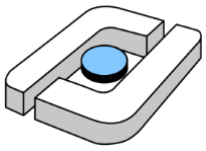


Hochschule Osnabrück

University of Applied Sciences

Fakultät
Ingenieurwissenschaften und Informatik

Schriftliche Ausarbeitung zum Thema:

Entwicklung einer Web-Anwendung mit Java, Jakarta EE und Quarkus

im Rahmen des Moduls
Software-Architektur – Konzepte und Anwendungen,
des Studiengangs Informatik-Medieninformatik

Autor:	Patrick Felschen
	Julian Voß
Matr.-Nr.:	932056
	934505
E-Mail:	patrick.felschen@hs-osnabrueck.de
	julian.voss@hs-osnabrueck.de
Themensteller:	Prof. Dr. Rainer Roosmann

Abgabedatum: 17.02.2023

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Source-Code Verzeichnis

Abkürzungsverzeichnis

API	Application Programming Interface
CDI	Contexts and Dependency Injection
CSS	Cascading Style Sheets
DBMS	Database Management System
DI	Dependency Injection
DTO	Data Transfer Object
ECB	Entity Control Boundary
EE	Enterprise Edition
HTTP	Hypertext Transfer Protocol
JAX-RS	Jakarta RESTful Web Services
JDBC	Java Database Connectivity
JPA	Jakarta Persistence API
JPQL	Java Persistence Query Language
JSON	JavaScript Object Notation
JWT	JSON Web Token
OIDC	OpenID Connect
REST	Representational State Transfer
RMM	Richardson Maturity Model
SQL	Structured Query Language
UI	User Interface
URI	Uniform Resource Identifier

1 Einleitung (Voß, Felschen)

In dieser Ausarbeitung wird die Planung und Implementation einer Web-Applikation unter Nutzung von Jakarta Enterprise Edition und Quarkus dargestellt. Dabei werden verschiedene Konzepte der Softwarearchitektur erläutert, welche die Grundlage der umgesetzten Anwendung bilden. Weiterhin wird die Umsetzung verschiedener Technologien, die genutzt wurden, um eine sichere Webanwendung inklusive Benutzeroberfläche bereitzustellen, anhand von Beispielen erläutert.

1.1 Vorstellung des Themas

Die umgesetzte Anwendung stellt ein Tippspiel für Fußballspiele dar. Nutzern ist es möglich, nach einer Registrierung Wetten auf angebotene Spiele abzugeben. Ein Administrator ist dafür zuständig, anstehende Spiele in das System einzutragen, sowie abgeschlossene Spiele mit dem entsprechenden Ergebnis zu aktualisieren. Wurde ein Spiel beendet, nimmt das System eine automatische Punkteverteilung vor, die Punkte werden anhand der Genauigkeit des Tipps vergeben. Sagt der Nutzer das Ergebnis exakt vor, werden ihm vier Punkte gutgeschrieben, drei Punkte für die richtige Tordifferenz und zwei Punkte, falls lediglich das Siegerteam richtig bestimmt wurde. Das System kann über eine Web-App und eine REST-Schnittstelle bedient werden.

1.2 Ziel der Ausarbeitung

Das Ziel der Ausarbeitung ist es zu zeigen, wie unter Quarkus eine sichere und erweiterbare Softwarelösung zu erstellen ist. Dabei werden Aspekte wie Sicherheit, Erfassung von Metriken, Persistierung von Anwendungsdaten, Bereitstellung einer Web-Oberfläche durch Server Side Rendering und das Testen des Systems behandelt.

1.3 Aufbau der Hausarbeit

Nachdem das Thema und das Ziel der Arbeit erläutert wurde, wird nun beschrieben, wie die Hausarbeit strukturiert ist. Zunächst gibt es einen Einblick in die grundlegende Softwarearchitektur des Projektes. Diese wird anhand von einem Context- und einem Containerdiagramm dargestellt. Anschließend werden Entwurfsmuster und Prinzipien der Softwarearchitektur erklärt, welche bei der Entwicklung der Anwendung berücksichtigt werden sollen.

Um die Anforderungen umzusetzen, werden danach Komponenten gezeigt, welche unter Verwendung des genutzten Frameworks, angeboten werden. Nachdem die Grundlagen dargestellt wurden, wird sich mit der Planung und Umsetzung der Schnittstelle und der Web-Anwendung auseinandergesetzt. Neben der Persistierung und Validierung von Daten in einer Datenbank, wird sich mit Transaktionen und Sicherheit befasst. Des Weiteren wird die verwendete Frontend-Technologie und die Oberfläche beschrieben. Abschließend wird gezeigt, wie die Funktionen der entwickelten Anwendung untersucht und getestet wurden.

2 Darstellung der Grundlagen

Zunächst wird die Struktur des Projektes mit Diagrammen veranschaulicht. Es wird gezeigt, wie Personen, Systeme und externe Systeme miteinander in Beziehung stehen. Zudem wird verallgemeinert gezeigt, welche Informationen zwischen den Einzelnen Akteuren ausgetauscht wird.

2.1 Context Diagramm (Voß)

Unter Verwendung eines C4 Context-Diagramms (siehe Abbildung 1: C4 Context Diagramm (Voß)) wird der grobe Aufbau des Systems erläutert.

Da sich dieses System auf das Tippen von Fußball Ergebnissen bezieht, wird als Persona ein Nutzer gewählt, der sich für das Tippen von Sportergebnissen interessiert. Dieser interagiert mit einer grafischen Oberfläche, um sich Fußballspiele ansehen zu können. Des Weiteren ist es über die Oberfläche möglich, Vorhersagen für das Ergebnis eines Spiels zu treffen.

Die Oberfläche verwendet zur Verwaltung dieser Nutzersteuerung einen Webservice.

Auf der anderen Seite wird es einem externen System ermöglicht auf den gleichen Service über eine REST-Schnittstelle zuzugreifen.

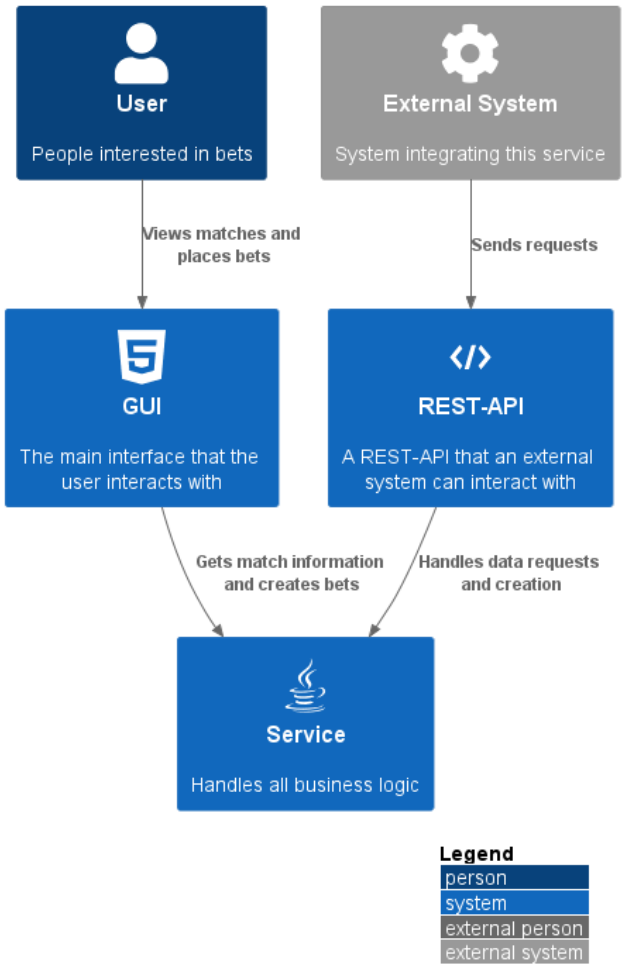


Abbildung 1: C4 Context Diagramm (Voß)

2.2 Container Diagramm (Voß)

In dieser Übersicht (siehe Abbildung 2: C4 Container Diagramm (Voß)) wird das Verhalten des Systems unter Verwendung eines C4 Container-Diagramms erklärt. Wie im vorherigen Context-Diagramm beschrieben, interagiert ein Nutzer mit einer Benutzungsoberfläche, dies ist hier eine Web-App, welche über Qute realisiert wird.

Vom Nutzer werden HTTP-Anfragen empfangen, woraufhin über Qute eine HTML-Seite als Antwort bereitgestellt wird. Über den Quarkus Service werden die Anfragen verarbeitet. Die Nutzerverwaltung wird über Keycloak und dem Standard OpenID Connect realisiert. Systemdaten, wie Wetten, Teams, Spiele und Nutzerprofile mit Punkten werden über die JDBC-Schnittstelle mittels einer PostgreSQL Datenbank persistiert. Sowohl der Keycloak als auch der PostgreSQL Dienst werden über Docker in einem eigenen Container gestartet und laufen damit isoliert voneinander.

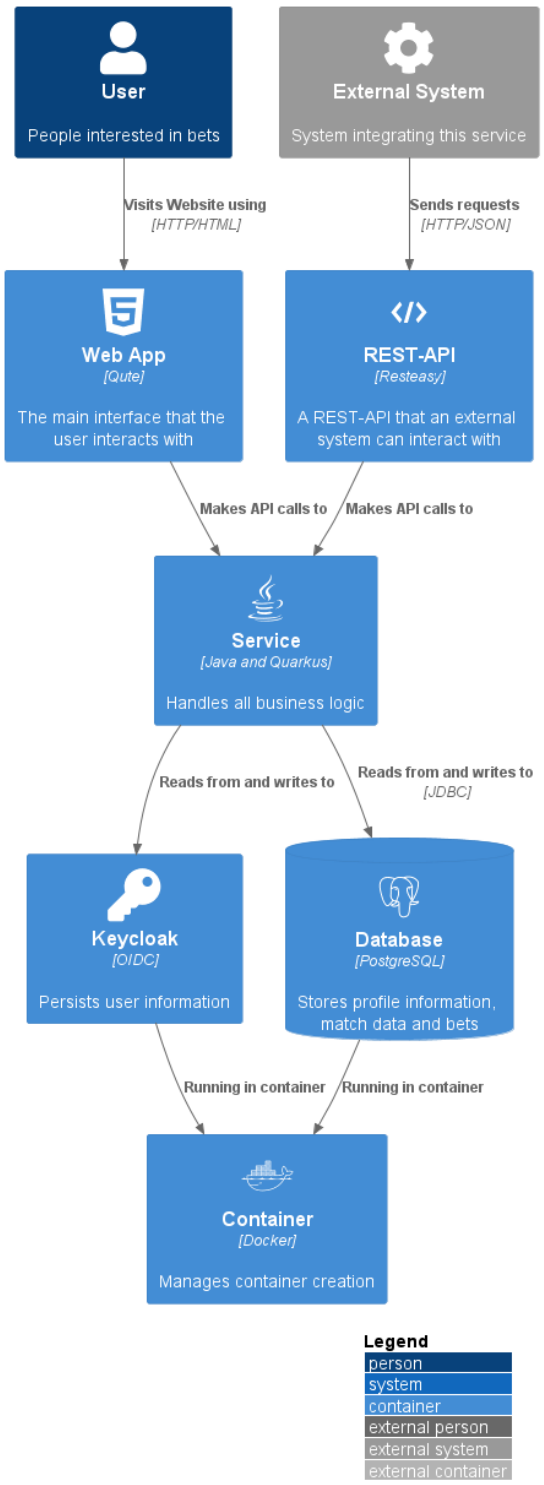


Abbildung 2: C4 Container Diagramm (Voß)

2.3 Layered Architecture (Voß)

Durch eine Layered Architecture werden die Probleme der Vermischung von UI-, Geschäftslogik und Infrastruktur behandelt. Technisch zusammengehörende Elemente dieser Ebenen werden in Layer zusammengefasst, die möglichst unabhängig voneinander agieren. Eine Schicht ist dabei lediglich von der darunterliegenden Schicht abhängig. Die oberste Schicht ist meist die Präsentationsschicht, welche die UI-Komponenten enthält. Darunter befindet sich die Applikationsschicht. Diese steuert das Verhalten der Anwendung. Die darunterliegende Domänenschicht enthält die Entities und gibt die Verwaltung dieser vor. Die unterste Schicht beinhaltet die Infrastruktur, welche die Persistierung der Entities vornimmt. (Anhang 6.1 – 6.4)

2.4 Modul-Pattern (Voß)

Die einzelnen Schichten der Layered Architecture beinhalten technisch zusammengehörende Elemente, wobei Module einen fachlichen Zusammenhang, teilweise mehrerer Layer, realisieren. Diese Module sind, wie die Schichten der Layered Architecture, lose voneinander gekoppelt. In diesem System sind die Module aufgeteilt in Bet-, Match-, Team- und Usermanagement.

2.5 Entity-Controller-Boundary-Gateway Pattern (Voß)

Durch das ECB-Pattern wird eine Layered Architecture umgesetzt. Die Boundary stellt dabei die Schnittstelle nach außen dar. In der Control-Ebene werden die Anfragen je nach Anforderung (durch Use-Cases definiert) umgesetzt. Die Business-Logik mit den dazugehörigen Modellen (Entities) liegt eine Schicht darunter und ist von der eigentlichen Anwendungssteuerung gekoppelt. Zur Persistierung der Entities wird in diesem System eine weitere Ebene, die Gateway-Schicht, eingeführt. In dieser Schicht wird ein Repository implementiert, welche vorgegebene Methoden aus der Business-Schicht umsetzt.

2.6 SOLID (Voß)

SOLID umfasst 5 Prinzipien der Softwareentwicklung, die die Basis verschiedener Softwarearchitekturen bilden. Die einzelnen Prinzipien werden in diesem Kapitel anhand von Beispielen aus der Anwendung erläutert.

2.6.1 Single Responsibility Principle (SRP) (Voß)

Ein Softwarebaustein sollte nur **eine** Verantwortlichkeit haben, sodass Änderungen lokal bleiben, wodurch die Wartbarkeit erhöht wird. Zur Umsetzung dieses Prinzips gibt es mehrere Ansätze. Eine Variante wäre das Anbieten einer Facade, welche Funktionen beinhaltet, die in einzelnen Klassen implementiert werden. Somit ist jede Klasse nur für einige wenige Methoden zuständig und lässt sich leicht warten oder bearbeiten. In diesem Projekt sind beispielsweise die Manager nur für die Verwaltung der Moduleigenen Anfragen zuständig und haben somit nur eine Verantwortlichkeit. (Anhang 6.1 – 6.4 Control Layer)

2.6.2 Open Closed Principle (OCP) (Voß)

Das OCP gibt vor, dass ein System offen für Erweiterung, aber geschlossen für Modifikation sein soll. Das bedeutet, dass sich das Verhalten anpassen lässt, ohne den Source-Code zu ändern. Erreichen lässt sich dies über die Einführung von Interfaces welche als Erweiterungspunkt dienen. Die Implementierung eines Interfaces kann beliebig ausgetauscht werden.

Die Kommunikation zwischen den einzelnen Layers wurde hier durch Interfaces umgesetzt. Dadurch kann zum Beispiel ein Manager in der Control-Ebene ausgetauscht werden, ohne in der Boundary eine Resource anpassen zu müssen. (Anhang 6.1 – 6.4 Control Layer)

2.6.3 Liskov Substitution Principle (LSP) (Voß)

Dieses Prinzip besagt, dass sich Objekte nach folgendem Muster austauschen lassen: Objekt O1 von Typ T kann durch Objekt O2 von Typ S ausgetauscht werden, falls S ein Subtyp von T ist. Dieses Verhalten wird beispielsweise durch das Strategy-Pattern realisiert.

Durch das Strategy-Pattern wird ein Interface mit einer oder mehrerer Methoden angeboten, welche in Klassen implementiert werden. Die Implementation kann hier von Klasse zu Klasse sehr unterschiedlich sein. Objekte der Klassen, die das Interface implementieren, können allerding beliebig ausgetauscht werden und bieten nach außen alle dieselben Methoden.

2.6.4 Interface Segregation Principle (ISP) (Voß)

Durch das ISP werden Funktionalitäten eines Bausteins in Interfaces gekapselt. Somit werden nach außen nur benötigte und erwartete Funktionen angeboten.

Beispiel in diesem System: Nur benötigte Funktionen werden von Facaden für die Resources angeboten. Weitere Funktionalitäten des Managers, der die Facade implementiert, sind der Resource unbekannt. (Anhang 6.1 – 6.4 Control zu Boundary Layer)

2.6.5 Dependency Inversion Principle (DIP) (Voß)

Durch das DIP lassen sich Abhängigkeiten umkehren, indem Interfaces zur Abstraktion zwischen abhängigen Softwarebausteinen eingefügt werden.

Eine Umkehrung der Abhängigkeit findet zwischen Entity- und Gateway-Ebene statt. Die im Repository benötigten Funktionen werden durch ein Interface (Catalog) vorgegeben. Dieser Vertrag stellt sicher, dass die Business-Logik, welche sich in der Entity-Schicht befindet, nicht von der darunterliegenden Infrastruktur abhängig ist. (Anhang 6.1 – 6.4 Entity zu Gateway Layer)

3 Darstellung der Umsetzung

Nachdem die Grundlagen und Prinzipien der Softwarearchitektur erläutert wurde, wird in diesem Abschnitt die konkrete Umsetzung des Projektes beschrieben. Als nächstes werden wichtige Komponenten erklärt, welche für die Umsetzung von Abhängigkeiten eingesetzt werden.

3.1 Contexts and Dependency Injection (Voß)

Um Abhängigkeiten zu vereinfachen, gibt es unter dem Framework Quarkus die Contexts and Dependency Injection. Über die Dependency Injection ist es möglich, die Erzeugung und Bereitstellung von Objekten von einem DI-Container übernehmen zu lassen. Bei der DI gibt es drei verschiedene Möglichkeiten, Objekte bereitzustellen:

3.1.1 Field Injection (Voß)

Über die Annotation wird dem DI-Container bekanntgegeben, dass eine Abhängigkeit zwischen einer UserResource und einer UserFacade besteht. Kann der UserResource keine UserFacade bereitgestellt werden, schlägt der Build-Vorgang fehl.

```
public class UserResource {
    @Inject
    private UserFacade userFacade;
[…]
}
```

Snippet 1: Klasse mit Field Injection

3.1.2 Constructor Injection (Voß)

Die Inject Annotation kann ebenfalls über einem Konstruktor platziert werden, wodurch die in der Parameterliste übergebenen Objekte bereitgestellt werden, sobald eine UserResource erzeugt wird, da dort der Konstruktor aufgerufen wird. Die Art der Constructor Injection wird in diesem Projekt überwiegend verwendet.

```
public class UserResource {
    private UserFacade userFacade;
    @Inject
    public UserResource(UserFacade userFacade) {
        this.userFacade = userFacade;
    }
[…]
}
```

Snippet 2: Klasse mit Constructor Injection

3.1.3 Method Injection (Voß)

Die Inject Annotation kann ebenfalls über einer Methode angegeben werden, wodurch die im Parameter angegebene Abhängigkeit erst beim Aufruf der Methode bereitgestellt wird. [1]

```
public class UserResource {
    private UserFacade userFacade;
    […]
    @Inject
    void setDeps(UserFacade userFacade) {
        this.userFacade = userFacade;
    }
```

Snippet 3: Klasse mit Method Injection

3.1.4 Eventbasierte Kommunikation der Module (Voß)

Ein weiteres Werkzeug der CDI ist das Event-System. Dies ermöglicht eine lose gekoppelte Kommunikation zwischen Objekten. Einem Event kann ein beliebiger Payload mitgegeben werden. Das Event kann an einem gewünschten Punkt abgeschickt werden, dabei muss der fire-Methode der entsprechende Payload angegeben werden. An einem anderen Punkt kann dieses Event über eine Observes Annotation abgefangen werden. Der Payload des gefeuerten Events kann ausgelesen und entsprechend verwendet werden.

```
public class BetManager implements BetFacade {
    […]
    @Inject
    Event<IncreaseProfilePointsDTO> increaseProfilePointsEvent;
    […]
    increaseProfilePointsEvent.fire(
        new IncreaseProfilePointsDTO(profilePoints)
    );
}
```

Snippet 4: Klasse mit CDI-Event

Wird ein Match von einem Admin beendet, so feuert der MatchManager ein Event, welches als Payload die Match ID und das Endergebnis des Spiels beinhaltet. Der BetManager reagiert auf dieses Event und liest alle Wetten aus, die auf das entsprechende Match platziert wurden. Der BetManager feuert ein neues Event für jede Wette, für die eine Punktevergabe vorgenommen werden muss. Das neue Event des BetManagers beinhaltet die Profil ID und die Punkteanzahl, die dem Nutzer gutgeschrieben werden müssen. Dieses Event wird vom ProfileManager verarbeitet, welcher dem Profil des Nutzers die im Event angegebene Punktzahl gutschreibt. Eine Nachricht kann so modulübergreifend versendet werden, ohne dass eine direkte Abhängigkeit zwischen diesen existiert.

3.2 REST-API (Voß)

Der von der Eclipse Foundation angebotene Standard JAX-RS definiert eine Java Schnittstelle, welche das Erstellen von Webservices im REST-Architekturstil ermöglicht. [2] Es existieren mehrere Implementationen von JAX-RS, eine Implementation ist RESTEasy von JBoss.

Um Web-APIs weiter zu klassifizieren, gibt es das Richardson Maturity Model. Hier werden die Schnittstellen in vier Klassen eingeteilt, die unterschiedliche Funktionalitäten bieten. Level 0 beschreibt ein einfaches Nachfrage – Antwort Prinzip. Anfragen werden per HTTP an einen bestimmten Endpunkt gesendet, woraufhin eine Antwort erfolgt. In Level 1 wird das Konzept der Resources eingeführt. Hier findet nun eine Unterteilung der Endpunkte in verschiedene Resources statt, welche einzeln angesprochen werden können. In Level 2 werden die Resources mit HTTP-Verben angesprochen (GET, POST, DELETE, …). Hierbei kann nun zwischen sicheren und unsicheren Anfragen unterschieden werden. Beispielsweise wird eine GET-Anfrage als sicher angesehen, da sie den Zustand des Systems nicht verändert. Zusätzlich dazu liefern die Antworten nun entsprechende Fehlercodes, je nach Bearbeitung der Anfrage. In Level 3 beinhalten die Antworten des System Links mit URIs zu weiteren möglichen Anfragen. Somit kann eine nächste Anfrage direkt über die vorherige Antwort gesteuert werden. [3]

Diese Web-API erreicht das Level 2 des RMMs. Es werden verschiedene Resources bereitgestellt, welche gängige HTTP-Anfragen verarbeiten (GET, POST, PATCH, DELETE). Hier folgt nun das API-Design der einzelnen Schnittstellen:

Tabelle 1: User Schnittstelle

Resource	**http-Method**	**Request: Param**	**Request: Body**	**Response: Status**	**Response: Body**
/users	POST	--	UserCreationDTO	200	UserID
				400	ErrorDTO

Tabelle 2: Profil Schnittstelle

/profiles	POST	--	ProfileCreationDTO	200	ProfileDTO
	GET		--	200	ProfileDTO Array
				204	Empty
/profiles /{profileId}	GET	profileId	--	200	ProfileDTO
				400	ErrorDTO
/profiles /me	GET	--	--	200	ProfileDTO
				400	ErrorDTO

Tabelle 3: Team Schnittstelle

/teams	POST	--	TeamCreationDTO	200	TeamDTO
				400	ErrorDTO
	GET		--	200	TeamDTO Array
				400	ErrorDTO
/teams /{teamId}	GET	teamId	--	200	TeamDTO
				400	ErrorDTO
	PATCH		TeamUpdateDTO	200	TeamDTO
				400	ErrorDTO
	DELETE		--	200	--
				400	ErrorDTO

Tabelle 4: Match Schnittstelle

/matches	POST	--	MatchCreationDTO	200	MatchDTO
				400	ErrorDTO
	GET		--	200	MatchDTO Array
				400	ErrorDTO
/matches /{matchId}	GET	matchId	--	200	MatchDTO
				400	ErrorDTO
	PATCH		MatchUpdateDTO	200	MatchDTO
				400	ErrorDTO
	DELETE		--	200	--
				400	ErrorDTO

Tabelle 5: Bet Schnittstelle

/bets	POST	--	BetCreationDTO	200	BetDTO
				400	ErrorDTO
	GET		--	200	BetDTO Array
				400	ErrorDTO
/bets /{betId}	GET	betId	--	200	BetDTO
				400	ErrorDTO
	PATCH		BetUpdateDTO	200	BetDTO
				400	ErrorDTO
	DELETE		--	200	--
				400	ErrorDTO
/bets /me	GET	--	--	200	BetDTO Array
				400	ErrorDTO

3.3 Persistierung (Felschen)

Persistenz ist ein Konzept in der Datenbankentwicklung, das sich auf die Fähigkeit eines Systems bezieht, Daten auf Dauer zu speichern, auch wenn das System selbst beendet wird. Dies garantiert, dass die Daten auch bei unvorhergesehenen Ereignissen erhalten bleiben. Durch eine Reihe von Technologien ist es möglich, alle Daten konsistent und vollständig in Datenbanken zu halten oder gegebenenfalls auszulesen. Ebenso werden Mechanismen bereitgestellt, welche Konflikte bei gleichzeitiger Verarbeitung beseitigen.

Zunächst wird auf das verwendete Datenbankmanagement-System eingegangen.

3.3.1 Database Management System – PostgreSQL (Felschen)

Ein DBMS bildet eine Schnittstelle zwischen Anwendung und Daten in einer Datenbank. Es werden Funktionen zum Lesen und Schreiben verschiedener Datentypen angeboten. Des Weiteren ist es möglich Daten über Beziehungen zu Organisieren.

Beim gewählten System namens PostgreSQL handelt es sich um eine relationale Datenbank. PostgreSQL ist ein leistungsstarkes, objektorientiertes, Open-Source-Datenbanksystem, welches eine Vielzahl von Datentypen, einschließlich numerischer und zeichenbasierter-Typen, unterstützt. [4]

Damit die Schnittstelle eines Datenbankmanagement-Systems in Java verwendet werden kann, wird ein JDBC-Treiber verwendet, welcher eine einheitliche Bedienung ermöglicht. [5]

Die Struktur im Rahmen der Projektarbeit wurde wie folgt modelliert. (siehe Abbildung 3: Entity-Relationship-Modell). Es existieren vier verschiedene Entitäten, welche miteinander in Beziehungen stehen. Alle Entitäten sind eindeutig über das Attribut „id“ identifizierbar.

Ein Spiel setzt sich aus zwei verschiedenen Teams zusammen. Teams können in vielen verschiedenen Teams vorkommen. Eine Wette besteht aus einem Spiel, für welches die Wette erstellt wird und einem Profil. Ein Spiel kann in vielen unterschiedlichen Wetten vorkommen. Ebenso kann ein Profil in mehreren Wetten vorkommen. Über das Profil in einer Wette wird identifiziert welcher Benutzer dessen Ersteller ist.

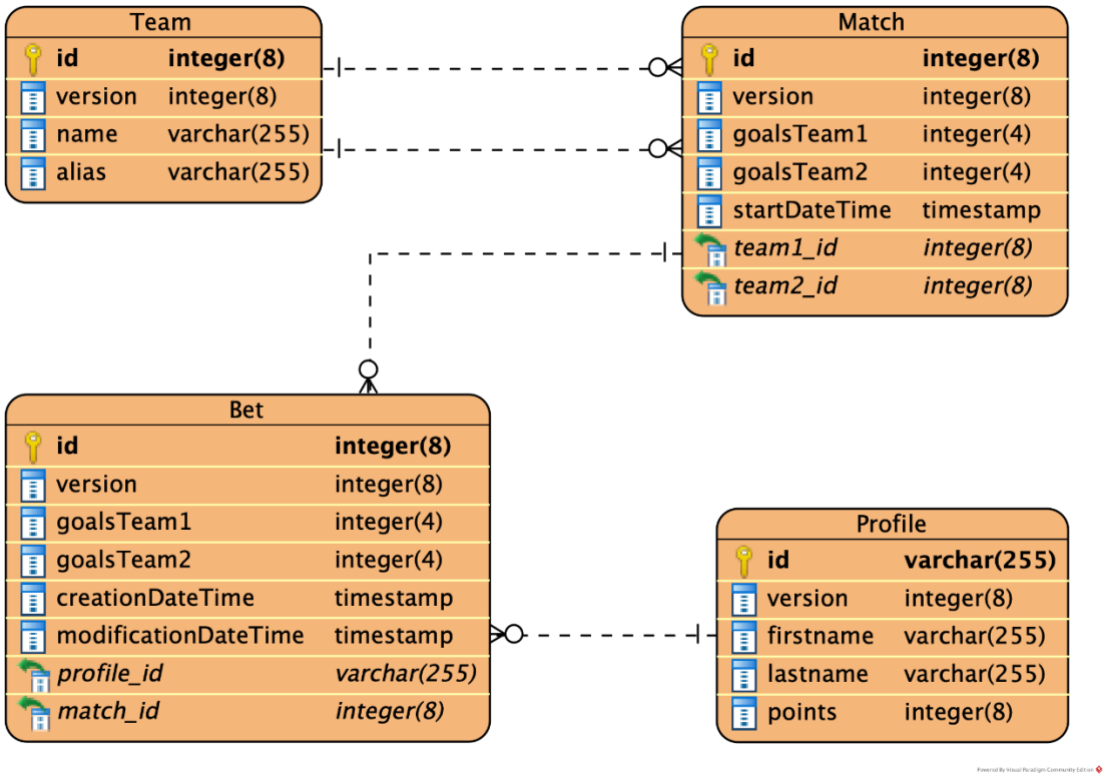


Abbildung 3: Entity-Relationship-Modell (Felschen)

3.3.2 Jakarta Persistence API – Hibernate (Felschen)

Die Java Persistence API bietet Java Entwicklern eine objektrelationale Zuordnungsfunktion, auch „mapping“ genannt, für die Verwaltung von relationalen Daten in Java Anwendungen. Die Schnittstelle ermöglicht unter Anderem das Anlegen von Entitäten und ein kontrolliertes Auslesen über sogenannte „Queries“.

Entitäten stellen im Normalfall eine Tabelle in einer relationalen Datenbank dar. Eine Instanz von einer Entität entspricht einer Zeile in dieser Tabelle. Jede Entität erhält eine eindeutige Identifikationsnummer.

Hibernate setzt die JPA-Spezifikation um und stellt diese Funktionen unter Quarkus bereit. Über Annotationen im Quelltext lassen sich Eigenschaften einzelner Objektvariablen definieren. Wie diese im Projekt verwendet wurden, wird im folgenden Beispiel einer Team Entität erläutert.

Der Codeausschnitt (siehe Snippet 5: Java Klasse eines Teams) zeigt die Java Klasse, welche die Entität eines Teams definiert. Über eine „Entity“-Annotation wird einleitend festgelegt, dass die Klasse eine Entität der Datenbank beschreibt. Um eine eindeutige Identifizierung zu garantieren ist es möglich die „Id“-Annotation zu verwenden. Dieser Wert ist der Primär-Schlüssel in der Datenbanktabelle. Beim Erstellen einer Instanz wird ein eindeutiger Schlüssel generiert und der Entität zugewiesen. Des Weiteren werden im Ausschnitt Annotationen zur Validierung gezeigt, welche in einem folgenden Abschnitt beschrieben werden.

```java
@Entity
public class Team {
    @Id
    […]
    private Long id;
    @Version
    private Long version;
    @Column(nullable = false)
    @Size(min = 1, max = 50)
    private String name;
    @Column(nullable = false)
    @Size(min = 3, max = 3)
    private String alias;
    […]
}
```

Snippet 5: Java Klasse eines Teams

Aus der zuvor erstellten Klassendefinition, wird mittels Hibernate ein „create table“-Statement (siehe Snippet 6: SQL-Statement der Teamtabelle) zur Erstellung der Tabelle in der Datenbank generiert. Um neue Einträge in einer Tabelle zu erstellen wir ein „insert“-Statement für die Instanz generiert, welche persistiert werden soll.

```sql
create table team (
    id int8 not null,
    alias varchar(255) not null,
    name varchar(255) not null,
    version int8,
    primary key (id)
)

insert into
    team
    (alias, name, version, id)
values
    (?, ?, ?, ?)
```

Snippet 6: SQL-Statement der Teamtabelle

Hibernate abstrahiert somit den Zugriff auf die Datenbank und ermöglicht auf Ebene der Business Logik eine objektorientierte Programmierung. Dies wird über das „mapping“ der Tabellen und Einträge der Datenbank auf Objekt Instanzen realisiert.

Eine wichtige Rolle, um diese Funktionalität zu realisieren, spielt der im folgenden Abschnitt beschriebene Manager.

3.3.3 Entity Manager (Felschen)

Entitäten werden von einem Entitätsmanager verwaltet, der durch EntityManager-Instanzen dargestellt wird. Jede EntityManager-Instanz ist mit einem „Persistence Context“ verbunden, welcher einem Satz verwalteter Entitäts-Instanzen entspricht, die in einem bestimmten Datenspeicher vorhanden sind. Ein „Persistence Context“ definiert den Bereich, in dem bestimmte Entitäts-Instanzen erstellt, persistiert oder entfernt werden können. Über eine Schnittstelle werden Methoden angeboten, die zur Interaktion mit dem Context verwendet werden können. [6]

Im Rahmen der Projektarbeit werden EntityManager-Instanzen jeweils in den Repositories im Gateway-Layer injiziert. Um Entitäts-Instanzen zu speichern, wird die „persist“-Methode des Managers (siehe Snippet 7: EntityManager Methoden) verwendet. Diese bekommt als Parameter eine zuvor instanziierte Entität. Ein anschließendes „flush“ synchronisiert den zugehörigen „Persistence Context“ und den darunter liegenden Datenspeicher.

Um Entitäten zu bearbeiten, wird eine „merge“-Methode angeboten. Diese erhält eine geänderte Entitäten-Instanz und aktualisiert den Zustand im „Persistence Context“.

```
@Inject
private EntityManager em;
   […]
Team entity = new Team(name, alias);
   […]
em.persist(entity);
em.flush();
   […]
em.merge(entity);
   […]
em.remove(entity);
   […]
Team team = em.find(Team.class, id);
```

Snippet 7: EntityManager Methoden

Damit Daten aus der Datenbank gelesen werden können, werden verschiedene Methoden angeboten. Wird lediglich nach einem Primärschlüssel in einer Datenbanktabelle gesucht, kann die „find“-Methode verwendet werden. Diese erhält eine Java-Klasse, um den Rückgabetyp zu definieren und einen Primärschlüssel, nach welchem gesucht werden soll.

Sobald eine komplexe Suche erfolgen soll, wird eine SQL ähnliche Sprache namens Java Persistence Query Language verwendet, welche im nächsten Abschnitt beschrieben wird.

3.3.4 Typed Queries (Felschen)

Bei komplexen Suchen nach Entitäten, werden sogenannte „TypedQueries“ verwendet (siehe Snippet 8: Named Query). Über eine „NamedQuery“ Annotation in der Klassendefintion einer Bet Entität, ist es möglich eine Query anzulegen, welche unter einem definierten Namen angesprochen werden kann. Die eigentliche Query wird als Zeichenkette in der JPQL angegeben. Unter anderem ist es möglich Platzhalter für Parameter zu erstellen. Diese werden mit einem einleitenden Fragezeichen und einer Positionsnummer gesetzt.

```
Bet.java
@NamedQueries({
     @NamedQuery(
        name = "Bet.findByProfileId",
        query = "SELECT b FROM Bet b WHERE b.profile.id = ?1"),
})
@Entity
public class Bet {
    […]
}

BetRepository.java
public class BetRepository […] {
    @Inject
    private EntityManager em;
    […]
    TypedQuery<Bet> query = em.createNamedQuery("Bet.findByProfileId", Bet.class);
    query.setParameter(1, profileId);
    return query.getResultList();
    […]
}
```

Snippet 8: Named Query

Der EntityManager bietet die Methode „createNamedQuery“ an, welche es ermöglicht den Namen einer zuvor definierten „NamedQuery“ anzugeben. Zudem wird der Rückgabetyp der Query angegeben. Bevor die Query ausgeführt werden soll, ist es möglich, Query Parameter zu übergeben. Diese werden mit der „setParameter“-Methode gesetzt, indem eine Parameter Position und ein Wert angegeben wird. [7]

Damit die Query ausgeführt wird, existiert eine „getResultList“- und eine „getSingleResult“-Methode, welche entweder eine Liste oder eine einzelne Entität zurückgeben.

3.3.5 Jakarta Transaction API (Felschen)

Aufgrund der Tatsache, dass mehrere Systeme auf eine Datenbank zugreifen wollen, wobei die Daten eine wichtige Rolle in der Business Logik haben, ist sicherzustellen, dass die Informationen genau, aktuell und zuverlässig sind. Die Datenintegrität würde verloren gehen, wenn mehrere Programme gleichzeitig dieselben Informationen aktualisieren könnten oder wenn ein System, das während der Verarbeitung einer Transaktion ausfällt, die betroffenen Daten nur teilweise aktualisiert lassen würde.

Transaktionen steuern genau diese Zugriffe auf Daten durch mehrere Programme. Im Falle eines Systemausfalls stellen diese sicher, dass die Daten nach der Wiederherstellung in einem konsistenten Zustand sind. [8]

Transaktionen lassen sich mit der Annotation „Transactional“ in Java steuern. Im Rahmen der Projektarbeit werden Transaktionen auf Ebene der Boundary pro Request erstellt. Dies geschieht durch den Parameter „TxType.REQUIRES_NEW“ der Transaktionsannotation.

Auf der Ebene der Repositories werden bestehende Transaktion betreten. Kommt es in einer Methode zu einer Exception, wird ein Rollback ausgeführt. Dies bedeutet, dass die zuvor in der Transaktion ausgeführten Datenbankaktionen rückgängig gemacht werden. Dadurch ist ein konsistenter Zustand wiederhergestellt.

3.3.6 Jakarta Bean Validierung (Felschen)

Damit sichergestellt ist, dass sich nur valide Daten in der Datenbank befinden, werden vor allem Entitäten, welche aus Informationen bestehen, die vom Benutzer eingegeben wurden, untersucht. Über eine Bean Validation lassen sich Attribute syntaktisch und semantisch validieren.

Über einen injizierten Validator werden im Repository Entitäten nach dem Instanziieren auf ConstraintViolations geprüft. Die „validate“ Methode (siehe Snippet 9: Validator) liefert eine Reihe an Regelverstößen, welche ausgelesen und dem Benutzer lesbar angezeigt werden können.

```
@Inject
private Validator validator;
[…]
Set<ConstraintViolation<Team>> violations =
    validator.validate(entity);

if (!violations.isEmpty()) {
    throw new […]
}
```

Snippet 9: Validator

Die Definition dieser Regeln wird in den jeweiligen Entity Klassen erstellt. Es gibt eine Menge an Annotationen, welche verwendet werden können, um Entity Attribute semantisch zu prüfen. Unter anderem werden im Projekt die „Min“- und „Max“-Annotationen verwendet, um Zahlenwerte nur in einem gewissen Bereich zuzulassen. Des Weiteren werden Zeichenketten mit der „Size“-Annotation auf ihre Länge beschränkt.

```
@Min(value = 0)
@Max(value = 99)
private Integer goalsTeam1;

@Size(min = 1, max = 50)
private String name;

@AssertTrue(message = "match teams must be different")
private boolean isDifferentTeams() {
    return !team1.equals(team2);
}
```

Snippet 10: Validator Annotationen

Außerdem besteht die Möglichkeit mit der „AssertTrue“-Annotation (siehe Snippet 10: Validator Annotationen) einen boolschen-Ausdruck zu validieren. Im Projekt wird diese Validierung verwendet, um sicherzustellen, dass kein Spiel (beziehungsweise Match) existieren kann, welches zwei gleiche Teams enthält.

3.4 Verfügbarkeit / Resilienz (Voß)

Um die Anwendung widerstandfähiger gegen Fehler zu gestalten, wird die SmallRye Fault Tolerance Implementation des Eclipse MicroProfile Standards eingesetzt. Durch diese Technologie können Annotationen verwendet werden, die ein Verhalten des Systems im Fehlerzustand vorgeben. Hier verwendet werden dabei folgende Annotationen: [9]

- Retry: Falls das System eine Anfrage fehlerhaft verarbeitet, kann über diese Annotation die Anzahl an Wiederholungsversuchen festgelegt werden.
- Timeout: Über die Timeout Annotation kann der Verarbeitung einer Anfrage eine maximale Dauer zugewiesen werden. Dies ist nützlich, falls das System überladen ist.
- Circuit Breaker: Mit der Circuit Breaker Annotation kann die Anzahl an Fehlern limitiert werden. Die Angabe eines „requestVolumeThreshold“ legt fest, wie viele der letzten Anfragen für den Circuit Breaker betrachtet werden sollen. Der Standardwert der „failureRatio“ beträgt 0.5. Werden nun diese beiden Werte betrachtet, ergibt sich, dass bei einem Fehleraufkommen von zwei der vier letzten Anfragen, der Circuit Breaker öffnet und weitere Anfragen für fünf Sekunden (Standardwert) unterbindet.
- Des Weiteren gibt es die Möglichkeit über die „Fallback“ Annotation eine Methode anzugeben, die im Falle eines Fehlerzustandes ausgeführt wird.

Am Beispiel des ProfileManagers wird die Verwendung im Projekt erklärt. Werden die Annotationen über der Klasse angegeben, wird das gewünschte Verhalten für jede Methode der Klasse erreicht. Fehlerhafte Anfragen werden viermal neu ausgeführt und erreichen bei einer Verarbeitungsdauer von über 250 Millisekunden einen Timeout Zustand. Der Circuit Breaker wurde so konfiguriert, dass er die in „skipOn“ angegebenen Exceptions ignoriert und das System nicht unterbricht. Diese Exceptions werden zu Fehlerausgabe genutzt.

```
@Retry(maxRetries = 4)
@Timeout(250)
@CircuitBreaker(requestVolumeThreshold = 4, skipOn = {
        EntityCreationException.class,
        EntityDeleteException.class,
        EntityQueryException.class,
        EntityReadException.class,
        EntityUpdateException.class,
        ReferenceNotFoundException.class,
})
public class ProfileManager implements ProfileFacade {
    […]
}
```

Snippet 11: Resilienz / Verfügbarkeit im ProfileManager

3.5 Metriken / Logging (Voß)

Um Statistiken und Metriken des Systems zu erfassen wird die SmallRye Metrics Implementation des MicroProfile Metrics Standards verwendet. Dadurch können Einblicke in das Verhalten der Anwendung gewährt werden. Diese Statistiken können im JSON-Format abgerufen werden. Auch diese Technologie wird über Annotationen gesteuert. [10]

- Counted: Wird einer Methode die Counted Annotation hinzugefügt, kann unter Angabe eines Namens und einer Beschreibung dieser Metrik gemessen werden, wie oft diese Methode aufgerufen wurde.
- Timed: Wie die Counted Annotation wird auch der Timer-Metrik einen Namen und eine Beschreibung gegeben, sowie eine Einheit, in welcher die Metrik erfasst werden soll.
- Gauge: Gauge ermöglicht es eine selbst definierte Metrik zu erfassen. Hierzu wird wie zuvor einen Namen und eine Beschreibung der Metrik definiert und eine zu erfassende Einheit gewählt.

Für die „create“ Funktion des ProfileManagers werden über die Counted Annotation die Anzahl der Funktionsaufrufe aufgezeichnet. Anhand dieser Metrik lässt sich ablesen, wie viel Profile erstellt wurden. Die Timed Annotation misst die Dauer des Funktionsablaufs in Millisekunden und stellt dafür verschiedene Statistiken bereit, wie zum Beispiel die kürzeste und längste Ausführungsdauer oder Durchschnittszeit.

```
@Override
@Counted(name = "createdProfiles",
         description = "How many profiles have been created.")
@Timed(name = "createdProfilesTimer",
       description = "A measure of how long it takes create a profile",
       unit = MetricUnits.MILLISECONDS)
public Profile create(String profileId, ProfileCreationDTO creationDTO) {
    […]
}
```

Snippet 12: Metriken der "create" Funktion

Quarkus verwendet den JBoss Log Manager und die JBoss Logging facade. Das Log-Verhalten kann in der „application.properties“ Konfigurationsdatei eingestellt werden. Es gibt verschiedene Log-Levels [11], wie zum Beispiel:

- Error: Für ernstzunehmende Fehlermeldungen
- Warn: Nicht-kritische Fehler oder Probleme
- Info: Informationen über Lifecycle Events
- Debug: Extrainformationen

```
public Profile create(String profileId, String firstname, String lastname, Long points) {
    LOG.infof("ProfileRepository#create called with
               profileId: %s, firstname: %s, lastname: %s, points: %d",
               profileId, firstname, lastname, points
    );
    […]
}
```

Snippet 13: Log Ausgabe in ProfileRepository

Hier zu sehen ist eine Ausgabe auf Informationsebene, die beim Anlegen eines Profil Informationen darüber angibt, welche Klasse und welche Methode aufgerufen wurden. Zusätzlich werden die angegebenen Parameter mit ausgegeben.

3.6 Dokumentation – Swagger UI (Felschen)

Swagger UI bietet eine übersichtliche Auflistung der umgesetzten Schnittstellen. Neben den zulässigen Methoden wird zudem der relative Pfad zu den jeweiligen Endpunkten dargestellt.

Die Abbildung 4: Swagger UI Auszug zeigt die umgesetzte Schnittstelle, für den Zugriff auf Spielwetten. Um diese Schnittstelle auszuprobieren ist es möglich einzelne Requests direkt über die Swagger UI abzuschicken.

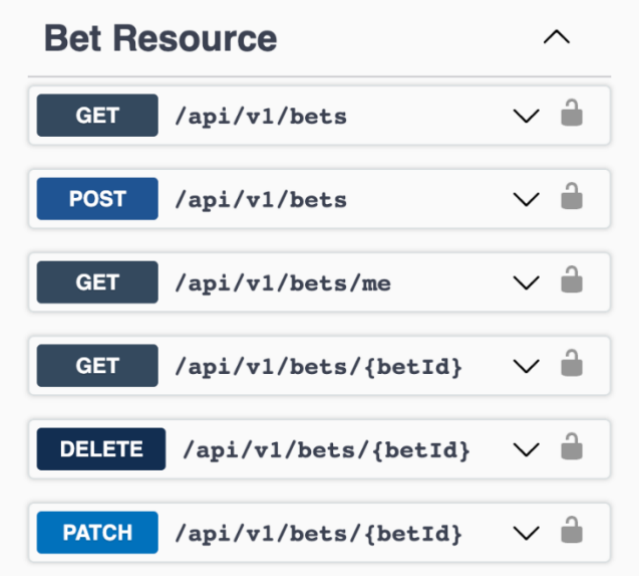


Abbildung 4: Swagger UI Auszug

Die gesendete Antwort der API kann danach betrachtet werden. Sind für einen Request einzelne Parameter oder Daten notwendig, lassen sich diese über bereitgestellte Eingabefelder definieren.

Eine weitere Funktion der UI ist eine Übersicht der erwarteten Antwort Status-Codes pro Request. Dem Benutzer wird somit visualisiert, mit welchen Antworten er rechnen kann, im Erfolgs- oder Fehlerfall.

3.7 Web-Frontend (Felschen)

Bei der Umsetzung des Frontends wurde sich für ein Server-Side Rendering (SSR) entschieden. Bei dieser Methode wird die Web-Anwendung auf dem Server generiert und an den Client gesendet, bevor sie im Browser des Benutzers angezeigt werden kann.

Damit aus der Java Anwendung eine benutzerfreundliche Web-Anwendung generiert werden kann, kommen sogenannte Template Engines zum Einsatz. Wie diese eingesetzt werden wird im folgenden Abschnitt gezeigt.

3.7.1 Quarkus Template Engine (Qute) (Felschen)

Eine Template Engine ist eine Software-Komponente, die es ermöglicht, dynamische Inhalte innerhalb statischer Vorlagen zu generieren. Sie wird verwendet, um die Trennung von Inhalt und Präsentation in Web-Anwendungen sicherzustellen.

Template Engines verarbeiten Vorlagen, die eine Kombination aus statischen und dynamischen Inhalten enthalten, und generieren daraus eine Ausgabe, die nur aus statischen Inhalten besteht. Die dynamischen Inhalte werden auf der Grundlage von Daten berechnet, die zur Laufzeit an die Template Engine übergeben werden.

Im Rahmen der Projektarbeit wurden strukturierte Vorlagen bestehend aus der Hypertext Markup Language erstellt. Qute bietet innerhalb der Vorlagen eine eigene Notation, um Inhalte einzufügen, Bedingungen zu verarbeiten oder Schleifen auszuführen. Des Weiteren bietet Qute die Möglichkeit HTML-Dateien ineinander zu integrieren. Somit verringert sich die HTML-Text Duplizierung erheblich.

Ein Beispiel in diesem Fall ist die Navigationszeile, welche sich auf jeder Seite am oberen Rand befindet. Diese wurde bewusst in einer eigenen HTML-Datei ausgelagert, was garantiert, dass diese wiederverwendet werden kann. Qute bietet zudem eine Möglichkeit Parameter beim „#include“ (siehe Snippet 14: Qute include) zu übergeben. Diese können per „#insert“ in der anderen Datei ausgelesen werden. Im Rahmen der Projektarbeit wurde diese Funktion jedoch nicht verwendet.

```
<html>
    […]
    {#include navbar}{/include}
    […]
</html>
```

Snippet 14: Qute include

Um eine Liste von Daten zu visualisieren, ist es möglich mit der „#for“ Notation eine Iteration zu starten. In diesem Fall (siehe Snippet 15: Qute Schleife) wird eine „teams“ Liste durchgegangen, innerhalb der Schleife sind die einzelnen Teams unter der Bezeichnung „team“ erreichbar. Um über Qute an Instanzvariablen zu gelangen, werden geschweifte Klammern verwenden. Diese sorgen dafür, dass die Template Engine den Wert der Variable ausliest und in HTML darstellt.

```
{#for team in teams}
    <tr>
        <td>{team.name}</td>
        <td>{team.alias}</td>
    </tr>
{/for}
```

Snippet 15: Qute Schleife

Soll eine Bedingung geprüft werden, lässt sich diese mit einer „#if“ Notation (siehe Snippet 16: Qute If-Anweisung) einleiten. Der HTML-Text innerhalb der if-Anweisung wir nur ausgegeben, falls die Bedingung der Umfassenden Anweisung wahr ist. Wird zusätzlich ein „#else“ angegeben, kann HTML-Text angegeben werden, welcher ansonsten angezeigt werden soll. Im Projekt wird diese Methode verwendet, um unter anderem je nach Anmeldestatus eine passende Navigationszeile bereit zu stellen.

```
<html>
    {#if isAuthenticated}
        […]
    {#else}
        […]
    {/if}
</html>
```

Snippet 16: Qute If-Anweisung

Nachdem die Vorlagen für die Template Engine definiert wurden, müssen diese von der Anwendung mit Daten gefüllt und instanziiert werden.

Qute in Verbindung mit Quarkus bietet eine Reihe an Funktionen, um Templates zu verwenden. Zunächst müssen die zuvor beschriebenen Templates gefunden werden. Durch eine von Qute definierte Ordnerstruktur und einer Template Bezeichnung lassen sich die einzelnen Templates als „TemplateInstance“ (siehe Snippet 17: Template Instanziierung) definieren. Um eine Typensicherheit zu garantieren, werden bei der Definition direkt Parameter angegeben, welche das Template erhalten muss, um erfolgreich generiert zu werden.

```
@CheckedTemplate
public static class Templates {
    public static native TemplateInstance TeamListTemplate(
        Collection<TeamDTO> teams
    );
}
[…]
@GET
public TemplateInstance get() {
    […]
    return Templates.TeamListTemplate(dtoList);
}
```

Snippet 17: Template Instanziierung

Um die Instanzen nach außen anzubieten, werden in WebResourcen auf der Boundary Ebene nun GET- und POST-Anfragen erwartet. Wird ein Request empfangen, wird eine Template Instanz erstellt und mit Daten gefüllt. Zuletzt wird die Instanz dem aufrufenden Benutzer zurückgeliefert. [12]

Damit dem Benutzer eine Oberfläche bereitgestellt wird, welche einheitliche Komponenten zum Interagieren besitzt, wurde das im folgenden Abschnitt beschriebene Toolkit verwendet.

3.7.2 Bootstrap (Felschen)

Bootstrap ist ein leistungsstarkes Toolkit mit vielen Funktionen, welches das Erstellen vom Frontend einer Web-Anwendung erheblich beschleunigen kann. Durch das Einbetten einer gegebenen CSS-Datei und Skripte, geschrieben mit JavaScript, werden eine Vielzahl an Komponenten bereitgestellt. Darunter zählen Elemente zum Ausrichten und Positionieren auf der Website, Eingabefelder, um beispielsweise Texteingaben vom Benutzer zu erhalten und Schaltflächen wie Buttons. [13]

3.7.3 Aufbau der Oberfläche (Felschen)

Auf der Web-Oberfläche befindet sich jeweils eine Navigationszeile am oberen Ende der Seite. Diese besitzt je nach Anmeldestatus und Berechtigung des Nutzers unterschiedliche Navigationsmöglichkeiten. Die Startseite der Oberfläche bildet die „Tippen“-Seite (siehe Anhang 5: UI – Tippen), auf dieser ist es als nicht angemeldeter Benutzer möglich, die kommenden Spiele einzusehen. Ist der Benutzer angemeldet, besteht die Möglichkeit einen Tipp zu erstellen. Die abgegebenen Tipps werden in einer Liste oberhalb der einzelnen Spiele dargestellt. Ein Tipp kann bis zum Start des Spiels beliebig bearbeitet werden. Der Spielstart steht jeweils rechts unten. Um ein Tipp zu bearbeiten, muss der Benutzer ein neues Ergebnis eintragen und die „Bearbeiten“ Schaltfläche (siehe Abbildung 5: Darstellung eines offenen Tipps) betätigen. Ist das Spiel bereits angefangen, werden die Eingabefelder ausgegraut (siehe Abbildung 6: Darstellung eines Tipps) und die Schaltfläche zum Bearbeiten wird unsichtbar. Das aktuelle Ergebnis wird neben den Team Bezeichnungen visualisiert und kann somit mit dem abgegebenen Tipp verglichen werden.

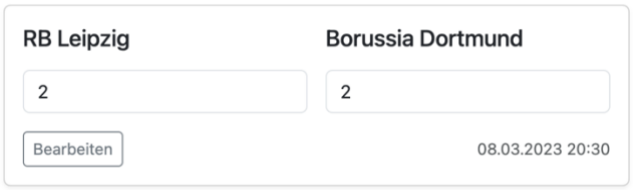


Abbildung 5: Darstellung eines offenen Tipps

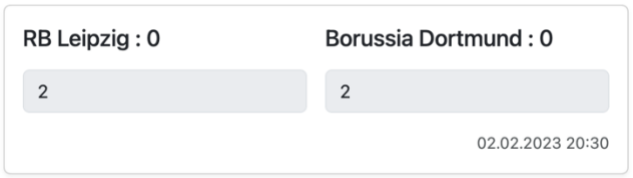


Abbildung 6: Darstellung eines Tipps

Um einen Einblick zu erhalten, mit welchem Konto der Benutzer angemeldet ist und wie viele Punkte er bereits durch das Tippen erreicht hat, kann über die oben genannte Menüzeile zum Konto navigiert werden. Auf dieser Seite (siehe Anhang 9: UI – Konto) wird die Punktzahl und der Vor- und Nachname des registrierten Benutzers dargestellt. Über eine Schaltfläche kann der Benutzer sich vom System abmelden. Möchte der Benutzer sich alle Profile ansehen, kann er dies auf der Seite der Rangliste. Hier wird eine Liste aller Profile (siehe Anhang 5: UI – Tippen) sortiert nach den jeweiligen Punktzahlen dargestellt.

Damit das System mit Spieldaten gefüllt wird, hat ein Administrator eine extra Navigationszeile unter seiner Profilseite. Hier ist es möglich, zur Teams- und zur Spielplanseite zu gelangen.

Auf der Teams-Seite (siehe Anhang 6: UI – Rangliste) können neue Teams angelegt werden oder bestehende Teams bearbeitet werden. Über eine Schaltfläche lässt sich ein sogenanntes Modal öffnen (siehe Abbildung 7: Team Modal), welches Eingabefelder bereitstellt, um einen Namen und ein Alias für ein Team festzulegen. Das Erstellen wird über die Schaltfläche „Fertig“ abgeschlossen.

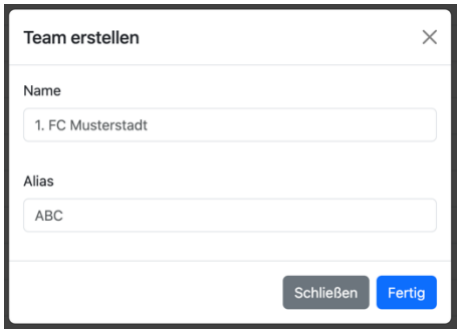


Abbildung 7: Team Modal

Damit Teams anschließend bearbeitet werden können lässt sich dieses Modal pro Team erneut öffnen. Dafür muss der Benutzer ein Team aus der Teamlist auswählen, welches bearbeitet werden soll. Die Eingabefelder werden dann mit den aktuellen Teamdaten gefüllt und können geändert werden.

Die existierenden Teams können nun zu einem neuen Spiel hinzugefügt werden. Auf der Spielplan-Seite (siehe Anhang 8: UI – Matches) besteht ebenfalls eine Schaltfläche zum Erstellen eines neuen Spiels.

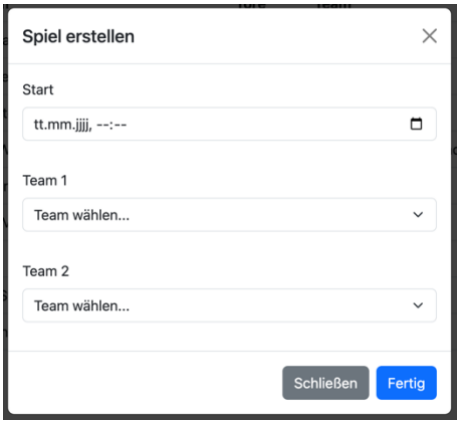


Abbildung 8: Spiel Modal

Damit der Start des Spiels vom Benutzer festgelegt werden kann, lässt sich das Datum und die Uhrzeit formatiert über ein Feld eingeben (siehe Abbildung 8: Spiel Modal). Anschließend müssen zwei Teams ausgewählt werden. Die Schaltfläche listet alle vorhandenen Teams auf und der Benutzer muss lediglich das passende Team betätigen damit dieses ausgewählt wird. Abschließend muss der Benutzer die Schaltfläche „Fertig“ betätigen, womit der Vorgang beendet wird.

Um ein Spiel mit aktuellen Ergebnissen zu vervollständigen, ist über ein zusätzliches Modal (siehe Abbildung 9: Spiel bearbeiten - Modal) die Eingabe der erzielten Tore möglich. Wenn das Ergebnis feststeht und das Spiel beendet ist, kann dies über eine Kontrollbox aktualisiert werden.

Nachdem das Spiel beendet wurde, beginnt die Punkteverteilung der Benutzer, welche für dieses Spiel einen Tipp abgegeben haben.

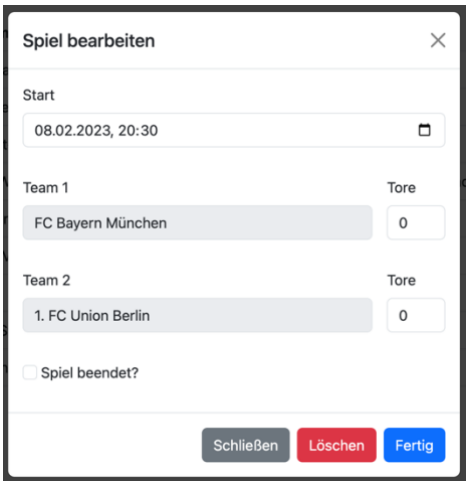


Abbildung 9: Spiel bearbeiten - Modal

3.8 Fehlerbehandlung (Exceptions) (Felschen)

Die Fehlerbehandlung eines Systems beschreibt den Prozess, welcher ausgelöst wird, falls eine ausgeführte Funktion oder Methode nicht die erwartete Aufgabe erfüllt, welche angestrebt wurde. Im Rahmen des Projektes wurden Klassen umgesetzt, um dem Benutzer möglichst genau anzuzeigen, welcher Fehler aufgetreten ist und wie er diesen gegebenenfalls beheben könnte.

Damit dem Benutzer verständliche Fehlernachrichten angezeigt werden können, wurden eigene Exception-Klassen definiert. Beispielsweise wird eine EntityCreationException geworfen, falls beim Erstellen einer neuen Entität etwas nicht funktioniert hat. Analog dazu wird beispielsweise eine EntityReadException geworfen, falls etwas beim Auslesen von Entitäten nicht funktioniert hat.

Das Werfen der einzelnen Exceptions übernehmen im Projekt die einzelnen Repositories in der Gateway-Schicht. Diese fangen beim Datenbankzugriff und beim Validieren von Entitäten alle Exceptions, welche der Zugriff werfen kann, ab und wandeln diese in die definierten Exceptions um. Die definierten Exceptions enthalten nun vor allem eine Fehlernachricht und den Namen der Entität, welche einen Fehler verursacht hat.

Wird eine solche Exception geworfen, kommt eine weitere Komponente zum Einsatz. Ein globaler ExceptionMapper wandelt die geworfenen Exceptions über die Resteasy-ServerExceptionMapper-Annotation in eine darstellbare Response um.

Aus den erfassten Daten der Exception wird ein DTO erstellt und als strukturiertes JSON-Objekt als Antwort gesendet. Im Fehlerfall wird dem Benutzer somit eine Einheitliche Antwort (siehe Snippet 18: Antwortbeispiel im Fehlerfall) geliefert welche von ihm ausgewertet werden kann.

```json
{
  "code": "error",
  "status": 400,
  "source": "[…].Team",
  "title": "EntityCreationException",
  "detail": "name : Größe muss zwischen 1 und 50 sein"
}
```

Snippet 18: Antwortbeispiel im Fehlerfall

3.9 Sicherheit – Keycloak (Voß)

OpenID Connect ist eine Schicht auf dem OAuth 2.0 Protokoll, welche eine Verifizierung des Endnutzers über einen Autorisierungsserver, in diesem Fall Keycloak, realisiert. [14] Quarkus ermöglicht es Keycloak über eine Dependency hinzuzufügen. Über die Dev Services kann im Entwicklungs- und Testmodus ein Docker Container gestartet werden, in dem ein Keycloak Server läuft. Hierbei werden einige Standardeinstellungen geladen, welche sich über die „application.properties“ konfigurieren lassen.

In Snippet 19: Keycloak Server Konfiguration ist die Konfiguration des Keycloak Servers zu sehen. Der Testcontainer ist für den Entwicklermodus aktiviert und für den Produktionsmodus deaktiviert. Der Port wird auf 8180 gesetzt (Quarkus nutzt 8080). Die auskommentierte Zeile würde ermöglichen eine Konfigurationsdatei für einen Keycloak Realm zu laden. Wird dies nicht genutzt, lädt der Testcontainer eine standardmäßige Realm Konfiguration. Ein Realm stellt in

```
# Keycloak Config
%dev.quarkus.keycloak.devservices.enabled=true
%prod.quarkus.keycloak.devservices.enabled=false
quarkus.keycloak.devservices.port=8180
# quarkus.keycloak.devservices.realm-path=quarkus-testrealm.json
```

Snippet 19: Keycloak Server Konfiguration

Keycloak eine Verwaltungseinheit dar. In dieser werden Nutzer, Anwendungen, Rollen und Gruppen zusammengefasst. Der Master Realm beinhaltet den Admin Account und wird nicht für die Verwaltung von anderen Daten genutzt, [15] hierfür werden eigene Realms angelegt, wie in dieser Anwendung der Quarkus Realm. Zur Kommunikation mit Keycloak wird ein Client genutzt. Clients spiegeln Anwendungen und Services wider, welche Keycloak nutzen, um Sicherheit zu gewähren. [15]

Um eine Anwendung zu schützen, werden Tokens verwendet. Ein Token, zum Beispiel ein JWT, wird zum Nachweis der Identität verwendet. In der realen Welt kann ein Token mit einem Personalausweis verglichen werden. In einem Token sind verschiedene Informationen gespeichert, wie zum Beispiel der genutzte Algorithmus, Name oder Nutzerrollen. [16] Zudem können weiter Attribute in dem Token gespeichert werden, welche sich über Claims auslesen lassen. Es gibt verschiedene Verfahren zur Prüfung solcher Tokens.

Wird über ein Tool wie Insomnia auf unseren Service zugegriffen, muss vorher für geschützte Endpunkte ein Bearer Token angefragt (siehe Abbildung 10: Bearer Token Authorization [17]) und angegeben werden. Um ein Bearer Token zu beantragen, wird der Client unserer Anwendung genutzt.

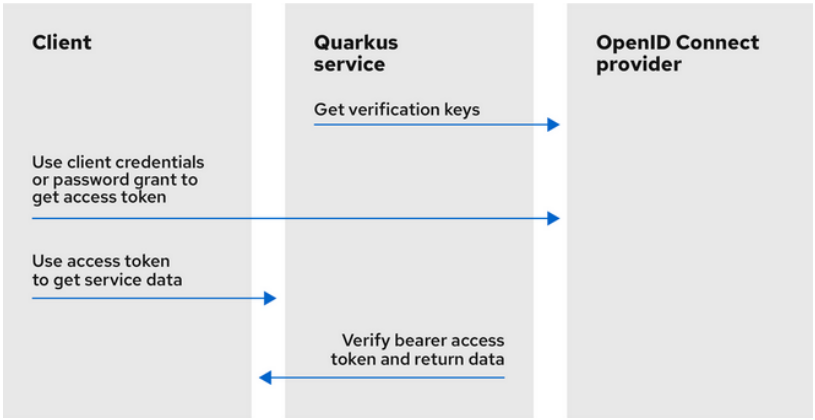


Abbildung 10: Bearer Token Authorization [17]

Im ersten Schritt werden Verifikationsschlüssel bereitgestellt, die genutzt werden, um die Signatur des Bearer Tokens zu prüfen. Über die Anmeldedaten des Clients kann dieser im nächsten Schritt dazu verwendet werden, einen Token abzurufen. Dieser Token kann nun in Anfragen an den Service verwendet werden. Die im ersten Schritt angefragten Verifikationsschlüssel überprüfen nun die Gültigkeit des Bearer Tokens. Bei erfolgreicher Prüfung wird die Anfrage verarbeitet. [17]

Greift ein Nutzer über die Web-App auf den Service zu, wird der Authorization Code Flow (siehe Abbildung 11: Authorization Code Flow [18]) verwendet.

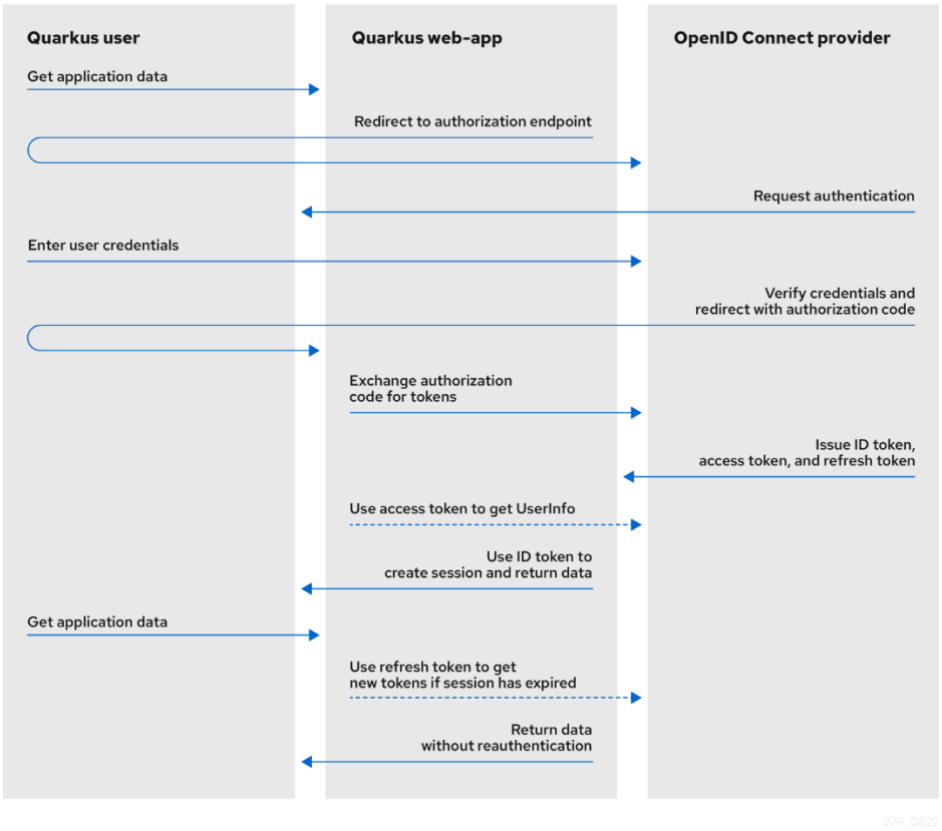


Abbildung 11: Authorization Code Flow [18]

Beim Zugriff eines nicht-angemeldeten Nutzer auf die Web-App wird eine Weiterleitung auf die Login-Form von Keycloak ausgeführt. Hier kann der Nutzer sich mit seinen Anmeldedaten einloggen. Nach der Überprüfung der Daten wird ein Berechtigungscode von Keycloak ausgestellt und eine Weiterleitung auf die ursprünglich angeforderte Seite der Web-App ausgeführt. Der Berechtigungscode wird verwendet, um drei Tokens, ID-, Access- und Refresh Token, zu erhalten. Das ID-Token beinhaltet die Identität des Nutzers, das Access Token wird genutzt, um weitere Nutzerdaten von der UserInfo API abzurufen und über das Refresh Token können neue ID- und Access Tokens angefordert werden, falls diese ablaufen. [18]

Die Endpunkte der Anwendung werden durch eine rollenbasierte Zugriffkontrolle geschützt. Im Keycloak Realm sind den Nutzern eine oder mehrere Rollen, wie zum Beispiel User oder Admin, zugewiesen. Über Annotationen kann festgelegt werden, welche Nutzer auf welche Endpunkte Zugreifen dürfen. Die Annotationen können entweder auf Methoden- oder auf Klassenebene definiert werden.

Soll ein Endpunkt für alle Internetnutzer zugänglich sein, so wird die PermitAll Annotation verwendet. Der Zugriff wird so unabhängig von Rolle oder Authentifizierung gewährt. Die im Beispiel angegebene get Methode darf somit von allen Nutzern aufgerufen werden.

```
    @GET
    @PermitAll
    public Response get() {
    […]
}
```

Snippet 20: Keine Zugangsbeschränkung

Falls nur angemeldete Nutzer, unabhängig von der Rolle, Zugriff erhalten sollen, wird die Authenticated Annotation verwendet. Die Angabe über der ProfileResource Klasse bedeutet, dass alle Methoden dieser Klasse nur von angemeldeten Nutzern aufgerufen werden dürfen.

```
@Authenticated
public class ProfileResource {
    […]
}
```

Snippet 21: Zugang nur angemeldete Nutzer

Um nur gewissen Rollen Zugriff zu erteilen, kann die RolesAllowed Annotation verwendet werden. Hier können eine oder mehrere Rollen angegeben werden, welchen der Zugriff gewährt werden soll. Im angegebenen Beispiel erhalten nur Nutzer mit der Admin Rolle Zugriff auf die post Methode.

```
    @POST
    @RolesAllowed({ "admin" })
    public Response post(MatchCreationDTO
creationDTO) {
      […]
    }
```

Snippet 22: Zugang nur Admins

```
public boolean removeById(String profileId, Long betId) {
    Bet bet = this.betCatalog.findById(betId);
    // Nur ein Admin oder der Ersteller haben Zugriff
    if (!securityIdentity.hasRole("admin") && !bet.getProfile().getId().equals(profileId)) {
        throw new OperationNotAllowedException(Bet.class.getName(), betId.toString(),
                "only an admin or the creator has access");
    }
    return betCatalog.removeById(betId);
}
```

Snippet 23: Zugang attributbasiert

Um sicherzustellen, dass Nutzer nur ihre eigenen Wetten bearbeiten oder löschen dürfen, wird eine attributbasierte Kontrolle verwendet. Der „removeById“ Methode wird eine profileId übergeben. Dieser Wert kann aus dem JWT, welcher in der Resource injiziert wird, über den subject Claim ausgelesen werden. Jede abgegebene Wette enthält das zugehörige Profil, in welcher die entsprechende ID ebenfalls abgespeichert ist. Somit kann beim Aufruf dieser Methode geprüft werden, ob der aktuell zugreifende

Nutzer auch der Nutzer ist, der diese Wette ursprünglich erstellt hat. Ist dies nicht der Fall, wird der Zugriff verwehrt. Das gleiche Konzept wird ebenfalls beim Bearbeiten einer Wette verwendet. Um trotzdem weiterhin Admins den Zugriff auf alle Wetten zu gewähren, wird aus der SecurityIdentity die Admin Rolle geprüft. Ist die Admin Rolle vorhanden, wird der Zugriff unabhängig von der passenden profileId gewährt.
Ein Nutzer kann sich über ein Post Request mit Nutzername, Password, Vor- und Nachname über die UserResource registrieren. Zum Anlegen des Nutzers wird im UserRepository der Keycloak Admin Client verwendet. Durch das Registrieren wird im Quarkus Realm ein neuer Nutzer mit der User Rolle angelegt. Keycloak vergibt für den Nutzer eine ID, welche nach dem Registriervorgang verwendet wird, um unter der gleichen ID im Service ein Profil anzulegen, welches die Punkte des Nutzers verwaltet.

3.10 Test (Voß)

Die Tests der Anwendung wurden im Rahmen der Projektarbeit in Quarkus über REST-assured und über den API Client Insomnia durchgeführt.

3.10.1 REST-assured (Voß)

Um das Verhalten des Services zu testen, wurde REST-assured verwendet. Dabei werden Requests an die einzelnen Endpunkte geschickt und geprüft, ob die erwarteten Statuscodes erreicht werden. Des Weiteren werden die Zugangsbeschränkungen über die Nutzung verschiedener Tokens getestet.
Zur Ausführung der Tests wurde eine Hilfsklasse geschrieben, die Funktionalitäten bietet, um Tokens zu erstellen und GET-, POST-, PATCH- und DELETE-Requests auszuführen. Im Testmanager wird die Client ID und das zugehörige Passwort aus der „application.properties“ Konfigurationsdatei ausgelesen. Um die Tests mit verschiedenen Nutzern auszuführen, werden beim Erzeugen des TestManagers ein Konto für die Nutzerrolle User und ein Konto für die Nutzerrolle Admin über die UserFacade angelegt.

```
@Singleton
public class TestManager {
    @ConfigProperty(name = "quarkus.oidc.client-id")
    private String clientId;
    @ConfigProperty(name = "quarkus.oidc.credentials.secret")
    private String clientSecret;

    @Inject
    public TestManager(UserFacade userFacade) {
        this.userFacade = userFacade;

        // Test users
        this.userFacade.createUser(new UserCreationDTO("user", "user", "Test", "User"));
        this.userFacade.createAdmin(new UserCreationDTO("admin", "admin", "Test", "Admin"));
    }
```

Snippet 24: TestManager

Über die „getToken“ Methode wird ein Request an die Keycloak Authorization URL gesendet. Genutzt wird dafür der Quarkus Service Client über eine Basic Authorization, im Header wird der Content-Type auf „URL encoded Form data“ gesetzt. Im Body werden Nutzerdaten, der Grant Type (Password) und ein Scope (Open ID) angegeben werden. REST-assured ermöglicht es, die Antwort im JSON-Format nach bestimmten Feldern zu durchsuchen. [19] Darüber ist es möglich, den ID-Token der Antwort auszugeben.

```
private String getToken(String username, String password, String
grantType, String scope) {
    String idToken = given().auth().preemptive()
            .basic(clientId, clientSecret)
            .header("Content-Type", "application/x-www-form-urlencoded")
            .body(String.format("username=%s&password=%s&grant_type=
%s&scope=%s", username, password, grantType,
                    scope))
            .post(tokenUrl)
            .then().extract().response().jsonPath().getString("id_token");
    return idToken;
```

Snippet 25: getToken Methode

Um einen Endpunkt zu Testen kann eine Hilfsmethode verwendet werden. Hier kann ein Token angegeben werden, welcher im Authorization Header gesetzt wird. Der Request wird an die angegebene URL gesendet und anschließend der erwartete Statuscode überprüft.

```
public Response testGetRequest(String url, String token,
Integer expectedStatus) {
    ValidatableResponse res = given()
            .header("Authorization", "Bearer " + token)
            .when().get(url)
            .then()
            .statusCode(expectedStatus);
    return res.extract().response();
```

Snippet 26: Hilfsmethode Get Request

Der Test (Snippet 27: Test Get Endpunkt) testet den Endpunkt der TeamResource, der alle Teams zurückliefert. Da diese Methode nur Admins zur Verfügung steht, werden drei Requests mit jeweils unterschiedlichen Token (keine Anmeldung, normaler Nutzer, Admin) ausgeführt. Nur wenn alle Requests die erwarteten Statuscodes zurückliefern, ist der Test erfolgreich.

```
@Test
public void testGetEndpoint() {
    testManager.testGetRequest(teamsUrl, "",
401);
    testManager.testGetRequest(teamsUrl,
testManager.getUserToken(), 403);
    testManager.testGetRequest(teamsUrl,
testManager.getAdminToken(), 200);
}
```

Snippet 27: Test Get Endpunkt

3.10.2 Insomnia (Felschen)

Insomnia wurde parallel zur Entwicklung verwendet, um die umgesetzte Schnittstelle zu testen und vorhandene Fehler festzustellen. Die Endpunkte der Funktionen der einzelnen Resourcen wurden in Insomnia als Request angelegt. Gegebenenfalls wurden Daten als JSON-Body angegeben oder Authentifizierungen hinzugefügt.

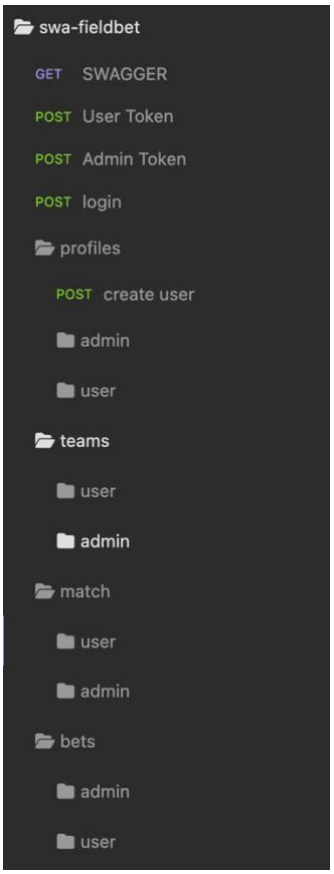


Abbildung 12: Auszug aus Insomnia

Damit die Funktionen in verschiedenen Rollen ausgeführt werden können, wurde eine übersichtliche Ordnerstruktur (siehe Abbildung 12: Auszug aus Insomnia) angelegt. Pro Resource existiert somit ein „user“-Ordner und ein „admin“-Ordner, welche die Rollen definieren.

Damit eine Authentifizierung erfolgt, werden Requests zum Anmelden miteinander verknüpft. Wird also eine Authentifizierung benötigt, wird erst ein Request gesendet, um einen Token zu erhalten und danach wird erst der eigentliche Request ausgeführt. Der resultierende Admin oder User-Token wird als Bearer Token für den eigentlichen Request angegeben.

4 Zusammenfassung und Fazit (Voß, Felschen)

Im Rahmen dieser Projektarbeit wurde eine Jakarta Enterprise Edition Anwendung unter Verwendung von Quarkus umgesetzt, welche typische Funktionalitäten eines Fußballtippspiels umsetzt. Der Nutzer kann sich über die Benutzungsoberfläche, welche über Qute realisiert wurde, registrieren und anschließend Wetten auf verschiedene Fußballspiele vornehmen. Einem Administrator ist es möglich, Spiele anzulegen und zu bearbeiten. Eine Rollenverteilung sowie die Nutzerdatenverwaltung wurde durch einen Keycloak Server umgesetzt. Beendet der Admin ein Spiel, wird Anhand des Ergebnisses eine automatische Punktevergabe ausgelöst. Zur Übersicht der Punkte existiert eine Bestenliste, in der die Nutzer nach Punktzahl sortiert aufgelistet sind.

Eine strukturiere Architektur der Anwendung wird durch eine Einteilung in Schichten und durch die Erstellung von verschiedenen Modulen garantiert. Aufgrund der SOLID Prinzipien ist die Anwendung wartbar und erweiterbar. Die Kommunikation über mehrere Module ist durch die Verwendung von Events, lose gekoppelt.

Keycloak wird eingesetzt, um die Anwendung gegenüber unautorisierten Zugriffen zu sichern, außerdem wurden die Nutzer in Rollen mit verschiedenen Rechten eingeteilt. Die Verwendung von Testcontainern schränkt die Konfigurationsmöglichkeiten allerdings etwas ein, wodurch die Verwendung eines EventListeners für Keycloak Events nicht realisiert werden konnte.

Um mögliche Fehler nachvollziehen zu können werden Funktionsaufrufe auf der Info-Ebene geloggt. Weiter geschützt ist die Anwendung durch die Verwendung von SmallRye Fault Tolerance. Auftretende Fehler werden über ein Exception-Mapping in geeigneter Form ausgegeben.

Durch die Nutzung von GitLab war die Zusammenarbeit an dem Projekt strukturiert umsetzbar. Da die Verwendung von Versionskontrollsystemen in der Softwareentwicklung eine wichtige Rolle spielt, bereitet diese Technologie vor allem auf zukünftige Projekte und Teamarbeit vor.

Im Laufe der Projektarbeit wurde sich mit der Softwarearchitektur und den aktuellen Technologien auseinandergesetzt. Es wurde erreicht, verschiedene Technologien anzuwenden und zusammen in einem Projekt einzubinden. Des Weiteren wurde das Arbeiten in einem Team gefördert. Somit wurde das Ziel, eine lauffähige Anwendung zu planen und zu entwickeln erreicht.

5 Literaturverzeichnis

[1] Quarkus, „Introduction to Contexts and Dependency Injection - Quarkus,“ [Online]. Available: https://quarkus.io/guides/cdi. [Zugriff am 09 02 2023].

[2] Eclipse Foundation, „Jakarta RESTful Web Services,“ [Online]. Available: https://jakarta.ee/specifications/restful-ws/3.0/jakarta-restful-ws-spec-3.0.html. [Zugriff am 09 02 2023].

[3] M. Fowler, „martinFowler,“ [Online]. Available: https://martinfowler.com/articles/richardsonMaturityModel.html. [Zugriff am 09 02 2023].

[4] PostgreSQL Global Development Group, „PostgreSQL,“ [Online]. Available: https://www.postgresql.org/about/. [Zugriff am 08 02 2023].

[5] Oracle, „Java Documentation - Java JDBC API,“ [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/. [Zugriff am 08 02 2023].

[6] Eclipse Foundation, „Managing Entities,“ [Online]. Available: https://eclipse-ee4j.github.io/jakartaee-tutorial/#managing-entities. [Zugriff am 09 02 2023].

[7] Eclipse Foundation, „Positional Parameters in Queries,“ [Online]. Available: https://eclipse-ee4j.github.io/jakartaee-tutorial/#positional-parameters-in-queries. [Zugriff am 09 02 2023].

[8] Eclipse Foundation, „What Is a Transaction?,“ [Online]. Available: https://eclipse-ee4j.github.io/jakartaee-tutorial/#what-is-a-transaction. [Zugriff am 09 02 2023].

[9] SmallRye, „Fault Tolerance Documentation,“ [Online]. Available: https://smallrye.io/docs/smallrye-fault-tolerance/5.2.0/index.html. [Zugriff am 09 02 2023].

[10] Quarkus, „Smallrye Metrics - Quarkus,“ [Online]. Available: https://quarkus.io/guides/smallrye-metrics. [Zugriff am 09 02 2023].

[11] Quarkus, „Quarkus - Configure Logging,“ [Online]. Available: https://quarkus.io/guides/logging. [Zugriff am 09 02 2023].

[12] Quarkus, „Qute Template Engine,“ [Online]. Available: https://quarkus.io/guides/qute. [Zugriff am 10 02 2023].

[13] Bootstrap, „Get started with Bootstrap,“ [Online]. Available: https://getbootstrap.com/docs/5.3/getting-started/introduction/. [Zugriff am 10 02 2023].

[14] OpenID, „OpenID Connect,“ [Online]. Available: https://openid.net/connect/. [Zugriff am 10 02 2023].

[15] Keycloak, „Keycloak Server Administration Guide,“ [Online]. Available: https://www.keycloak.org/docs/latest/server_admin/. [Zugriff am 10 02 2023].

[16] Okta, „Introduction to JSON Web Tokens,“ [Online]. Available: https://jwt.io/introduction. [Zugriff am 10 02 2023].

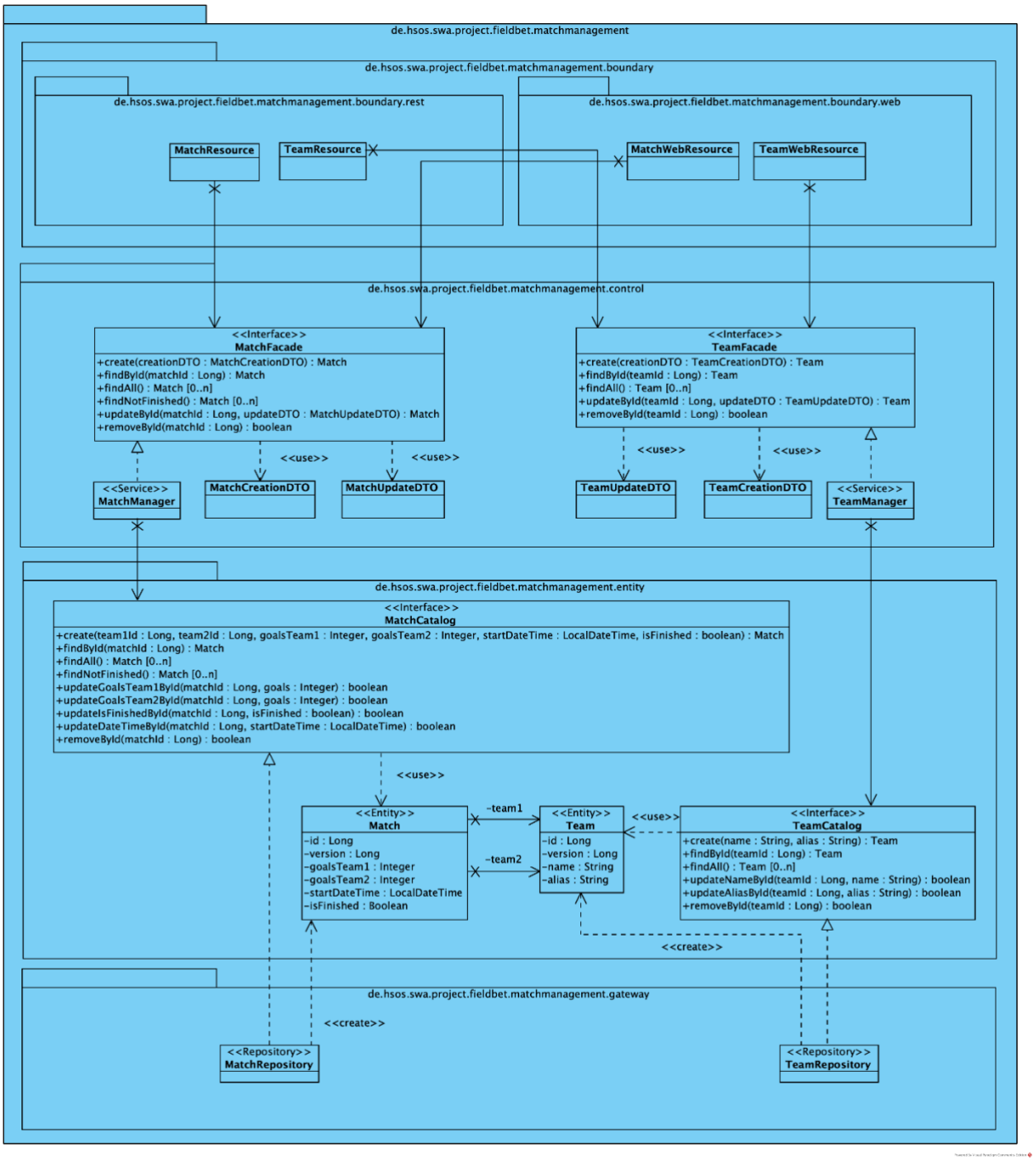
[17] Quarkus, „Using OpenID Connect (OIDC) to Protect Service Applications using Bearer Token Authorization,“ [Online]. Available: https://quarkus.io/guides/security-openid-connect. [Zugriff am 10 02 2023].

[18] Quarkus, „OpenID Connect (OIDC) authorization code flow mechanism,“ [Online]. Available: https://quarkus.io/guides/security-openid-connect-web-authentication. [Zugriff am 11 02 2023].

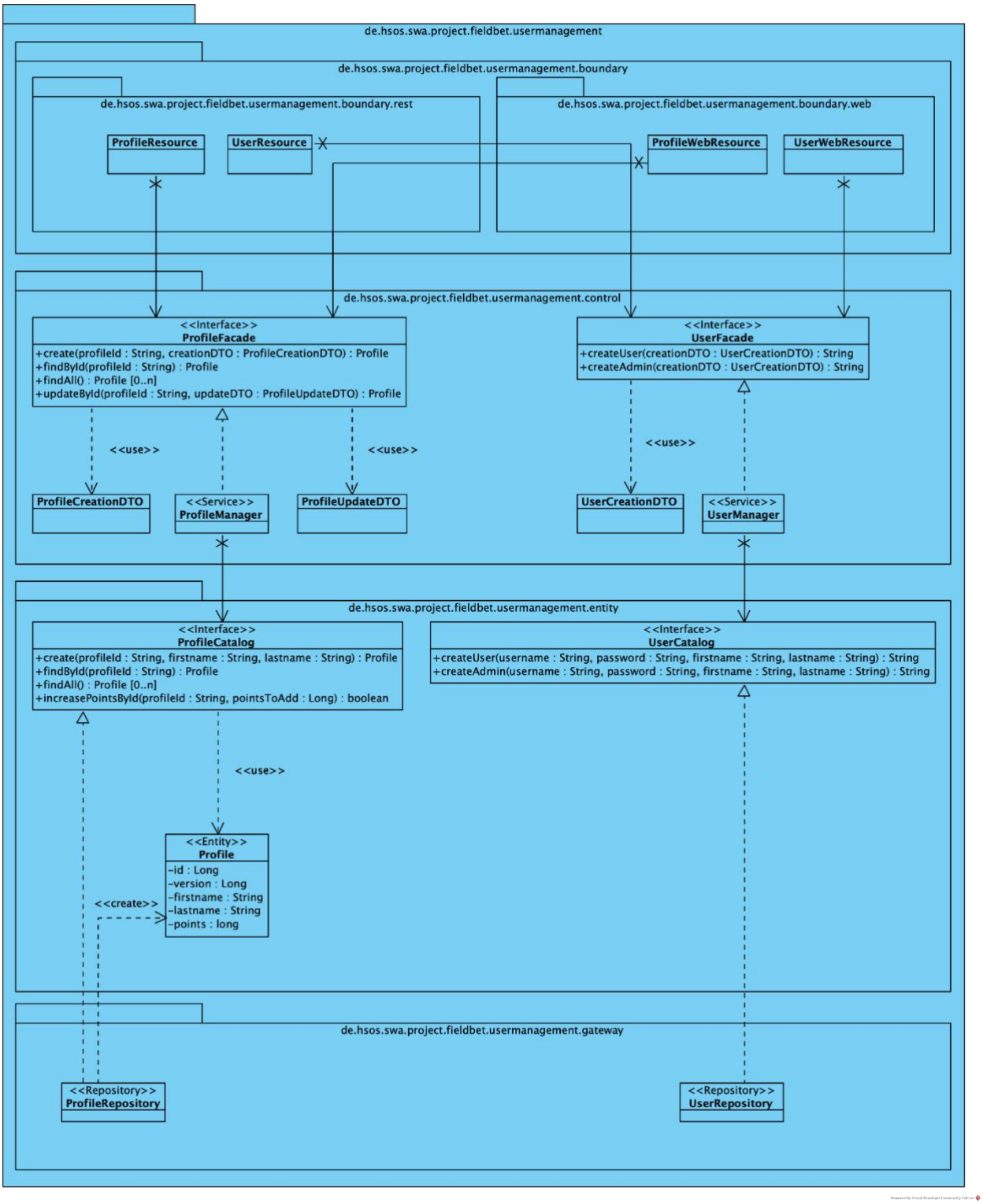
[19] RestAssured, „RestAssured - Wiki,“ [Online]. Available: https://github.com/rest-assured/rest-assured/wiki/Usage. [Zugriff am 11 02 2023].

6 Anhang

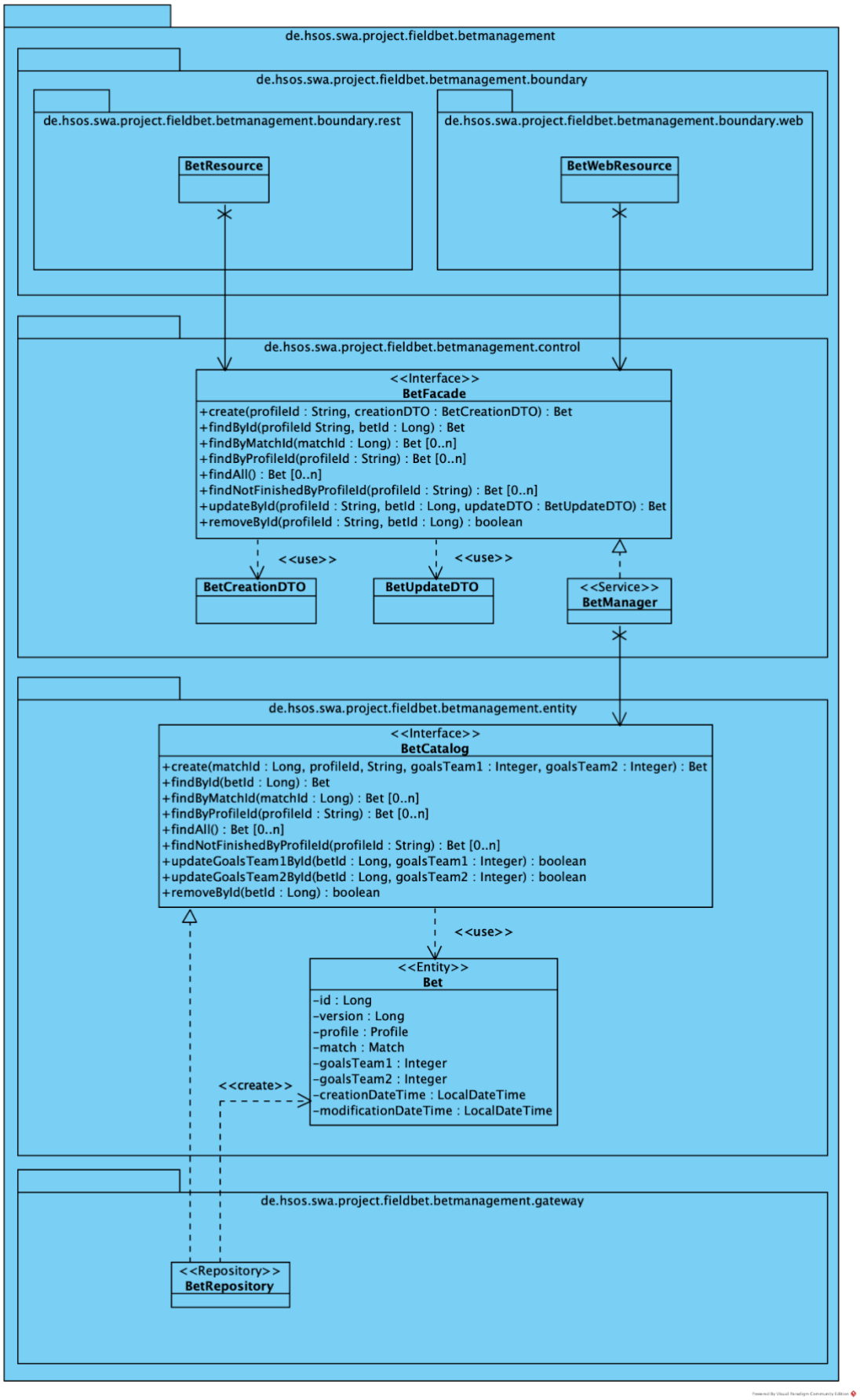
6.1 Anhang 1: Match Management Modul (Felschen)



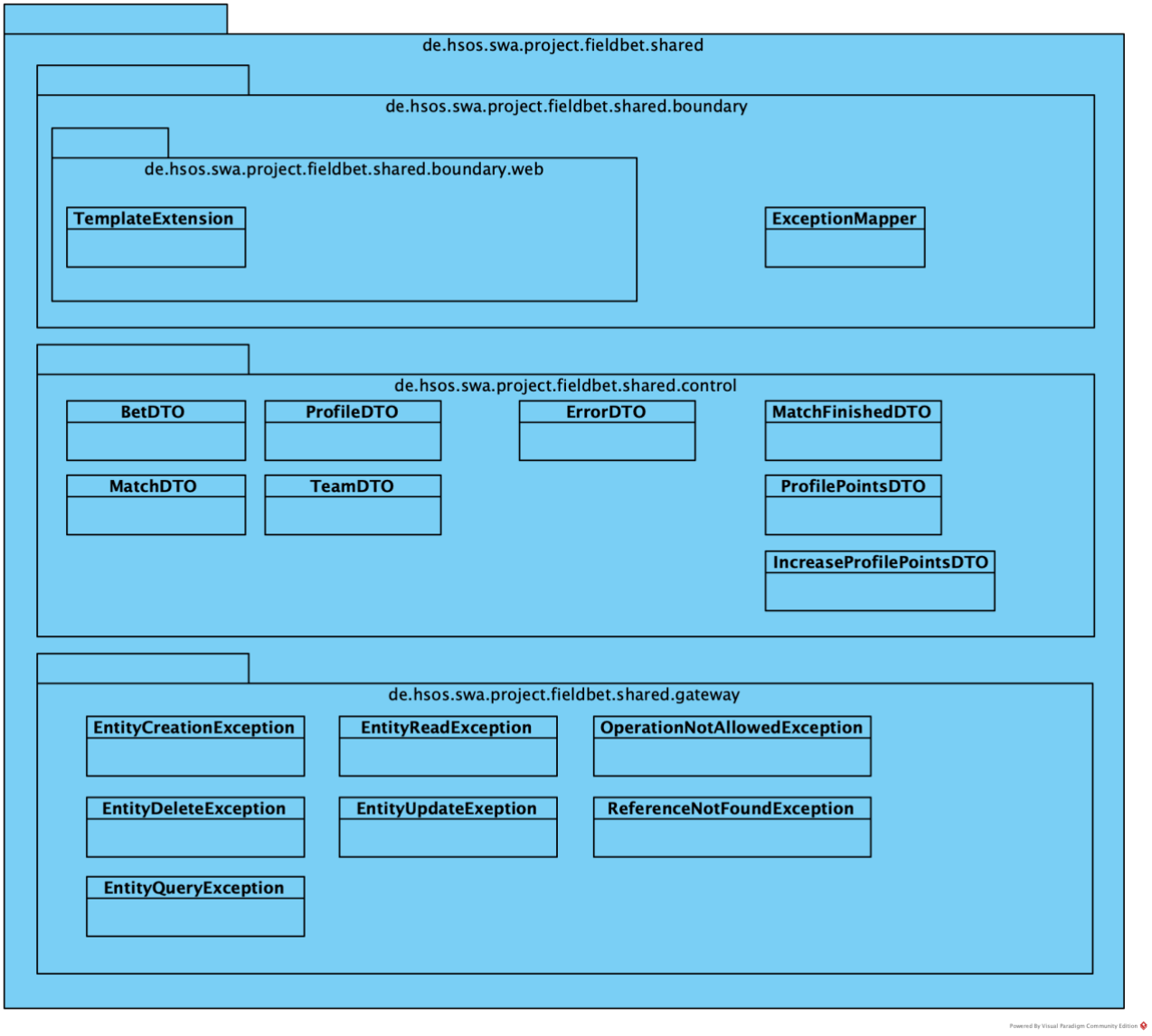
6.2 Anhang 2: User Management Modul (Felschen)



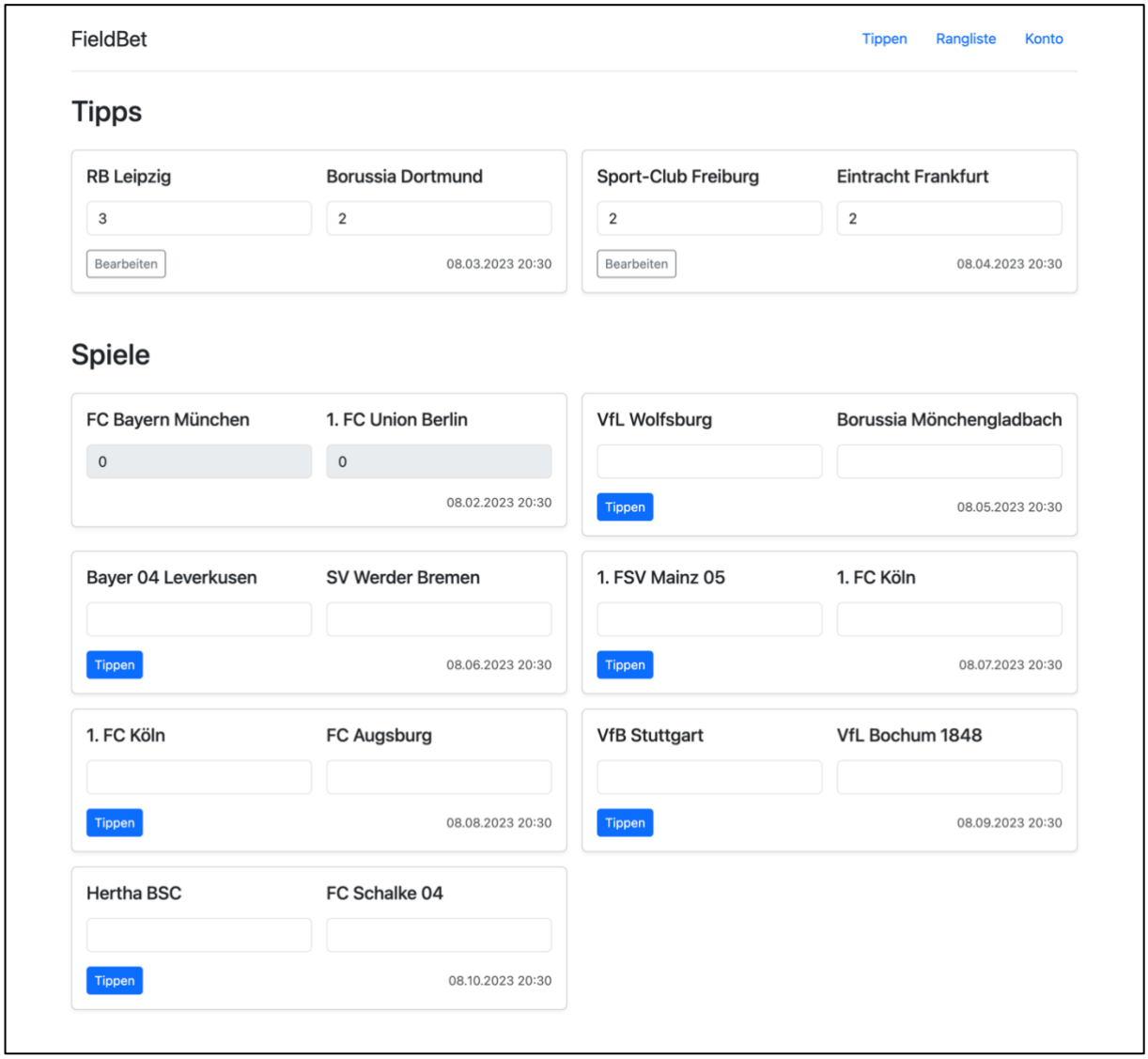
6.3 Anhang 3: Bet Management Modul (Felschen)



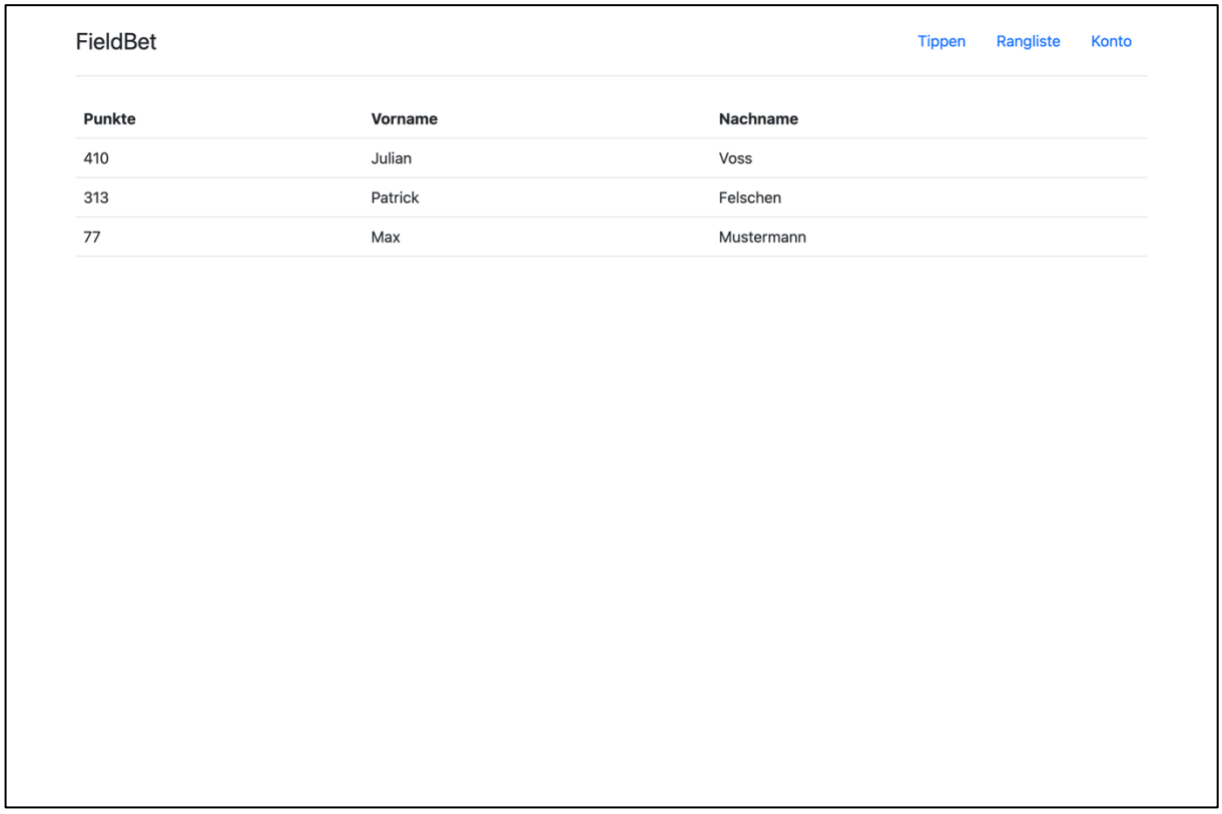
6.4 Anhang 4: Shared Modul (Felschen)



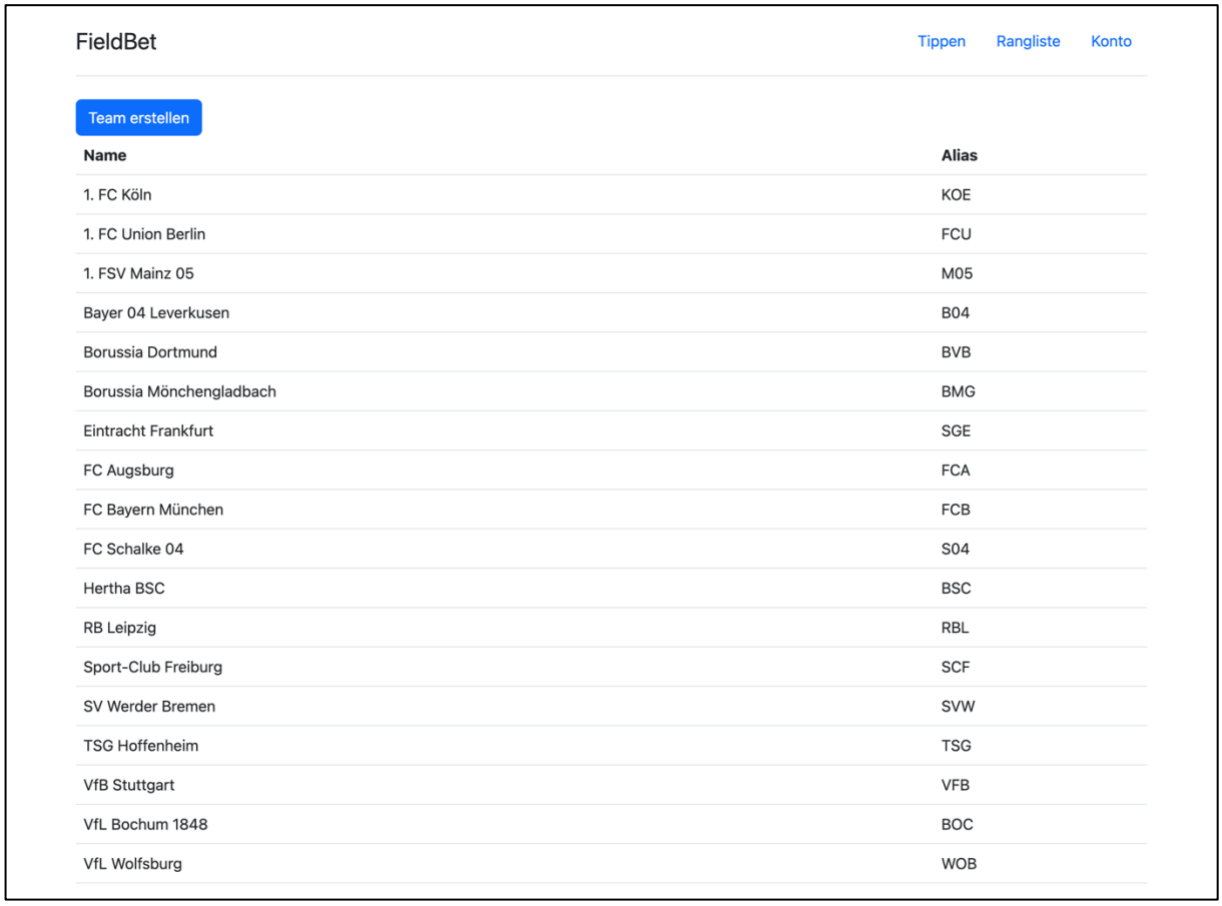
6.5 Anhang 5: UI – Tippen (Felschen)



6.6 Anhang 6: UI – Rangliste (Felschen)

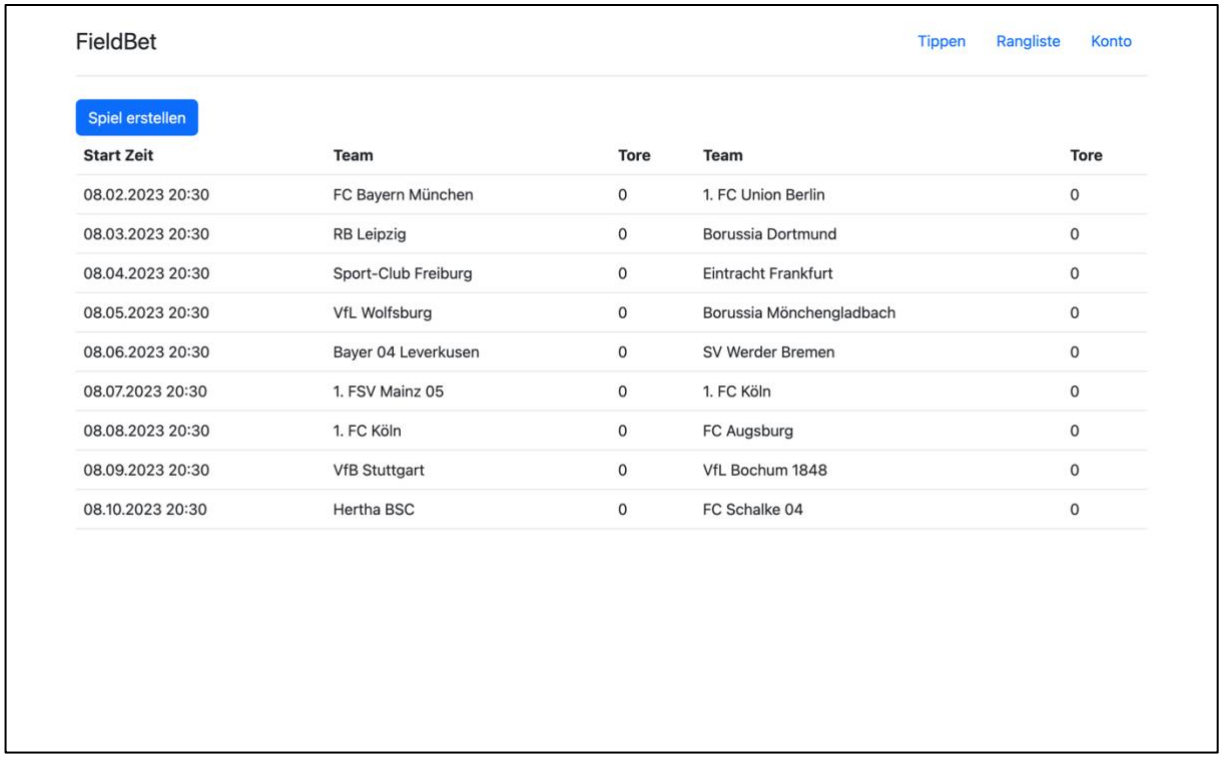


6.7 Anhang 7: UI – Teams (Felschen)



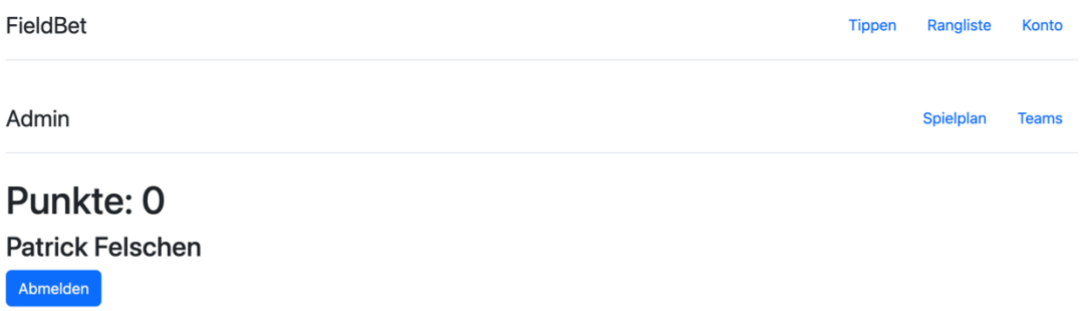
Name	Alias
1. FC Köln	KOE
1. FC Union Berlin	FCU
1. FSV Mainz 05	M05
Bayer 04 Leverkusen	B04
Borussia Dortmund	BVB
Borussia Mönchengladbach	BMG
Eintracht Frankfurt	SGE
FC Augsburg	FCA
FC Bayern München	FCB
FC Schalke 04	S04
Hertha BSC	BSC
RB Leipzig	RBL
Sport-Club Freiburg	SCF
SV Werder Bremen	SVW
TSG Hoffenheim	TSG
VfB Stuttgart	VFB
VfL Bochum 1848	BOC
VfL Wolfsburg	WOB

6.8 Anhang 8: UI – Matches



Start Zeit	Team	Tore	Team	Tore
08.02.2023 20:30	FC Bayern München	0	1. FC Union Berlin	0
08.03.2023 20:30	RB Leipzig	0	Borussia Dortmund	0
08.04.2023 20:30	Sport-Club Freiburg	0	Eintracht Frankfurt	0
08.05.2023 20:30	VfL Wolfsburg	0	Borussia Mönchengladbach	0
08.06.2023 20:30	Bayer 04 Leverkusen	0	SV Werder Bremen	0
08.07.2023 20:30	1. FSV Mainz 05	0	1. FC Köln	0
08.08.2023 20:30	1. FC Köln	0	FC Augsburg	0
08.09.2023 20:30	VfB Stuttgart	0	VfL Bochum 1848	0
08.10.2023 20:30	Hertha BSC	0	FC Schalke 04	0

6.9 Anhang 9: UI – Konto (Felschen)



Eidesstattliche Erklärung

Hiermit erklären wir an Eides statt, dass wir die vorliegende Arbeit selbständig und ohne fremde Hilfe angefertigt haben. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche einzeln kenntlich gemacht. Es wurden keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Osnabrück, 16.02.2023 Felschen.....
Ort, Datum Unterschrift

Osnabrück, 16.02.2023 .....
Ort, Datum Unterschrift

Urheberrechtliche Einwilligungserklärung

Hiermit erklären wir, dass wir damit einverstanden sind, dass unsere Arbeit zum Zwecke des Plagiatsschutzes bei der Fa. Ephorus BV bis zu 5 Jahren in einer Datenbank für die Hochschule Osnabrück archiviert werden kann. Diese Einwilligung kann jederzeit widerrufen werden.

Osnabrück, 16.02.2023 Felschen.....
Ort, Datum Unterschrift

Osnabrück, 16.02.2023 .....
Ort, Datum Unterschrift