Dokumentation der Eigenleistung: Safae Kartite (6535706)

1. Beschreibung der Funktion:

Die Funktionalität, die ich entwickelt habe, umfasst das Speichern von Spielerdaten (Name, Punktestand und Rank) in der Datenbank. Im Frontend werden die Spielerinformationen erfasst und berechnet, bevor sie an das Backend gesendet werden, welches die Daten speichert.

2. Die Einbindung, Funktionalität und der Quellcode :

- Voraussetzungen:

Zunächst werden die Namen der Spieler als Formulareingaben auf der Startseite (*startpage.html*) angegeben. Anschließend werden diese an die Logik von *game.js* weitergeleitet, sodass jedem Spieler Versuche und Punkte zugewiesen werden.

- Frontend-Komponente: *saveScores()* Funktion:

```
async saveScores() {
    let calculateScorePlayer1 = this.player1.score / this.player1.attempts *
10;
    let calculateScorePlayer2;
    if (this.player2) {
        calculateScorePlayer2 = this.player2.score / this.player2.attempts *
10;
    }
    const url = 'http://localhost:8080/player';

    let scores;

    if (this.player2) {
        scores = [
            {name: this.player1.name, score: calculateScorePlayer1},
            {name: this.player2.name, score: calculateScorePlayer2}
        ];
    } else {
        scores = [
            {name: this.player1.name, score: calculateScorePlayer1}
        ];
    }


    for (const player of scores) {
        const response = await fetch(url, {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json'
            },
            body: JSON.stringify(player)
        });
        if (!response.ok) {
            console.error('Failed to save player score');
        }
    }

    window.location.href = 'scores.html';
}
```

1. Punktzahlberechnung:
    - Die Funktion berechnet zuerst die Punktzahl von Spieler 1. Diese wird berechnet, indem der aktuelle Punktestand (this.player1.score) durch die Anzahl der Versuche (this.player1.attempts) geteilt und anschließend mit 10 multipliziert wird.
    - Falls es einen zweiten Spieler gibt (this.player2), wird dessen Punktzahl auf die gleiche Weise berechnet.

2. Die URL für den Backend-Endpunkt wird in der Variablen url gespeichert. Dies ist der Endpunkt, an den die Spielergebnisse gesendet werden.

3. Ein Array scores wird erstellt, das die berechneten Punktzahlen der Spieler enthält. Falls es einen zweiten Spieler gibt, wird das Array mit den Punktzahlen beider Spieler gefüllt. Falls nur ein Spieler vorhanden ist, wird das Array nur mit dessen Punktzahl gefüllt.

4. Senden der Daten an das Backend:
    - Die Funktion durchläuft das scores-Array und sendet die Daten jedes Spielers einzeln an das Backend.
    - Dafür wird die fetch-API verwendet, um eine HTTP-POST-Anfrage an die zuvor definierte URL zu senden.
    - Die Anfrage enthält die Spielerinformationen im JSON-Format im Anfragetext (body).

5. Fehlerbehandlung:
    - Nach jeder Anfrage wird überprüft, ob die Antwort des Servers (response) erfolgreich war (response.ok).
    - Falls nicht, wird eine Fehlermeldung in der Konsole ausgegeben.

6. Nachdem alle Spielergebnisse erfolgreich gesendet wurden, wird der Benutzer zur scores.html Seite weitergeleitet, auf der die aktuellen Punktzahlen und Ranglisten angezeigt werden.

2. Backend-Komponente:

PlayerService Klasse: createPlayer Funktion

- Eingabedaten: Nimmt ein Objekt PlayerEntity als Parameter (newPlayer), das die Daten des neuen Spielers enthält.
- Rückgabewert: Gibt das gespeicherte PlayerEntity-Objekt zurück.
- Prüft, ob der Name (newPlayer.getName()) oder die Punktzahl (newPlayer.getScore()) null ist. Falls ja, wird eine InvalidArgumentException ausgelöst.
- Wenn die Eingabedaten valide sind, wird der Spieler in der Datenbank gespeichert. Verwendet dafür das playerRepository und dessen save-

Methode. Gibt das gespeicherte PlayerEntity-Objekt zurück, das die vom Datenbankmanagementsystem generierte ID und die gespeicherten Daten enthält.

```
public PlayerEntity createPlayer(PlayerEntity
newPlayer) throws InvalidArgumentException{
    if (newPlayer.getName()==null ||
newPlayer.getScore()==null){
        throw new InvalidArgumentException();
    }
    return playerRepository.save(newPlayer);
}
```

PlayerController Klasse: POST-Endpoint

- Annotations: Definiert den HTTP-POST-Endpunkt unter der URL '/player'.Und gibt an, dass die Methode JSON-Daten im Anforderungstext konsumiert.
- Das PlayerEntity-Objekt wird aus dem JSON-Body der Anfrage extrahiert.
- Übergibt das PlayerEntity-Objekt an die createPlayer-Methode des PlayerService, um den Spieler zu erstellen und zu speichern.
- Gibt eine ResponseEntity mit dem gespeicherten PlayerEntity-Objekt und dem HTTP-Status CREATED (201) zurück.
- Falls eine InvalidArgumentException ausgelöst wird, gibt die Methode eine ResponseEntity mit dem HTTP-Status BAD_REQUEST (400) zurück.

```
@PostMapping(path = "/player", consumes =
MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<PlayerEntity>
createPlayer(@RequestBody PlayerEntity newPlayer){
    try {
        PlayerEntity createdPlayer =
playerService.createPlayer(newPlayer);
        return new ResponseEntity<>(createdPlayer,
HttpStatus.CREATED);
    } catch (InvalidArgumentException e){
        return new
ResponseEntity<>(HttpStatus.BAD_REQUEST);
    }
}
```

3. Zusammenfassung:

Eingabe der Spielernamen (startpage.html) → Übertragung an game.js→ Aufzählung der Versuche und Berechnung der Punkte (game.js) → Senden an das Backend → Speichern in der Datenbank (PlayerService & PlayerController)

Dokumentation der Eigenleistung: Paula Kropfinger (8118206):

Nach beenden des Spiels (Timeout oder alle Paare aufgedeckt) wird der Spieler auf die Score Page weitergeleitet, dort werden alle in der Datenbank gespeicherten Spielerinformationen (Name, Rank, Score) in einer Tabelle ausgegeben.

Ich habe die Funktion für das Auslesen der in der Datenbank gespeicherten Werte und die Ausgabe dieser Werte auf der Score Seite unseres Spiels gemacht. Im Backend werden die Daten von allen vorhandenen Spielern ausgelesen und sortiert. Im Frontend wird eine Tabelle generiert, um die Daten anzuzeigen. Die Funktion wird beim Öffnen der Webseite Score Page aufgerufen.

Frontend Code:

Zur Funktion gehört deshalb der Code zum Generieren der Tabelle:

scores.html:

```
<div class="container">
    <h1>Scores</h1>
    <table id="scoreTable">
        <thead>
        <tr>
            <th>Rank</th>
            <th>Name</th>
            <th>Score</th>
        </tr>
        </thead>
        <tbody>
        </tbody>
    </table>
</div>
```

- **<thead>**: Der Tabellenkopf mit einer Zeile (<tr>) und drei Spalten (<th>) für "Rank", "Name" und "Score".
- **<tbody>**: Der Tabellenkörper, in den die Datenzeilen eingefügt werden.

Daten abrufen:

Scores.js:

```
const fetchPlayerData = async () => {
    const url = 'http://localhost:8080/player';

    try {
        const response = await fetch(url);
        if (!response.ok) {
            throw new Error('Failed to fetch player data');
        }
        const data = await response.json();
        return data;
    } catch (error) {
        console.error('Error fetching player data:', error);
        return []; // Return empty array in case of error
    }
};
```

- Die fetch-Funktion wird verwendet, um Daten von der angegebenen URL abzurufen.
- Wenn die Antwort nicht erfolgreich ist (!response.ok), wird eine Fehlermeldung erzeugt.
- Die Antwort wird mit response.json() in JSON-Format umgewandelt und zurückgegeben.
- Im Fehlerfall wird der Fehler mit console.error ausgegeben und ein leeres Array [] zurückgegeben.

und Daten in der Tabelle anzeigen:

Scores.js:

```
const populateTable = async () => {
    const tableBody = document.querySelector('#scoreTable tbody');
    const playerData = await fetchPlayerData();


    playerData.forEach(item => {
        let row = document.createElement('tr');
        row.innerHTML =
`<td>${item.rank}</td><td>${item.name}</td><td>${item.score}</td>`;
        tableBody.appendChild(row);
    });
};

// Execute populateTable function when the DOM is fully loaded
document.addEventListener('DOMContentLoaded', () => {
    populateTable();
});
```

- Asynchrone Funktion „populateTable“, die keine Parameter erwartet.
- Selektiert das tbody-Element innerhalb der Tabelle mit der ID scoreTable.
- Ruft die Funktion fetchPlayerData auf, um die Spielerdaten asynchron zu laden und in der Variablen playerData zu speichern.
- Schleife zur Erstellung der Tabellenzeilen:
    - Iteriert über jedes Element (item) in playerData
    - Für jedes item wird eine neue Zeile (tr-Element) erstellt.
    - Der Inhalt der Zeile wird mit den Daten (rank, name, score) des jeweiligen Spielers gefüllt.
    - Die erstellte Zeile wird dem tbody der Tabelle hinzugefügt
- Fügt einen Eventlistener hinzu, der populateTable() aufruft, wenn das DOM vollständig geladen ist.
- Dies stellt sicher, dass die Tabelle mit den Spielerdaten erst dann befüllt wird, wenn das gesamte HTML-Dokument geladen und bereit ist.

sowie der Backend Code:

PlayerService.java

```
public List<PlayerEntity> getAllPlayersWithRanks() {
    List<PlayerEntity> players = playerRepository.findAll();
    List<PlayerEntity> sortedPlayers = players.stream()

.sorted(Comparator.comparingInt(PlayerEntity::getScore).reversed())
            .collect(Collectors.toList());

    for (int i = 0; i < sortedPlayers.size(); i++) {
        sortedPlayers.get(i).setRank(i + 1);
    }
    return sortedPlayers;
}
```

Methode „getAllPlayersWithRanks“, die eine Liste von Spielerobjekten (PlayerEntity) zurückgibt, wobei jeder Spieler basierend auf seiner Punktzahl (Score) sortiert und mit einem Rang versehen wird:

- „playerRepository.findAll()“ ruft alle PlayerEntity-Objekte ab und speichert sie in einer Liste namens players
- Die Spieler werden nach ihrer Punktzahl in aufsteigender Reihenfolge sortiert. „reversed()“ kehrt diese Reihenfolge um, sodass die Spieler in absteigender Reihenfolge nach ihrer Punktzahl sortiert werden (höchste Punktzahl zuerst).
- Eine for-Schleife iteriert durch die sortierte Liste sortedPlayers.
- Für jeden Spieler in der sortierten Liste wird der Rang gesetzt. Der Rang ist der Index des Spielers in der Liste plus eins (da Listenindizes bei 0 beginnen, die Ränge jedoch bei 1 beginnen sollen).

PlayerController.java:

```
@GetMapping(path="/player", produces = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<List<PlayerEntity>> getAllPlayers(){
    List<PlayerEntity> allPlayersWithRanks =
playerService.getAllPlayersWithRanks();
    return new ResponseEntity<>(allPlayersWithRanks, HttpStatus.OK);
}
```

- Methode, die eine HTTP GET-Anfrage auf den Pfad „player“ verarbeitet und eine Liste von PlayerEntity-Objekten im JSON-Format zurückgibt.
- Ruft die Methode „getAllPlayersWithRanks()” aus dem playerService auf, um alle Spielerdaten mit Rängen abzurufen. Das Ergebnis wird in „allPlayersWithRanks“ gespeichert.
- Erstellt eine neue ResponseEntity, die die Liste der Spieler (allPlayersWithRanks) und den HTTP-Status OK (200) enthält. HttpStatus.OK signalisiert, dass die Anfrage erfolgreich war.

Zusammenfassung

Spieler gewinnt (alle Paare aufgedeckt) / verliert (Timeout) -> Weiteleitung score page -> Player Data vom Backend abrufen -> Tabelle mit Spielern sortiert nach Rank wird geladen und dann ausgegeben