliert und effizient getestet werden.

Testobjekt: UserService

1. Übersicht

Das Testobjekt **UserService** bildet die Geschäftslogik zur Benutzerverwaltung und Favoriten-Funktionalität ab.

Es ist verantwortlich für:

- das Laden eines Benutzers anhand des Usernames (für Login & Authentifizierung),
- die Fehlerbehandlung, wenn ein Benutzer nicht existiert,
- das Hinzufügen von Charakteren zu den Favoriten eines Benutzers.

Die Tests im **UserServiceTest** prüfen das Verhalten dieser Geschäftslogik mithilfe von Mockito und gemockten Repositorys.

2. Verantwortlichkeiten des UserService

Der UserService ist zuständig für:

2.1 Laden eines Benutzers anhand des Usernames

- Methode: loadUserByUsername(String username)
- Aufgabe:
 - Benutzer im UserRepository anhand des Usernames finden
 - Wenn vorhanden → in ein UserDetails-Objekt konvertieren
 - Wenn nicht vorhanden → UsernameNotFoundException werfen
 - Benutzerrolle korrekt in Spring-Security-Authorities umwandeln

2.2 Hinzufügen eines Favoriten

- Methode: addFavourite(Long userId, Character character)
- Aufgabe:
 - Benutzer anhand der ID aus dem Repository laden
 - Prüfen, ob Favoritenliste existiert (ggf. anlegen)
 - Charakter zur Liste hinzufügen
 - Aktualisierten Benutzer speichern

3. Abhangigkeiten

Der UserService nutzt im Testkontext folgende Komponenten:

Abhangigkeit	Beschreibung
UserRepository	Zugriff auf Benutzerobjekte; wird vollstandig gemockt, um Datenbankzugriffe zu simulieren.
Character-Entity	Domain-Objekt eines Charakters, der einem Benutzer als Favorit hinzugefgt werden kann.
Mockito	Wird verwendet, um Repository-Aufrufe zu simulieren (z. B. when, doReturn, verify).
JUnit 5	Testframework zur Ausführung der Unit-Tests.
Spring Security (UserDetails)	Wird genutzt, um Benutzerinformationen fur Authentifizierung zu erzeugen.

4. Zweck der Tests

Die Tests des UserService stellen sicher, dass:

- Benutzer korrekt geladen und in Sicherheitstypen konvertiert werden,
- Fehlersituationen fachgerecht abgefangen werden,
- Favoriten aktualisiert und gespeichert werden,
- die interne Logik unabhangig vom Web-Layer korrekt arbeitet.

5. Testfalle (bersicht)

US-01 – loadUserByUsername_returnsUserDetails

Testfallname:

loadUserByUsername_returnsUserDetails

Beschreibung / Zweck:

Pruft, ob ein existierender Benutzer korrekt aus dem Repository geladen und in ein Spring-Security-konformes UserDetails-Objekt umgewandelt wird.

Eingaben / Vorbereitung:

- Repository liefert Benutzer "player" mit Passwort "secret" und Rolle "PLAYER".
- Username "player" wird an den Service bergeben.

Erwartetes Ergebnis:

- UserDetails enthlt:

- richtigen Username,
- richtiges Passwort,
- korrekte Authority "ROLE_PLAYER".

Resultat: ✓ bestanden

US-02 – loadUserByUsername_throwsException_whenUserNotFound

Testfallname:

loadUserByUsername_throwsException_whenUserNotFound

Beschreibung / Zweck:

Prüft, ob der Service korrekt reagiert, wenn ein Benutzer nicht existiert.

Eingaben / Vorbereitung:

- Repository gibt Optional.empty().
- Username "unknown" wird angefragt.

Erwartetes Ergebnis:

- UsernameNotFoundException wird geworfen.
- Repository-Aufruf wurde genau einmal ausgeführt.

Resultat: ✓ bestanden

US-03 – addFavourite_addsCharacterToUserList

Testfallname:

addFavourite_addsCharacterToUserList

Beschreibung / Zweck:

Prüft, ob ein Charakter korrekt zu den Favoriten eines Benutzers hinzugefügt wird.

Eingaben / Vorbereitung:

- MockUser mit leerer Favoritenliste.
- MockCharacter (z. B. „Goku“).
- Repository speichert Benutzer nach Änderung.

Erwartetes Ergebnis:

- Charakter wird der Favoritenliste hinzugefügt.
- userRepository.save(user) wurde aufgerufen.
- Favoritenliste enthält exakt 1 neuen Eintrag.

Resultat: ✓ bestanden

6. Zusammenfassung

Die Tests des **UserService** zeigen, dass:

- Benutzer korrekt geladen und aus Repository-Daten in sichere UserDetails umgewandelt werden,
- Fehler korrekt erkannt und durch Exceptions behandelt werden,
- Favoritenoperationen logisch korrekt umgesetzt und dauerhaft gespeichert werden.

Damit gilt das **Testobjekt UserService als vollständig und erfolgreich getestet** im Bereich Benutzeroberfläche & Favoriten-Logik.

❖ Frontend

Testobjekt: ProtectedRoute

1. Übersicht

Das Testobjekt ProtectedRoute ist eine React-Komponente, die Zugriffe auf bestimmte Routen schützt.

Sie entscheidet anhand des Authentifizierungsstatus und der Benutzerrolle, ob:

- der geschützte Inhalt gerendert wird,
- oder eine Umleitung z. B. auf /login oder /forbidden erfolgt.

Die Tests im ProtectedRoute-Testfile prüfen das Verhalten dieser Komponente mithilfe von React Testing Library, React Router (MemoryRouter) und einem gemockten AuthContext (useAuth).

2. Verantwortlichkeiten der ProtectedRoute

Die ProtectedRoute ist zuständig für:

2.1 Weiterleitung nicht authentifizierter Benutzer

- HTTP-ähnliches Verhalten in der SPA:
 - Wenn isAuthenticated === false, wird nicht der geschützte Inhalt, sondern die Login-Seite gerendert.
- Typisches Ziel: Route /login.

2.2 Rollenprüfung für geschützte Routen

- Wenn requiredRole gesetzt ist:
 - Prüfen, ob user.role der geforderten Rolle entspricht.
 - Falls Rolle unzureichend → Weiterleitung auf /forbidden.

2.3 Zugriff für Benutzer mit ausreichender Rolle

- Wenn Benutzer authentifiziert ist und user.role der requiredRole entspricht:

- Der Kindinhalt (children) wird direkt gerendert.
- Keine Umleitung erfolgt.

3. Abhängigkeiten

Die ProtectedRoute nutzt im Testkontext folgende Komponenten:

Abhängigkeit Beschreibung

useAuth (AuthContext) Liefert Authentifizierungsstatus, Benutzerobjekt und Ladezustand; im Test durch vi.mock gemockt.

MemoryRouter / Routes / Route Router-Simulation aus react-router-dom; ermöglicht das Testen von Navigation und Routenwechseln.

React Testing Library (render, screen) Zum Rendern der Komponente und Abfragen des DOM.

Vitest (describe, it, expect, beforeEach, vi) Testframework und Mocking-Utilities.

Geschützte Seiten-Komponenten (z. B. <div>Protected Content</div>) Dummy-Inhalte, um das Verhalten der Route sichtbar zu machen.

4. Zweck der Tests

Die Tests der ProtectedRoute überprüfen, ob:

- nicht authentifizierte Benutzer korrekt zur Login-Seite umgeleitet werden,
- Benutzer mit unzureichender Rolle zur Forbidden-Seite umgeleitet werden,
- Benutzer mit der richtigen Rolle den geschützten Inhalt sehen dürfen,
- das Verhalten unabhängig vom echten Backend nur aufgrund der Authentifizierungsdaten funktioniert.

5. Testfälle (Übersicht)

PR-01 – redirects unauthenticated users to /login

Testfallname:

redirects unauthenticated users to /login

Beschreibung / Zweck:

Prüft, ob ein nicht authentifizierter Benutzer beim Aufruf einer geschützten Route auf /login umgeleitet wird.

Eingaben / Vorbereitung:

- mockUseAuth liefert:
 - isAuthenticated: false
 - user: null
 - loading: false
- MemoryRouter startet auf Route /protected.
- /protected ist mit ProtectedRoute umschlossen.
- Route /login rendert „Login Page“.

Erwartetes Ergebnis:

- Im gerenderten DOM ist der Text „Login Page“ vorhanden.
- Der geschützte Inhalt („Protected Content“) wird nicht angezeigt.

Resultat: ✓ bestanden

PR-02 – redirects users with insufficient role to /forbidden

Testfallname:

redirects users with insufficient role to /forbidden

Beschreibung / Zweck:

Prüft, ob ein authentifizierter Benutzer mit Rolle PLAYER beim Zugriff auf eine Admin-Route korrekt zur Forbidden-Seite umgeleitet wird.

Eingaben / Vorbereitung:

- mockUseAuth liefert:
 - isAuthenticated: true
 - user: { role: "PLAYER" }
 - loading: false
- MemoryRouter startet auf Route /admin.
- Route /admin ist mit ProtectedRoute requiredRole="ADMIN" geschützt.
- Route /forbidden rendert „Forbidden Page“.

Erwartetes Ergebnis:

- Im gerenderten DOM ist der Text „Forbidden Page“ vorhanden.
- Der Admin-Inhalt („Admin Content“) wird nicht angezeigt.

Resultat: ✓ bestanden

PR-03 – allows access when the user has the required role

Testfallname:

allows access when the user has the required role

Beschreibung / Zweck:

Prüft, ob ein Benutzer mit Rolle ADMIN beim Zugriff auf eine Admin-Route den geschützten Inhalt angezeigt bekommt.

Eingaben / Vorbereitung:

- mockUseAuth liefert:
 - isAuthenticated: true

- user: { role: "ADMIN" }
- loading: false
- MemoryRouter startet auf Route /admin.
- Route /admin ist mit ProtectedRoute requiredRole="ADMIN" geschützt.
- Als Child wird <div>Admin Content</div> gerendert.

Erwartetes Ergebnis:

- Im gerenderten DOM ist der Text „Admin Content“ vorhanden.
- Es erfolgt keine Umleitung auf /login oder /forbidden.

Resultat: ✓ bestanden

6. Zusammenfassung

Die Tests der ProtectedRoute zeigen, dass:

- die Komponente nicht authentifizierte Benutzer zuverlässig auf /login umleitet,
- Benutzer mit unzureichender Rolle korrekt auf /forbidden umgeleitet werden,
- Benutzer mit der richtigen Rolle den geschützten Inhalt sehen,
- die Autorisierungslogik im Frontend klar und deterministisch funktioniert.

Damit gilt das Testobjekt ProtectedRoute als erfolgreich getestet im Kontext von Route-Schutz, Rollenprüfung und Umleitungslogik.

Testobjekt: AuthContext / AuthProvider

1. Übersicht

Das Testobjekt AuthContext bzw. AuthProvider bildet die zentrale Authentifizierungslogik im Frontend ab.

Es verwaltet:

- den aktuellen Benutzer (user),
- den Authentifizierungsstatus (isAuthenticated),
- Login-Funktionalität inklusive Speichern von Token und Benutzerdaten in localStorage.

Die Tests im AuthContext-Testfile prüfen das Verhalten des Contexts mit einem echten AuthService, jedoch mit gemockten Funktionen des authService.

2. Verantwortlichkeiten des AuthContext

Der AuthContext / AuthProvider ist zuständig für:

2.1 Bereitstellen von Authentifizierungszustand

- Stellt über useAuth() folgende Werte und Funktionen zur Verfügung:
 - user

- isAuthenticated
- login(...)
- (ggf. weitere wie logout, werden hier nicht direkt getestet)

2.2 Login-Verarbeitung

- Aufruf von authService.login(username, password).
- Übernahme der von authService gelieferten Daten:
 - Speichern des Tokens in localStorage (z. B. Schlüssel authToken).
 - Speichern der Benutzerdaten (z. B. userData) in localStorage.
 - Aktualisieren von user und isAuthenticated im Context.

2.3 Persistenz im Browser

- Nutzung von localStorage, um Authentifizierungsdaten zwischen Sessions zu halten.
- Beim Login werden Token und Benutzerobjektpersistiert.

3. Abhängigkeiten

Der AuthContext nutzt im Testkontext folgende Komponenten:

Abhängigkeit Beschreibung

authService (login, logout, getToken, getUserData) Wird vollständig mit vi.mock gemockt; login liefert einen Fake-Token und Fake-User.

localStorage Browser-Speicher; im Test über jsdom verfügbar, vor jedem Test geleert.

AuthProvider Kontext-Provider, der den AuthContext um die Test-Komponente legt.

useAuth Hook, um im Test-Komponentchen auf Context-Werte zuzugreifen.

React Testing Library (render, screen, waitFor) Zum Rendern und asynchronen Warten auf State-Updates.

userEvent Simuliert Benutzerinteraktionen (z. B. Button-Klick).

Vitest (describe, it, expect, beforeEach, vi) Testframework, Assertion- und Mocking-Funktionen.

4. Zweck der Tests

Die Tests des AuthContext überprüfen, ob:

- ein Login-Aufruf den Context-Zustand (user, isAuthenticated) korrekt aktualisiert,
- die Benutzerrolle aus der Antwort des authService korrekt übernommen wird,
- der Token in localStorage gespeichert wird,
- die Benutzerdaten in localStorage persistiert werden.

5. Testfälle (Übersicht)

AC-01 – should log in and expose user role

Testfallname:

should log in and expose user role

Beschreibung / Zweck:

Prüft, ob der AuthContext nach einem Login den Authentifizierungsstatus auf „true“ setzt, die Benutzerrolle zur Verfügung stellt und Token + Userdaten im localStorage speichert.

Eingaben / Vorbereitung:

- authService.login ist gemockt und liefert:
 - token: "fake.jwt.token"
 - user: { username: "player", role: "PLAYER" }
- localStorage wird vor dem Test geleert.
- Test-Komponente LoginTester verwendet useAuth() und:
 - zeigt aktuelle Rolle (user?.role oder "none"),
 - zeigt isAuthenticated (true/false),
 - bietet einen Button, der login("player", "player123") auslöst.
- AuthProvider umschliesst LoginTester.

Erwartetes Ergebnis:

- Vor dem Klick:
 - angezeigte Rolle ist "none".
 - isAuthenticated ist "false".
- Nach Klick auf den Login-Button:
 - isAuthenticated ändert sich auf "true".
 - Rolle wird als "PLAYER" angezeigt.
- In localStorage:
 - authToken ist "fake.jwt.token".
 - userData ist JSON mit { username: "player", role: "PLAYER" }.

Resultat: ✓ bestanden

6. Zusammenfassung

Die Tests des AuthContext / AuthProvider zeigen, dass:

- der Login-Prozess korrekt mit dem (gemockten) authService interagiert,
- der Authentifizierungszustand im Context zuverlässig aktualisiert wird,
- die Benutzerrolle im Frontend verfügbar ist,
- Token und Benutzerinformationen erfolgreich im localStorage gespeichert werden.

Damit gilt das Testobjekt AuthContext / AuthProvider als erfolgreich getestet im Hinblick auf Login-Verhalten, Rollenbereitstellung und Persistenz der Authentifizierungsdaten im Frontend.

Installationsanleitung

Dieses Kapitel beschreibt alle Schritte, die notwendig sind, um das Projekt lokal auszuführen.

Das System besteht aus zwei getrennten Teilen:

- **Backend** – Spring Boot (Java)
- **Frontend** – React (Vite)

Voraussetzungen

Backend

Komponente Version

Java JDK 17 oder höher

Maven 3.8+

IDE empfohlen IntelliJ IDEA / Eclipse / VS Code

Frontend

Komponente Version

Node.js 18+

npm 8+

Browser Chrome, Firefox oder Edge

❖ Backend – Installation & Start

1. Projekt klonen

```
git clone <https://github.com/Thierno-hamidou5/Dragonball-Z.git>
```

```
cd backend
```

2. Abhängigkeiten installieren

Maven lädt automatisch alle Dependencies:

```
mvn clean install
```

3. Datenbank konfigurieren

Das Projekt nutzt **H2 In-Memory** (kein Setup nötig).

Falls du auf MySQL wechseln willst → application.properties anpassen.

4. Backend starten

```
mvn spring-boot:run
```

Oder in IntelliJ:

Rechtsklick auf BackendApplication.java → **Run**

5. Backend erreichbar unter

<http://localhost:8080>

6. Wichtige Endpunkte (Beispiele)

Methode	Endpoint	Beschreibung
GET	/api/users/favourites	Favoriten lesen
POST	/api/users/favourites/{id}	Favorit hinzufügen
DELETE	/api/users/favourites/{id}	Favorit entfernen
POST	/api/auth/login	Login

❖ Frontend – Installation & Start

1. In das Frontend-Verzeichnis wechseln

```
cd frontend
```

2. Abhängigkeiten installieren

```
npm install
```

3. Entwicklungsserver starten

```
npm run dev
```

4. Frontend erreichbar unter

Normalerweise:

<http://localhost:5176>

5. Backend-Adresse konfigurieren

Falls das Backend nicht auf localhost:8080 läuft:

In .env oder services/api.js ändern:

```
export const API_URL = "http://localhost:8080";
```

Login für Testzwecke

Testdaten aus Backend

Falls dein Backend folgende Default-Benutzer lädt:

Username	Passwort	Rolle
admin	admin123	ADMIN
player	player123	PLAYER

Damit kann der Login im Frontend vollständig getestet werden.

Tests ausführen

Backend-Tests

`mvn test`

Frontend-Tests

`npm run test`

Projekt stoppen

Backend stoppen

Terminal schliessen oder mit Ctrl + C.

Frontend stoppen

Ctrl + C im Vite-Dev-Terminal.

Technologiestack

❖ Backend

Technologie	Version	Verwendung
Java	21	Programmiersprache fuer das Backend
Spring Boot	3.5.0	Zentrales Backend-Framework
Spring Web	3.5.0 (Starter)	REST-Controller und HTTP-Handling
Spring Security	6.x (ueber Boot)	Authentifizierung und Autorisierung
JWT (io.jsonwebtoken, jjwt)	0.11.5	Erzeugen und Validieren der JWT-Tokens

Technologie	Version	Verwendung
Spring Data JPA / Hibernate	ueber Spring Boot	ORM fuer die Datenbankzugriffe
PostgreSQL JDBC Driver	42.7.7	Zugriff auf PostgreSQL-Datenbank (geplante Prod-Umgebung)
H2	2.x (runtime)	In-Memory / File-DB fuer Entwicklung und Tests
springdoc-openapi	2.7.0	Swagger / OpenAPI UI fuer die REST-API
Maven	3.x	Build-Tool und Dependency-Management
JUnit, Spring Test, Mockito	ueber Test-Starter	Unit- und Integrationstests

Beschreibung

Das Backend verwendet **Spring Boot**, um eine klare Struktur und schnelle Entwicklung zu ermöglichen.

Die **REST-Endpunkte** werden mit Spring Web erstellt.

Spring Security schützt alle wichtigen Routen und verwendet **JWT-Tokens**, um Benutzer zu authentifizieren.

Nach dem Login erstellt der JwtService einen Token, der vom Frontend an jeden Request angehängt wird.

Der JwtFilter prüft diesen Token und liest daraus den User.

Über **Spring Data JPA** greift das Backend auf die Datenbank zu, ohne dass man SQL schreiben muss.

Passwörter werden mit **BCrypt** sicher gespeichert.

Tests laufen über **JUnit** und **Mockito**, um die Qualität des Codes zu sichern.

❖ Frontend

Technologie	Version	Verwendung
React	18.x	Grundlegendes UI-Framework fuer das Frontend
Vite	5.x	Development-Server und Build-Tool (schneller als CRA)
React Router	6.x	Client-Side Routing, Navigation zwischen Seiten
JavaScript (ES202x)	Standard	Programmiersprache fuer das Frontend
Axios	1.x	HTTP-Client; hängt JWT automatisch an Requests
HTML / CSS (Vanilla)	–	Styling, Layout, Komponentenstruktur
Vitest	1.x	Test-Runner fuer Frontend-Unit-Tests

Technologie	Version	Verwendung
React Testing Library (RTL)	14.x	Komponenten-Tests (DOM-Interaktionen, Rendering)
Node.js	18+	Ausfuehren des Development-Servers und Build-Prozesses
npm	8+	Paketmanager fuer Frontend-Abhaengigkeiten
LocalStorage API		Browser-API Speicherung von Token & Userdaten
ESLint (optional)	-	Code-Qualitaet, Linting-Regeln

Beschreibung

Das Frontend nutzt **React** als Bibliothek. Damit kann man einzelne UI-Bausteine bauen, die man auf verschiedenen Seiten wiederverwenden kann. Alle Seiten wie *Home*, *Z-Fighters*, *Villains*, *Login*, *Favorites* und *Manage Characters* bestehen aus React-Komponenten.

Für die Entwicklung wird **Vite** benutzt. Vite startet sehr schnell, lädt Änderungen sofort nach und baut die App später für die Produktion optimiert.

Die Navigation wird mit **React Router** umgesetzt. Damit kann der User zwischen verschiedenen Seiten wechseln. Geschützte Seiten (z. B. Manage Characters) sind über **Protected Routes** abgesichert, damit nur eingeloggte oder Admin-Benutzer darauf zugreifen können.

Für alle API-Anfragen verwendet das Frontend **Axios**.

Ein eigener *apiClient* hängt automatisch den JWT-Token über Authorization: Bearer <token> an jeden Request und kümmert sich darum, Fehler wie **401 / 403** richtig zu behandeln.

Das Login speichert **authToken** und **userData** im **localStorage**. Dadurch bleibt der Benutzer auch nach einem Reload noch eingeloggt.

Tests werden mit **Vitest** und **React Testing Library** durchgeführt. Damit prüft man, ob Komponenten richtig angezeigt werden, ob die Login-Funktion klappt oder ob die Favoriten richtig funktionieren.

Feedback

Ich bin mit meinem Dragonball-Projekt insgesamt **sehr zufrieden**. Das Backend funktioniert stabil, und alle wichtigen Sicherheitsmechanismen – wie **JWT-Login**, **Rollenprüfung**, **Passwort-Hashing mit BCrypt** sowie der sichere Umgang mit Benutzerdaten – sind sauber und korrekt umgesetzt. Besonders das **Favoriten-System** arbeitet zuverlässig, und die gesamte **Datenbankstruktur** ist klar, logisch und gut erweiterbar.

Auch das Frontend konnte ich erfolgreich mit einer modernen und sauberen Architektur umsetzen. Dank **React**, **React Router**, **Axios**, eigenen Services und dem **globalen Axios-Interceptor**, der automatisch den JWT-Token an alle API-Requests anhängt, fühlt sich die App wie eine professionelle

Single-Page-Application an. Die Benutzeroberfläche ist übersichtlich, modern und alle Funktionen – Login, Charakterdetailseiten, Favoriten, Verwaltung – sind einfach zugänglich.

Technische Herausforderungen & wie ich sie gelöst habe

1. Security war eine grosse Herausforderung

Die grösste Schwierigkeit war es, die Spring-Security-Konfiguration richtig zu verstehen – besonders den JWT-Filter, die Rollenprüfung und die Absicherung der Endpunkte. Anfangs hatte ich zu viele Routen mit `permitAll()` freigegeben, was ein Sicherheitsrisiko war. Dank den Dozentenunterlagen konnte ich meine Security schrittweise korrigieren und vollständig korrekt umsetzen.

2. UserDetailsService

Am Anfang hatte ich keinen eigenen UserDetailsService implementiert und meine Security funktionierte deshalb nicht korrekt. Durch das kurze Feedback des Lehrers und die klaren Unterlagen verstand ich, dass ich mich entscheiden muss: entweder eine eigene UserDetailsService-Klasse oder die Implementierung direkt im UserService. Danach konnte ich den Service sauber aufbauen, richtig in den AuthenticationProvider einbinden und das gesamte Login-Handling zuverlässig zum Laufen bringen

3. Mono-Repo & Git (sehr grosse Herausforderung)

Die Arbeit im Mono-Repository brachte viele neue Themen wie Branching, sauberes Committen und das Lösen von Merge-Konflikten. Es war ungewohnt, Backend und Frontend im selben Projekt zu verwalten. Am Ende habe ich viel über professionelle Projektstruktur und Team-Entwicklung gelernt.

4. Transaktionen im Backend

`@Transactional` korrekt einzusetzen – vor allem beim Löschen von Charakteren und dem Entfernen der Favoriten – war anfangs anspruchsvoll. Ich musste verstehen, wie Spring Transaktionen öffnet und wann Rollbacks ausgelöst werden. Schlussendlich funktioniert das gesamte Delete-Handling stabil und fehlerfrei.

Rolle der Unterrichtsunterlagen

Die vom Dozenten bereitgestellten Unterrichtsunterlagen hatten eine zentrale Bedeutung für die erfolgreiche Umsetzung des Projekts.

Sie erklärten alle wichtigen Komponenten wie **SecurityConfig**, **JwtService**, **JwtFilter** und **UserDetailsService** klar und nachvollziehbar und zeigten gleichzeitig, wie diese Bausteine korrekt zusammenspielen.

Dank der verständlichen Struktur war es möglich, das **komplette Backend und Frontend sauber aufzubauen**.

Mit zusätzlicher Zeit hätte man die Unterlagen noch intensiver nutzen können, etwa zur **Code-Optimierung** oder für **erweiterte Tests**.

Trotzdem boten sie eine **stabile und zuverlässige Grundlage**, auf der das gesamte Projekt erfolgreich realisiert wurde.

Reflexion

Dieses Projekt hat mir sehr deutlich gezeigt, wie wichtig eine saubere Struktur und gute Sicherheitsmechanismen in einer modernen Web-Anwendung sind. Besonders das Zusammenspiel von **JWT-Authentifizierung, Spring Security, Rollenmodellen** und der Kommunikation zwischen Frontend und Backend war am Anfang für mich schwierig. Ich musste zuerst verstehen, wie ein Token erstellt, gespeichert und bei jeder Anfrage wieder überprüft wird. Durch das Arbeiten mit den verschiedenen Security-Bausteinen habe ich ein viel tieferes Verständnis dafür gewonnen, wie professionelle Systeme Benutzer schützen.

Ein grosses Lernfeld war für mich die **Security-Thematik insgesamt** – vom JWT-Filter über die richtige Konfiguration von permitAll(), bis hin zu Rollenprüfung und Endpoint-Absicherung. Genau diese Themen haben mir geholfen zu erkennen, wie viele Fehler man unbewusst einbauen kann, wenn man die Sicherheitsarchitektur nicht genau versteht. Heute weiss ich viel besser, worauf es bei sicherheitsrelevanter Softwareentwicklung ankommt.

Auch im Frontend habe ich wichtige Erfahrungen gesammelt. Ich habe gelernt, wie Routing, Zustand, Login-Flow und API-Kommunikation wirklich zusammenhängen. Mir wurde klar, dass das Frontend genauso Teil der Sicherheitskette ist, zum Beispiel beim korrekten Umgang mit Tokens, bei Protected Routes oder beim globalen Fehlerhandling.

Ein weiterer wichtiger Erkenntnispunkt war der Umgang mit **DataInitializer**. Ich habe verstanden, dass solche Klassen zwar praktisch für Tests und Entwicklung sind, aber in echten Produktivsystemen ein Sicherheitsrisiko darstellen – vor allem, wenn Standardpasswörter im Code gespeichert werden. Durch dieses Projekt wurde mir bewusst, wie wichtig es ist, **Passwörter, JWT-Secrets und andere sensible Daten nie im Code aufzubewahren**.

Insgesamt hat mir das Projekt sehr geholfen, mein Wissen im Backend und Frontend zusammenzuführen. Ich habe nun ein viel klareres Bild davon, wie ein vollständiges System von Login über Security bis hin zur Datenbank funktioniert.

Ich bin auf einem guten Weg Richtung Full-Stack-Entwicklung und habe gelernt, wie man stabile, sichere und gut strukturierte Anwendungen aufbaut.

Quellen & Hilfsmittel

- **Unterricht WISS – Modul M223**
Grundlage für Architektur, Security, Spring Boot, Git und Best Practices.
- **Dozent Graziano**
Fachliche Unterstützung, besonders bei Security (permitAll, UserDetailsService), Git-Struktur, Mono-Repo und Build-Prozessen.
- **Nico**
Technischer Austausch, Feedback zur Struktur und Hilfe bei Git-Konflikten.

- **Dokumentation**
Spring Boot, React, Vite, Axios, JWT, JPA/Hibernate, H2.
- **draw.io**
Erstellung von Klassen- und Architekturdiagrammen.
- **ChatGPT**
Unterstützung bei Strukturierung, Fehleranalyse, Textformulierung und Sicherheitskonzepten.
- **Codex**
Kommentar für Backend / Frontend