

2020

# Dokumentation Vokabeltrainer-Spiel

KOMPONENTENBASIERTE ENTWICKLUNG KOMPLEXER ANWENDUNGEN  
EVANDIO PRIYANTO; CLEMENS BRAUER

# Inhaltsverzeichnis

<b>1</b>	<b>KOMPONENTENSCHNITT .....</b>	<b>3</b>
<b>2</b>	<b>SCHNITTSTELLENBESCHREIBUNG .....</b>	<b>4</b>
2.1	PLAYERMANAGEMENT – PLAYERMANAGEMENT_SERVICE .....	4
2.2	VOCABULARYMANAGEMENT – VOCABULARYMANAGEMENT_SERVICE .....	5
2.3	VOCABULARYMANAGEMENT – VOCABULARYSUPPLY_SERVICE.....	5
2.4	GAMEMANAGEMENT – GAMEMANAGEMENT_SERVICE .....	6
<b>3</b>	<b>KONZEPTIONELLES DATENMODELL .....</b>	<b>8</b>
<b>4</b>	<b>PRÄSENTATIONSSCHICHT .....</b>	<b>11</b>
4.1	ARCHITEKTUR .....	11
4.1.1	<i>View und Controller</i> .....	11
4.1.2	<i>Client Adapter</i> .....	12
4.2	MAINPANE .....	13
4.3	PLAYERPANE .....	13
4.3.1	<i>Angewandte Services der Schnittstellen</i> .....	13
4.3.2	<i>PlayerPane Oberfläche</i> .....	14
4.4	GAMEPANE.....	15
4.4.1	<i>Angewandte Services der Schnittstellen</i> .....	15
4.4.2	<i>GamePane Oberfläche</i> .....	17
4.4.3	<i>NewGameDialog Oberfläche</i> .....	19
4.5	VOCABPANE.....	19
4.5.1	<i>Angewandte Services der Schnittstellen</i> .....	19
4.5.2	<i>VocabPane Oberfläche</i> .....	20
<b>5</b>	<b>FRAMEWORKS .....</b>	<b>22</b>
5.1	JUNIT .....	22
5.2	MOCKITO .....	23
5.3	JPA MIT HIBERNATE UND JDBC DRIVER .....	24
5.4	PICOCONTAINER.....	27
5.5	SPRING.....	28
5.6	RETROFIT2.....	29
5.7	JAVAFX .....	31
<b>6</b>	<b>ABLAUFUMGEBUNG .....</b>	<b>33</b>

## Abbildungsverzeichnis

Abbildung 1 Schnittstellen ohne RestAdapter.....	3
Abbildung 2 Datenmodell.....	8
Abbildung 3 Controller Aufrufe .....	12
Abbildung 4 MainPane .....	13
Abbildung 5 PlayerPane.....	14
Abbildung 6 Hinweis bei der Löschung eines Spieler .....	15
Abbildung 7 Spieler Ladeaktivität.....	15
Abbildung 8 GamePane .....	17
Abbildung 9 Ansicht Spiel beendet.....	18
Abbildung 10 Ansicht Fragen beantworten.....	18
Abbildung 11 NewGameDialog Pane.....	19
Abbildung 12 VocabPane.....	20
Abbildung 13 Frameworks in Projekten .....	22
Abbildung 14 JUnit Test.....	23
Abbildung 15 Mockito .....	24
Abbildung 16 JPA persistence.xml.....	25
Abbildung 17 Hibernate Annotation .....	26
Abbildung 18 Picocontainer .....	27
Abbildung 19 Spring SpringBootApplication .....	28
Abbildung 20 SpringRestController .....	28
Abbildung 21 Spring Error handling.....	29
Abbildung 22 Retrofit2 Interface.....	30
Abbildung 23 Retrofit2 Builder.....	30
Abbildung 24 JavaFX Scene Builder 2.0 .....	31
Abbildung 25 JavaFX Controller.....	32
Abbildung 26 App Klasse mit main Methode .....	32
Abbildung 27 Ablaufumgebung .....	33

## Tabellenverzeichnis

Tabelle 1 Score Attribute.....	8
Tabelle 2 Player Attribute.....	9
Tabelle 3 Game Attribute .....	9
Tabelle 4 Round Attribute .....	9
Tabelle 5 Answer Attribute.....	10
Tabelle 6 QuizQuestion Attribute.....	10
Tabelle 7 Word Attribute.....	10
Tabelle 8 Vocabulary Attribute.....	10
Tabelle 9 VocabularyList Attribute .....	11
Tabelle 10 JavaFX Views .....	11
Tabelle 11 Annotationsbeschreibungen.....	27

# 1 Komponentenschnitt

In diesem Kapitel werden die Komponenten des Informationssystems sowie deren Schnittstellenkommunikation anhand des in Abbildung 1 dargestellten Komponentendiagramms vorgestellt.

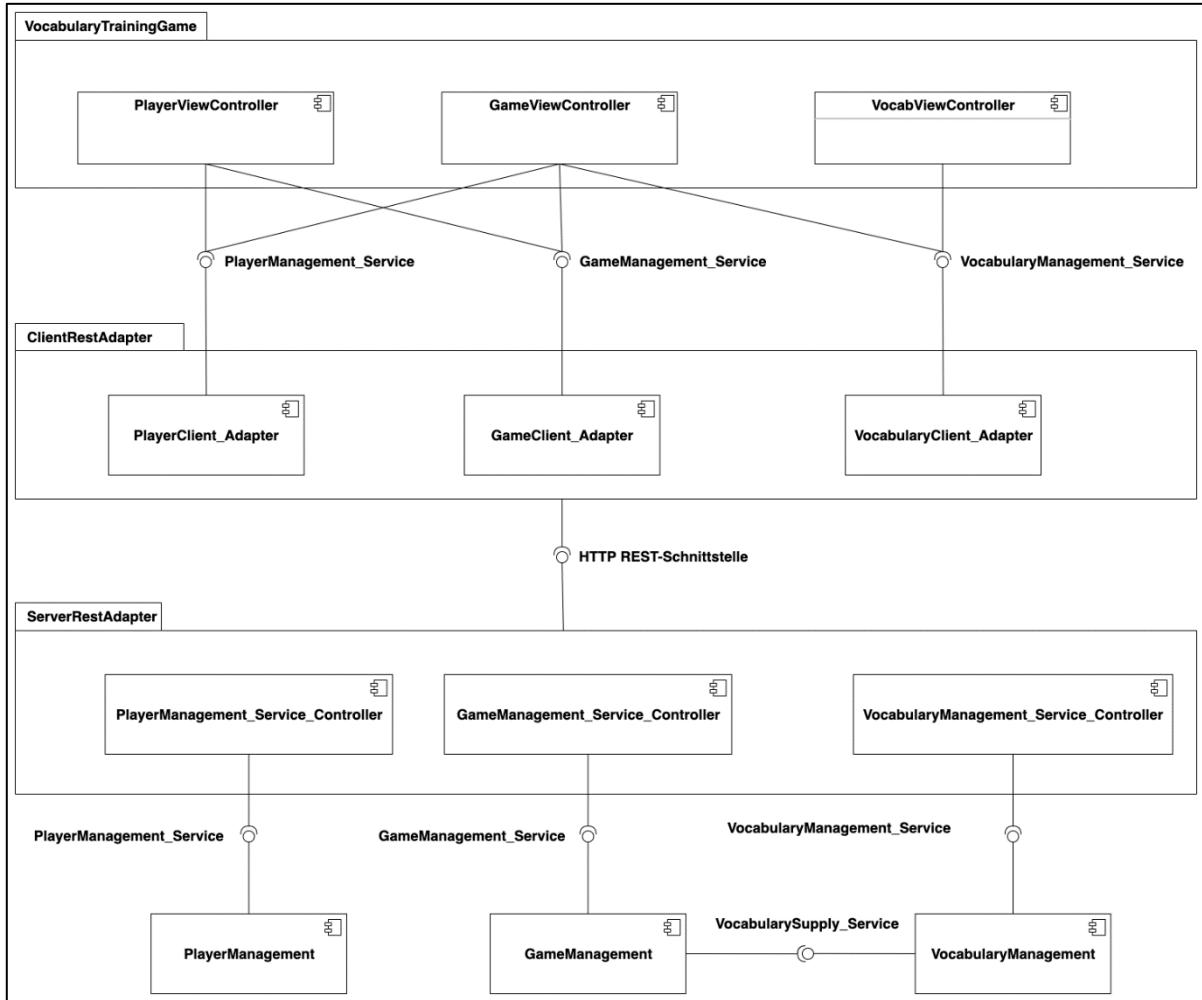


Abbildung 1 Schnittstellen ohne RestAdapter

Die Darstellung lässt sich grob in zwei Bereiche unterteilen. Im unteren Bereich befinden sich die Server-Komponenten und im oberen die Client-Komponenten der Anwendung.

## Server-Komponenten:

Die `PlayerManagement`, `GameManagement`- und `VocabularyManagement` Komponenten exportieren jeweils ihre Service Schnittstellen, welche von der jeweiligen `Service Controller` Komponente im `RestAdapter` importiert werden. Die `VocabularyManagement` Komponente bietet zusätzlich eine `VocabularySupply_Service` Schnittstelle an, die von der `GameManagement` Komponente importiert wird.

## Client-Komponenten:

Die `PlayerClient_Adapter`, `GameClient_Adapter`- und `VocabularyClient_Adapter` Komponenten exportieren jeweils ihre Service Schnittstellen, welche von der jeweiligen `ViewController` Komponente im `VocabularyTrainingGame` Packet importiert werden. Zusätzlich importiert die `PlayerViewController` Komponente die `GameManagement_Service`-

und die *GameViewController* Komponente die *PlayerManagement\_Service*- und *VocabularyManagement\_Service* Schnittstelle.

Der Datenfluss zwischen *ServerRestAdapter* und *ClientRestAdapter* wird durch eine HTTP REST-Schnittstelle realisiert.

## 2 Schnittstellenbeschreibung

In diesem Kapitel werden die Schnittstellen *PlayerManagement\_Service*, *GameManagement\_Service*, *VocabularyManagement\_Service* und *VocabularySupply\_Service* im Java-Dock Form beschrieben.

### 2.1 PlayerManagement – *PlayerManagement\_Service*

Die *PlayerManagement\_Service* Schnittstelle realisiert die Erstellung, die Bearbeitung, die Löschung und die Abfrage der *Player* Objekte. Folgend werden die einzelnen Methodenköpfe beschrieben:

```
/**  
 * Die Methode erstellt ein neues Player Objekt, gibt es zurück und speichert es in der  
 * Datenbank ab. Um sicherzustellen, dass kein Name doppelt vorkommt, wird vorab überprüft, ob  
 * der übergebene Name bereits von einem anderen Spieler belegt ist.  
 *  
 * @param name : Der Name des neuen Players als String.  
 * @return Player Objekt : Falls ein Player mit demselben Namen noch nicht vorhanden ist  
 * @throws DuplicatePlayerException : Falls bereits ein Player mit demselben Namen existiert.  
 * @throws PlayerManagementServiceException : Falls ein technischer Fehler auftritt.  
 */  
public Player createPlayer(String name) throws DuplicatePlayerException,  
PlayerManagementServiceException;  
  
/**  
 * Die Methode ändert den Namen eines Players und gibt das aktualisierte Player Objekt zurück.  
 * Um sicherzustellen, dass kein Name doppelt vorkommt, wird vorab überprüft, ob der übergebene  
 * Name bereits von einem anderen Spieler belegt ist. Weiterhin wird überprüft, ob der Player  
 * noch in der Datenbank existiert.  
 *  
 * @param player : Der Player, dessen Name geändert werden soll.  
 * @param name : Der neue Name für den Player.  
 * @return aktualisiertes Player Objekt : Falls ein Player mit demselben Namen noch nicht  
 * vorhanden ist.  
 * @throws DuplicatePlayerException : Falls bereits ein Player mit demselben Namen existiert.  
 * @throws PlayerManagementServiceException : Falls ein technischer Fehler auftritt.  
 * @throws PlayerNotFoundException : Falls der Player in der Datenbank nicht mehr existiert.  
 */  
public Player editPlayerName(Player player, String name) throws DuplicatePlayerException,  
PlayerManagementServiceException, PlayerNotFoundException;  
  
/**  
 * Ein Player wird anhand seiner ID gelöscht. Dafür wird vorab überprüft, ob das Player  
 * Objekt noch in der Datenbank existiert.  
 *  
 * @param playerID : Die ID des Players, welcher gelöscht werden soll.  
 * @return true -> Falls der Player erfolgreich gelöscht wurde.  
 * @throws PlayerNotFoundException : Falls der Player in der Datenbank nicht mehr existiert.  
 * @throws PlayerManagementServiceException : Falls ein technischer Fehler auftritt.  
 */  
public boolean deletePlayer(int playerID) throws PlayerNotFoundException,  
PlayerManagementServiceException;
```

---

<sup>1</sup> Es wird das komplette Objekt übergeben, um bei der Aktualisierung die Versionierung (Optimistic-Locking) zu prüfen

```

/**
 * Die Methode gibt eine Liste aller Player Objekte zurück.
 *
 * @return List<Player> : Eine Liste aller Player Objekte.
 *         leere Liste : Falls kein Player vorhanden ist
 * @throws PlayerManagementServiceException : Falls ein technischer Fehler auftritt.
 */
public List<Player> getAllPlayer() throws PlayerManagementServiceException;

```

## 2.2 VocabularyManagement – VocabularyManagement\_Service

Die *VocabularyManagement\_Service* Schnittstelle realisiert die Erstellung und Abfrage der *VocabularyList* Objekte. Folgend werden die einzelnen Methodenköpfe beschrieben:

```

/**
 * Die Methode erstellt ein VocabularyList Objekt. Um sicherzustellen, dass noch kein
 * VocabularyList Objekt mit demselben Buch- und Kapiteltitel existiert, wird vorab überprüft,
 * ob diese (in Kombination) bereits in der Datenbank vorkommen.
 *
 * @param book : Der Buchtitel.
 * @param chapter : Der Titel des Kapitels.
 * @param vocabularys : Die Liste an Vokabeln des Buch-Kapitels.
 * @param language1 ; Eine Übersetzungssprache des Buch-Kapitels.
 * @param language2 ; Eine weitere Übersetzungssprache des Buch-Kapitels.
 * @return VocabularyList Objekt : Das erstellte VocabularyList Objekt.
 * @throws DuplicateVocabularyListException : Falls es bereits eine VocabularyList mit
 *         demselben Buch- und Kapiteltitel existiert.
 * @throws VocabManagementServiceException : Falls ein technischer Fehler auftritt.
 */
public VocabularyList createVocabularyList(String book, String chapter, List<Vocabulary>
vocabularys, String language1, String language2) throws DuplicateVocabularyListException,
VocabManagementServiceException;

/**
 * Die Methode gibt eine Liste aller VocabularyList Objekte zurück.
 *
 * @return List<VocabularyList> : Eine Liste aller VocabularyList Objekte.
 *         eine leere Liste : Falls kein VocabularyList Objekt existiert.
 * @throws VocabManagementServiceException : Falls ein technischer Fehler auftritt.
 */
public List<VocabularyList> getAllVocabularyLists() throws VocabManagementServiceException;

/**
 * Die Methode gibt eine Liste der Hauptinformationen aller VocabularyList Objekte zurück.
 * Diese Informationen werden jeweils in einem VocabularyListDTO2 Objekt gespeichert und
 * bestehen aus dem Buch- und Kapiteltitel, der ID sowie den beiden Sprachen einer
 * VocabularyList.
 *
 * @return List<VocabularyListDTO> : Eine Liste aller VocabularyListDTO Objekte.
 *         eine leere Liste : Falls kein VocabularyList Objekt existieren.
 * @throws VocabManagementServiceException : Falls ein technischer Fehler auftritt.
 */
public List<VocabularyListDTO> getAllVocabularyListsMainInformation() throws
VocabManagementServiceException;

```

## 2.3 VocabularyManagement – VocabularySupply\_Service

Die *VocabularySupply\_Service* Schnittstelle realisiert die Abfrage von Vocabulary- und Word Objekten. Folgend werden die einzelnen Methodenköpfe beschrieben:

---

<sup>2</sup> Das VocabularyDTO Objekt wird genutzt, um Ladezeiten zu minimieren. Dies kann genutzt werden, wenn nur wesentliche Informationen eines VocabularyList Objektes benötigt werden.

```

/**
 * Die Methode gibt eine Liste aller, in der Datenbank verfügbaren Wörter einer Sprache
 * zurück.
 *
 * @param language : Die Sprache, für welche alle Wörter benötigt werden.
 * @return List<Word> : Die Liste aller verfügbaren Wörter einer Sprache.
 * @throws NoWordFoundException : Falls kein Wort gefunden wird.
 * @throws VocabManagementServiceException : Falls ein technischer Fehler auftritt.
 */
public List<Word> getWords( String language) throws NoWordFoundException,
VocabManagementServiceException;

/**
 * Die Methode gibt ein VocabularyList Objekt anhand der übergebenen Id zurück.
 *
 * @param vocabularyListId : ID der gesuchten VocabularyListe
 * @return VocabularyList : Die gefundene VocabularyList.
 * @throws NoVocabularyListFoundException : Falls kein VocabularyListObjekt mit der ID
 * existiert.
 * @throws VocabManagementServiceException : Falls ein technischer Fehler auftritt.
 */
public VocabularyList getVocabularyList(int vocabularyListId) throws
NoVocabularyListFoundException, VocabManagementServiceException;

```

## 2.4 GameManagement – GameManagement\_Service

Die *GameManagement\_Service* Schnittstelle realisiert die Erstellung, die Bearbeitung, die Löschung und die Abfrage der *Game* Objekte. Folgend werden die einzelnen Methodenköpfe und Variablen beschrieben:

```

/**
 * Die Anzahl der zu spielenden Runden.
 */
final int NUMBER_OF_ROUNDS = 3;

/**
 * Die Anzahl an falschen Antworten in einer Frage.
 */
final int NUMBER_OF_WRONG_ANSWERS = 3;

/**
 * Diese Methode erstellt ein neues Game, speichert es in der Datenbank und
 * gibt das erstellte Game Objekt zurück.
 *
 * @param player1 : Erster Spieler
 * @param player2 : Zweiter Spieler
 * @param from : Ausgangssprache für das Spiel (Frage)
 * @param to : Übersetzungssprache für das Spiel (Antwort)
 * @param vocabularyListId : ID der ausgewählten VocabularyListe
 * @return Game Objekt : Falls ein Game erfolgreich erstellt wurde
 * @throws GameNotCreatableException : Falls das Spiel aufgrund einer der folgenden
        Exception: (NoVocabularyFoundException | NoWordNotFoundException |
        NoVocabularyListFoundException | GameManagementServiceException)
        nicht erstellt werden kann.
 * @throws OnePlayerTwoTimesInGameException : Falls player1 und player2 der selbe Spieler ist.
 * @throws GameManagementServiceException : Falls ein technischer Fehler aufgetreten ist.
 */
public Game createGame(Player player1, Player player2, String from, String to, int
vocabularyListId) throws OnePlayerTwoTimesInGameException, GameNotCreatableException,
GameManagementServiceException;

```

```

/**
 * Die Methode löscht ein Game Objekt anhand der Game-ID. Dafür wird vorab überprüft, ob das
 * Game Objekt noch in der Datenbank existiert.
 *
 * @param gameId : Die Id des Games, welches gelöscht werden soll.
 * @return True : Wenn das Löschen erfolgreich war.
 * @throws GameNotFoundException : Falls kein Spiel anhand der ID gefunden werden kann.
 * @throws GameManagementServiceException : Falls ein technischer Fehler aufgetreten ist.
 */
public boolean deleteGame(int gameId) throws GameNotFoundException,
GameManagementServiceException;

/**
 * Die Methode aktualisiert ein Game Objekt. Dafür wird vorab überprüft, ob das Game Objekt
 * noch in der Datenbank existiert.
 *
 * @param game : Das Game, welches aktualisiert werden soll.3
 * @return True : Wenn die Aktualisierung erfolgreich war.
 * @throws GameNotFoundException : Falls das Spiel in der Datenbank nicht mehr existiert.
 * @throws GameManagementServiceException : Falls ein technischer Fehler aufgetreten ist.
 */
public boolean updateGame(Game game) throws GameNotFoundException,
GameManagementServiceException;

/**
 * Die Methode gibt alle Game Objekte als Liste zurück, an welchen ein Spieler mit der
 * übergebenen ID beteiligt ist.
 *
 * @param playerID : Die ID eines Spielers.
 * @return List<Game> : Liste mit allen Game Objekten an denen der Spieler beteiligt ist
 *         leere Liste : Falls keine Games mit dem gewünschten Spieler vorhanden sind.
 * @throws GameManagementServiceException : Falls ein technischer Fehler aufgetreten ist.
 */
public List<Game> getAllGamesByPlayer(int playerID) throws GameManagementServiceException;

/**
 * Die Methode löscht alle Spiele, an denen ein Spieler beteiligt ist
 *
 * @param playerID : Die PlayerID des Spielers, wessen Spiele gelöscht werden sollen.
 * @return true : Wenn das Löschen erfolgreich war.
 * @throws GameManagementServiceException : Falls ein technischer Fehler aufgetreten ist.
 */
public boolean deleteAllGamesByPlayer(int playerID) throws GameManagementServiceException;

/**
 * Die Methode löscht alle Spiele, an denen ein Spieler beteiligt ist
 *
 * @param playerID : Die ID des Spielers, wessen Spiele gelöscht werden sollen.
 * @return true : Wenn die Löschung der Spiele erfolgreich, bzw. aufgrund von keiner
 *         Spielbeteiligung nicht nötig war.
 * @throws GameManagementServiceException : Falls ein technischer Fehler aufgetreten ist.
 */
public boolean deleteAllGamesByPlayer(int playerID) throws GameManagementServiceException;

```

---

<sup>3</sup> Es wird das komplette Objekt übergeben, um bei der Aktualisierung die Versionierung (Optimistic-Locking) zu prüfen

### 3 Konzeptionelles Datenmodell

Wie in Abbildung 2 dargestellt, besteht das Datenmodell aus insgesamt neun Klassen. Die Persistenz des Datenmodells wurde in diesem Projekt mithilfe der Java-Persistence-API (JPA) durch den JPA-Provider Hibernate in einer MySQL Datenbank realisiert.

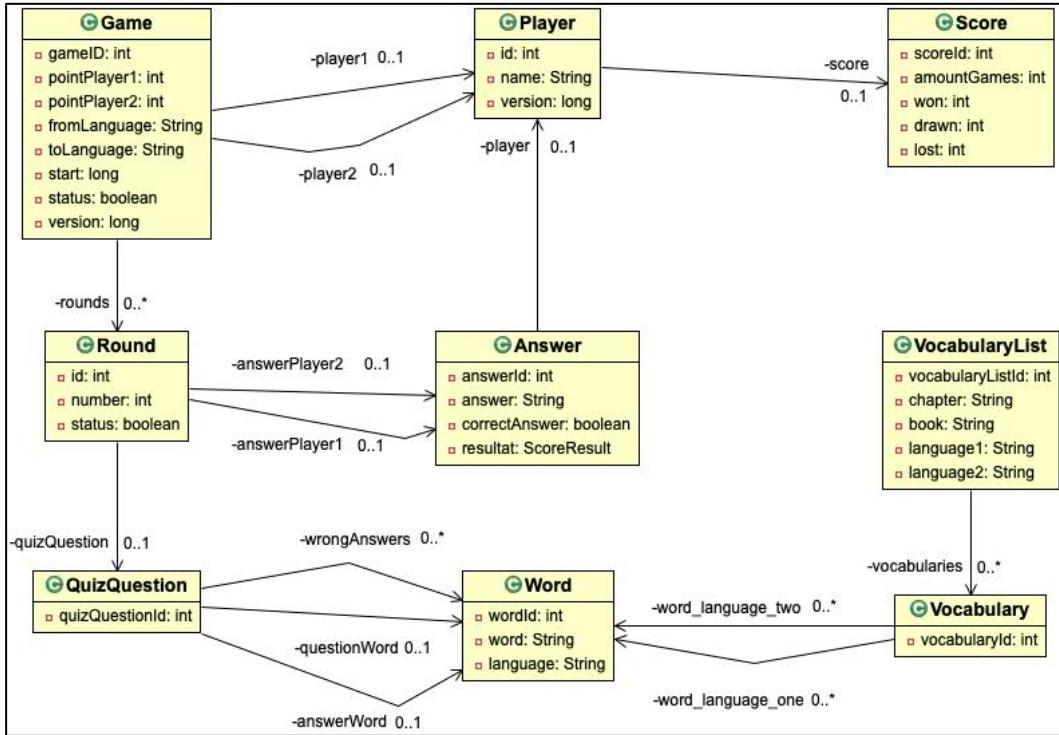


Abbildung 2 Datenmodell

Die Datenhoheit über die Klassen ist wie folgt auf die Komponenten aufgeteilt:

GameManagement:	Game, Round, Answer, QuizQuestion
PlayerManagement:	Player, Score,
VocabularyManagement:	VocabularyList, Vocabulary, Word

Die Klassen und deren Attribute werden nachfolgend beschrieben:

#### Score:

Die Score Klasse definiert die Spielergebnisse eines Spiels. Die Attribute werden in Tabelle 1 beschrieben.

Attribut : Type	Beschreibung
scoreId : int	Unique Identifier
amountGames : int	Gesamtanzahl der gespielten Spiele
won : int	Anzahl der gewonnenen Spiele
draw : int	Anzahl der unentschiedenen Spiele
lost : int	Anzahl der verlorenen Spiele

Tabelle 1 Score Attribute

**Player:**

Die Player Klasse definiert einen Spieler. Die Attribute werden in Tabelle 2 beschrieben.

Attribut : Type	Beschreibung
id : int	Unique Identifier
name : String	Name des Spielers
version : long	Versionsnummer des Spielerobjektes für Concurrency Strategie Optimistic-Locking
score : Score	Spielergebnisse des Spielers

Tabelle 2 Player Attribute

**Game:**

Die Game Klasse definiert ein Spiel. Die Attribute werden in Tabelle 3 beschrieben.

Attribut : Type	Beschreibung
gameID : int	Unique Identifier
pointPlayer1 : int	Anzahl der richtigen Antworten des ersten Spielers
pointPlayer2 : int	Anzahl der richtigen Antworten des zweiten Spielers
fromLanguage : String	Ausgangssprache des Spiels
toLanguage : String	Übersetzungssprache des Spiels
start : long	Startzeit des Spieles in Millisekunden
status : boolean	Signalisiert, ob das Spiel aktiv (true) oder beendet (false) ist
version : long	Versionsnummer des Spielerobjektes für Concurrency Strategie Optimistic-Locking
player1 : Player	Erster Spieler
player2 : Player	Zweiter Spieler
rounds : List<Round>	Liste mit Spielrunden

Tabelle 3 Game Attribute

**Round:**

Die Round Klasse definiert eine Spielrunde. Die Attribute werden in Tabelle 4 beschrieben.

Attribut : Type	Beschreibung
id : int	Unique Identifier
number : int	Rundennummer
status : boolean	Signalisiert, ob die Runde noch gespielt werden muss (true) oder beendet (false) ist
answerPlayer1 : Answer	Antwort des ersten Spielers auf die Quizfrage
answerPlayer2 : Answer	Antwort des zweiten Spielers auf die Quizfrage
quizQuestion : QuizQuestion	Quizfrage

Tabelle 4 Round Attribute

**Answer:**

Die Answer Klasse definiert eine Antwort eines Spielers auf eine Quizfrage. Die Attribute werden in Tabelle 5 beschrieben.

Attribut : Type	Beschreibung
answerId : int	Unique Identifier
answer : String	Antwort auf eine Frage
correctAnswer : boolean	Signalisiert, ob eine Antwort richtig (true) oder falsch (false) ist
resultat : ScoreResult	Beschreibt, ob ein Spieler in der Runde mit der Antwort gewonnen (WON), verloren (LOST) oder Gleichstand (DRAW) errungen hat
player : Player	Spieler, welcher die Antwort gegeben hat

Tabelle 5 Answer Attribute

**QuizQuestion:**

Die QuizQuestion Klasse definiert eine Quizfrage einer Runde. Die Attribute werden in Tabelle 6 beschrieben.

Attribut : Type	Beschreibung
quizQuestionId : int	Unique Identifier
questionWord : Word	Fragewort
answerWord : Word	Richtige Antwort
wrongAnswers : List<Word>	Liste an falschen Antworten

Tabelle 6 QuizQuestion Attribute

**Word:**

Die Word Klasse definiert ein Wort einer Sprache. Die Attribute werden in Tabelle 7 beschrieben.

Attribut : Type	Beschreibung
wordId : int	Unique Identifier
word : String	Wort
language : String	Sprache des Wortes

Tabelle 7 Word Attribute

**Vocabulary:**

Die Vocabulary Klasse definiert eine Vokabel. Die Attribute werden in Tabelle 8 beschrieben.

Attribut : Type	Beschreibung
vocabularyId : int	Unique Identifier
word_language_one : List<Word>	Liste von Vokabelwörtern der Ausgangssprache
word_language_two : List<Word>	Liste von Vokabelwörtern der Übersetzungssprache

Tabelle 8 Vocabulary Attribute

### VocabularyList:

Die VocabularyList Klasse definiert die Liste von Vokabeln eines Buchkapitels. Die Attribute werden in Tabelle 9 beschrieben.

Attribut : Type	Beschreibung
vocabularyListId : int	Unique Identifier
book : String	Buchtitel
chapter : String	Kapiteltitel
language_one : String	Ausgangssprache
language_two: String	Übersetzungssprache
vocabularies : List<Vocabulary>	Liste von Vokabeln

Tabelle 9 VocabularyList Attribute

## 4 Präsentationsschicht

In diesem Kapitel wird der Aufbau und die Funktionsweise der Präsentationsschicht beschrieben. In dieser Arbeit wurde die Präsentationsschicht in Form einer JavaFX Gui erstellt. Dafür wurden mithilfe des JavaFX Scene Builder<sup>4</sup> Oberflächen (Panes) erstellt und als fxml Dateien gespeichert.

Folgend wird die Architektur der gesamten Gui sowie die einzelnen Oberflächen beschrieben.

### 4.1 Architektur

In diesem Unterkapitel wird die Architektur der Präsentationsschicht beschrieben.

#### 4.1.1 View und Controller

Insgesamt wurden mit dem JavaFX Scene Builder die in Tabelle 10 aufgelisteten fünf Oberflächen gestaltet und als fxml Dateien im Projekt abgelegt. Diese Dateien beinhalten Informationen der Gui Elemente im XML Format.

Name	Controller	Funktion
Main.fxml	MainController.java	Stellt das Hauptfenster mit einer Tab-View dar
PlayerPane.fxml	PlayerController.java	View für die Spielerverwaltung
GamePane.fxml	GameController.java	View für die Spielverwaltung
NewGameDialog.fxml	NewGameDialogController.java	Dialog zur Erstellung von neuen Spielen
VocabPane.fxml	VocabController.java	View für das Einlesen von Vokabellisten

Tabelle 10 JavaFX Views

Um eine strikte Trennung von Oberflächengestaltung und Funktionslogik zu realisieren, wurde für jede fxml View Datei eine Controllerklasse erstellt. Mithilfe dieser Controllerklasse kann auf die einzelnen Gui Elemente zugegriffen und deren Action-Listenermethoden implementiert werden.

Beim Start der App wird die Main.fxml Ansicht und deren Main Controller geladen. Innerhalb des Main Controllers werden in Abhängigkeit des ausgewählten Tabs die entsprechende fxml

<sup>4</sup> <https://www.oracle.com/java/technologies/javase/javafxscenebuilder-info.html>

Ansicht sowie der dazugehörige Controller geladen. In Abbildung 3 wird zur Verdeutlichung der Aufrufbaum der einzelnen Ansichten/Controller dargestellt.

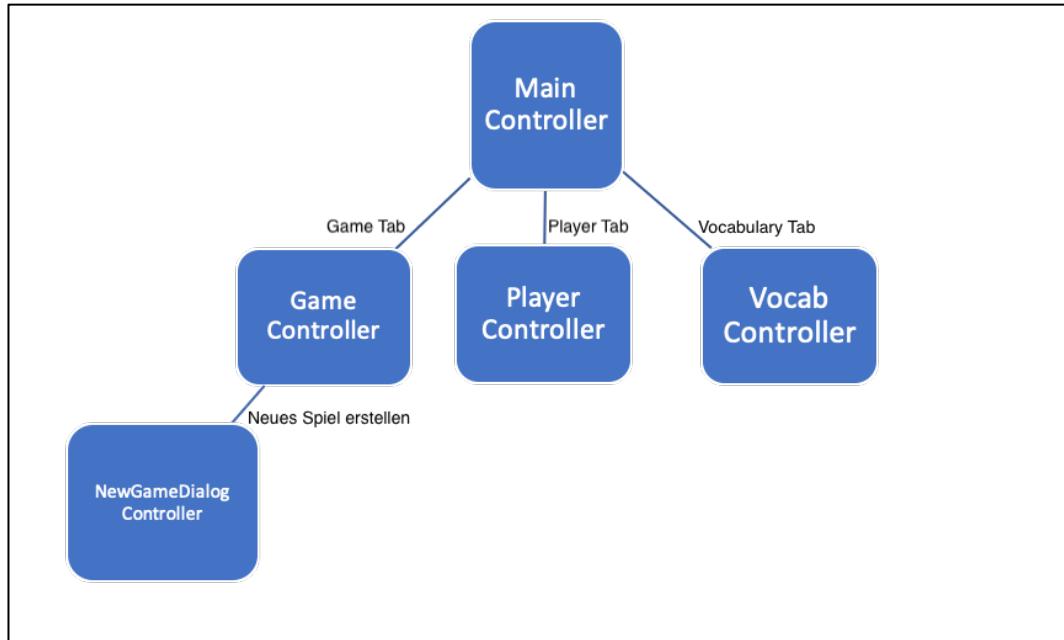


Abbildung 3 Controller Aufrufe

Die NewGameDialog Ansicht wird bei der Erstellung eines neuen Spiels aufgerufen. Diese wird durch den Klick eines Buttons in der GamePane View geladen. Eine genauere Beschreibung dazu erfolgt in Unterkapitel 2.4.

#### 4.1.2 Client Adapter

Um mit dem Server zu kommunizieren, wurden die folgenden Client Adapter erstellt:

- ⇒ GameClientAdapter → GameManagement\_Service
- ⇒ PlayerClientAdapter → PlayerManagement\_Service
- ⇒ VocabularyClientAdapter → VocabularyManagement\_Service

Diese realisieren die Serveranfrage mithilfe ihrer zugehörigen RestService Schnittstellen. Durch die Kommunikation mit dem Restadapter des Servers werden die entsprechenden Services angesprochen.

Um einen möglichst flüssigen Spielablauf zu gewährleisten, werden die meisten Server-Anfragen in einem separaten Thread durchgeführt. Dies hat den Vorteil, dass die Gui während der Client Server Kommunikation nicht einfriert. Dafür befinden sich im Paket model diverse Worker Klassen, welche von der Klasse Task erben. Diese rufen über einen der Client Adapter jeweils einen Service auf. In den Controllern wurden weiterhin spezifische onWorkerDone Methoden implementiert, welche aufgerufen werden, sobald die Client Server Kommunikation beendet wurde.

Ein Beispiel für die Erstellung eines Worker Thread:

```

DeletePlayerGamesWorker worker = new DeletePlayerGamesWorker(player);
setOnGameDeleteWorkerDone(worker);
Thread workerThread = new Thread(worker);
workerThread.start();
  
```

## 4.2 MainPane

Die MainPane stellt eine leere TabPane Oberfläche dar. Die einzigen Gui-Elemente befinden sich mit den Tabbuttons Player, Game und Vocabulary im oberen Bereich.

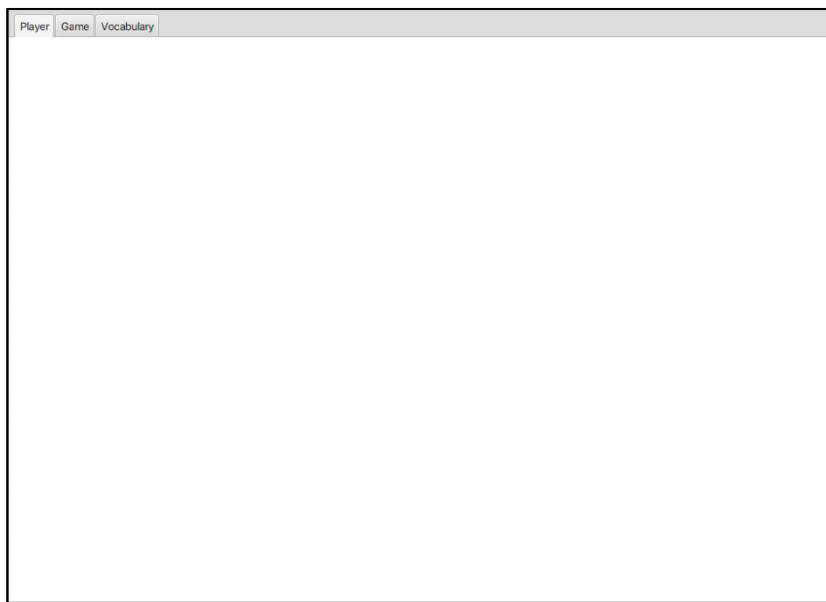


Abbildung 4 MainPane

In diese Oberfläche werden während der Ausführung des Programmes die in den folgenden Unterkapiteln vorgestellten Panes in Abhängigkeit des ausgewählten Tabs geladen.

## 4.3 PlayerPane

Auf der PlayerPane können Spieler der Anwendung betrachtet, gelöscht, bearbeitet und neu erstellt werden. In den folgenden Unterkapiteln werden genutzte Methoden der Schnittstellen sowie die Oberfläche und deren Verhalten beschrieben.

### 4.3.1 Angewandte Services der Schnittstellen

Alle im Bezug zur Datenbank stattfindenden Aktionen werden mithilfe von PlayerClientAdapter und GameClientAdapter Instanzen von der Client Seite angestoßen.

#### **PlayerClientAdapter**

Der PlayerClientAdapter spricht über die Rest-Schnittstelle die folgenden Methoden der in Unterkapitel 2.1 vorgestellten PlayerManagement\_Service Schnittstelle an :

- createPlayer              -> CreatePlayerWorker
- editPlayerName (ohne Worker)
- deletePlayer              -> DeletePlayerWorker
- getAllPlayer              -> GetAllPlayerWorker

#### **GameClientAdapter**

Der GameClientAdapter spricht über die Rest-Schnittstelle die folgende Methode der in Unterkapitel 2.4 vorgestellten GameManagement\_Service Schnittstelle an :

- deleteAllGamesByPlayer    -> DeletePlayerGamesWorker

#### 4.3.2 PlayerPane Oberfläche

Die PlayerPane wird als erste Oberfläche beim Start des Programmes geladen und angezeigt. In Abbildung 5 wird die PlayerPane Ansicht mit exemplarischen Spielerdaten dargestellt.

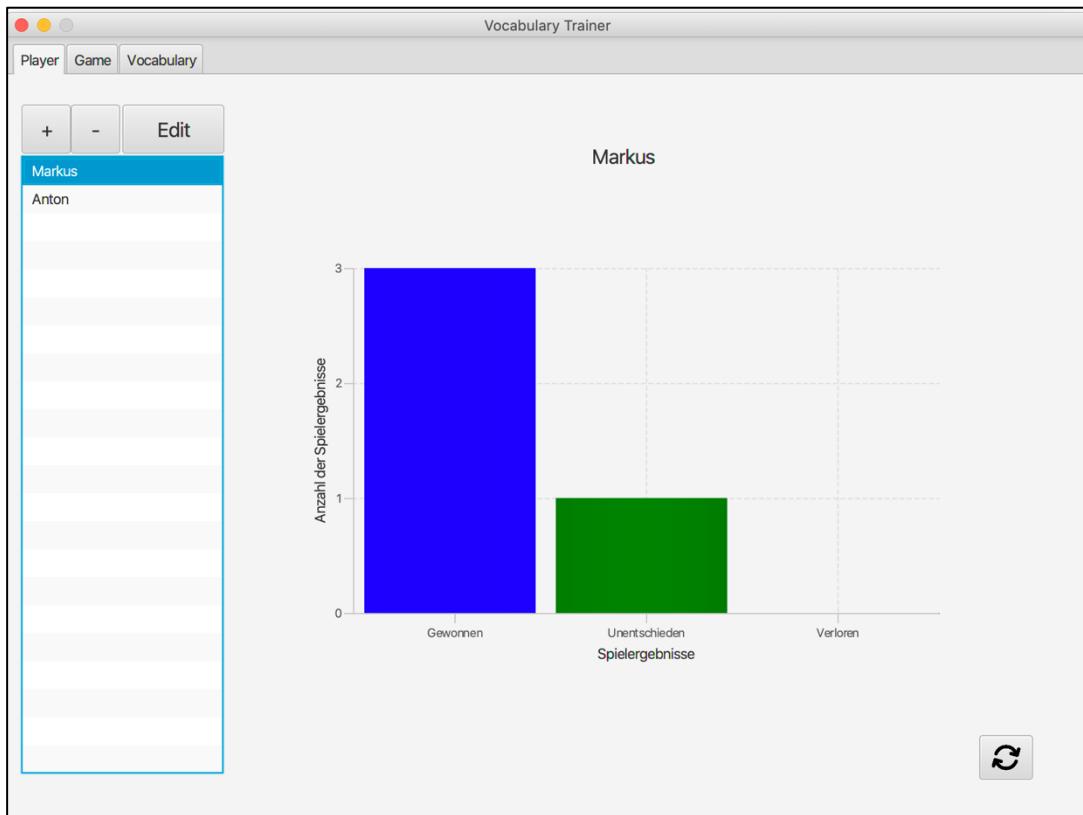


Abbildung 5 PlayerPane

Auf der linken Seite ist eine Liste mit allen Spielernamen sowie einem Plus- , Minus- und Bearbeitungsbutton sichtbar. Die Spielerliste wird bei der Initialisierung der Ansicht durch die `getAllPlayer` Servicemethode mit allen Spielern aus der Datenbank befüllt. Durch einen Klick auf den Plusbutton öffnet sich ein Eingabedialog, welcher den User auffordert, den Namen eines neuen Spielers einzugeben. Nach der Eingabe des Namens und der Bestätigung des OK Buttons wird der Spieler durch die `createPlayer` Servicemethode in der Datenbank gespeichert und der Liste hinzugefügt. Falls dieser Name bereits an einen anderen Spieler vergeben ist, erhält der User eine Mitteilung, dass der Spieler aufgrund einer Namensdoppelung nicht angelegt werden konnte. Äquivalent dazu funktioniert der Bearbeitungsbutton, wobei die Auswahl eines Spielers aus der Liste vor dessen Klick vorausgesetzt wird. Durch eine erfolgreiche Bearbeitung wird durch die `editPlayerName` Servicemethode der neue Name für den Spieler in der Datenbank hinterlegt und in der Spielerliste entsprechend geändert.

Die Auswahl des Minusbuttons ermöglicht es dem User einen Spieler zu löschen. Auch hierbei wird die Auswahl eines Spielers aus der Liste vorausgesetzt. Vor der Löschung wird der User, wie in Abbildung 6 dargestellt, darauf hingewiesen, dass neben dem Spieler auch alle mit ihm verbundenen Spiele gelöscht werden.

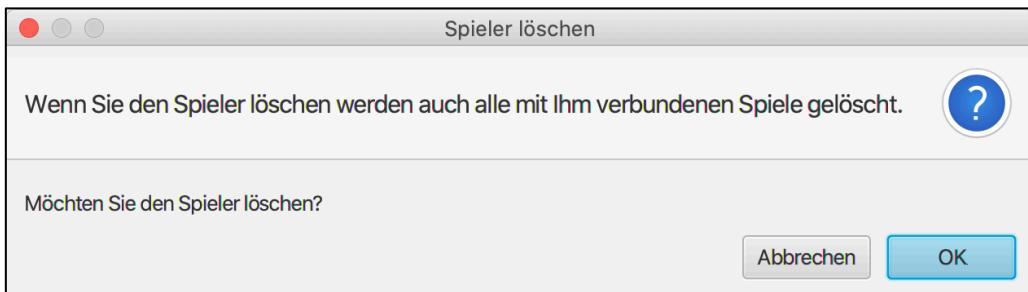


Abbildung 6 Hinweis bei der Löschung eines Spieler

Durch einen Klick auf OK wird die Löschung mithilfe der `deleteAllGamesByPlayer` und `deletePlayer` Servicemethoden durchgeführt.

Weiterhin ist in Abbildung 5 im mittleren Bereich ein Balkendiagramm sowie der Spielername des ausgewählten Spielers zu sehen. In dem Balkendiagramm wird die jeweilige Anzahl an verlorenen, gewonnenen oder in Gleichstand geendeten Spiele des Spielers angezeigt.

Im unteren rechten Bereich befindet sich ein Aktualisierungsbutton . Durch einen Klick auf den Button wird die PlayerPane mithilfe der `getAllPlayer` Servicemethode aktualisiert. Weiterhin dient dieser Button als Indikator für Hintergrundaktivitäten (siehe Abbildung 7).



Abbildung 7 Spieler Ladeaktivität

In diesem Fall wird links neben dem Button ein Schriftzug mit der Beschreibung der Aktivität angezeigt. Zusätzlich dreht sich der Pfeil des Buttons. Der Button ist währenddessen deaktiviert und kann nicht angeklickt werden.

#### 4.4 GamePane

Auf der GamePane können Spiele gespielt und gelöscht werden. Ausgehend von der GamePane kann die NewGameDialog Ansicht als Dialogfenster geöffnet werden, um ein neues Spiel zu erstellen. Da diese Panes eng miteinander verbunden sind, werden diese in diesem Unterkapitel gemeinsam betrachtet.

In den folgenden Abschnitten werden genutzte Methoden der Schnittstellen sowie die Oberfläche und deren Verhalten beschrieben.

##### 4.4.1 Angewandte Services der Schnittstellen

Alle im Bezug zur Datenbank stattfindenden Aktionen werden mithilfe von PlayerClientAdapter, GameClientAdapter und VocabularyClientAdapter Instanzen von der Client Seite angestoßen.

### **PlayerClientAdapter**

Der PlayerClientAdapter spricht über die Rest-Schnittstelle die folgenden Methoden der in Unterkapitel 2.1 vorgestellten PlayerManagement\_Service Schnittstelle an:

- getAllPlayer                      -> GetAllPlayerWorker

### **GameClientAdapter**

Der GameClientAdapter spricht über die Rest-Schnittstelle die folgenden Methoden der in Unterkapitel 2.4 vorgestellten GameManagement\_Service Schnittstelle an:

- createGame                      -> CreateGameWorker
- deleteGame                      -> DeleteGameWorker
- getAllGamesByPlayer            -> GetAllGameWorker
- updateGame                      -> UpdateGameWorker

Weiterhin werden innerhalb der GameManagement\_Service Schnittstelle die folgenden VocabularySupply\_Service Schnittstellenmethoden (siehe Unterkapitel 2.3) aufgerufen:

- getVocabularyList
- getWords

### **VocabularyClientAdapter**

Der VocabularyClientAdapter spricht über die Rest-Schnittstelle die folgende Methode der in Unterkapitel 2.2 vorgestellten VocabularyManagement\_Service Schnittstelle an:

- getAllVocabularyListsMainInformation-> GetAllVocabularyListDTOWorker

#### 4.4.2 GamePane Oberfläche

Die GamePane wird durch einen Klick auf den Tab Game im oberen Bereich des Programmes geladen und angezeigt. In Abbildung 8 wird die GamePane Ansicht mit exemplarischen Spieler- und Spieldaten dargestellt.

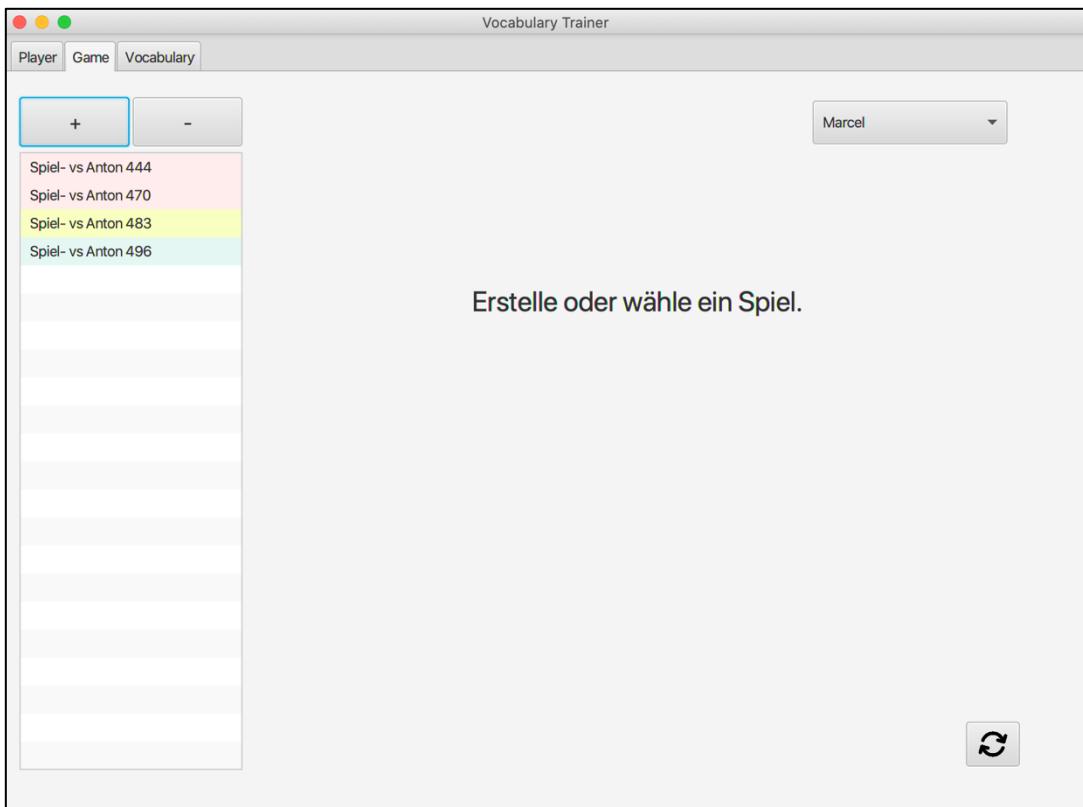


Abbildung 8 GamePane

Grundlegend ähneln sich viele der vorab gezeigten PlayerPane Elemente den GamePane Elementen. Auf der linken Seite ist eine Liste mit allen Spielen sowie einem Plus- und Minusbutton sichtbar.

Durch einen Klick auf den Plusbutton öffnet sich der in dem folgenden Abschnitt 4.4.3 beschriebene NewGameDialog um eine neue Spiel zu erstellen. Die Auswahl des Minusbuttons ermöglicht es dem User ein Spiel zu löschen. Hierbei wird die Auswahl eines Spielers aus der Liste vorausgesetzt. Die Löschung wird mithilfe der `deleteGame` Servicemethode in der Datenbank realisiert.

Weiterhin ist ein Dropdownbutton im rechten oberen Bereich zu sehen. Bei der Initialisierung der GamePane wird dieser Dropdownbutton durch die `getAllPlayer` Servicemethode mit allen Spielern aus der Datenbank gefüllt. Sobald der User über diesen Button einen Spieler auswählt, werden mithilfe der `getAllGamesByPlayer` Servicemethode alle Spiele eines Spielers, absteigend nach Erstellungsdatum geordnet, in die linke Liste geladen. Dabei wird jedes Spiel mit einer Spiel ID und dem Namen des Gegenspielers angezeigt. Außerdem färbt sich der Hintergrund eines Spieles in eine von drei unterschiedlichen Farben.

## **Rot**

Die rote Farbe signalisiert, dass ein Spiel beendet ist. Durch einen Klick auf ein rotes Spiel wird, wie in Abbildung 9 dargestellt, das Ergebnis sowie eine Auswertung angezeigt.

Gleichstand!					
Runde	Fragewort	Anton		Marcel	
		Antwort	Resultat	Antwort	Resultat
1	el zoo	groß	Falsch	Zoo	Falsch
2	la ciudad	Stadt	Richtig	Stadt	Richtig
3	el apellido	der Affe	Falsch	Kuh	Falsch

Abbildung 9 Ansicht Spiel beendet

Die Spiele können einsehen, welche Frage, von wem, wie beantwortet wurde.

## **Gelb**

Die gelbe Farbe signalisiert, dass der Mitspieler am Zug ist. Durch einen Klick auf ein gelbes Spiel wird dies dem User als Schriftzug angezeigt.

## **Grün**

Die grüne Farbe signalisiert, dass der User an der Reihe ist um Fragen zu beantworten. Durch einen Klick auf ein grünes Spiel werden, wie in Abbildung 10 dargestellt, vier verschiedene Antwortmöglichkeiten für ein Fragewort in Form von Buttons angezeigt.

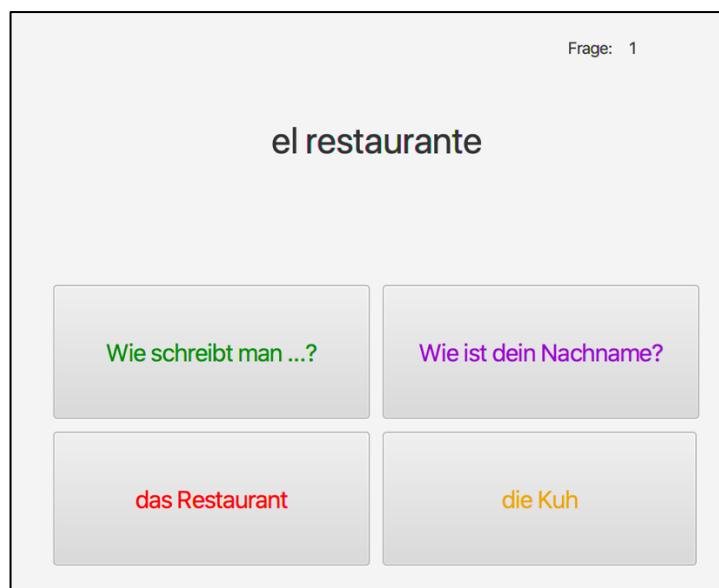


Abbildung 10 Ansicht Fragen beantworten

Sobald der User auf eine Antwort klickt, wird diese auf ihre Richtigkeit überprüft und mithilfe der `updateGame` Servicemethode in der Datenbank hinterlegt. In dieser Implementierung spielt jeder Spieler drei Runden. Sobald die letzte Runde gespielt wurde, wird neben der Richtigkeitsprüfung das Endergebnis bestimmt.

Wie bereits in Abschnitt 4.3.2 beschrieben, befindet sich auch auf der GamePane im unteren rechten Bereich ein Aktualisierungsbutton . Durch einen Klick auf den Button wird die GamePane mithilfe der `getAllPlayer` Servicemethode aktualisiert. Weiterhin dient dieser Button auch hier als Indikator für Hintergrundaktivitäten.

#### 4.4.3 NewGameDialog Oberfläche

Der NewGameDialog wird, wie bereits vorab beschrieben, durch einen Klick auf den Plusbutton in der GamePane Oberfläche geladen und angezeigt. In Abbildung 11 wird die NewGameDialog Ansicht mit exemplarischen Spieler- und Vokabeldaten dargestellt.

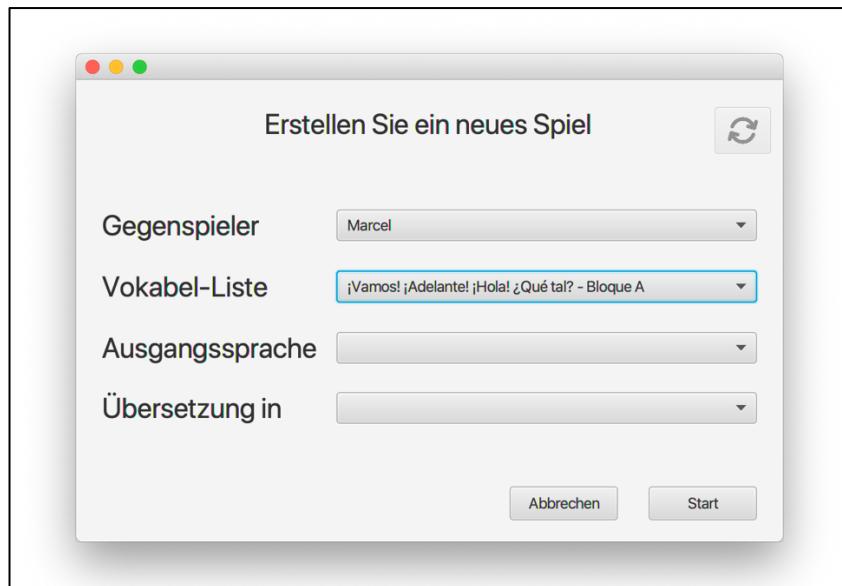


Abbildung 11 NewGameDialog Pane

In dem Dialog werden vier verschiedene Dropdownbuttons angezeigt. Alle Dropdownbuttons werden bei der Initialisierung des Dialoges mithilfe der `getAllVocabularyListsMainInformation` und `getAllPlayer` Servicemethoden aus der Datenbank geladen. Sobald der User nach der Auswahl auf den Start-Button klickt, wird das Spiel durch die `createGame` Servicemethode in der Datenbank abgelegt und der Dialog geschlossen. Diese greift dabei auf die `getVocabularyList` und `getWords` Servicemethoden der `VocabularySupply_Service` Schnittstelle zu.

### 4.5 VocabPane

Auf der VocabPane können Vokabellisten eingesehen und abgelegt/eingelesen werden.

In den folgenden Abschnitten werden genutzte Methoden der Schnittstellen sowie die Oberfläche und deren Verhalten beschrieben.

#### 4.5.1 Angewandte Services der Schnittstellen

Alle im Bezug zur Datenbank stattfindenden Aktionen werden mithilfe einer `VocabularyClientAdapter` Instanz von der Client Seite angestoßen.

## VocabularyClientAdapter

Der VocabularyClientAdapter spricht über die Rest-Schnittstelle die folgenden Methoden der in Unterkapitel 2.2VocabularyManagement – VocabularyManagement\_Service vorgestellten VocabularyManagement\_Service Schnittstelle an:

- createVocabularyList      -> CreateVocabularyListWorker
- getAllVocabularyLists      -> GetAllVocabularyListsWorker

### 4.5.2 VocabPane Oberfläche

Die VocabPane wird durch einen Klick auf den Tab Vocabulary im oberen Bereich des Programmes geladen und angezeigt. In Abbildung 12 wird die VocabPane Ansicht mit exemplarischen Vokabeldaten dargestellt.

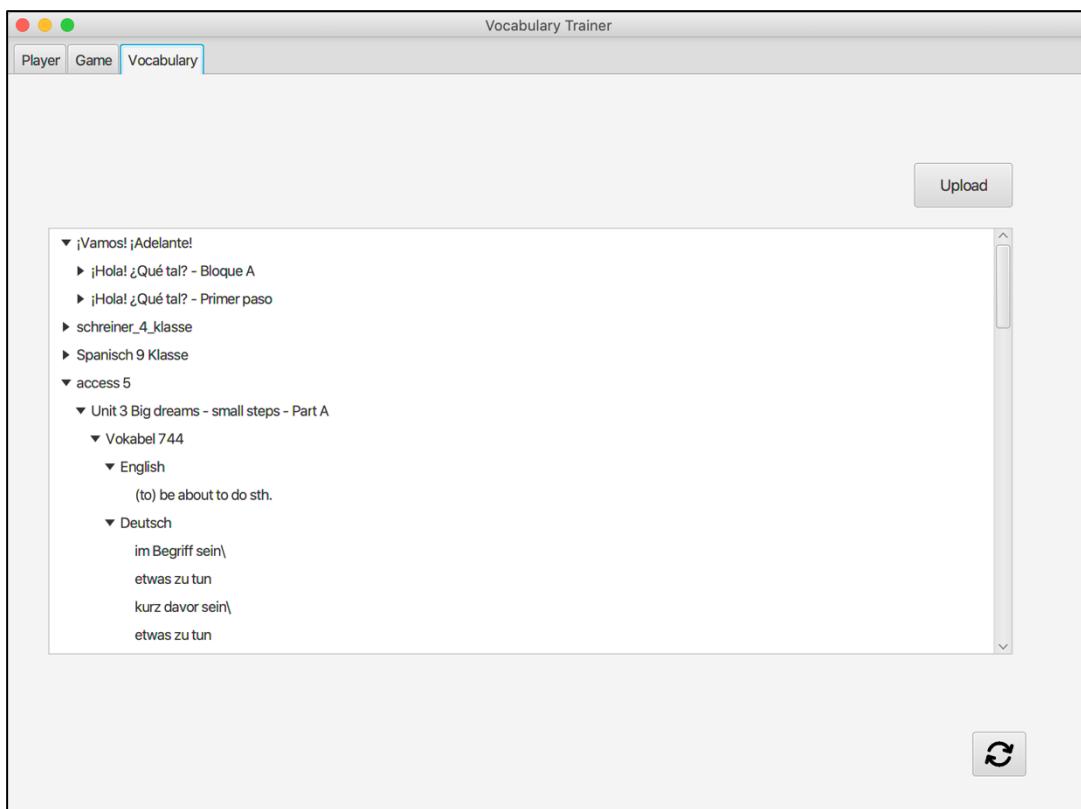


Abbildung 12 VocabPane

Die VocabularyPane ist in ihrem Funktionsumfang recht einfach aufgebaut. Mittig ist ein DataTree Fenster, welches die in der Datenbank verfügbaren Vokabellisten beinhaltet, zu sehen. Bei der Initialisierung der VocabularyPane werden diese VocabularyList Objekte mithilfe der *getAllVocabularyLists* Servicemethode geladen und in das DataTree Objekt abgelegt.

Der Tree wird dabei folgendermaßen aufgebaut:

```
⇒ Buchtitel
  ⇒ Kapiteltitel
    ⇒ VokabellID
      ⇒ Sprache1
        ⇒ Wort
        ⇒ Wort
      ⇒ Sprache2
        ⇒ Wort
        ⇒ Wort
```

Durch einen Klick auf den Uploadbutton öffnet sich ein FileChooser Fenster. Dadurch hat der User die Möglichkeit die gewünschte Vokabelliste (TXT Datei im entsprechenden Format) auszuwählen. Sobald er die Auswahl mit OK bestätigt, wird mithilfe der `createVocabularyList` Servicemethode die Vokabelliste erstellt und in der Datenbank abgelegt. Dabei wird überprüft, ob sich bereits ein Buchtitel mit dem Kapiteltitel in der Datenbank befindet. Duplikate werden nicht zugelassen und mit einer Nachricht angezeigt.

Wie bereits bei den anderen Panes beschrieben, befindet sich auch auf der VocabPane im unteren rechten Bereich ein Aktualisierungsbutton . Durch einen Klick auf den Button wird die VocabPane mithilfe der `getAllVocabularyLists` Servicemethode aktualisiert. Weiterhin dient dieser Button auch hier als Indikator für Hintergrundaktivitäten.

## 5 Frameworks

In diesem Kapitel werden die in dem Projekt verwendeten Frameworks beschrieben. Um einen Überblick über die verwendeten Frameworks zu erhalten, werden diese, repräsentiert durch ihr Logo, in Abbildung 13 verteilt auf die einzelnen Maven Projekte dargestellt.

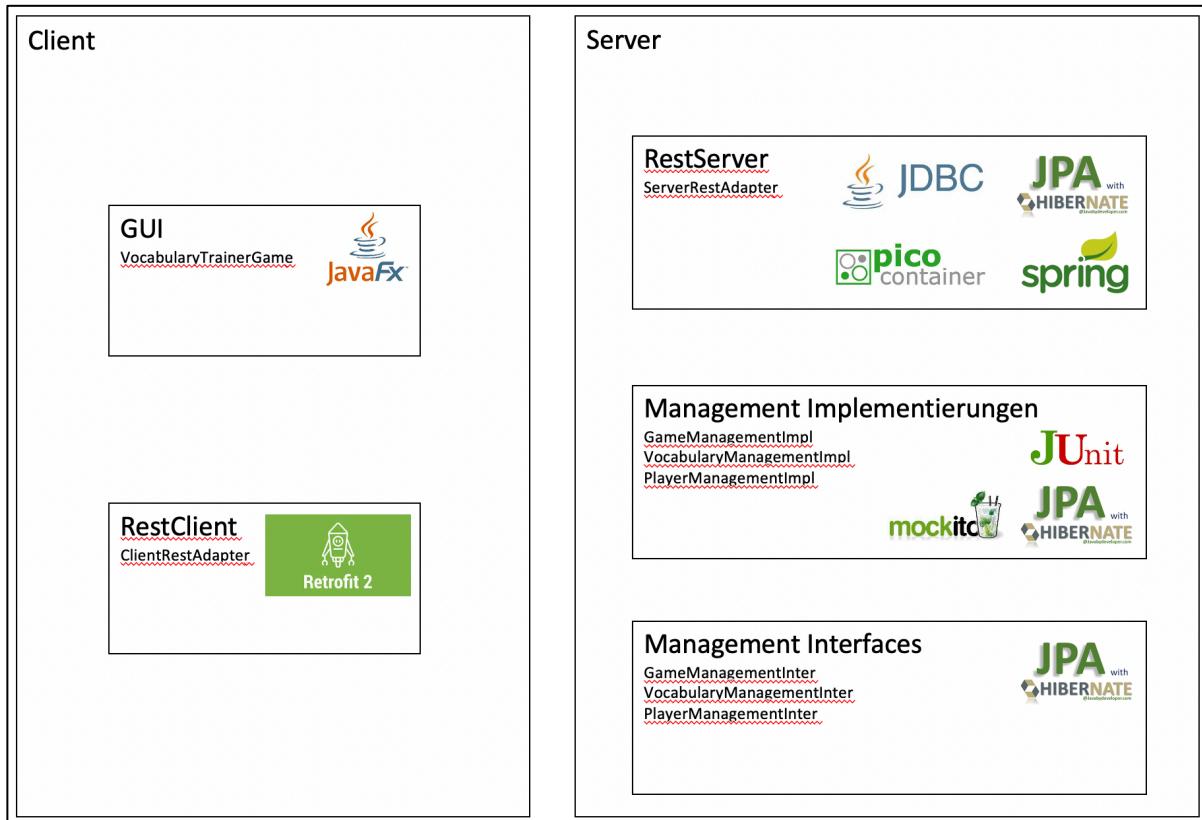


Abbildung 13 Frameworks in Projekten

Für den Überblick wurden die Maven Projekte in Gruppen zusammengefasst. Jede Gruppe wurde wiederum der Server- oder Client-Seite zugeordnet.

In den folgenden Unterkapiteln werden die verwendeten Frameworks und ihre Konfiguration genauer beschrieben.

### 5.1 JUnit

Bei JUnit handelt es sich um ein Test Framework, das in dieser Arbeit in den Vocabulary-, Game- und Player-ManagementImpl Maven-Projekten zum Testen der jeweiligen Service\_Impl Klassen genutzt wird.

#### Konfiguration:

Um das JUnit Framework zu nutzen, muss dessen Dependency in die pom.xml des Maven-Projektes eingebunden werden. Im Anschluss wird eine oder mehrere Testklassen mit beliebigen Testmethoden, wie in Abbildung 14 beispielhaft dargestellt, erzeugt.

```

@Before
public void setUp() {
    // run before each test
}

@After
public void onFinish() {
    // run after each test
}

@Test
public void testMethode1() {
    int expectedResult = 3;
    int actualMaxNumber = Math.max(3, 2);
    boolean assumption = 3 > 2;

    Assert.assertEquals(expectedResult, actualMaxNumber);
    Assert.assertTrue(assumption);
}

@Test(expected = IndexOutOfBoundsException.class)
public void testMethode2() {
    String[] wordArray = {"word1", "word2"};

    String word3 = wordArray[2];
}

```

Abbildung 14 JUnit Test

JUnit bietet verschiedene Annotationsmöglichkeiten, um die Testklassen nach Belieben zu konfigurieren. Beispielsweise müssen alle implementierten Testmethoden mit einer `@Test` Annotation versehen werden, damit diese bei einem Testvalidierungsdurchlauf erkannt werden. Weiterhin bietet JUnit mit der `@Before` und `@After` Annotation die Möglichkeit Methoden vor und nach jeder Ausführung einer Testmethode aufzurufen.

Um zu überprüfen, ob Tests erfolgreich durchlaufen werden oder fehlschlagen, kann die JUnit Assert-Klasse, wie beispielhaft in `testMethode1` gezeigt, genutzt werden. So kann beispielsweise überprüft werden, ob sich Objekte gleichen oder ein Boolean True ist. Weiterhin wurde die `@Test` Annotation der `testMethode2` um die `expected` Variable erweitert. So kann eine erwartete `IndexOutOfBoundsException` als Testergebnis angegeben werden.

Die Testklassen wurden in dieser Arbeit unter dem Pfad `src/test/java` in den jeweiligen Maven-Projekten implementiert. Um die Tests bei der Maven-Install zu berücksichtigen, sollten die Testklassen gemäß der Maven-Konvention<sup>5</sup> benannt werden.

## 5.2 Mockito

Das Mockito Framework wird in dieser Arbeit verwendet, um in Testklassen Mock-Objekte als Platzhalter für „echte“ Objekte zu erzeugen. Dadurch wird eine Isolation der Tests von anderen Klassen realisiert.

### Konfiguration:

Um das Mockito Framework zu nutzen, muss dessen Dependency in die pom.xml des Maven-Projektes eingebunden werden. Im Anschluss kann die Mockito Klasse in dem Projekt, wie in Abbildung 15 beispielhaft dargestellt, aufgerufen und genutzt werden.

---

<sup>5</sup> <https://maven.apache.org/surefire/maven-surefire-plugin/examples/inclusion-exclusion.html>

```

21     private PlayerManagementDao pmDao = Mockito.mock(PlayerManagementDao.class);
22     private PlayerManagement_Service pmS = new PlayerManagement_ServiceImpl(pmDao);
23
24     @Test
25     public void testDeletePlayerSuccess() throws PlayerManagementServiceException, PlayerNotFoundException {
26         Player player = new Player("Sascha");
27
28         Mockito.when(pmDao.getPlayer(player.getId())).thenReturn(player);
29         Mockito.when(pmDao.deletePlayer(player)).thenReturn(true);
30
31         boolean result = pmS.deletePlayer(player.getId());
32         Assert.assertTrue(result);
33     }
34
35     @Test(expected = PlayerNotFoundException.class)
36     public void testDeletePlayerFails() throws PlayerManagementServiceException, PlayerNotFoundException {
37         int unknownPlayerID = 1;
38         Mockito.when(pmDao.getPlayer(unknownPlayerID)).thenThrow(PlayerNotFoundException.class);
39
40         pmS.deletePlayer(unknownPlayerID);
41     }

```

Abbildung 15 Mockito

In dem abgebildeten Beispiel soll die Methode *deletePlayer* der PlayerManagement\_Service\_implementation Klasse getestet werden. Die Methode greift intern mithilfe einer DAO Klasse und deren Methoden *getPlayer* und *deletePlayer* auf die Datenbank zu, um Objekte zu schreiben und zu lesen. Da die Tests jedoch isoliert von der Funktionalität der PlayerManagementDao Klasse realisiert werden sollen, wird diese mithilfe von Mockito in Zeile 21 gemockt und im Anschluss der PlayerManagement\_Service\_implementation Klasse übergeben.

In der Testmethode *testDeletePlayerSuccess* soll getestet werden, ob die *getDeletePlayer* Methode unter gegebenen Bedingungen einen Spieler aus der Datenbank löscht und dies mit einem True als Rückgabewert quittiert. Innerhalb dieser Methode werden die *getPlayer* und *deletePlayer* Methoden der PlayerManagementDao Klasse aufgerufen, um den Player als Objekt in der Datenbank anhand der playerID zu finden und diesen im Anschluss aus der Datenbank zu löschen. Um dieses Verhalten für den Testfall zu simulieren, wird in Zeile 28 die Mockito Klasse genutzt, um den Player als Rückgabewert für den Aufruf der *getPlayer* Methode mit dem vorbestimmten Übertragungsparameter (ID des Spielers) zu bestimmen. Der Mockito Aufruf kann dabei als Wenn (when) -> Dann (thenReturn) verstanden werden. Äquivalent dazu wird ein Wiedergabewert in Zeile 29 für die *deletePlayer* Methode (pmDao) bestimmt. Sobald nun die *deletePlayer* Methode (pmS) aufgerufen wird, wird auf Grundlage des Verhaltens der internen Abläufe True als Ergebnis erwartet.

In der Testmethode *testDeletePlayerFails* wird erwartet, dass die Löschung eines Players nicht möglich ist, da kein Player mit der übergebenen ID in der Datenbank existiert. Dafür wird in Zeile 38 der Mockito Aufruf mit einer *thenThrow* Methode beendet. Die Aussage dieses Aufrufes kann als „Wenn die *deletePlayer* Methode (pmDao) mit dem Übertragungsparameter 1 aufgerufen wird, dann werfe eine PlayerNotFoundException“ verstanden werden. Sobald nun die *deletePlayer* Methode (pmS) aufgerufen wird, wird die PlayerNotFoundException geworfen.<sup>6</sup>

### 5.3 JPA mit Hibernate und JDBC Driver

Die Persistenz des Datenmodells wurden in diesem Projekt mithilfe der Java-Persistence-API (JPA) durch den JPA-Provider Hibernate und einem JDBC Driver realisiert. Hibernate ist ein Objekt-relationales Mapping-Framework, mit welchem objektorientierte Domänenmodelle in

---

<sup>6</sup> Dieses Verhalten entspricht nicht der tatsächlichen Implementierung, sondern wird hier nur als Beispiel aufgeführt.

einer relationalen Datenbank abgebildet werden können. Der JDBC Driver dient als Connector zu der verwendeten MySQL Datenbank. In dieser Arbeit wurde dies in dem *ServerRestAdapter* Maven Projekt realisiert.

### Konfiguration:

Im ersten Schritt müssen die entsprechenden Dependencies des JDBC Drivers (mysql-connector-java) und Hibernate in die pom.xml des Maven-Projektes eingebunden werden.

Ein zentrales Element der JPA Konfiguration stellt die, in Abbildung 16 exemplarisch dargestellte, persistence.xml Datei dar.

```
1 <?xml version="1.0"?>
2 <persistence version="2.1"
3   xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/xml/ns/p
4   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5   xmlns="http://xmlns.jcp.org/xml/ns/persistence">
6   <persistence-unit transaction-type="RESOURCE_LOCAL"
7     name="managerMysql">
8
9     <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
10
11    <class>htw.GameManagementInter.Game</class>
12    <class>htw.GameManagementInter.Answer</class>
13    <class>htw.GameManagementInter.Round</class>
14    <class>htw.GameManagementInter.QuizQuestion</class>
15    <class>htw.PlayerManagementInter.Player</class>
16    <class>htw.PlayerManagementInter.Score</class>
17    <class>htw.VocabularyManagementInter.VocabularyList</class>
18    <class>htw.VocabularyManagementInter.Vocabulary</class>
19    <class>htw.VocabularyManagementInter.Word</class>
20
21    <properties>
22      <property name="javax.persistence.jdbc.driver"
23        value="com.mysql.jdbc.Driver" />
24      <property name="javax.persistence.jdbc.url"
25        value="jdbc:mysql://db.f4.htw-berlin.de:3306/htw" />
26      <property name="javax.persistence.jdbc.user" value="htw" />
27      <property name="javax.persistence.jdbc.password" value="htw" />
28      <property name="hibernate.dialect" value="org.hibernate.dialect.MySQLDialect" />
29      <property name="hibernate.show_sql" value="true" />
30      <property name="hibernate.hbm2ddl.auto" value="update" />
31      <!-- clean, update, clean-drop -->
32    </properties>
33  </persistence-unit>
34
35 </persistence>
```

Abbildung 16 JPA persistence.xml

In dieser Datei wird die Verknüpfung zwischen Hibernate und dem MySQL JDBC Driver (Zeile 21 - 32), sowie zwischen Hibernate und den Datenmodellklassen (Zeile 9 - 19) hergestellt. Mithilfe der JPA Persistence Klasse kann im Projekt ein EntityManagerFactory Objekt instanziert werden, welches über den in Zeile 7 bestimmten Namen die Konfiguration der persistence.xml Datei annimmt. Das EntityManagerFactory Objekt wiederum erstellt EntityManager, welche verwendet werden, um persistente Entitätsinstanzen zu erstellen und zu entfernen, um Entitäten anhand ihres Primärschlüssels zu finden und um Abfragen über Entitäten durchzuführen. Hibernate scannt während der Laufzeit die Annotationen der angegebenen Datenmodellklassen. In Abbildung 17 werden beispielhaft die Annotationen der Game Klasse dargestellt.

```

37 @Entity
38 @NamedQuery(name="Game.findAllByPlayer", query="SELECT i "
39             + "FROM Game AS i "
40             + "WHERE i.player1.id = :playerId or i.player2.id = :playerId "
41             + "Order By i.start DESC")
42 public class Game {
43
44     @Id
45     @GeneratedValue(strategy=GenerationType.AUTO)
46     private int gameID;
47
48     @ManyToOne(cascade = CascadeType.MERGE)
49     private Player player1;
50
51     @ManyToOne(cascade = CascadeType.MERGE)
52     private Player player2;
53
54     @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
55     private List<Round> rounds;
56     private int pointPlayer1;
57     private int pointPlayer2;
58     private String fromLanguage;
59     private String toLanguage;
60     private long start;
61     private boolean status;
62
63     @Version
64     private long version;

```

Abbildung 17 Hibernate Annotation

Die abgebildeten Annotationen werden folgend in Tabelle 11 genauer definiert.

Annotationen	Beschreibung
@Entity	Gibt an, dass die Klasse eine Entität ist.
@Id	Gibt den Primärschlüssel einer Entität an.
@Version	Gibt das Versionsfeld oder die Eigenschaft einer Entitätsklasse an, die als ihr optimistischer Sperrwert dient.
@GeneratedValue()	Ermöglicht die Spezifikation von Generierungsstrategien für die Werte von Primärschlüsseln.
strategy=GenerationType.AUTO	Der GenerationType definiert die Arten der Primärschlüssel-Generierungsstrategien. AUTO zeigt an, dass der Persistenzanbieter eine geeignete Strategie für die jeweilige Datenbank wählt, um die Nummerierung der ID automatisch zu generieren.
@ManyToOne	Gibt eine Einzelwert-Zuordnung zu einer anderen Entitätsklasse an, die eine viele-zu-eins-Multiplizität aufweist.
@OneToMany	Gibt eine many-valued Assoziation mit einer Eins-zu-viele-Multiplizität an.
cascade=CascadeType.MERGE	Der CascadeType definiert die Menge der kaskadierbaren Operationen, die auf die zugehörige Entität übertragen werden. MERGE signalisiert dabei eine Kaskadenverschmelzung. Das bedeutet in dem Beispiel, dass wenn ein Game Objekt aktualisiert wird, erfolgt die Aktualisierung auch auf dem Player Objekt.
cascade=CascadeType.ALL	ALL signalisiert, dass alle Operationen kaskadiert werden. Dazu gehört z.B. das Erstellen, Löschen und Aktualisieren. Das bedeutet in dem Beispiel, dass wenn ein Game Objekt aktualisiert, gelöscht oder

	erstellt wird, erfolgen die gleichen Operationen auch für die verbundenen Runden Objekte.
fetch=FetchType.LAZY	Der FetchType definiert Strategien zum Abrufen von Daten aus der Datenbank. Lazy signalisiert, dass die damit verbundenen Objekte nur bei Bedarf von der Datenbank abgerufen werden.
@NamedQuery	Gibt eine statisch, benannte Abfrage in der Java Persistence Abfragesprache an. Das Attribut name spezifiziert den Namen der Abfrage und query die Abfrage selbst.

Tabelle 11 Annotationsbeschreibungen

Es existieren weitaus mehr Annotationen, welche in der JPA Dokumentation<sup>7</sup> eingesehen werden können.

## 5.4 Picocontainer

Bei Picocontainer handelt es sich um ein Framework, mit welchem in diesem Projekt die Dependency Injection durchgeführt wird.

### Konfiguration:

Um das Picocontainer Framework zu nutzen, muss dessen Dependency in die pom.xml des Maven-Projektes eingebunden werden. Im Anschluss kann eine MutablePicoContainer Instanz, wie in Abbildung 18 beispielhaft dargestellt, erzeugt und dieser die voneinander abhängigen Komponenten als Klasse übergeben werden.

```

79④ private PlayerManagement_Service_Impl registerComponents() {
80     MutablePicoContainer container = new DefaultPicoContainer(new ConstructorInjection());
81     EntityManager em = EntityManagerService.getInstance().getEntityManagerFactory().createEntityManager();
82     container.addComponent(PlayerManagementDaoImpl.class);
83     container.addComponent(PlayerManagement_Service_Impl.class);
84     container.addComponent(em);
85     return container.getComponent(PlayerManagement_Service_Impl.class);
86 }
```

Abbildung 18 Picocontainer

In dem Projekt wird der Picocontainer mithilfe einer ConstructorInjection realisiert. Dies bedeutet, dass jede Klasse, welche in Abhängigkeit zu einer anderen steht, die Injektion über den Konstruktor realisiert. Weiterhin könnte die Dependency Injection durch

**SetterInjection:** Instanziert Komponenten mit leeren Konstruktoren über die entsprechenden Setter-Methoden.

**NameFieldInjection:** Instanziert Komponenten mit leeren Konstruktoren und injiziert weitere Komponenten anhand des Variablenamens.

**AnnotatedFieldInjection:** Instanziert Komponenten mit leeren Konstruktoren und injiziert die Abhängigkeiten automatisch durch gesetzte @Inject Annotationen.

realisiert werden. Mithilfe der getComponent Methode können die einzelnen Komponenten im Anschluss aufgerufen werden.

<sup>7</sup> <https://docs.oracle.com/javaee/7/api/javax/persistence/package-summary.html>

## 5.5 Spring

Mithilfe des Spring Framework wurde in diesem Projekt der Restserver implementiert und das Exception handling für die Restschnittstelle realisiert.

### Konfiguration:

Um das Spring Framework zu nutzen, muss dessen Dependency in die pom.xml des Maven-Projektes eingebunden werden. Spring bietet mithilfe des spring initializr<sup>8</sup> einen komfortablen Weg, um die richtigen Dependencies für sein Projektsetup zu finden. Die Initialisierung für den Restserver wird, wie in Abbildung 19 dargestellt, mithilfe der @SpringBootApplication Annotation und der SpringApplication Klasse realisiert.

```
6  @SpringBootApplication
7  public class RestAdapterApplication {
8
9    public static void main(String[] args) {
10      SpringApplication.run(RestAdapterApplication.class, args);
11    }
12
13 }
```

Abbildung 19 Spring SpringApplication

Durch die @SpringBootApplication Annotation wird eine Auto-Konfiguration durchgeführt und der Komponenten-Scan für das Package, in welchem sich die Klasse befindet, aktiviert. Die Auto-Konfiguration lässt den Server später via *localhost* über den Port 8080 starten. Durch die Komponenten-Scan Option überprüft Spring alle Spring-Annotationen der Package-Klassen und berücksichtigt die entsprechenden Anweisungen.

Insgesamt wurden drei Rest-Controller Klassen (PlayerManagement\_Service\_Controller, VocabularyManagement\_Service\_Controller und GameManagement\_Service\_Controller) in dem Package implementiert. In Abbildung 20 wird exemplarisch ein Auszug der PlayerManagement\_Service\_Controller Klasse dargestellt.

```
38 @RestController
39 @RequestMapping("/player_service")
40 public class PlayerManagement_Service_Controller {
41
42
43    @PutMapping("/create")
44    public Player createPlayer(@RequestParam String name) throws DuplicatePlayerException {
45      PlayerManagement_Service pms = registerComponents();
46      Player player;
47      player = pms.createPlayer(name);
48      return player;
49    }
50
51    @GetMapping("/get")
52    public List<Player> getAllPlayer() {
53      PlayerManagement_Service pms = registerComponents();
54      List<Player> player = pms.getAllPlayer();
55      return player;
56    }
57
58    @DeleteMapping("/delete/{id}")
59    public boolean deletePlayer(@PathVariable("id") int PlayerID) throws PlayerNotFoundException {
60      PlayerManagement_Service pms = registerComponents();
61      boolean result = pms.deletePlayer(PlayerID);
62      return result;
63    }
64
65    @PatchMapping("/edit")
66    public Player editPlayerName(@RequestBody Map<String, Object> payload) throws DuplicatePlayerException, PlayerNotFoundException {
67      ObjectMapper objectMapper = new ObjectMapper();
68      PlayerManagement_Service pms = registerComponents();
69      Player player = objectMapper.convertValue(payload.get("player"), Player.class);
70      String name = objectMapper.convertValue(payload.get("name"), String.class);
71
72      Player result = pms.editPlayerName(player, name);
73      return result;
74    }
}
```

Abbildung 20 Spring RestController

<sup>8</sup> <https://start.spring.io>

Die Rest-Controller Klassen werden jeweils mit einer `@RestController` Annotation markiert. Weiterhin erhält jede Klasse eine `@RequestMapping` Annotation, welche mit ihrem Parameter `value` den URL Pfad für die Klasse bestimmt. Zusätzlich wird an jeder Methode eine entsprechende `Mapping` Annotation mit einer jeweiligen URL Erweiterung gesetzt, wobei der Präfix der Annotation die HTTP-Methode signalisiert. Die erwarteten Parameter können dabei in verschiedenen Varianten bestimmt werden.

<code>@RequestBody :</code>	Methodenparameter, welcher aus dem Body des Web Request entnommen wird. Meist komplexe Strukturen.
<code>@PathVariable :</code>	Methodenparameter ist an eine URL Vorlagenvariable gebunden. Diese ist durch zwei geschweifte Klammern und einem Namen im URL Pfad angegeben.
<code>@RequestParam :</code>	Methodenparameter ist an den Request Parameter gebunden.

In demselben Package befindet sich in unserer praktischen Ausarbeitung die `ErrorHandler` Klasse, welche auszugsweise in Abbildung 21 dargestellt wird.

```

22 @ControllerAdvice
23 public class ErrorHandler {
24
25     @ExceptionHandler({ PlayerNotFoundException.class, NoVocabularyListFoundException.class,
26                         NoVocabularyListFoundException.class, NoWordNotFoundException.class,
27                         GameNotFoundException.class })
28     public void handleNotFound(HttpServletRequest response, Exception e) throws IOException {
29         response.sendError(HttpStatus.NOT_FOUND.value(), e.getMessage());
30     }
31
32     @ExceptionHandler({ GameNotCreatableException.class })
33     public void handleNotCreatable(HttpServletRequest response, GameNotCreatableException e) throws IOException {
34         response.sendError(HttpStatus.UNPROCESSABLE_ENTITY.value(), e.getMessage());
35     }
36
37     @ExceptionHandler({ DuplicatePlayerException.class, DouplicateVocabularyListException.class,
38                         OnePlayerTwoTimesInGameException.class })
39     public void handleConflict(HttpServletRequest response, Exception e) throws IOException {
40         response.sendError(HttpStatus.CONFLICT.value(), e.getMessage());
41     }
42

```

Abbildung 21 Spring Error handling

Mithilfe dieser Klasse werden auftretende `Exceptiones` global abgefangen und mit entsprechenden Error-Codes und Nachrichten als Response an den Client zurückgesandt. Die Konfiguration erfolgt durch die `@ControllerAdvice` und `@ExceptionHandler` Annotationen. Die `@ControllerAdvice` Annotation signalisiert Spring, dass in dieser Klasse die `Exceptiones` global abgefangen werden. Die `@ExceptionHandler` Annotation erwartet eine einzelne bzw. mehrere `Exceptiones` als Parameter. Sie wird an eine Methode gesetzt, welche ausgeführt werden soll, sobald eine entsprechende Exception geworfen wird. Die Methode kann beispielsweise `Exceptiones`- und ein `HttpServletResponse`- Objekt übergeben bekommen.

## 5.6 Retrofit2

Das Retrofit2 Framework wird in dieser Arbeit für die Realisierung des Clients genutzt, um HTTP-Requests an den Rest-Server zu senden und deren Respons zu empfangen. Dabei übernimmt Retrofit2 die Serialisierung und Deserialisierung.

## Konfiguration:

Um das Retrofit2 Framework zu nutzen, muss dessen Dependency in die pom.xml des Maven-Projektes eingebunden werden. Weiterhin wird für jeden Client-Service ein Interface erstellt. Ein Beispiel dafür wird in Abbildung 22 dargestellt.

```
16 public interface PlayerRestService {  
17     @GET("/player_service/get/{id}")  
18     Call<Player> getPlayer(@Path("id") int id);  
19  
20     @PUT("/player_service/create")  
21     Call<Player> createPlayer(@Query("name") String name );  
22  
23     @DELETE("/player_service/delete/{id}")  
24     Call<Boolean> deletePlayer(@Path("id") int id);  
25  
26     @PATCH("/player_service/edit")  
27     Call<Player> editPlayerName(@Body Map<String, Object> map);  
28  
29     @GET("/player_service/get")  
30     Call<List<Player>> getAllPlayer();  
31 }
```

Abbildung 22 Retrofit2 Interface

Die Methodenköpfe werden mit einer entsprechenden HTTP-Methode und dem entsprechenden URL Pfad annotiert. Weiterhin werden die zu sendenden Parameter, entsprechend den zu erwartenden Parametern auf der Serverseite, mit einer Annotation versehen. @Body entspricht dabei der Spring @RequestBody, @Path der Spring @PathVariable und @Query der Spring @RequestParam Annotation. Weiterhin werden die gewünschten Return-Werte als Typ einer Call Variable bestimmt.

```
29 public class PlayerClientAdapter implements PlayerManagement_Service {  
30  
31     private Retrofit rf = new Retrofit.Builder().baseUrl("http://localhost:8080")  
32         .addConverterFactory(ScalarsConverterFactory.create()).addConverterFactory(GsonConverterFactory.create())  
33         .build();  
34  
35     private PlayerRestService restService = rf.create(PlayerRestService.class);  
36     private Gson gson = new Gson();  
37  
38     @Override  
39     public Player createPlayer(String name) throws DuplicatePlayerException {  
40         Call<Player> call = restService.createPlayer(name);  
41         Response<Player> response = null;  
42         Player player = null;  
43  
44         try {  
45             response = call.execute();  
46  
47             if(response.isSuccessful()) {  
48                 player = response.body();  
49             } else if(response.code() == 409) {  
50                 throw new DuplicatePlayerException(name);  
51             } else if(response.code() == 500) {  
52                 ErrorMessage message = gson.fromJson(response.errorBody().charStream(),ErrorMessage.class);  
53                 throw new PlayerManagementServiceException(message.getMessage());  
54             }  
55  
56         } catch (IOException e) {  
57             throw new GuiIOException(e);  
58         }  
59         return player;  
60     }
```

Abbildung 23 Retrofit2 Builder

In der ClientAdapter Klasse, siehe Abbildung 23, wird ein Retrofit Objekt instanziert, wobei die Rest-Server URL sowie dieConverterFactory bestimmt wird. Mit dem erstellten Retrofit Objekt wird im Anschluss durch die *create* Methode ein Objekt des zuvor erstellten Interface PlayerRestService erstellt. Mit diesem werden im Anschluss, in der jeweiligen Methode, die Requests an den Server gesandt und die Antwort als Response zurückerhalten, sowie ausgewertet.

## 5.7 JavaFX

In dieser Arbeit wurde JavaFX zu Erstellung der grafischen Oberfläche verwendet. Die Gestaltung der Oberfläche wurde mit dem JavaFX Scene Builder 2.0 realisiert.

### Konfiguration:

JavaFX benötigt eine Installation der JavaFXSDK. Diese wurde für dieses Projekt über den Eclipse IDE Marketplace realisiert. Weiterhin wurde der JavaFX Scene Builder in Version 2.0 installiert<sup>9</sup>, mit welchem die grafische Oberfläche per Drag and Drop erstellt wurde. Der Scene Builder wird beispielhaft mit geöffneter Player-View in Abbildung 24 dargestellt.

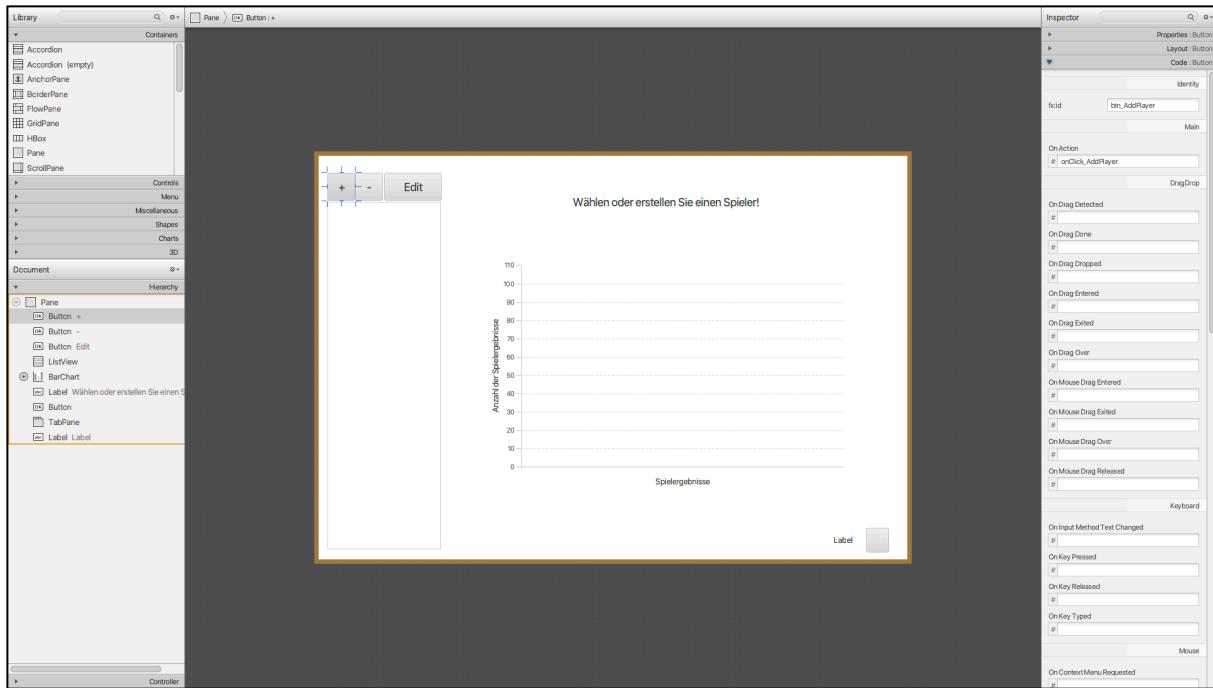


Abbildung 24 JavaFX Scene Builder 2.0

Am linken Rand befindet sich eine Liste von GUI Elementen, welche per Drag and Drop auf die Pane-Fläche in der Mitte gezogen werden können. Am rechten Rand kann jedes Element genauer konfiguriert werden. Dabei ist es beispielsweise möglich den Style zu verändern oder ActionListener-Bezeichnungen zu definieren. Die erstellte Oberfläche kann im Anschluss als fxml Datei abgespeichert und mit einer Controller Klasse verknüpft werden. Das „Skelett“ der Controller Klasse kann mit dem Scene Builder über die Funktion *Sample Skeleton* automatisch generiert werden. In dieser befinden sich dann eine entsprechende Deklaration aller UI Elemente sowie der Listener (siehe Ausschnitt in Abbildung 25).

<sup>9</sup> <https://www.oracle.com/java/technologies/javafx-scene-builder-source-code.html>

```

72 @FXML
73 private Button btn_EditPlayer;
74
75 @FXML
76 private CategoryAxis bar_xAxis;
77
78 @FXML
79 private Button updateButton;
80
81 @FXML
82 private Label informationLabel;
83
84
85 void updatePlayerView(ActionEvent event) {
86     setSpinningUpdateButton(true, "Die Spieler werden geladen ... ");
87     GetAllPlayerWorker worker = new GetAllPlayerWorker();
88     setOnGetAllPlayerUpdateWorkerDone(worker);
89     Thread workerThread = new Thread(worker);
90     workerThread.start();
91 }

```

Abbildung 25 JavaFX Controller

In dem gezeigten Ausschnitt der PlayerController Klasse sind im oberen Bereich die UI Elemente als Objekte zu erkennen. Im unteren Bereich ist die Action Methode *updatePlayerView* zu sehen, welche mit dem updateButton durch den Scene Builder verknüpft wurde. Alle Objekte und Methoden, welche im direkten Zusammenhang mit den UI Elementen der fxml Datei stehen, wurden mit der @FXML Annotation versehen.

Damit die Anwendung gestartet werden kann, erbt die App Klasse mit der main Methode von Application, siehe Abbildung 26.

```

16 public class App extends Application {
17
18
19 public void start(Stage primaryStage) throws IOException {
20     primaryStage.show();
21     Parent mainPane = FXMLLoader.load(App.class.getResource("/Main.fxml"));
22
23     primaryStage.setScene(new Scene(mainPane));
24     primaryStage.setResizable(false);
25
26     primaryStage.setTitle("Vocabulary Trainer");
27 }
28
29 public static void main(String[] args) {
30     Application.launch(args);
31 }
32 }
33

```

Abbildung 26 App Klasse mit main Methode

Die *start* Methode wird als *Entry Point* Methode durch die *Application* Klasse importiert und kann entsprechend der gewünschten Implementierung ausprogrammiert werden. In dem dargestellten Beispiel wird mithilfe des *FXMLLoader* die *Main.fxml* Datei geladen und dem *primaryStage* übergeben. Dieser wird vorab durch die *show* Methode sichtbar gemacht.

## 6 Ablaufumgebung

Die Ablaufumgebung der Anwendung wird in Abbildung 27 dargestellt.

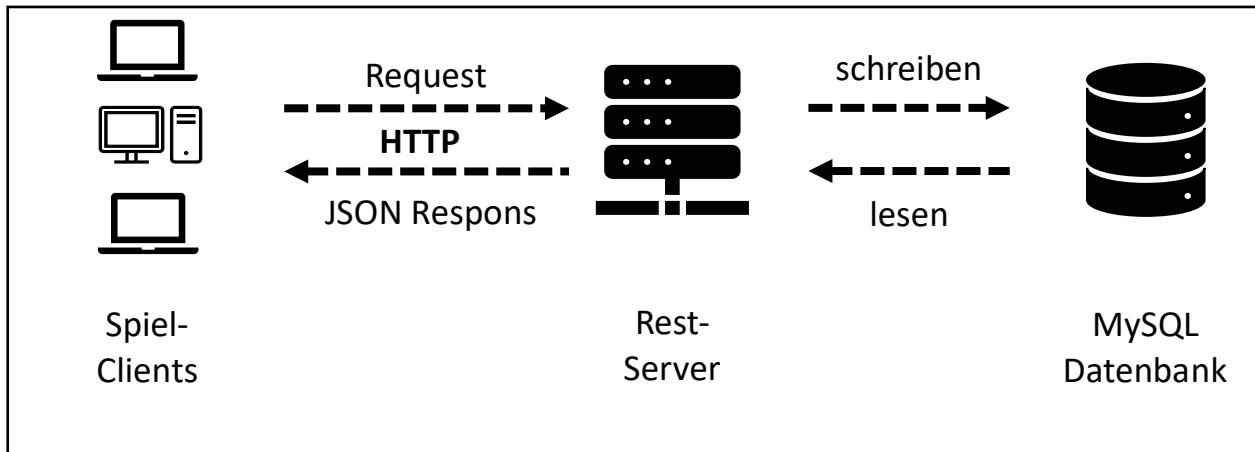


Abbildung 27 Ablaufumgebung

Der Rest-Server dient in der Anwendung als zentrales Element. Dieser nimmt die eintreffenden HTTP Requests der Clients entgegen, arbeitet diese mit einem schreibenden und lesenden Zugriff auf der MySQL Datenbank ab und sendet eine HTTP JSON Respons als Antwort an den entsprechenden Client zurück. Bei dem Vokabel-Trainer-Spiel handelt es sich um eine Mehrbenutzeranwendung. Das bedeutet, dass eine Vielzahl an Clients gleichzeitig Anfragen an den Server senden können und den aktuellen State der Datenbank als Antwort bekommen.