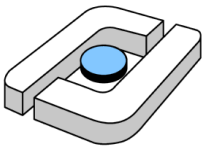

Hochschule Osnabrück

University of Applied Sciences

Fakultät
Ingenieurwissenschaften und Informatik

Schriftliche Ausarbeitung zum Thema:

Simple Chess

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

Autor:	Joe Kramer
Matr.-Nr.:	915505
E-Mail:	joe.kramer@hs-osnabrueck.de
Themensteller:	Prof. Dr. Rainer Roosmann

Abgabedatum: 17.02.2023

Inhaltsverzeichnis

Abbildungsverzeichnis

Source-Code Verzeichnis

Abkürzungsverzeichnis

CDI	Context and Dependency Injection for the Java EE Plattform
ECB	Entity-Controller-Boundary Pattern
EJB	Enterprise Java Beans
Java EE	Java Enterprise Edition, in der Version 7
JSF	Java Server Faces
SWA	Software-Architektur
SFLB	Statefull Session Bean
SLSB	Stateless-Session Bean
JWT	JSON-Web-Token
SSE	Server Sent Events
RBAC	Role-Based Access Control
ORM	Object Relational Mapper
DBMS	Datenbankmanagementsystem

1 Einleitung

Schach ist ein altes Spiel, das schon im Mittelalter gespielt wurde. Auch heute ist es mit diversen Schachvereinen, Turnieren und Meisterschaften ein Sport mit großer Beliebtheit. In dieser Projektarbeit wird eine Anwendung entwickelt, die genau dieses Spiel umsetzt. Die Anwendung mit Java und Quarkus entwickelt. Über eine bereitgestellte Rest-Schnittstelle soll die Anwendung am Ende bedient werden können.

1.1 Ziel der Ausarbeitung

Die Anwendung wird mit dem Ziel entwickelt möglichst unkompliziert Schachpartien erstellen und abhalten zu können. Dazu sollen beim erstellen von Partien Token erzeugt werden, mit denen Spieler dem Spiel beitreten können. Es soll außerdem möglich sein das auch Zuschauer mit einem Token das Spiel verfolgen können. Sobald beide Spieler einem Spiel beigetreten sind, können sie die Partei starten. Dabei wird zuerst in einem Minigame entschieden welcher Spieler weiß Spielen, also anfangen darf. Ein Spiel ist dann zu ende, wenn ein König geschmissen werden kann oder einer der Spieler aufgibt. Die Partien werden dabei für die Dauer einer Partie, maximal jedoch einen Monat in der Datenbank gespeichert. Um reaktiv die Aktionen des jeweils anderen Spielers mitzubekommen, werden Server-Sent-Events eingesetzt.

1.2 Aufbau der Hausarbeit

Im folgenden Kapitel werden zuerst die verwendeten Bibliotheken, Frameworks und allgemeine Grundlagen vermittelt. Darauffolgend wird die entwickelte Anwendung im Detail vorgestellt. Dabei wird erklärt wie es zu der verwendeten SWA kam und welche Aufgaben die einzelnen Teilbereiche der Anwendung haben. Zum Schluss wird das Projekt nochmal kurz zusammengefasst und ein Fazit gezogen.

2 Darstellung der Grundlagen

Zunächst werden einige der wichtigsten, verwendeten Technologien genauer erläutert. Die einzelnen Abschnitte beziehen sich im Allgemeinen auf die in der Anwendung verwendeten Frameworks und Bibliotheken, die Datenbank, sowie auf Softwarearchitektur Patterns.

2.1 REST-API

Die REST-API oder auch RESTful-API genannt, ist im Gegensatz zu vielen Anderen APIs mit jeder Programmiersprache umsetzbar. Sie stellt nämlich nur Anforderungen an den Grundsätzlichen Design der API. Dazu zählen z.B. einheitliche Schnittstellen, die Entkopplung von Client und Server und die Verwendung einer Mehrschichtigen Systemarchitektur. Des Weiteren kann noch festgehalten werden das über HTTP-Requests kommuniziert wird und Daten häufig im beliebten JSON Format ausgetauscht werden (vgl. [@IBM]).

2.2 Quarkus

Bei Quarkus handelt es sich um ein Java Framework. „Quarkus wurde speziell für beliebte Java-Standards, Frameworks und Libraries wie Eclipse MicroProfile, […] RESTeasy (JAX-RS), Hibernate ORM (JPA) […] und viele mehr konzipiert“ [@RED]. Das Hauptaugenmerk von Quarkus liegt darauf, eine gute Entwicklungserfahrung zu bieten. Dies schafft Quarkus, indem Java Programme effizient in Containern ausgeführt werden, was wiederum zu einer schnellen Kompilierbarkeit führt. Die Lesbarkeit und Komplexität von Programmen profitiert außerdem von CDI. Dabei handelt es sich um eine komfortable Möglichkeit Programme um Funktionalitäten zu erweitern ohne sich dabei um die weiter um die Dependencys kümmern zu müssen.

2.3 Entity-Control-Boundary Pattern (ECB)

Das Entity-Control-Boundary Pattern ist ein Architekturpattern. Es sieht vor das die Anwendung in drei Schichten eingeteilt wird. Die erste ist die Boundary Schicht. Sie stellen ermöglicht den Zugang zur Anwendung und zu den verwalteten Ressourcen. Sie sollte ausschließlich mit der Control Schicht kommunizieren. In der Control Schicht wird die Programmlogik definiert und Funktionalitäten realisiert. Über sie werden der Boundary die angeforderten Ressourcen übermittelt. Daten und Ressourcen werden durch die Entitäten in der Entity Schicht gespeichert. Durch diese Unterteilung wird die Verwaltung von Abhängigkeiten erleichtert und die Verantwortlichkeiten innerhalb der Anwendung getrennt.

2.4 Smallrye Fault Tolerance & Metrics

Smallrye ist eine Implementierung von MicroProfile, einer Standardisierung von Microservice-Technologien für Java EE. Ein Teil dieser Implementierung ist u.a. Fault Tolerance und Metrics. Fault Tolerance ermöglicht über Annotationen, Funktionalitäten bereitzustellen die, die Resilienz der Anwendung gewährleisten können. Mit Metrics wird die Anwendung um Funktionen erweitert, die eine einfache Überwachung und Protokollierung des Zustandes und der Leistung des Microservices ermöglicht.

2.5 Rest-Assured

Hierbei handelt es sich um eine Bibliothek, mit der sich REST-Web-Services automatisiert testen lassen. Mit ihr lassen sich bereitgestellte Endpoints einfach testen. Es ermöglicht die Formulierung von Anfragen mit URL-Parametern und unterstützt die Auswertung von Antwortinhalten und HTTP-Statuscodes.

2.6 Server-Sent-Events (SSE)

Server-Sent-Events sind Teil des HTML5-Standards. Sie ermöglichen es dem Server, Daten an Clients zu senden. Dazu müssen die Clients initial eine Verbindung zum Server aufbauen. Diese Verbindung wird nun einfach offengehalten und kann vom Server zum Verschicken von Nachrichten genutzt werden. Es ist so möglich mit den Clients separat

zu kommunizieren oder aber auch mit mehrere gleichzeitig und somit einen Broadcast zu realisieren.

2.7 Resteasy-Reactive

Dies ist eine Bibliothek die auf die einfache Erstellung von REST APIs ausgelegt ist. Sie stellt zahlreiche Annotationen bereit, die vor allem in der Boundary Verwendung finden werden. Die reaktive Version dieser Bibliothek, ermöglicht es nicht blockierende Endpoints zu realisieren. Diese bekommen andere Threads zugeordnet und somit können anfragen effizienter bearbeitet werden. Es ist jedoch nicht möglich alle Enpoints reaktiv zu schreiben. Werden z.B. Ressourcen von einer Datenbank angefordert, handelt es sich in der Regel um einen Blockierenden Zugriff und somit ist auch der Endpoint blockierend.

2.8 JSON Binding (Jsonb) -Reactive

Um Java Objekte in der Boundary entgegennehmen und zurückgeben zu können, werden diese in JSON-Objekte formatiert. Diese Formatierung übernimmt Jsonb. Objekte müssen zur automatischen Erkennung einige Voraussetzungen erfüllen. Es müssen zum Beispiel die Variablen frei zugänglich sein, also entweder mit public deklariert oder es müssen entsprechende Getter und Setter vorhanden sein. Alternativ kann das Mapping auch manuell in einem Adapter festgelegt werden.

2.9 JSON-Web-Token (JWT)

JSON-Web-Tokens sind ein Standardformat für die Übertragung von Informationen. Es handelt sich im Grunde um JSON-Objekte, die zusätzlich zu den enthaltenen Informationen um eine Signatur erweitert wurden. Anhand dieser Signatur kann mit dem Public-/Private Key Verfahren geprüft werden, ob Daten des Objektes manipuliert wurden. Um JWT zu entschlüsseln und neu generierte zu signieren, werden Lösungen von Smallrye verwendet.

2.10 Role-Based Access Control (RBAC)

Role-Based Access Control ist ein Zugriffskontrollmodell, bei dem Zugriff auf Ressourcen basierend auf der Rolle eines Benutzers vergeben werden. Diese Rollen werden den Nutzern einzeln zugewiesen. RBAC erleichtert die Verwaltung von Benutzerberechtigungen und kann die Sicherheit von Systemen und Daten erhöhen. Die Rollen werden i.d.R. in einem JWT gespeichert und beim Autorisieren an einem geschützten Endpoint abgefragt.

2.11 Quarkus-Schedule

Die Realisierung von zeitgesteuerten Aufgaben kann mithilfe der Quarkus Erweiterung, Quarkus-Schedule durchgeführt werden. Funktionen die zeitgesteuert aufgerufen werden sollen, müssen lediglich mit einer Annotation gekennzeichnet werden. Mit dieser können dann die genauen Parameter eingestellt werden. Genutzt werden kann dies z.B. für die Überprüfung und Aktualisierung von alten Datenbankeinträgen.

2.12 Hibernate Object Relational Mapper (Hibernate ORM)

Hibernate ORM ist eine Bibliothek zur Verwaltung relationaler Datenbanken in Java Anwendungen. Es bietet eine Reihe von Funktionen, die dabei helfen Datenbanken in Anwendungen abstrakt einzubinden, ohne dabei spezifische Anfragen zu formulieren. Eine Vielzahl von Datenbanken wird unterstützt, wie MySQL, PostgreSQL, Oracle oder auch Microsoft SQL Server. Ein nachträglicher Wechsel der Datenbank ist unproblematisch und kann i.d.R. ohne Änderungen am Code erfolgen.

2.13 PostgreSQL

Bei PostgreSQL handelt es sich um eine objektrelationales Datenbankmanagementsystem, das Open-Source und kostenlos ist. Es wird weitestgehend der SQL-Standard unterstützt und darüber hinaus werden weitere Features geboten. Das DBMS ist auf verschiedenen Plattformen lauffähig, und kann auf spezielle Anforderungen angepasst werden. Die Community hinter PostgreSQL sorgt ständig für die Weiterentwicklung und Verbesserung der Sicherheitsfeatures.

2.14 @Application- & @Request-Scoped

Hierbei handelt es sich um Annotationen, mit deren Hilfe der Lebenszyklus und Kontext von Klassen definiert wird. @ApplicationScoped sorgt dafür, dass die Klasse nur einmal instanziiert wird und danach stets mit dieser gearbeitet wird. Sie ist für die gesamte Dauer der Anwendung verfügbar. Mit @ReqestScoped markierte Klassen sind hingegen immer nur für die Dauer eines HTTP-Requests verfügbar. Sie werden für jede weiter Anfrage neu instanziiert. Neben diesen beiden Annotationen gibt es noch weitere, die aber in diesem Projekt nicht verwendet werden.

3 Anwendung

In diesem Kapitel wird die entwickelte Software im Detail vorgestellt. Zunächst wird erläutert welche Ziele beim Designen der SWA im Vordergrund standen und welche Patterns letztlich umgesetzt wurden. Dann wird der grobe Aufbau der Anwendung dargestellt. Genauer werden darauffolgend die einzelnen, zusammenhängenden Softwarestrukturen erläutert.

3.1 Softwarearchitektur

Bei der Entwicklung der Softwarearchitektur gab es einige Punkte die Berücksichtigt wurden. Zum einen sollte die spätere Software recht einfach und auf das Notwendigste beschränkt sein. Es sollen also z.B. keine Nutzer gespeichert werden oder Schachpartie Verläufe. Somit wird die Datenbank entlastet und die Rest-Schnittstellen übersichtlicher.

Ein weiteres Ziel war es die Erweiterbarkeit der Anwendung zu ermöglichen. Dies wurde z.B. bei der Autorisierung berücksichtigt, die durch JWT und RBAC erfolgt. Diese werden zunächst durch die Anwendung generiert und signiert, könnten aber später, wenn eine umfangreichere Nutzerverwaltung erforderlich ist, durch Tools wie Keycloak verwaltet werden. Weitere mögliche Erweiterungen wären auch das hinzufügen von abgeänderten Turnierschachregeln oder ein Wechsel der Datenbank. Ersteres ist gemäß des Open Closed Principles durch die Verwendung des Entity-Control-Boundary Patterns möglich. Es müssten in diesem Fall lediglich weitere entsprechende Rest-Schnittstellen und eine weitere Implementation des zuständigen Services hinzugefügt werden, in dem dann die Regeln festgehalten werden.

Der letzte Punkt auf den besonders Wert gelegt wurde, ist die Reaktivität des Systems. Damit ist vor allem gemeint, dass Aktionen, die von einem Spieler getätigt werden, möglichst schnell allen Spielern und Zuschauern angezeigt werden sollen. Es ist anzumerken das Reaktivität in diesem Zusammenhang keine allgemein gültige Definition besitzt. Quarkus definiert z.B. die Reaktivität mit Vier Eigenschaften. Eine reaktive Anwendung soll demnach, zeitnah auf Anfragen reagieren, sich gemäß der Auslastung anpassen, resilient sein und mithilfe von asynchronen Nachrichten kommunizieren. [QA01]

Um also möglichst schnell den Spielern und Zuschauern Änderungen mitzuteilen, ist es Notwendig, dass der Server selbständig Daten an Clients senden kann. Hierzu wird auf Server-Sent-Events gesetzt. Die Clients müssen dann nicht jedes Mal eine Anfrage an

den Server senden, wenn sie eine Änderung an einem Objekt vermuten, sondern registrieren sich einmalig und erhalten dann automatisch die aktualisierten Objekte.

3.2 Anwendungsaufbau

Die Anwendung ist nach dem ECB-Pattern, mit einer zusätzlichen Gateway Schicht aufgebaut. Die einzelnen Resourcen in der Boundary Schicht sind nach dem Dependency Inversion Principle, jeweils über ein Interface mit den jeweiligen Controllern verknüpft. Des Weiteren befinden sich noch die DTOs in dieser Schicht, da sie hier entweder Empfangen oder zurückgegeben werden. In der Control Ebene gibt es neben den Interfaces und Controllern noch eine Klasse, den GameValidator. Seine Funktion wird später erklärt. Um die Datenverwaltung kümmert sich die Gateway-Schicht. Die in ihr vorhanden Repositorys sind dabei entweder mit Entitäten und der Datenbank verbunden oder verwalten Ihre Daten in eigens organisierten Datenstrukturen. Bei den Entitys handelt es sich teilweise um Objekte, die von der Datenbank genutzt werden und teilweise um Objekte, die von den Repositorys direkt verwaltet werden. Neben diesen Schichten existieren noch das Shared-Package. In diesem wurden diejenigen Klassen platziert, die an verschiedensten Stellen innerhalb des Programms benötigt werden.

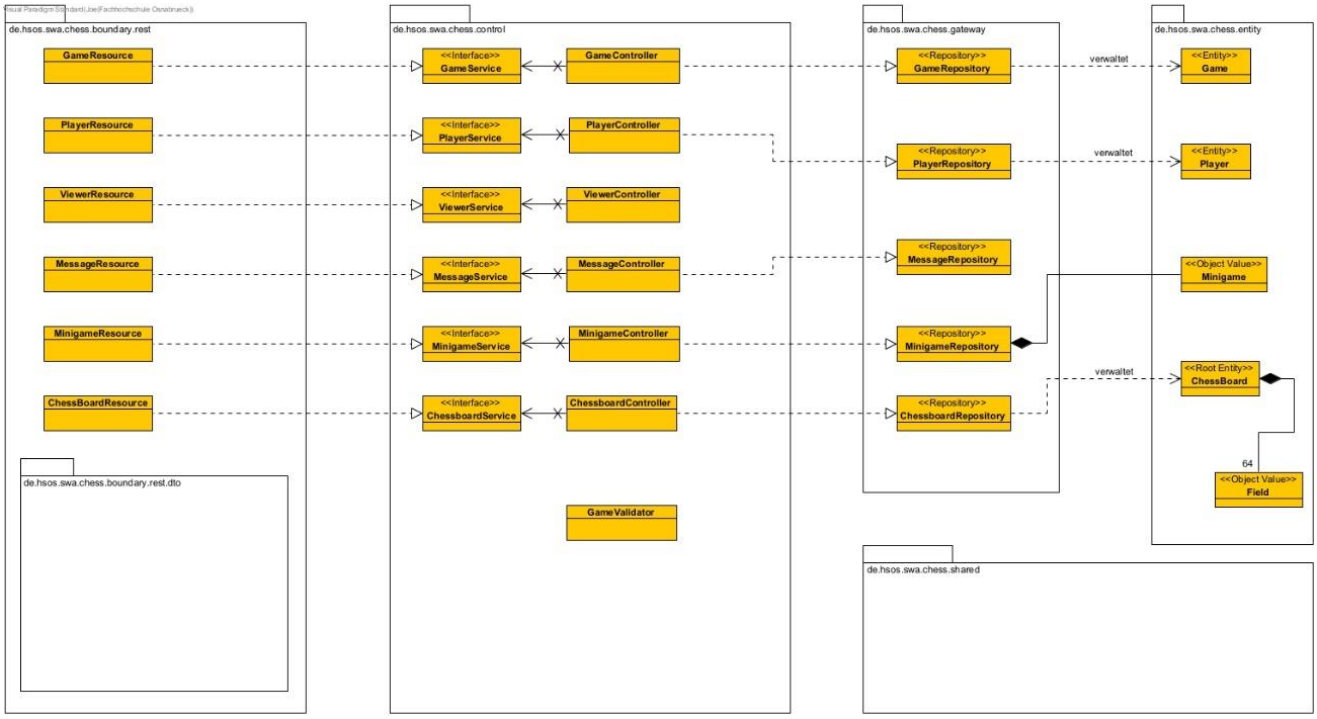

Abbildung 1: Klassendiagramm

3.3 Boundary

Der Zugriffspunkt auf die Anwendung ist die Boundary. Die Boundary besitzt den Zweck eingehende Anfragen initial zu verarbeiten und Responses zu bilden, die ggf. um Objekte ergänzt werden. Bei den Event-Endpoints gibt es leider nicht die Möglichkeit Responses zurückzugeben. Hier wird nach dem Schließen der Verbindung, automatisch der http-Code 200 zurückgegeben. Einzig beim Unautorisierten Aufruf einer solchen Rest-Schnittstelle werden zudem noch die http-Codes 401 oder 403 ausgegeben. Zur Autorisierung wird RBAC genutzt. Geschützte Endpoints werden dazu einfach mit @RolesAllowed versehen. Neben dieser Annotation gibt es in diesen Klassen weitere wichtige. Zu-

```
@GET
@RolesAllowed("player")
@Operation(
        summary = "Spiel abfragen",
        description = "Fragt das dir zugeordnete Spiel ab"
)
@APIResponse(
        responseCode = "201",
        description = "Spiel abgefragt",
        content = @Content(mediaType = MediaType.APPLICATION_JSON)
)
@APIResponse(
        responseCode = "204",
        description = "Spiel konnte nicht abgefragt werden"
)
public RestResponse<GameResponseDTO> getYourGame(@Context JsonWebToken jwt) {
    Long gameId = Long.valueOf(jwt.getClaim("gameId").toString());
    Optional<GameResponseDTO> game = gameService.findById(gameId);
    if(game.isEmpty()) {
        return RestResponse.status(RestResponse.Status.NO_CONTENT);
    }
    return RestResponse.status(RestResponse.Status.OK, game.get());
}
```

Snippet 1: GameResource

erst einmal wird die @Path Annotation benötigt, um kenntlich zu machen, dass es sich bei der Klasse um eine extern aufrufbare Schnittstelle handelt, die über den eingestellten Pfad erreichbar ist. Für einzelnen Zugriffspunkte ist es so ebenfalls möglichen einen spezielleren Pfad zu nutzten, zwingend notwendig ist aber vor allem die entsprechende HTTP-Verb Annotation. Mit @Produces und @Consumes muss des Weiteren noch angegeben werden welcher Parametertyp bei einem Post ggf. erwartet wird und in welcher Form Antworten zurückgegeben werden sollen. Neben dem Post-Parameter kommen

noch Pfad-Parameter zum Einsatz. Sie werden mit @RestPath angegeben. Handelt es sich um einen Event-Endpoint, ist zwangsläufig noch mit @Context das Objekt zu deklarieren, über das Nachrichten an den Client gesendet werden können. Werden Informationen aus dem JWT benötigt, kann dieser mit der gleichen Annotation zur Verfügung gestellt werden.

3.4 Controller

Innerhalb der Control-Ebene findet der Großteil der Anwendungslogik statt. Auf die eingehenden Anfragen der Boundary werden hier die Antworten gebildet. Die Informationen, die zum Bilden dieser Antworten benötigt werden, stammen i.d.R. von den Injizierten Repositorys. Eine Ausnahme ist aber z.B. die Lebensdauer der ausgestellten JWT beim Hinzufügen eines Spielers im PlayerController. Diese wird mithilfe von @ConfigProperty aus der „application.properties“ ausgelesen. Das ist Sinnvoll damit diese Konstanten nicht verteilt in der Anwendung festgelegt werden, sondern zentral an einem Punkt. Alle Controller sind @RequestScoped, da in ihnen keine Daten gespeichert werden. Die Rückgabe von Objekten an die Boundary erfolgt häufig als Optional-Objekt. Hingegen erhalten die Controller keine Optionals aus der Gateway-Schicht. Das liegt daran dass, in den Controllern meist recht viele Anfragen an die Repositorys gerichtet werden. Würden dann jedes Mal Optional-Objekte geliefert werden, würde der Code unübersichtlicher, weil die null Prüfung länger ausfällt und der Inhalt jedes Mal mit „get“ entnommen werden müsste.

```
@Inject
@ConfigProperty(name = "jointoken_length", defaultValue = "5")
Integer joinTokenLength;
```

Snippet 2: ConfigProperty

3.4.1 GameValidator

Die Klasse ist dafür verantwortlich, alte Inhalten aus den Datenstrukturen zu entfernen. Es handelt es sich um eine der wenigen mit @ApplicationScoped annotierten Klassen. Dies ist hier erforderlich, weil Quarkus-Scheduling zum Einsatz kommt. Es gibt zwei Methoden die mit @Scheduled markiert sind. Beide beziehen Ihr Aufrufin-

```
@Blocking
@Scheduled(cron="{cron.expr}")
public void deleteOldGames() {
    Set<Long> oldIds = gameRepository.deleteOldGames();
    deleteOldMinigames(oldIds);
}
```

Snippet 3: deleteOldGames()

tervall aus der „application.properties“. Mit „deleteOldGames“ wird im GameRepository die entsprechende Methode aufgerufen. Dabei werden die IDs der gelöschten Spiele gespeichert und an „deleteOldMinigames“ weitergegeben, die sich dann darum kümmert die Minigames der Spiele zu löschen, sofern noch vorhanden. Die zweite Methode trägt die IDs bereits gelöschter Spieler zusammen und sorgt dann für deren Entfernung aus dem MessageRepository.

3.5 Gateway

Die Repositorys können in dieser Anwendung grundsätzlich in zwei Kategorien eingeteilt werden. Die einen verwenden eigens erstellte Strukturen, um die namensgebenden Ressourcen zu verwalten und die anderen nutzen die dazu die Datenbank. Gründe, die für die Nutzung der Datenbank sprechen ist die Einfachheit und vor allem das Daten auch unabhängig von dem Zustand der Anwendung zur Verfügung stehen. In zwei Fällen wurde sich jedoch trotzdem dagegen entschieden. Die Gründe dafür werden in den beiden folgenden Abschnitt zu den jeweiligen Repositorys erläutert.

3.5.1 MessageRepository

Funktion dieses Repositorys ist es, Nachrichtenkanäle zu verwalten. Dafür stehen Methoden zur Verfügung, um neue zu erzeugen, vorhandene zu löschen oder aber auch um existierende zu abonnieren. Die Nachrichtenkanäle werden in einer Map angelegt. Als Key wird eine Long-Value benötigt. Bei diesem Key wird immer die ID des Objektes genutzt, das den Kanal erstellt hat. In dieser Anwendung sind das zum einen die Spieler und zum anderen das Schachbrett. Da in dieser Klasse Daten gespeichert werden, gehört sie wie auch das folgende Minigame-Repository und der GameValidator zu den einzigen Klassen der Anwendung, die als @ApplicationScoped markiert werden müssen.

Bei dem eigentlichen Nachrichtenkanal Objekt handelt es ich um einen sogenannten SseBrodcaster. In ein solches Objekt können Nachrichten gesendet werden, die dann an alle Abonnenten verteilt werden. Es handelt sich hierbei um SSE. Das ist der Grund,

```
private static final OutboundSseEvent.Builder builder = new OutboundSseEventImpl.BuilderImpl();
private Map<Long, SseBroadcaster> publisher = new HashMap<>();

public void createBroadcast(Long publisherId) {
    publisher.put(publisherId, new SseBroadcasterImpl());
}
```

Snippet 4: createBroadcast

warum dieses Repository lokal in der Anwendung arbeitet. Denn so können die Clients direkt mit den Nachrichtenkanälen verbunden bleiben. Würden die Nachrichtenkanäle in der Datenbank persistiert werden, würde sich diese Verbindung schließen.

3.5.2 MinigameRepository

Beim MinigameRepository gibt es das gleiche Problem, wie im MessageRepository. Es werden Minigame-Entitys verwaltet, die wiederum auf SSE setzten um reaktiv mit den Spielern zu kommunizieren. Ein Unterschied ist hier, dass es für das Minispiel theoretisch vorteilhaft sein könnte, den aktuellen Zustand in einer Datenbank zu sichern. Dadurch das dieser nicht gesichert wird müssen mögliche Fehler Szenarien eingeplant werden. Zum einen könnte es passieren das ein Spieler die Verbindung zum Spiel verliert, während das Minigame läuft. Dafür musste im Controller sichergestellt werden das Spieler wiederbeitreten können. Das andere Szenario wäre das die Anwendung unterbrochen und neugestartet wird. In diesem Fall muss es möglich sein ein neues Minispiel zu starten.

```
private Map<Long, Minigame> minigames = new HashMap<>();

public boolean subscribeAsPlayerOne(Long gameId, Long playerId, SseEventSink sink) {
    Minigame minigame = minigames.get(gameId);
    if(minigame == null) {
        return false;
    }
    minigame.setPlayerOne(Tuple2.of(playerId, sink));
    return true;
}
```

Snippet 5: MinigameRepo

3.5.3 GameRepository

```
@Inject
EntityManager em;

@Transactional
public Game createGame(CreateGameDTO newGame, String newPlayerJoinToken, String newViewerJoinToken) {
    Game game = new Game();
    game.setGameName(newGame.gameName);
    game.setPlayerJoinToken(newPlayerJoinToken);
    game.setViewerJoinToken(newViewerJoinToken);
    em.persist(game);
    return game;
}
```

Snippet 6: GameRepo

Das GameRepository nutzt den Entitymanager um auf Spiele in der Datenbank zuzugreifen. Methoden die neue Spiele erzeugen oder Änderungen an vorhanden vornehmen, müssen mit der @Transactional Annotation gekennzeichnet sein. Dies kommt z.B. bei der Methode „deleteOldGames“ vor. Diese Methode formuliert eine Datenbankabfrage und gibt dabei an das Objekte vom Typ „Game“ erwartet werden. Es handelt sich um eine sogenannte „TypedQuery“. In dieser speziellen Abfrage werden alle Spiele angefordert, die älter als die in der „application.properties“ festgelegte Lebensdauer sind. Anschließend wird jedes gefundene Spiel gelöscht und die IDs der entfernten Spiele zurückgegeben.

3.5.4 PlayerRepository

In diesem Repository gibt es nicht mehr besonders viel neues zu sehen. Eine Besonderheit ist bei diesem aber z.B. das es extra Methoden gibt um die ID und den Status des Gegenspielers zu erfahren. Diese sind funktionell nicht unbedingt notwendig, denn für diese Information könnte auch das Spiel abgefragt werden und aus der Liste der Spieler der Gegenspieler ausgemacht werden. Durch diese Methode wird jedoch die Lesbarkeit des Codes erhöht und zudem ist auch die Abfrage performanter.

3.5.5 ChessBoardRepository

Mit nur drei Methoden hat diese Repository die wenigsten von allen. Dadurch das aber in jeder dieser Methoden die Felder des Schachbrettes von einem zwei-dimensionalen Array in eine Liste oder andersherum gewandelt werden, ist es nicht so einfach wie es

zuerst scheint. Warum diese Umformung stattfindet, wird im Kapitel 3.6.3 zur Chess-Board-Entity erläutert.

3.6 Entitys

Die Entitys werden zum Speichern von Datenpaketen benötigt. Sie enthalten Variablen in denen die Informationen abgelegt werden und entsprechende Methoden um diese abzurufen oder zu beschreiben. Neben diesen genannten Strukturen kommen lediglich noch Annotationen vor. Auf Logik wie z.B. Schachregeln wird hier verzichtet, um gemäß des Single-Responsibility-Principle zu agieren.

3.6.1 Game-Entity

Die Game Entity hält die Chessboard- und die Player-Entitys. Damit hält sie alle Entitys, die in der Datenbank persistiert werden. Die Spieler werden in einer Liste mit der @O-neToMany Annotation angelegt. In dieser wird mit CascadeType.ALL festgelegt, dass alle Operationen, die auf die Game-Entity angewendet werden, auch Auswirkungen auf

```
@Enumerated
private GameStatus gameStatus = GameStatus.SETUP;

@OneToMany(cascade = CascadeType.ALL, orphanRemoval = true, fetch = FetchType.LAZY)
@JoinColumn(name = "game_id")
private List<Player> players = new ArrayList<>();

@CreationTimestamp
private Date createdAt;

@OneToOne(cascade = CascadeType.ALL, orphanRemoval = true, fetch = FetchType.LAZY)
@MapsId
@JoinColumn(name = "id")
private ChessBoard board = new ChessBoard();
```

Snippet 7: GameEntity

die Player-Entitys haben. Damit die Spieler fest mit dem Spiel verbunden sind und nicht ohne eines existieren können, wird die orphanRemoval Eigenschaft auf true gesetzt. Zuletzt wird noch der FetchType vorgegeben. Dieser bestimmet das Ladeverhalten von Entitäten, wenn untergeordnete Entitäten abgerufen werden. Hier wird der FetchType Lazy verwendet, weil er unnötige Abfragen vermeidet und so die Performance erhöht. Mit der Annotation @JoinColum wird festgelegt mit welchem Attribut Spieler zurück auf

ein Spiel zugeordnet werden können. In diesem Fall wird der Standardwert verwendet, also die Id der Game-Entity und als Spaltenname „game_id“ gewählt.

Das zum Spiel gehörige Schachbrett wird mit @OneToOne gemappt. Hier gibt es die Besonderheit, dass sinnvollerweise die Spiel-Id immer der Schachbrett-Id entspricht. Sichergestellt wird das mit der @MapsId Annotation. Mit dieser übernimmt das Spiel die Id des Schachbrettes. Die sonstigen Eigenschaften sind genauso wie bei den Player-Entitys gewählt. Der für den GameValidator wichtige Zeitstempel wird in einem Date Objekt über die @CreationTimestamp erzeugt. Ansonsten gibt es neben den selbsterklärenden Variablen: „gameName“, „playerJoinToken“ und „viewerJoinToken“ noch den „gameStatus“. Bei dieser Variable handelt es sich um eine Enumeration die angibt in welchen Spielstadium sich das Spiel bereits befindet.

3.6.2 Player-Entity

In der Player-Entity ist zusehen das bei der ID, neben der @Id Annotation noch @GeneratedValue und @SequenceGenerator benutzt wurden. @GeneratedValue gibt dabei an, dass die ID nicht manuell zugewiesen werden muss, sondern durch PostgreSQL erzeugt wird. Ferner wird festgelegt das dabei eine definierte Sequenz verwendet werden soll, die mit @SequenceGenerator beschrieben wird. Hier war es wichtig, dass die Spieler und das Schachbrett ihre ID aus derselben Sequenz beziehen, damit beide eindeutig zuordbare Brodcasts im MessageRepository erstellen können. Ansonsten gibt es wie mit dem Spielstatus beim Spiel in diesem Fall noch ein Spielerstatus. In diesem wird festgehalten, ob er bereit ist das Spiel zu starten, am Zug ist oder dabei ist einen Bauern aufzuwerten.

3.6.3 ChessBoard-Entity

Dieses Entity besitzt nur zwei Variablen. Zum einen Ihre ID die sie mit dem Spiel teilt und zum anderen eine Liste mit den Spielfeldern. Auf diese Form wurde zurückgegriffen, weil sich das Speichern und Verwalten eines zwei-dimensionalen Arrays in der Datenbank als problematisch herausstellte. Die Felder werden nicht mit @OneToMany Annotation referenziert, da es ich bei den Feldern um eine andere Entitätsklasse handelt. Die normalen mit @Entity definierten Entitäten, werden in der Datenbank als eigene Tabelle angelegt. Die Feld-Entity ist hingegen mit @Embeddable definiert. Sie besitzt dadurch

keine eigene Tabelle in der Datenbank, braucht deshalb auch keine eigene ID und benötigt stattdessen die Parent ID um zugeordnet werden zu können. Als Parent wird die Entität angesehen die entweder ein Feld mit @Embedded speichert oder wie hier mehrere Felder mit @ElementCollection besitzt. Letztendlich entsteht trotzdem eine weitere Tabelle mit Feldern in der Datenbank, diese besitzen jedoch keine eigenen Ids, sondern verweisen lediglich auf das Entsprechen Schachbrett.

3.6.4 Minigame-Entity

Alle bisher vorgestellten Entitys wurden in der Datenbank gespeichert. Bei der Minigame-Entity sieht das anders aus. Sie wird stattdessen in der MinigameRepository in einer Map zusammen mit einer Spiel-ID gespeichert und verwaltet. Sie benötigt deshalb intern keine zusätzliche ID. Sie besitzt wie auch schon das Spiel und die Spieler einen Status in Form einer Enumeration. Daneben ist mit der „choosenColor“ noch eine weitere Enum-Klasse vorhanden. In ihr wird die vom ersten Spieler gewählte und vom zweiten Spieler zu erratende Farbe gespeichert. Ähnlich wie bereits im MessageRepository gesehen, sind auch hier mit „playerOne“ und „playerTwo“ Strukturen vorhanden, um SSE zu verschicken. Der Unterschied ist das hier kein Broadcasting stattfindet, sondern Nachrichten immer nur an einen der Spieler adressiert werden.

4 Zusammenfassung und Fazit

Die in dieser Projektarbeit entwickelte „Simple Chess“ Anwendung, ermöglicht es wie vorgesehen Schachpartien abzuhalten. Die bereitgestellte Rest-Schnittstelle verhält sich resilient und es wurden Maßnahmen gegen SQL-Injections getroffen. Partien lassen sich mit gewählten oder zufälligen Namen erstellen. Als Spieler oder Viewer mit dem entsprechenden Token, ist es möglich Spielen beizutreten. Daraufhin können mit den JWT im Authorization-Header oder dem bereitgestellten Cookie Anfragen an das System gerichtet werden, um das Spielgeschehen zu verfolgen oder aktiv zu verändern. Spiele die nicht regulär beendet werden, werden automatisch nach einer festgelegten Zeit wieder entfernt.

Beim Einarbeiten in das Thema Quarkus-reactive habe ich gemerkt, dass es Vorteilhaft wäre, Mutiny-Responses zur Verfügung zu stellen. Besonders wenn mehrere Clients die Anwendung simultan nutzten, könnte dann durch die Nutzung von spezielleren I/O-Threads in der Theorie eine bessere Performance erreicht werden. Leider kam es beim Arbeiten mit diesen zu Problemen und mir fehlte die Zeit mich weiter in die Problematik einzulesen. Deshalb habe ich mich auf eine Umsetzung mit SSE konzentriert, mit denen ich im Vorfeld schon Erfahrungen gesammelt habe. Mit dem so erzielten Ergebnis bin ich insgesamt zufrieden.

5 Referenzen

Web-Seiten zuletzt am 17.02.2023 abgerufen.

[@QA1] Quarkus: „Quarkus Reactive Architecture“, https://quarkus.io/guides/quarkus-reactive-architecture

[@IBM] IBM: „Rest-Apis“, https://www.ibm.com/de-de/cloud/learn/rest-apis

[@RED] RedHat: „Was ist Quarkus“, https://www.redhat.com/de/topics/cloud-native-apps/what-is-quarkus

Eidesstattliche Erklärung

Hiermit erkläre ich/ erklären wir an Eides statt, dass ich / wir die vorliegende Arbeit selbständig und ohne fremde Hilfe angefertigt habe / 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.

....
Ort, Datum Unterschrift

(bei Gruppenarbeit die Unterschriften sämtlicher Gruppenmitglieder)

Urheberrechtliche Einwilligungserklärung

Hiermit erkläre ich/ Hiermit erklären wir, dass ich/wir damit einverstanden bin/sind, dass meine/ 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.

....
Ort, Datum Unterschrift

(bei Gruppenarbeit die Unterschriften sämtlicher Gruppenmitglieder)

Hinweis: Die urheberrechtliche Einwilligungserklärung ist freiwillig.