

# Racketsports Manager

Semesterarbeit ZHAW



**Autor: Raphael Marques**

**Betreuer: Michael Reiser**

9. August 2015

# Inhaltsverzeichnis

<b>1</b>	<b>Inhalt der Arbeit</b>	<b>4</b>
1.1	Ausgangslage . . . . .	4
1.2	Aufgabenstellung . . . . .	4
1.3	Erwartete Resultat . . . . .	5
1.4	Zielsetzung der Arbeit . . . . .	5
1.4.1	Grobplanung . . . . .	6
1.4.2	Termine . . . . .	6
<b>2</b>	<b>Analyse</b>	<b>7</b>
2.1	Marktumfeld . . . . .	7
2.1.1	Identifizierung von Partnern . . . . .	7
2.1.2	Terminvereinbarung . . . . .	8
2.1.3	Amateur-Liga Management . . . . .	8
2.1.4	Protokollierung eines Spiels . . . . .	8
2.2	Benutzergruppen/Ist-Analyse . . . . .	8
2.2.1	Gelegenheitsspieler . . . . .	9
2.2.2	Regelmässige Spieler . . . . .	9
2.2.3	Clubmitglieder . . . . .	10
2.3	Anforderungsanalyse . . . . .	12
2.3.1	Vision . . . . .	12
2.3.2	Ziele . . . . .	12
2.3.3	Rahmenbedingungen . . . . .	12
2.3.4	Schnittstellen . . . . .	13
2.3.5	Anwendungsfälle . . . . .	13
2.3.6	Anforderungen . . . . .	27
<b>3</b>	<b>Konzeption</b>	<b>30</b>
3.1	Technologiestack . . . . .	30
3.1.1	MVC Frameworks . . . . .	31
3.1.2	Kontext . . . . .	36
3.1.3	Businessschicht . . . . .	38
3.1.4	Präsentationsschicht . . . . .	39

<b>4</b>	<b>Implementation</b>	<b>42</b>
4.1	Projektaufbau . . . . .	42
4.1.1	app . . . . .	42
4.1.2	Documentation . . . . .	43
4.1.3	public . . . . .	43
4.2	REST API . . . . .	43
4.2.1	Routing . . . . .	44
4.2.2	Spiel Endpunkt /matches . . . . .	44
4.2.3	Liga Endpunkt . . . . .	46
4.2.4	Racketsportzentrum Endpunkt /courts . . . . .	48
4.2.5	Benutzer Endpunkt /users . . . . .	48
4.3	Web Applikation . . . . .	49
4.3.1	Google Maps Integration . . . . .	50
4.4	Android Applikation . . . . .	51
4.5	Workflows . . . . .	52
4.6	Allgemeine Workflows . . . . .	52
4.6.1	CRUD für Datenobjekte . . . . .	52
4.7	Court . . . . .	53
4.7.1	Court Registrierung . . . . .	53
4.8	Liga . . . . .	54
4.8.1	Liga Registrierung . . . . .	54
4.8.2	Automatische Herausforderung Liga . . . . .	55
4.9	Match . . . . .	59
4.9.1	Match Workflow . . . . .	59
4.10	User . . . . .	61
4.10.1	Freunde System . . . . .	61
<b>5</b>	<b>Test</b>	<b>62</b>
5.1	Unit Tests . . . . .	62
5.2	System Tests . . . . .	62
5.3	User Acceptance Tests . . . . .	62
<b>6</b>	<b>Reflektion</b>	<b>63</b>

# 1 Inhalt der Arbeit

Racket-Sportarten haben ein Problem. Man muss die richtige Zeit und den richtigen Partner finden. Die Applikation, welche ich für die Semester-Arbeit erstelle sollte dieses Problem lösen. Durch einen "Doodle-Style Termin und Spieler-Finder sowie eine Liga-Verwaltung werden die Teilnehmenden motiviert, öfters zu squashen.

## 1.1 Ausgangslage

Einen Partner für Racket-Sportarten zu finden, ist nicht immer sehr einfach. Wenn man schliesslich jemanden gefunden hat, ist es immer schwer, einen Termin zu finden. Spielt man regelmässig gegen die gleiche Person, stellt man sich oft auf diese ein und lernt mit der Zeit nur noch wenig dazu.

Diese App sollte diese Probleme lösen, indem man einfach miteinander Termine vereinbaren kann und das Termine vom System für die User automatisch generiert werden.

## 1.2 Aufgabenstellung

Um die Ziele zu erreichen, muss methodisch vorgegangen werden. In der Dokumentation müssen angewendete Technologien begründet werden sowie Konzepte und Architekturen der Applikation festgehalten werden.

1. Informieren über den momentanen Markt und Organisation rund um Racketsport
  - Andere Applikationen rund um Racketsport identifizieren
  - Zielgruppen- und Anwendungsfallanalyse
2. Anforderungsanalyse, welche Funktionalitäten dem Nutzer einen echten Mehrwert bieten
  - Basierend auf (1) eine Anforderungsanalyse erstellen
  - User Stories erstellen
  - Sprints planen
3. Konzeption der App, Technologien wählen
  - Entscheidungsmatrix, welche Technologien verwendet werden
  - DB Design
  - Klassen-Diagramm
  - GUI-Entwurf
4. Die Applikation umsetzen

- Applikation programmieren
- Unit Tests erstellen/ausführen
- 5. Systemtests durchführen
  - Systemtests konzipieren
  - Systemtests durchführen
  - End-to-End Testing
- 6. Benutzertests durchführen
  - Benutzer einladen für Alpha-Test
  - Feedback von Benutzer verlangen

## 1.3 Erwartete Resultat

Folgende Resultate werden erwartet:

- Marktanalyse & Benutzerverhaltensanalyse zur vereinbarung von Spielen
- Anforderungsanalyse
- Recherche für einzusetzende Techniken
- Konzepte und Implementationstechniken
- User Stories und Projektplanung
- Android App und Webapplikation
- Testplan und Umsetzung
- Benutzerfeedback des Alpha-Tests
- Fazit

## 1.4 Zielsetzung der Arbeit

Ziel der Arbeit ist eine WebApplikation sowie eine Android App mit bestimmten Funktionalitäten zu erstellen. Der Autor der Arbeit soll so Zugang zu neuen Technologien im Webbereich erhalten und Erfahrung in der Webapplikations-Programmierung sammeln.

Aufgabenstellung:

- Dokumentation über Entscheidungen und Implementationstechniken
- Webapplikation mit folgender Funktionalität:
  - RestAPI um alle untenstehenden Funktionalitäten
  - Liga erstellen und löschen
  - Einer Liga beitreten und eine Liga verlassen
  - Spiel vereinbaren
  - Termin für Spiel finden
  - Spielresultat eintragen
  - Rangliste der Liga berechnen
  - Automatische Spielvorschläge innerhalb der Liga (z.B. jeder Spieler spielt jede zweite Woche ein Spiel)
- Android App, welche auf die WebAPI zugreift mit gleicher Funktionalität wie WebApplikation

### 1.4.1 Grobplanung

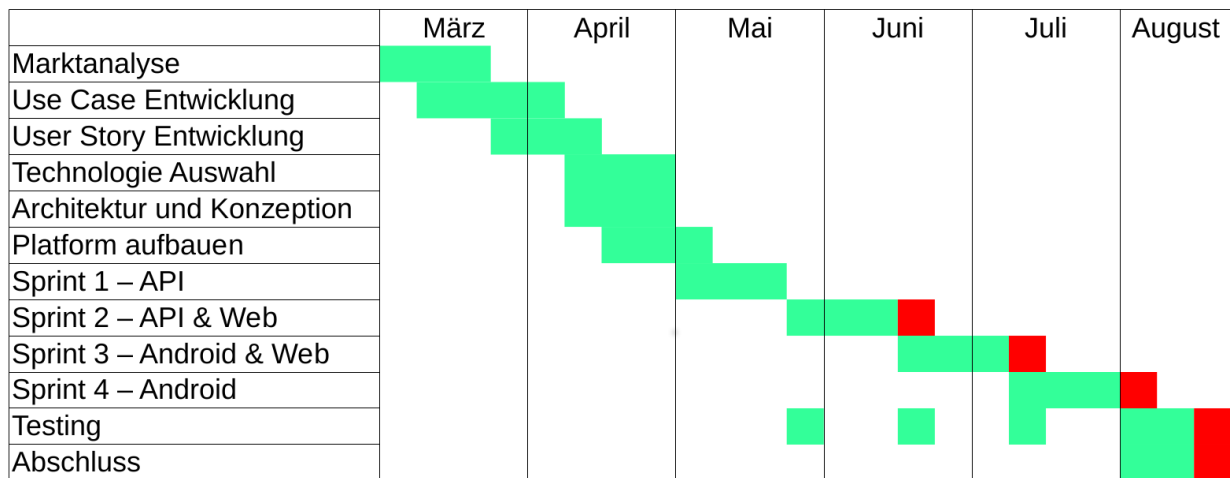


Abbildung 1.1: Grobplanung für Applikation

### 1.4.2 Termine

## 2 Analyse

Das Ziel der Software ist, den User in der Terminfindung, Protokollierung und Partnerfindung optimal zu unterstützen. Dieser Teil der Dokumentation dient zur Findung exakter Anforderungen an die Software, die den User optimal unterstützen. Zusätzlich wird Anfangs untersucht wie das Marktumfeld rund um die geplante Applikation aussieht, um einen möglichen Erfolg einer solchen Applikation zu schätzen sowie mögliche Synergieeffekte zu identifizieren.

### 2.1 Marktumfeld

Um mögliche Synergien oder Wettbewerber zu identifizieren, müssen zuerst die geplanten Basis Funktionalitäten aufgelistet werden:

- Vereinfachung zur Identifizierung von Partnern
- Vereinfachung zur Terminvereinbarung
- Vereinfachung eines Amateur-Liga Management
- Vereinfachung zu Protokollierung eines Spiels

#### 2.1.1 Identifizierung von Partnern

**GlobalTennisNetwork.com** ist eine Community zur Identifizierung von Tennispartnern. Die Website ist fokussiert auf Tennis. Andere Racket Sportarten werden nicht behandelt. Dadurch sieht der Autor dieser Service nicht als direkte Konkurrenz. Eine Verbindung mit dem Service um eine Grössere Population von Potenziellen Tennispartnern zu erreichen wäre vorstellbar.

**Spontacts** ist eine Android Applikation, welche Spontane Terminvereinbarungen ermöglicht. Es kann zusätzlich als Identifizierung von Partnern für jegliche Freizeitaktivitäten dienen. Es gibt jedoch keine Fokussierung auf Racket Sport. Nichts desto trotz kann Spontacts als konkurrenz zu der geplanten Applikation angesehen werden, da zwei Funktionalitäten (auch wenn der Fokus nicht auf Sport liegt teilweise abgedeckt werden können. Jedoch wird die Konkurrenz nicht als erheblich eingeschätzt.

**sport42.com** bietet auch eine Möglichkeit zur Identifizierung eines Potenziellen Sportpartners. Racket Sportarten werden hier auch abgedeckt. Nach Recherchen stellte sich jedoch heraus, das der Fokus der Applikation auf dem US Markt liegt. Bei einer potenziellen Expansion der geplanten Applikation müsse die Konkurrenz neu evaluiert werden. Da der Fokus

jedoch im Moment auf den deutschsprachigen Raum gerichtet ist, wird dieser Service nicht als Konkurrenz eingeschätzt.

**sportpartner.com** ist ein auf Sport fokussierter Service um Partner zu identifizieren. Der Service deckt auch Racketsportarten wie Tennis, Squash sowie Badminton ab. Bis jetzt gibt es allerdings nur wenig aktive Spieler in Europa.

### 2.1.2 Terminvereinbarung

**Doodle** ist der bekannteste Terminvereinbarungs Service. Ein ähnlicher Algorithmus, welcher in der geplanten Applikation verwendet wird, wurde von den Ersteller dieser Applikation entwickelt. Jedoch gibt es bei Doodle keinerlei Fokus auf Inhalt zum Termin. Es ist möglich Business wie auch Freizeit Termine zu erstellen. Die geplante Applikation kann davon profitieren einen ähnlichen, auf Sport Termine fokussierte Terminvereinbarungs Algorithmus zu entwickeln. Sollte dieser genügend intuitiv umgesetzt sein, wird Doodle nicht als Konkurrenz betrachtet.

**Moreganize.ch** wie Doodle benützt auch Moreganize einen Terminvereinbarungs Algorithmus.

### 2.1.3 Amateur-Liga Management

**LeagueManager Wordpress Plugin** ist eine Zusatzsoftware für Wordpress, mit welchen man Teams und Matches zusammenstellen kann und so Ligen erstellen kann. Der Fokus liegt hier grösstenteils auf Teamsports.

**Excel** von Microsoft wird oft als Tracking und Organisation für Ligen verwendet. Die Einfachheit und das breite Skillset spricht für diese Lösung. Jedoch ist Zusammenarbeit und Organisation umständlich.

### 2.1.4 Protokollierung eines Spiels

**Excel** von Microsoft wird oft als Tracking und Organisation für Ligen verwendet. Die Einfachheit und das breite Skillset spricht für diese Lösung. Jedoch ist Zusammenarbeit und Organisation umständlich. Als alternative, welche die Zusammenarbeit vereinfacht bietet sich auch **Google Docs** an.

## 2.2 Benutzergruppen/Ist-Analyse

Die geplante Applikation wird nicht für alle Benutzergruppen interessant sein. Um das Zielklientel zu finden wird erstellt diese Kapitel eine Kategorisierung in verschiedene Benutzergruppen. Anschliessend wird abgeschätzt, wie welche Benutzergruppe von der Applikation profitieren kann.



## 2.2.1 Gelegenheitsspieler

### 2.2.1.1 Analyse

Gelegenheitsspieler spielen nicht regelmässig (wöchentlich) Racketsport. Sie vereinbaren mit - meistens wenige Personen in engem oder erweitertem privaten Umfeld - unregelmässig Spiele. Oft pausieren Gelegenheitsspieler mehrere Monate und haben anschliessend intensivere Phasen mit mehreren Spielen innerhalb wenigen Wochen. Der Aufwand um ein Spiel zu vereinbaren und einen passenden Court zu finden ist dementsprechend gross. Viel fehlt auch an einem passenden Partner im privaten Umfeld, insbesondere wenn der Spieler Ambitionen hegt sich zu verbessern. Schlussendlich ist die Motivation zu einer „gewissen Regelmässigkeit“ nicht hoch, da die Vereinbarung eines Spieles aufwändig ist, sowie eine Verbesserung des Spiels gegenüber anderen Spielern nicht schnell ersichtlich ist.

Zusammengefasst lassen sich folgende Charakteristiken zusammenfassen:

- Einzelne oder wenige Partner im privatem Umfeld
- Oft längere Pausen gefolgt von intensiveren Phasen
- Grosse Aufwand um Spiel zu vereinbaren
- Fehlende Partner bei gewollter Verbesserung im Spiel
- Fehlende Vergleichsmöglichkeiten, da wenige Partner und meist keine Protokollierung der Ergebnisse
- Wenig Anreize um nächstes Spiel zu organisieren

### 2.2.1.2 Use Cases

Gelegenheitsspieler können ausserordentlich von der geplante Applikation profitieren.

Neue **Partner** in der Nähe können unkompliziert mit der Applikation identifiziert werden. So ist es möglich längere Pausen, wenn der originale Partner in den Ferien ist oder schlicht verhindert ist, zu verhindern.

**Courts und Spielzeiten** können schnell vereinbart werden. Dies verringert den Aufwand um ein Spiel durchzuführen und erhöht somit die Motivation öfters Squash zu spielen. Spiele können **protokolliert** werden und so kann eine Statistik erstellt werden, welche wiederum als Anreiz dienen kann um besser zu werden.

Durch ein **Liga Management** können zusätzliche Anreize, neue Partner sowie eine Regelmässigkeit gefunden werden. Möglicherweise sind ein Grossteil der Gelegenheitsspieler nicht an einer solchen Liga interessiert, jedoch gibt es ein gewisses Potenzial dafür.

## 2.2.2 Regelmässige Spieler

### 2.2.2.1 Analyse

Regelmässige Spieler spielen regelmässig zu einem vereinbarten Termin mit einem oder mehreren Partner einen Racketsport. Meist werden genannte vereinbarte Termine mit den gleichen Partnern ausgetragen. Bei den vereinbarten Terminen gibt es gewisse Variationen. So

kann der Zeitpunkt und auch der Wochentag variieren.

Zusätzlich zu den regelmässigen Terminen, kommen meist noch unregelmässige Termine in der Charakteristik der Gelegenheitsspieler hinzu.

Durch die Regelmässigkeit ist diese Spielergruppe meist einiges ambitionierter als Gelegenheitsspieler. Dadurch das Sie regelmässig mit den gleichen Spielern spielen, wollen Sie besser sein als die anderen. Da Sie jedoch nicht in einer Liga spielen gibt es meist kein Protokoll der Resultate oder ein Ranking.

#### 2.2.2.2 Use Cases

Regelmässige Spieler können durch die geplante Applikation sehr unkompliziert ihre Spiele **protokollieren**.

Eine **Liga** kann zwischen mehreren regelmässigen Spielern arrangiert werden. Diese kann durch regelmässige **Terminvereinbarungen vom System** dazu beitragen das es keine Pausen gibt bei Krankheit oder Abwesenheit. Spieler die ausserhalb der eigenen Stammgruppe spielen wollen, können gleichzeitig bei **öffentlichen Ligen** weitere Erfahrungen sammeln.

### 2.2.3 Clubmitglieder

#### 2.2.3.1 Analyse

Clubmitglieder Spielen regelmässig und haben dafür fix Vereinbarte Clubtrainings. Diese sind nicht flexibel. Zusätzlich besteht auch eine Infrastruktur für Ranking über die Liga sowie eine Protokollierung von Spielen in den meisten Fällen. Clubmitglieder sind somit nicht im Hauptfokus dieser Applikation. Höchstens Ausserhalb des Clublebens ist es gut möglich das ein Clubmitglied diese Applikation braucht um neue Spiele zu vereinbaren oder in einer privaten Liga mitzuspielen.

#### 2.2.3.2 Use Cases

Neue Clubs ohne Infrastruktur könnten diese Applikation für ihre Administration verwenden.

## 2.3 Anforderungsanalyse

### 2.3.1 Vision

Erleichterung zur Terminfindung und Administration von Racketsportspielern

### 2.3.2 Ziele

Folgende Business Prozesse sollten unterstützt werden:

- Identifizierung von Partnern
- Terminvereinbarung
- Amateur-Liga Management
- Protokollierung eines Spiels

### 2.3.3 Rahmenbedingungen

#### 2.3.3.1 Allgemein

Die Applikation hat keine konkreten Rahmenbedingungen. Sie ist eine Standalone Applikation und hat keine externen Abhängigkeiten.

#### 2.3.3.2 Technologie

Es gibt keine Einschränkungen, welche Technologie benutzt werden sollte, solange die Anforderungen erfüllt werden.

#### 2.3.3.3 Erweiterbarkeit

Die Applikation soll möglichst einfach erweiterbar sein. Zusätzliche Sicherheit, neue Funktionalitäten sowie skalierbarkeit in Performance sowie Stabilität sollten mit der eingesetzten Technologie möglich sein.

#### 2.3.3.4 Sicherheit

Technologien sollten Standard Sicherheitsanforderungen im Web umsetzen können. Privacy und Integrity Anforderungen sind in diesem Proof of Concept noch nicht geplant.

#### 2.3.3.5 Stabilität und Performance

Für den Proof of Concept sind keine Stabilitäts- und Performance Anforderungen nötig.

### 2.3.4 Schnittstellen

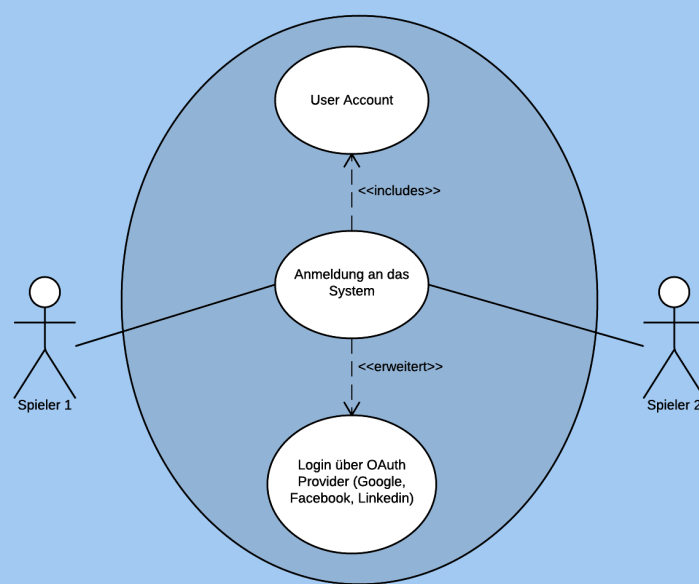
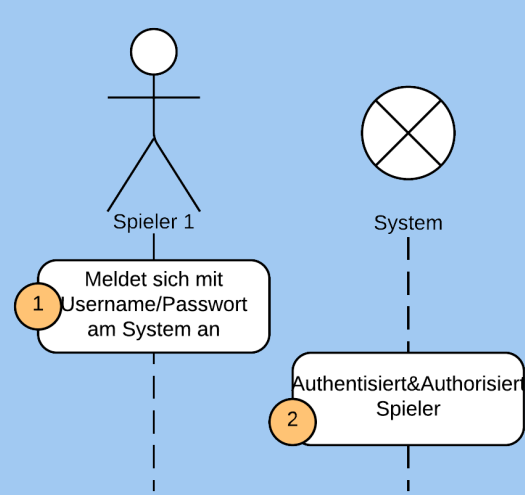
Folgende Tabelle zeigt interne sowie externe Schnittstellen auf:

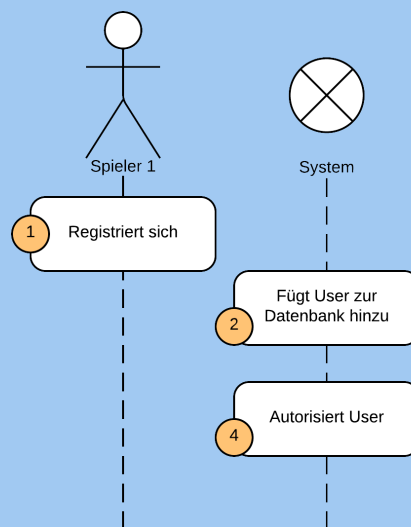
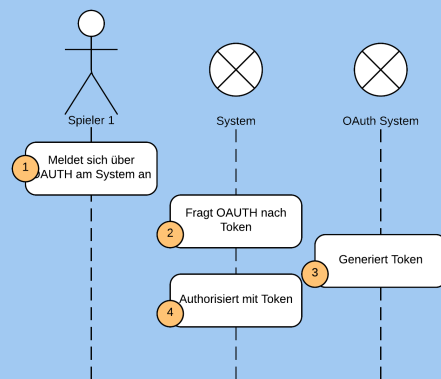
ID	Name	Beschreibung
I1	API → Client	Der Client greift auf die API zu um die aktuellen Daten dem Benutzers anzuzeigen. API sollte HTTP REST benutzen.
I2	Client → Browser	Der Client läuft innerhalb eines Browsers. Der Browser führt die Applikation - bestehend aus HTML, CSS und Javascript - aus.
I3	Browser → Android	Der Browser läuft innerhalb der Android Applikation. Die Android Applikation Emuliert den Browser und zeigt die Applikation als App dem Benutzer an.

### 2.3.5 Anwendungsfälle

#### 2.3.5.1 UC0 - Login in das System

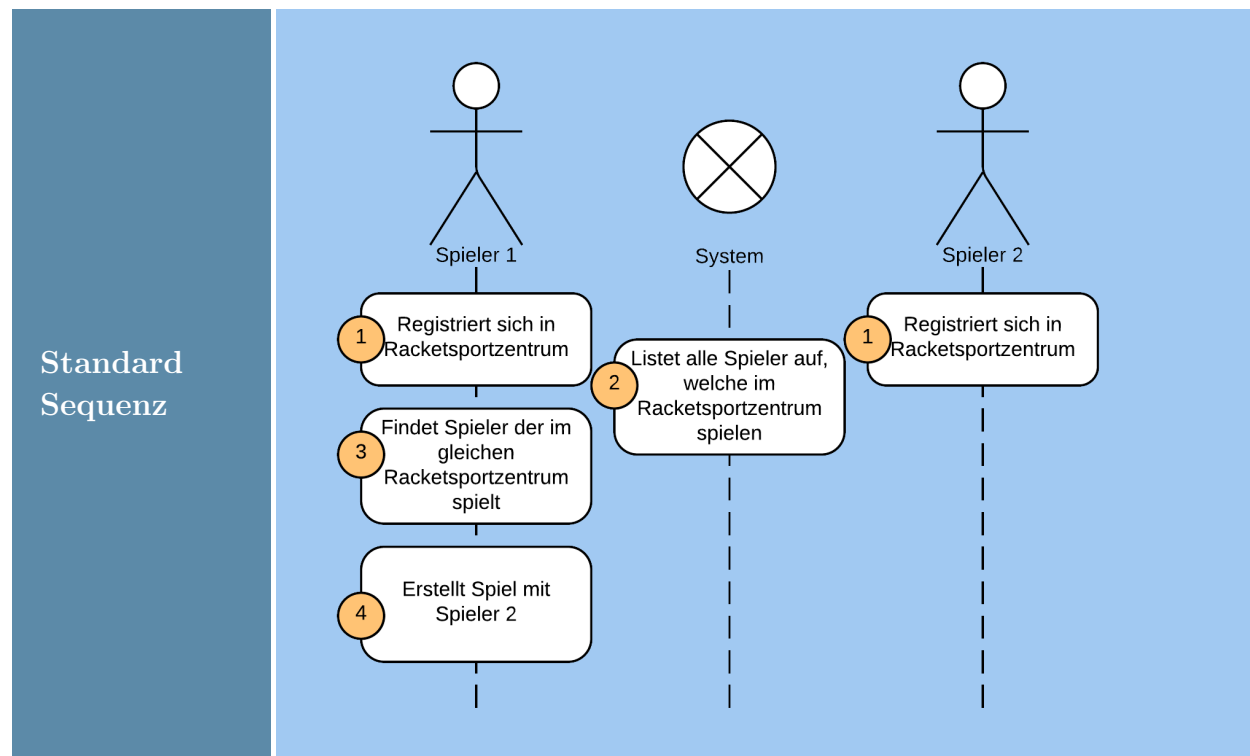
UC0	Login in das System
Beschreibung	Ein User des Systems will sich mit dem System authentisieren. Der Benutzer bekommt durch die Authentisierung Autorisierungen zugesprochen sowie ein Profil und Daten zugeordnet. Es ermöglicht dem User seine eigene Ansicht in die Daten des Systems zu erhalten. Da dieses System eine Applikation individualisiert für den User darstellt, sind alle Ansichten in das System einer Autorisierung zugeordnet und können nur von authentisierten Benutzern benutzt werden.

Diagramm	
Version	1.0
Vorbedingung	-
Anforderungen	REQ0.01 - Anmeldung an das System REQ0.02 - Registrierung eines Users mit dem System REQ0.03 - Authentisierung über OAuth
Testfälle	Test0.1: Anmeldung über Username/Passwort Test0.2: Anmeldung über OAuth Test0.3: Registrierung über Username/Passwort Test0.4: Registrierung über OAuth
Standard Sequenz	

Alternative  
Sequenzen

## 2.3.5.2 UC1 - Identifizierung von Partnern

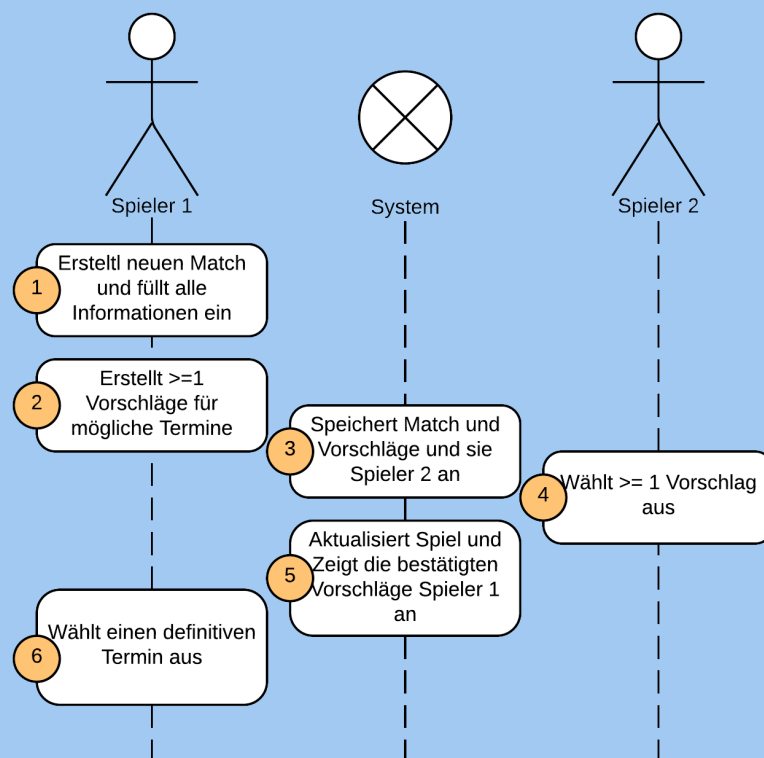
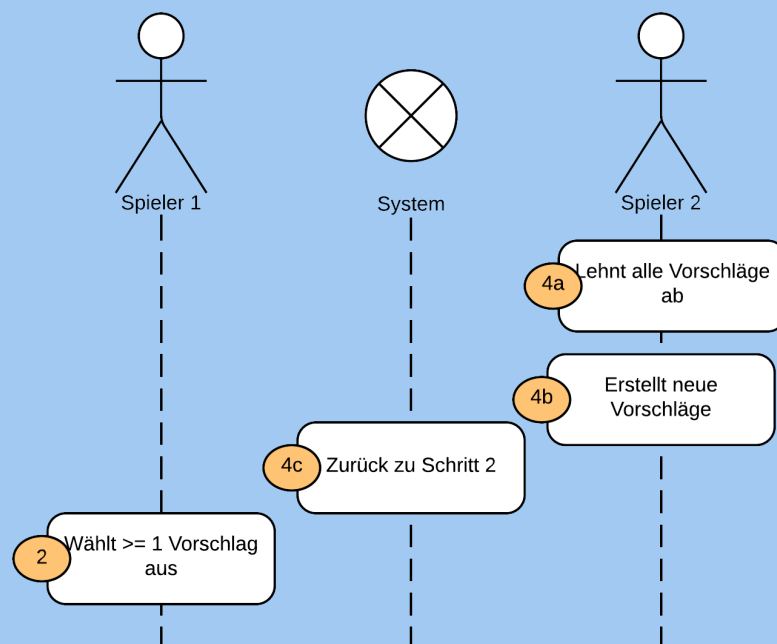
UC1	Identifizierung von Partnern
Beschreibung	Zwei Partner - A und B - wollen einen bestimmten Racketsport in einer bestimmten Region spielen. Der User A oder B kann Spielvorschläge an andere User senden. Dafür kann er nach Region filtern.
Diagramm	<pre> graph TD     subgraph System         UC1((Racketsportzentrum))         UC2((Registriert sich in Racketsportzentrum))         UC3((Vereinbart Match mit Spieler gleichen Racketsportzentrums))         UC2 -.-&gt; &lt;&lt;includes&gt;&gt;  UC1     end     S1((Spieler 1)) --- UC1     S1 --- UC2     S1 --- UC3     S2((Spieler 2)) --- UC1     S2 --- UC2     S2 --- UC3 </pre>
Version	1.0
Vorbedingung	UC0 - Anmeldung an das System
Anforderungen	REQ1.01 - Zuteilung von User zu Racketsportzentren REQ1.02 - Details zu Spielern REQ1.03 - Friend System REQ3.01 - Court erstellen REQ3.02 - Spieler können sich in Courts eintragen
Testfälle	Test1.1: TBD Test 1.2: TBD





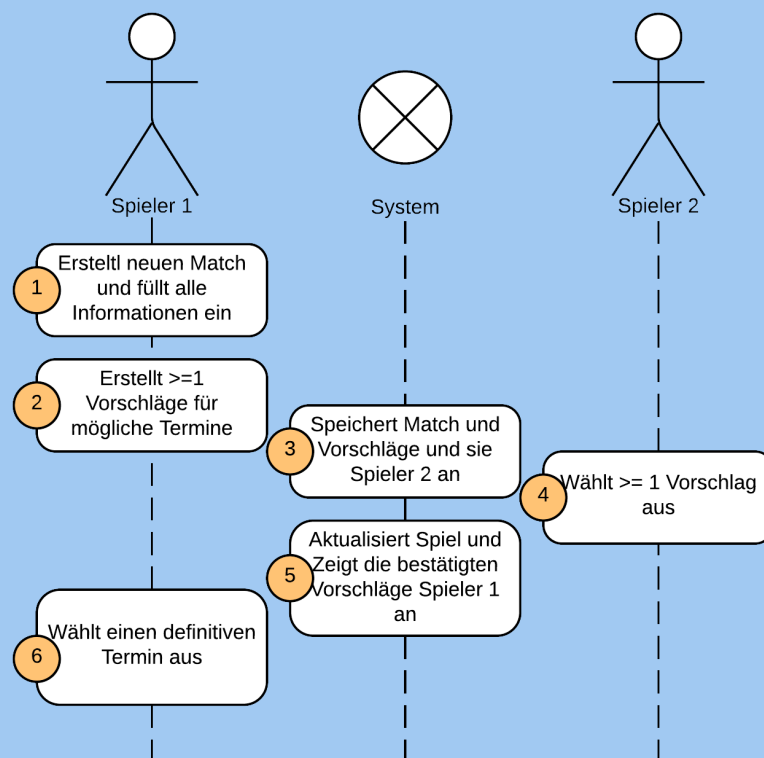
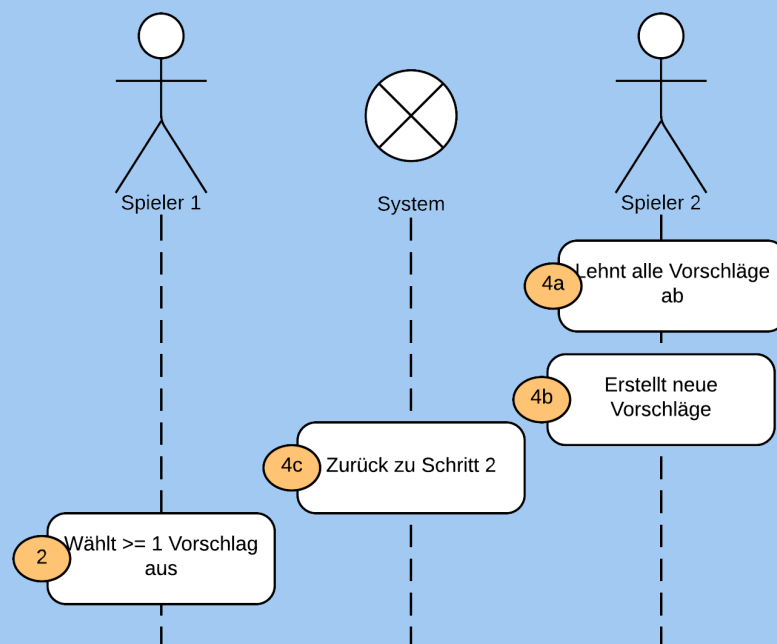
## 2.3.5.3 UC2 - Terminvereinbarung

UC2	Terminvereinbarung
Beschreibung	Der Spieler A kann einen oder mehrere Spielvorschläge senden. Anschliessend kann der Spieler B einer dieser Spielvorschläge <b>annehmen</b> , <b>ablehnen</b> oder <b>neue Spielvorschläge senden</b> . Sobald beide Nutzer einen Spielvorschlag angenommen hat, gilt der Termin als vereinbart.
Diagramm	
Version	1.0
Vorbedingung	UC0 - Anmeldung an das System
Anforderungen	REQ2.01 - Spiel erstellen REQ2.02 - Spielterminvorschläge erstellen REQ2.03 - Spielterminvorschläge annehmen und ablehnen REQ2.04 - Wiederkehrende Spiele erstellen REQ2.06 - Privatsphäreinstellungen berücksichtigen
Testfälle	Test2.1: TBD Test2.2: TBD

Standard  
SequenzAlternative  
Sequenzen

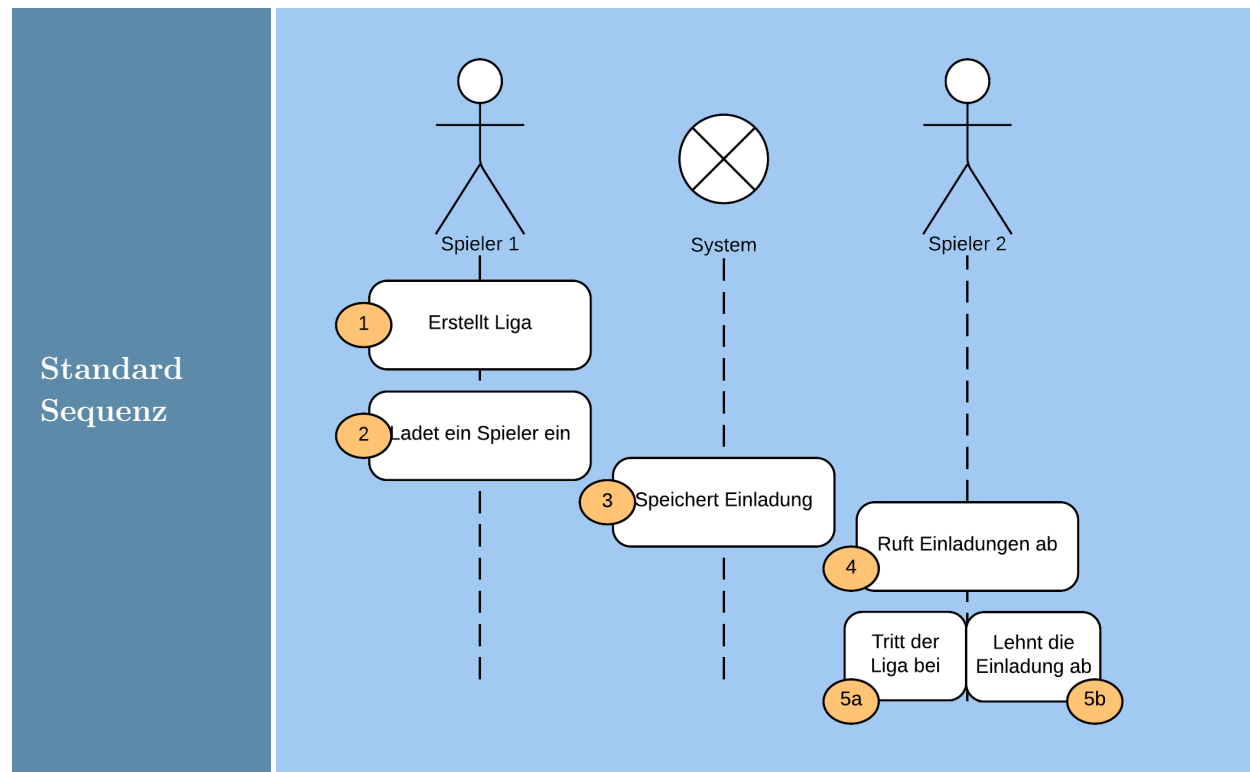
## 2.3.5.4 UC 3 - Spiel Protokollieren

UC3	Spiel protokollieren
Beschreibung	Nachdem ein Spiel durchgeführt wurde, kann der Spieler B sowie der Spieler A die Scores eintragen oder bestätigen. Anschliessend können beide Spieler die jeweiligen vergangenen Spiele jederzeit in der Applikation nachschlagen. Optional wird das protokollierte Spiel in einer Liga gewertet (falls vor dem Spiel so deklariert).
Diagramm	<pre> graph TD     subgraph UC3 [Spiel protokollieren]         direction TB         UC1((Spiel))         UC2((Resultate))         UC3((Spiel spielen))         UC4((Resultate eintragen))         UC1 -.-&gt; &lt;&lt;erweitert&gt;&gt;  UC2         UC3 -.-&gt; &lt;&lt;includes&gt;&gt;  UC1         UC3 --- UC4     end     S1[Spieler 1] --- UC3     S2[Spieler 2] --- UC4 </pre> <p>The diagram illustrates the 'Spiel protokollieren' use case. It features a large blue oval boundary containing four use cases: 'Spiel' (top), 'Resultate' (right), 'Spiel spielen' (center), and 'Resultate eintragen' (bottom). A dashed arrow labeled '&lt;&lt;erweitert&gt;&gt;' points from 'Spiel' to 'Resultate'. A dashed arrow labeled '&lt;&lt;includes&gt;&gt;' points from 'Spiel spielen' to 'Spiel'. A solid line connects 'Spiel spielen' and 'Resultate eintragen'. Outside the boundary, two actor stick figures are shown: 'Spieler 1' on the left and 'Spieler 2' on the right. 'Spieler 1' is connected to 'Spiel spielen' by a solid line, and 'Spieler 2' is connected to 'Resultate eintragen' by a solid line.</p>
Version	1.0
Vorbedingung	UC0 - Anmeldung an das System UC1 - Terminvereinbarung
Anforderungen	REQ2.07 - Ergebnisses des Spiels eintragen REQ2.08 - Bestätigung des Spiels eintragen
Testfälle	Test3.1: TBD Test1.2: TBD

Standard  
SequenzAlternative  
Sequenzen

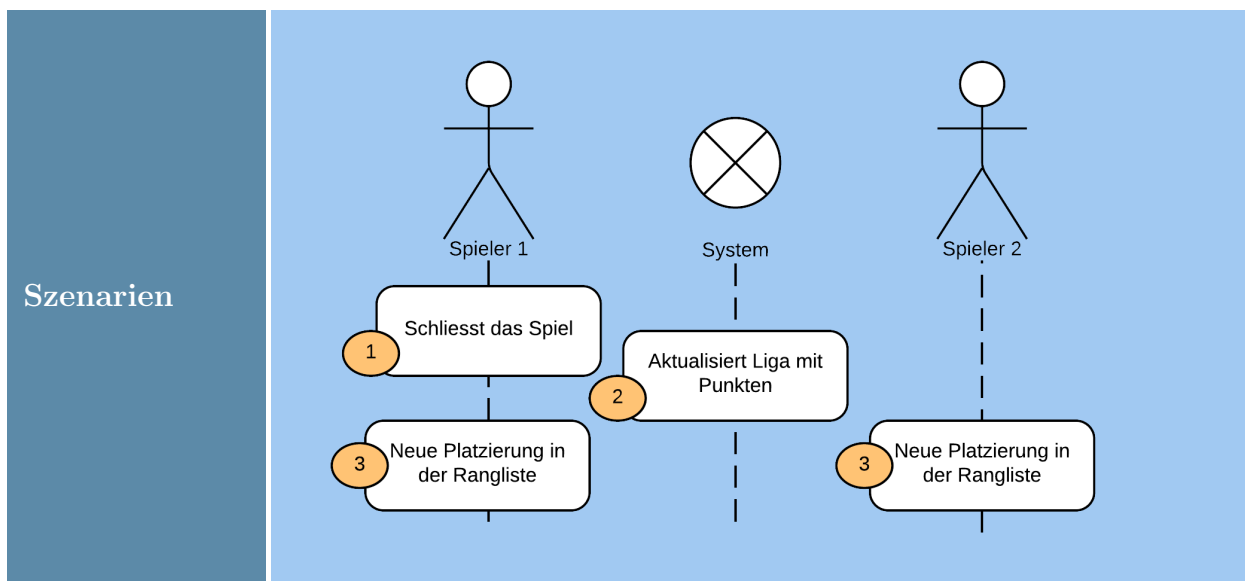
## 2.3.5.5 UC 4.1 - Liga erstellen

UC4.1	Liga erstellen
Beschreibung	Jeder Spieler kann eine Liga erstellen. Diese Liga generiert eine Rangliste. Der Spieler kann anschliessend andere Spieler in die Liege einladen oder die Spieler treten der Liga bei
Diagramm	
Version	1.0
Vorbedingung	UC0 - Anmeldung an das System
Anforderungen	REQ4.01 - Liga erstellen REQ4.02 - Spieler können sich in Liga eintragen
Testfälle	Test4.1: TBD Test4.2: TBD



## 2.3.5.6 UC 4.2 -Rangliste durch Spiele aktualisieren

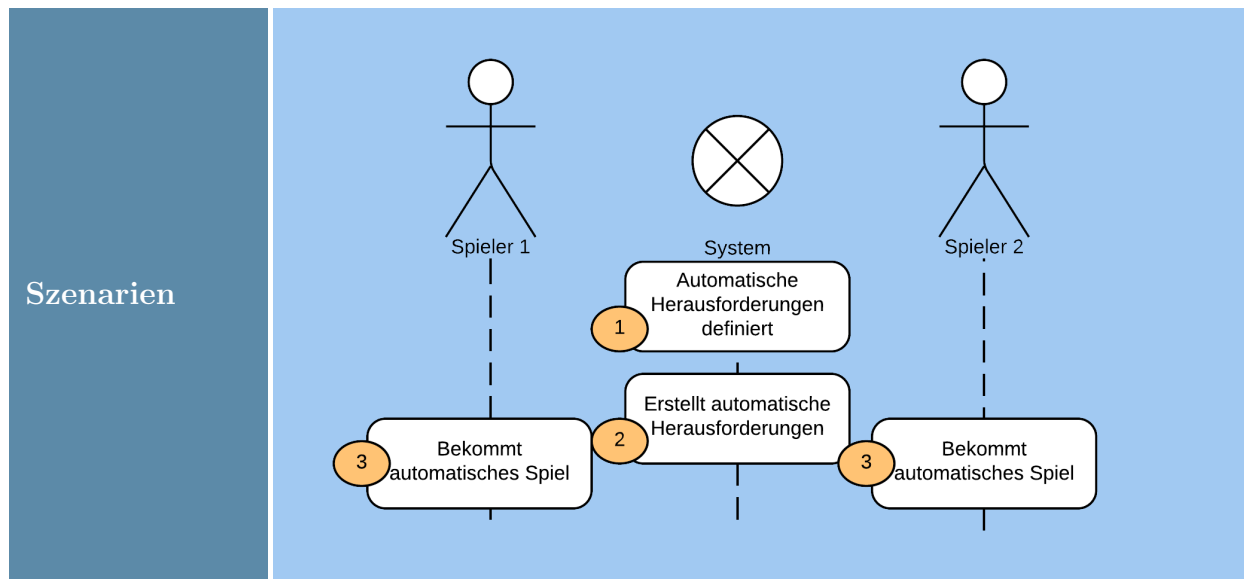
UC4.2	Rangliste durch Spiele aktualisieren
Beschreibung	Nachdem ein Spiel gespielt ist, können beide Spieler ihre Scores eintragen. Stimmt diese überein oder bestätigt ein Spieler der eingetragene Score des anderen Spielers, wird dieser zur Rangliste hinzugerechnet.
Diagramm	<pre> graph TD     subgraph UC2         Spiel((Spiel))         Resultate((Resultate))         Spiel_spielen((Spiel spielen))         Resultate_eintragen((Resultate eintragen))         Liga_Aktualisieren((Liga Aktualisieren))         Liga((Liga))         Rangliste_erstellen((Rangliste erstellen))                  Spiel_spielen -- "&lt;&lt;includes&gt;&gt;" --&gt; Spiel         Spiel_spielen -. "&lt;&lt;erweitert&gt;&gt;" .-&gt; Resultate         Resultate_eintragen -.-&gt; Rangliste_erstellen         Rangliste_erstellen -.-&gt; Liga_Aktualisieren         Liga_Aktualisieren -.-&gt; erweitert  Liga     end          Spieler1[Spieler 1] --- Spiel_spielen     Spieler1 --- Resultate_eintragen     Spieler2[Spieler 2] --- Spiel_spielen     Spieler2 --- Resultate_eintragen </pre>
Version	1.0
Vorbedingung	UC4.1 - Liga erstellen
Anforderungen	REQ4.03 - Rangliste der Liga durch Spiele aktualisieren
Testfälle	Test1.1: Test Test 1.2: Test2





## 2.3.5.7 UC 4.3 - Spiele automatisch anfordern

UC4.3	Spiele automatisch anfordern
Beschreibung	Bei der Erstellung der Liga kann einen Algorithmus zur automatischen Spielvereinbarung ausgewählt werden. Dieser Algorithmus schlägt jedem Spieler nach der definierten Regelmässigkeit Terminvorschläge vor und stellt so sicher, dass alle Spieler regelmässig gegen alle andere Spieler spielen. Die Spieler können sich auf einen der vorgeschlagenen Termine - oder ein anderer individuell vorgeschlagener Termin festlegen.
Diagramm	<pre> graph LR     subgraph UC43 [ ]         direction TB         Liga((Liga))         Match((Matchautomatismus))         Spiel((Spiel erstellen))         Liga -- "&lt;&lt;includes&gt;&gt;" --&gt; Liga_erstellen         Spiel -- "&lt;&lt;includes&gt;&gt;" --&gt; Match         Match -- "&lt;&lt;erweitert&gt;&gt;" --&gt; Liga     end     Spieler1((Spieler 1)) --&gt; Liga_erstellen     Spieler1 --&gt; Spiel     Spieler2((Spieler 2)) --&gt; Spiel     </pre>
Version	1.0
Vorbedingung	UC4.1 - Liga erstellen
Anforderungen	REQ4.04 - Erstellen eines automatischen Spiels
Testfälle	Test1.1: Test Test 1.2: Test2



### 2.3.6 Anforderungen

Folgende Anforderungen resultieren aus den Use Cases und sind kategorisiert in funktionale- sowie nicht funktionale Anforderungen. Zusätzlich werden die Anforderungen bewertet mit den Attributen Notwendigkeit und Kritikalität.

Folgende Kategorien können für Notwendigkeit vergeben werden:

- Essential - Anforderung in der Aufgabenstellung enthalten und/oder Notwendig für den PoC
- Conditional - Anforderung verbessert die App massiv und würde Lösung erweitern. Ein nicht Erfüllen der Anforderung macht die Lösung jedoch nicht inakzeptabel und kann in zukünftigen Releases umgesetzt werden
- Optional - Können umgesetzt werden, müssen jedoch nicht. Je nach Feedback von zukünftigen Kunden könnten optionale Anforderungen in Zukunft berücksichtigt werden

Für Kritikalität gibt es folgende Kategorien:

- High - Anforderung ist Basis von anderen essential Anforderungen und muss schnell erstellt werden
- Medium - Anforderung sollte innerhalb des POC implementiert werden um Benutzerfälle vollständig abzudecken
- Low - Anforderung kann in zukünftigen Releases umgesetzt werden

Die Anforderung in diesem Abschnitt werden in User Stories übertragen in das Tool Yodiz: <https://app.yodiz.com/plan/pages/task-board.vz?cid=14274&pid=1&iid=1>

### 2.3.6.1 Funktionale Anforderungen

Funktionale Anforderungen definierten spezifische, aus den Anwendungsfällen benötigte, Funktionalitäten der Applikation.

Anforderung	Notwendigkeit	Kritikalität
Kategorie Bedienbarkeit		
REQ0.01 - Anmeldung an das System	Essential	High
REQ0.02 - Registrierung eines Users mit dem System	Essential	High
REQ0.02 - Authentisierung über OAuth	Conditional	High
Kategorie User		
REQ1.01 - Zuteilung von User zu Racketsportzentren	Essential	High
REQ1.02 - Details zu Spielern	Essential	High
REQ1.03 - Friend System	Conditional	Medium
Kategorie Spiele		
REQ2.01 - Spiel erstellen	Essential	High
REQ2.02 - Spiel Terminvorschläge erstellen	Essential	High
REQ2.03 - Spiel Terminvorschläge annehmen und ablehnen	Essential	High
REQ2.04 - Wiederkehrende Spiele erstellen	Essential	High
REQ2.05 - Spiele erstellen	Conditional	Medium
REQ2.06 - Privatsphäreinstellungen berücksichtigen	Optional	Low
REQ2.07 - Ergebnisses des Spiels Eintragen	Essential	High
REQ2.08 - Bestätigung des Spiels eintragen	Essential	High
Kategorie Courts		
REQ3.01 - Courts erstellen	Essential	High
REQ3.02 - Spieler können sich in Courts eintragen	Essential	High
Kategorie Liga		
REQ4.01 - Liga erstellen	Essential	High
REQ4.02 - Spieler können sich in Liga eintragen	Essential	High
REQ4.03 - Rangliste der Liga durch Spiele aktualisieren	Essential	High
REQ4.04 - Erstellen eines automatischen Spiels	Essential	High

### 2.3.6.2 Nicht funktionale Anforderungen

Die nicht funktionalen Anforderungen definieren Eigenschaften der Applikation, welche erfüllt werden sollten, jedoch nicht für den User ersichtlich sind. Diese Anforderungen sind wichtig

sobald eine solche Applikation in einen produktiven Status übergeht. Im Moment ist die Applikation in einem PoC Status geplant, deshalb sind viele nicht funktionale Anforderungen optional.

Anforderung	Notwendigkeit	Kritikalität
Qualitätsmerkmal: Funktionalität		
NREQ0.01 - Sicherheit	Conditional	High
Qualitätsmerkmal: Zuverlässigkeit		
NREQ1.01 - Fehlertoleranz	Conditional	High
NREQ1.02 - Wiederherstellbarkeit	Essential	High
Qualitätsmerkmal: Benutzbarkeit		
NREQ2.01 - Verständlichkeit	Optional	Medium
NREQ2.02 - Bedienbarkeit	Conditional	High
Qualitätsmerkmal: Effizienz		
NREQ3.01 - Effizient in Programmierung	Essential	High
NREQ3.02 - Effizient in Installation	Essential	High
Qualitätsmerkmal: Wartbarkeit		
NREQ4.01 - Einfach erweiterbar/änderbar	Essential	High
NREQ4.02 - Stabilität	Optional	High
NREQ4.03 - Testbarkeit	Essential	High
NREQ4.04 - Analysierbarkeit	Optional	High

Diese nicht funktionalen Anforderungen müssen immer im PoC angeschaut werden. Optionale Anforderungen können für einen produktiven Einsatz neu als essential ne kategorisiert werden.

## 3 Konzeption

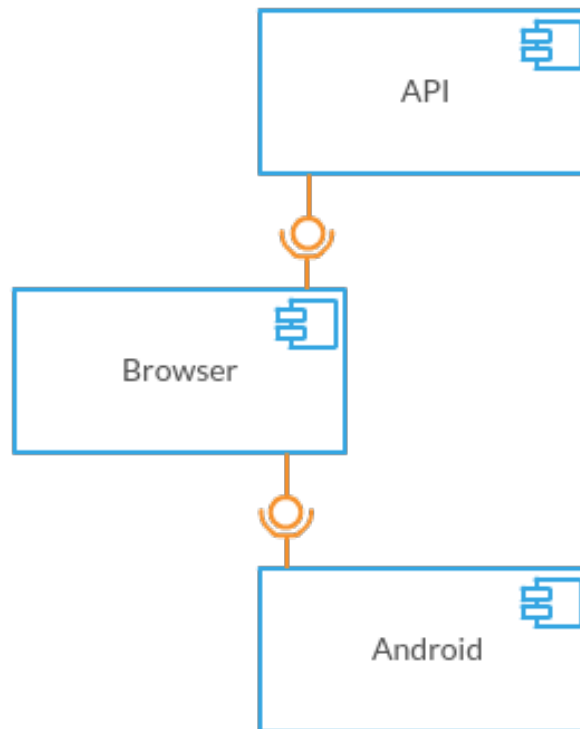


Abbildung 3.1: Grobkonzept für Applikation

Die Applikation besteht aus drei Teilen. Einem Webserver, der eine API und statische Clientfiles zur Verfügung stellt, der Client im Browser, welcher die API konsumiert sowie eine Android Applikation welche die Website lädt.

### 3.1 Technologiestack

Wie in dem Grobkonzept beschrieben, wird für die Applikation einen Technologiestack gebraucht, welcher eine skalierbare API sowie eine gute Integration der API mit einer Browser Frontend Anwendung bietet. Folgende Anforderungen werden an den Technologiestack gestellt:

- Skalierbare REST API

- Einfacher und schneller Umgang mit AJAX
- Gute Integration zwischen API und HTML/JS Client
- Responsive Design, Integration mit OAUTH für 3rd Party Authentisierung
- Persistence Layer (Datenbankunterstützung)

### 3.1.1 MVC Frameworks

Komplexe Applikationen werden vorzugsweise mit MVC Frameworks erstellt. Über MVC APIs sind Routing, Logik sowie Präsentationsschicht gut voneinander Abstrahiert. Es ist gut möglich, für Views mit verschiedenen Daten über ein Model zu versehen, oder auch ein Logisches Routing für eine API zu entwickeln. Folgende Graphik zeigt ein MVC Konzept. Der Client sendet einen Request zu dem Server und wird vom Routing zur Logik im System weitergeleitet. Die Logik findet das richtige Model sowie die dazugehörige View. Die View wird mit dem Model gerendert und es entsteht eine Antwort, welche dem Client zurückgesendet wird. In einer REST API ist die View JSON. Das Model wird in JSON umgewandelt und versendet.

Folgendes Code-Beispiel - die Funktion `list()` - zeigt, wie alle Courts aus dem Persistence Layer selektiert werden und per JSON zum Client gesendet werden. Dies ist ein API Endpunkt zur auflistung von Courts (`http://webserver/courts`). Gut zu sehen ist, dass die `jsonp()` Funktion als Renderer gebraucht wird anstatt eine Standard-View.

```
1 exports.list = function(req, res) {  
2   Court.find().sort('-created').populate('user')  
3   .exec(function(err, courts) {  
4     if (err) {  
5       return res.status(400).send({  
6         message:  
7         errorHandler.getMessage(err)  
8       });  
9     } else {  
10      res.jsonp(courts);  
11    }  
12  });  
13};
```

Mit einem MVC Framework auf der Server-Seite kann man so gut abstrahierte und ausbaufähige - skalierbare - APIs erstellen. Diese APIs müssen nun jedoch vom Client Browser verarbeitet werden können. Folgende Möglichkeiten bieten sich an:

- Parallel zu der REST API werden Renderer gebaut, welche das Model mit einer View in eine - für den Client statische - Website rendern. Der Client verfügt hier ausschliesslich Logik um die verschiedenen Webseiten abzurufen.

- Ein Website-Skelett mit Logik wird beim ersten Aufruf an den Client verschickt. Der Client bezieht nun Daten aus der REST API und reichert die schon vorhandenen Views mit den Objekten - gesendet über AJAX - selber an.

Eine parallele Implementierung zur Rest API geht entgegen dem Basis Konzept, dass alle Applikationen so gut wie möglich von der REST API profitieren. Zusätzlich würde bei einer parallelen Implementierung jeder Klick in einer Aktualisierung der Applikation resultieren. Dies ist unerwünscht, da sich die Website nicht schnell und intuitiv anfühlt. Man hat bei jedem Klick eine Downtime, da viel Daten übertragen werden müssen, und der Browser den DOM jedes mal neu aufbauen muss. Bei der zweiten Option wird die Website nur einmal heruntergeladen. Der DOM wird nach dem Download aufgebaut und von der Logik verändert. Klicks lösen einen viel geringeren Aufwand von Server bis Client aus und somit ist die Downtime viel kleiner. Die Applikation fühlt sich schneller und intuitiver an.

Wie bei dem Server, kann man auch bei der Applikation ein MVC Pattern implementieren. Ein Routing definiert, bei welcher URL welche View aufgerufen wird. Bei dem Aufruf einer View ist ein Controller hinterlegt, welcher bei der API das Model und Objekt besorgt. Die View rendert die vom Controller generierten Daten

„Figure Angular JS “

Server und Client MVCs können so miteinander Kombiniert werden und es entsteht eine skalierbare und wartbare Applikationsumgebung.

#### 3.1.1.1 Server MVCs

MVCs für den Server gibt es verschiedene:

- Spring Framework - Java
- ExpressJS - JSON
- Rails - Ruby

#### 3.1.1.2 Client MVCs

MVC für den Server sind ausschliesslich in Javascript geschrieben:

- AngularJS
- Backbone.js
- Ember.js

### 3.1.1.3 Stacks

Eine Konfiguration des Client- sowie Server MVCs, damit beide gut miteinander Funktionieren ist zusätzlich wichtig. In der Evaluation wird somit folgende Konfigurationen abgewogen:

- Spring Framework
- Express Framework

### 3.1.1.4 Anforderungen an Stacks

Die Anforderungen an einen Frameworkstack gehen aus den nicht funktionalen Anforderungen heraus. Einige Anforderungen, welche sich auf die User Experience beziehen (Benutzbarkeit) sind für die Anforderungen an den Stack nicht relevant.

Anforderung	Notwendigkeit	Kritikalität
Qualitätsmerkmal: Funktionalität		
NREQ0.01 - Sicherheit	Conditional	High
Qualitätsmerkmal: Zuverlässigkeit		
NREQ1.01 - Fehlertoleranz	Conditional	High
NREQ1.02 - Wiederherstellbarkeit	Essential	High
Qualitätsmerkmal: Effizienz		
NREQ3.01 - Effizient in Programmierung	Essential	High
NREQ3.02 - Effizient in Installation	Essential	High
Qualitätsmerkmal: Wartbarkeit		
NREQ4.01 - Einfach erweiterbar/änderbar	Essential	High
NREQ4.02 - Stabilität	Optional	High
NREQ4.03 - Testbarkeit	Essential	High
NREQ4.04 - Analysierbarkeit	Optional	High

### 3.1.1.5 Spring

Spring ist ein MVC Framework in Java. Man programmiert in der J2EE Umgebung und bietet eine API zum Client. Gleichzeitig sendet man den AngularJS Stack zum Client, welcher anschliessend die API konsumiert. Als Persistence Layer können Relationale Datenbanken wie MySQL, Oracle oder Sybase verwendet werden. Über Data Access Object wird dieser Layer angesprochen und in Models Emuliert.

**Installation und Konfiguration** Die Installation und Konfiguration von Spring mit Ember.js ist komplex. Die Installation funktioniert über Maven, einzelne Komponenten nach der



Installation müssen aufeinander konfiguriert werden. Security sowie ORM für den Persistence Layer sind nicht im Spring Package enthalten und müssen zusätzlich hinzugefügt und auf Spring konfiguriert werden. Es gibt Templates, welche dies etwas einfacher gestalten, jedoch in meinen Recherchen habe ich kein Funktionierendes Modell gefunden

**Portabilität** Durch Maven ist eine Spring installation einfach auf einer anderen Maschine installierbar. Einmal eingerichtet ist es einfach möglich den Code auszutauschen und die Applikation laufen zu lassen.

**Funktionalität** Out of the Box bringt Spring keine Funktionalität ausser den MVC Workflow. Es gibt kein Basis User Management, kein Front End MVC, keine CSS Frameworks oder sonstiges. Ember.js muss zusätzlich selber installiert und integriert werden. **QUELLEN!!!!**

**Stabilität** Spring gilt das eines der meist eingesetzten Frameworks auf dem Markt. Es ist bekannt für seine Stabilität und Enterprise Readiness.

**Skalierbarkeit** Spring ist skalierbar innerhalb des Application Servers. Es gibt jedoch limitationen im Clustering.

**Erweiterbarkeit** Die Programmiersprache von Spring ist Java oder Scala. Diese Programmiersprachen bieten eine breite Palette an Funktionen und Objekten, welche benutzt werden können. Spring ist somit sehr gut erweiterbar.

**Stacks** Ein Stack für Spring, der alle Plug-Ins und Client MVCs mitbringt heisst JHipster . Dieser Stack bringt eine Userverwaltung, Integration von Persistence Layer, Client MVC (AngularJS) und vieles weiter. Die Installation ist komplex und schwer verständlich.

**Objekte / API** Ein integraler Bestandteil der Applikation soll eine API mit JSON sein. Bei Spring muss man Javaobjekte zu JSON objekte serialisieren und umgekehrt. Folgender Workflow existiert für Objekte von Persistence Layer bis zum Ausgang der API:

1. Relationale Datenbankfelder
2. Konvertierung zu einem Objekt über ein ORM(Object-relational Mapping) Interface
3. Konvertierung von Objekt zu einem JSON-Objekt
4. JSON Objekt wird versendet

Diese Konversionen müssen oft manuell erstellt werden und sind negativ um effizient zu programmieren.

### 3.1.1.6 NodeJS / ExpressJS

**Installation und Konfiguration** ExpressJS wird über npm installiert. Die Installation ist sehr simpel. Zusätzlich existieren für NodeJS und ExpressJS eine grosse Anzahl Stacks, welche sehr einfach zu installieren sind. Der Persistence Layer benutzt MongoDB, d.h. Objekte werden nativ im JSON Format in der Datenbank gespeichert. Konfigurationen über die Stacks sind sehr intuitiv.

**Portabilität** NPM lässt sich wie Maven so konfigurieren, dass eine automatische Installation ausgeführt wird der benötigten Plug-ins. Die Portabilität ist somit wie bei Spring sehr gut.

**Funktionalität** NodeJS bzw. ExpressJS besitzen out of the box keine Funktionalität. Diverse Stacks unterstützen jedoch Userverwaltung, Plugin-Handling, Client sowie Server MVCs.

**Stabilität** NodeJS ist nicht so viel benutzt wie Spring. Die Stabilität ist somit nicht endgültig bewiesen. einige grosse Firmen setzen jedoch schon NodeJS ein und es ist bis jetzt noch nichts bekannt über Probleme mit der Stabilität.

**Skalierbarkeit** NodeJS ist sehr gut skalierbar. Über Loadbalancer kann man HTTP Anfragen an mehrere NodeJS prozesse verteilen. Da der ganze Kontext im Persistence Layer existiert - und MongoDB im Cluster läuft - ist NodeJS super skalierbar.

**Erweiterbarkeit** JavaScript ist sehr populär und es existieren Zahlreiche Bibliotheken. Stacks sind in Module aufgebaut und die Erweiterbarkeit und Wartbarkeit von Code ist sehr effizient.

**Stacks** In NodeJS ist der MEAN Stack weit verbreitet. Der MEAM Stack besteht aus folgenden Produkten:

- M - MongoDB, der Skalierbare Persistence Layer
- E - ExpressJS, ein MVC um APIs zu entwickeln
- A - AngularJS, ein MVC auf dem Client um Single-Page Applikationen zu erstellen, welche auf die ExpressJS API zugreifen.
- N - NodeJS, JavaScript Applikationsserver, welcher sehr gut skalierbar ist.

**Objekte / API** Ein integraler Bestandteil der Applikation soll eine API mit JSON sein. Bei einem Mean Stack gibt es keine Konversionen. Von der Datenbank bis zum Output der API existiert immer genau das gleiche Objekt.

### 3.1.1.7 Evaluation

Für die Stacks MEAN sowie JHipster wird nun eine Evaluation durchgeführt. Es soll herausgefunden werden, welcher Stack besser geeignet ist für eine effiziente Programmierung.

Die nicht funktionalen Anforderungen werden somit mit einer Bewertung von 1-10 für jedes Framework versehen. Die Bewertung entsteht auf Basis der Erfahrung des Autors dieser Arbeit aufgrund von Recherchen. Argumente sind in dem Kapitel der einzelnen Frameworks aufgelistet.

Anforderung	MEAN	JHipster
Qualitätsmerkmal: Funktionalität		
NREQ0.01 - Sicherheit	9	10
Qualitätsmerkmal: Zuverlässigkeit		
NREQ1.01 - Fehlertoleranz	10	9
NREQ1.02 - Wiederherstellbarkeit	10	9
Qualitätsmerkmal: Effizienz		
NREQ3.01 - Effizient in Programmierung	10	6
NREQ3.02 - Effizient in Installation	10	7
Qualitätsmerkmal: Wartbarkeit		
NREQ4.01 - Einfach erweiterbar/änderbar	10	7
NREQ4.02 - Stabilität	8	10
NREQ4.03 - Testbarkeit	10	10
NREQ4.04 - Analysierbarkeit	9	9
Total	86	77

Die Applikation wird somit mit dem MEAN Stack entwickelt.

### 3.1.2 Kontext

Die Applikation ist aufgeteilt auf einen Server, sowie auf einen Client, welcher ein Browser oder eine Android App ist. Auf dem Server sind alle Daten hinterlegt:

- Gespeicherte Objekte in MongoDB
- Server Logik
- Client Daten, welche vom Browser über HTTP abgefragt werden

Im Anfangszustand hat der Client keine Daten. Der Client bekommt die Daten bei dem Abruf der Applikations URL über HTTP. Er baut nun die Logik im Browser Cache auf und startet das JavaScript Programm. Das JavaScript Programm lädt nun die auf dem Server gespeicherten Objekte über HTTP AJAX Abrufe und stellt diese dar.

Die Logik von Server wie auch Client benutzt das M(V)C Pattern. Objecte werden in Models - inklusive Business Logik - gespeichert, der Controller beinhaltet die Applikationslogik, welche das Model sowie die View auswählt. Die View rendert nun das Model in ein bestimmtes Schema (siehe Abbildung 3.2).

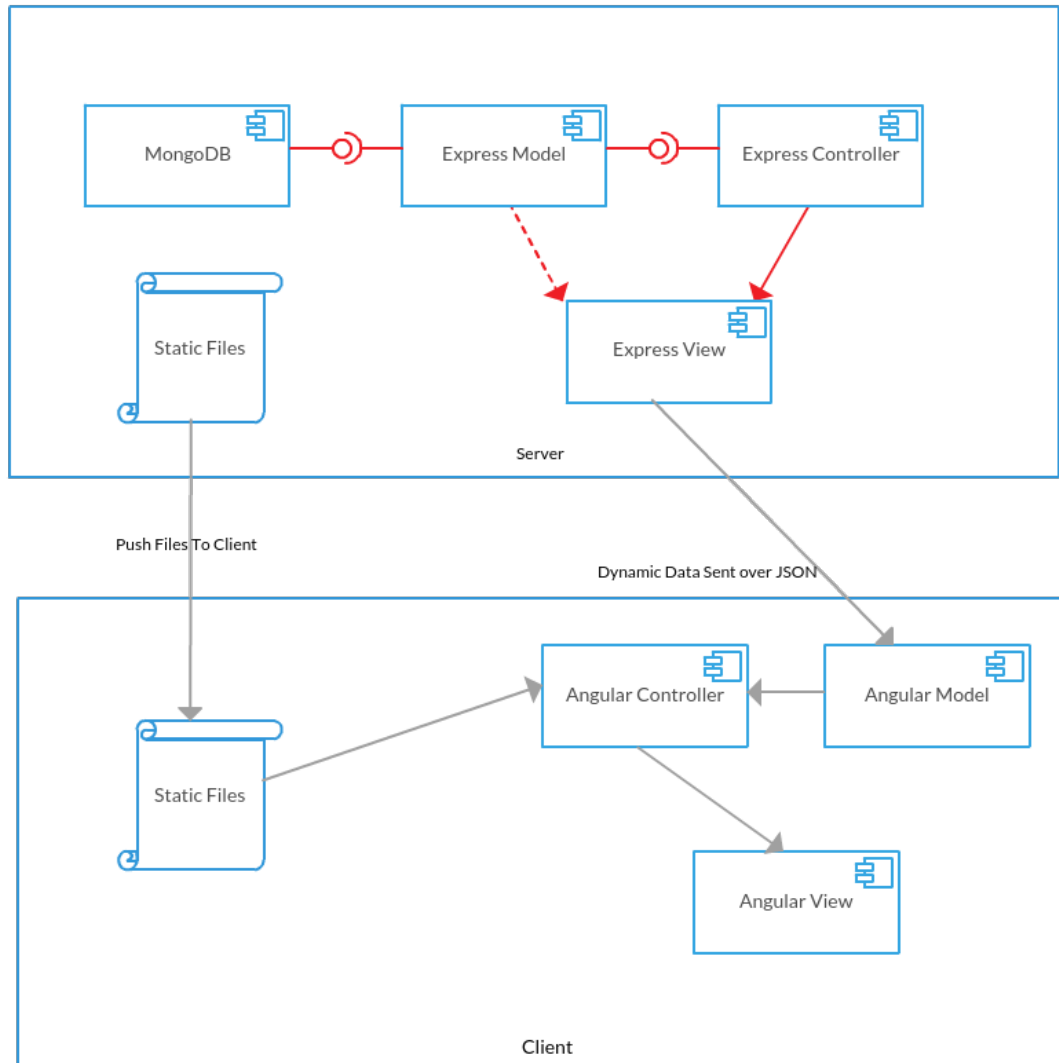


Abbildung 3.2: Detaillierte Applikations Architektur

### 3.1.2.1 Continues Integration

Die Applikation wird auf dem Server des Autors gehostet.

Um Updates der Applikation direkt zu installieren besitzt der Server eine Applikation für Continues Integration namens Jenkins. Jenkins führt ein Deployment der Applikation automatisch aus bei Anforderung eines solchen Deployments. In Jenkins werden diese Deployments Jobs genannt.

Dieser Jenkins Job wird bei jedem Push nach Github ausgeführt. Das heisst bei jedem Update der Applikation wird die neuste Version direkt auf dem Server unter der Adresse <https://racket.marques.pw> aktualisiert.

### 3.1.3 Businessschicht

#### 3.1.3.1 DB Design

In der Applikation gibt es kein Relationales Datenbankmodell. MongoDB arbeitet mit Dokumenten, sowie Referenzen. Dokumente sind JSON-Objekte in JavaScript, welche in MongoDB als Dokument gespeichert werden. Ein Objekt ist eine Representation von Business Objekten in der Applikation.

Das User-Objekt repräsentiert der User der Applikation. Der User hat einen Namen, einen Usernamen, ein Passwort (encrypted und salted), Berechtigungen und Freunde. Zusätzlich werden ihm andere Objekte zugeordnet sowie andere User (Repräsentation als Freund).

Das Court-Objekt repräsentiert ein Racketsportzentrum der Applikation. Dieses Objekt wird benötigt um den Physikalischen Austragungsort eines Spieles zu definieren. Das Objekt beinhaltet einen Namen, eine Adresse (inklusive Koordinaten für eine zukünftige Umkreissuche), was für Sportarten gespielt werden können und welche User in diesem Racketsportzentrum spielen wollen.

Das Match-Objekt repräsentiert das Spiel, welches geplant, ausgetragen oder beendet ist. Das Spiel-Modell definiert zwei oder einen Spieler, einen Status, eine Sportart, ein Court, mehrere Datumvorschläge, maximal fixes Datum, eine Punktzahl, sowie ein Gewinner. Hinter dem Match-Objekt existiert ein relativ grosser Business-Workflow, welcher im Kapitel `IMPLEMENTATION Matchmaking` definiert ist.

Das Liga-Objekt repräsentiert eine Liga. Verschiedene Benutzer können einer Liga beitreten und sind nach Beitritt bestimmten regeln unterworfen. Dafür können die Benutzer spiele für die Liga spielen und so Punkte für einen optionalen Preis sammeln. Die Liga beinhaltet neben einem Namen, einer Sportart, einem Standort (inklusive Koordinaten, für zukünftige Umkreissuche), einem Niveau, Start- und Enddatum, einem Preis und einem Matchmaking Plan (wird später im Dokument erläutert`REF`).

Folgende Grafik zeigt die Beziehung der Verschiedenen Schemas auf, das Datenbankmodell ist nicht Relational, und somit nicht normalisiert.

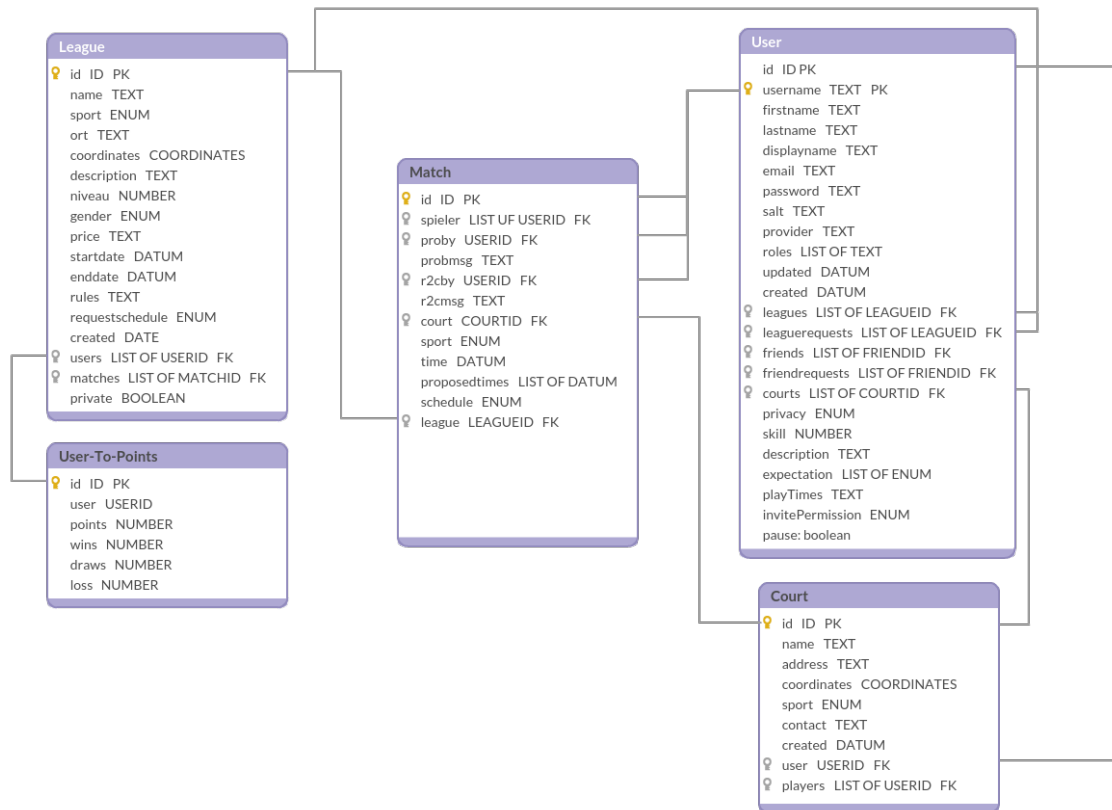


Abbildung 3.3: Datenbank Modell

### 3.1.4 Präsentationsschicht

Als GUI wird ein standard Bootstrap Design verwendet. Ohne Authentisierung kann nur die Home-Page gesehen werden sowie die Login und Signup Page. Für alle anderen Seiten muss der Benutzer authentisiert sein. Sobald die Authentisierung durchgeführt wurde, gibt es können die Elemente (Nach Datenbank) Benutzer, Liga, Racketsportzentrum sowie Spiele selektiert werden. Innerhalb der einzelnen Menu kann man verschiedene Operationen direkt ansteuern, einige nur über andere Operationen. Folgendes Diagramm zeigt die Interaktion durch die verschiedenen Views.

#### 3.1.4.1 User Section

Die User Section einhaltet drei Views die direkt aus dem Menu erreichbar sind. Die erste View User Profile ermöglicht dem User, Details über sich preiszugeben. Er kann zusätzlich das Passwort ändern. In der Friends view kann er neue Friendrequests erstellen, und pendente Friendrequests annehmen oder ablehnen. Die Social Account View bietet eine Verknüpfung von Social Accounts mit der Applikation an.

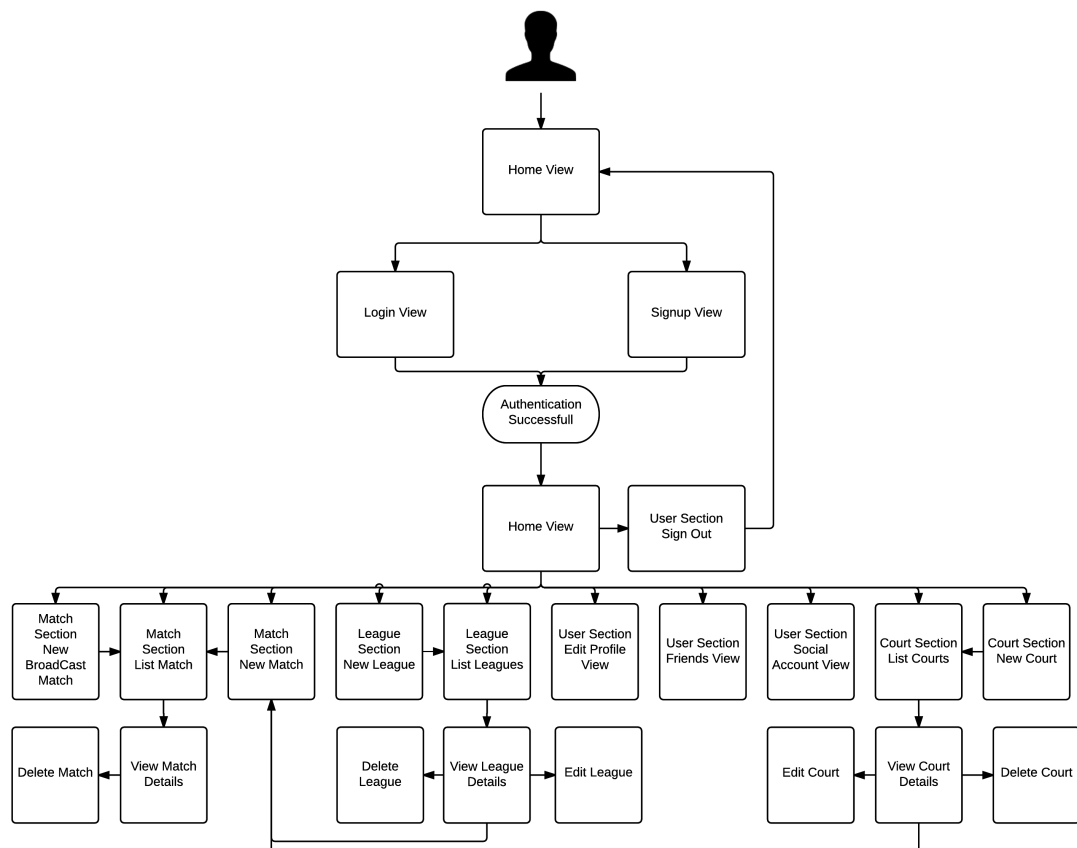


Abbildung 3.4: GUI Interaktions Modell

### 3.1.4.2 Court Section

Die Court Section beinhaltet vier Views sowie eine Aktion. In der new Court View kann ein neues Racketsportzentrum registriert werden. In der List Courts view findet man alle Racketsportzentren und erreicht bei klick auf ein Zentrum die View Court Details View. In dieser View kann man alle Details des Racketsportzentrum anschauen, sowie alle Spieler, welche in diesem Racketsportzentrum spielen. Durch klick auf den Spieler kann in die New Match View gewechselt werden, um einen Spieler herauszufordern. Von der Detail View kann man zusätzlich das Court löschen, sofern man das Court erstellt hat oder ein Admin ist.

### 3.1.4.3 League Section

Die League Section beinhaltet vier Views sowie eine Aktion. In der New League View kann ein neues Liga registriert werden. In der List League View findet man alle Ligen und erreicht bei klick auf eine Liga die View League Details View. In dieser View kann man alle Details die Liga anschauen, sowie alle Spieler, welche in dieser Liga spielen. Durch klick auf den Spieler kann in die New Match View gewechselt werden, um einen Spieler herauszufordern.

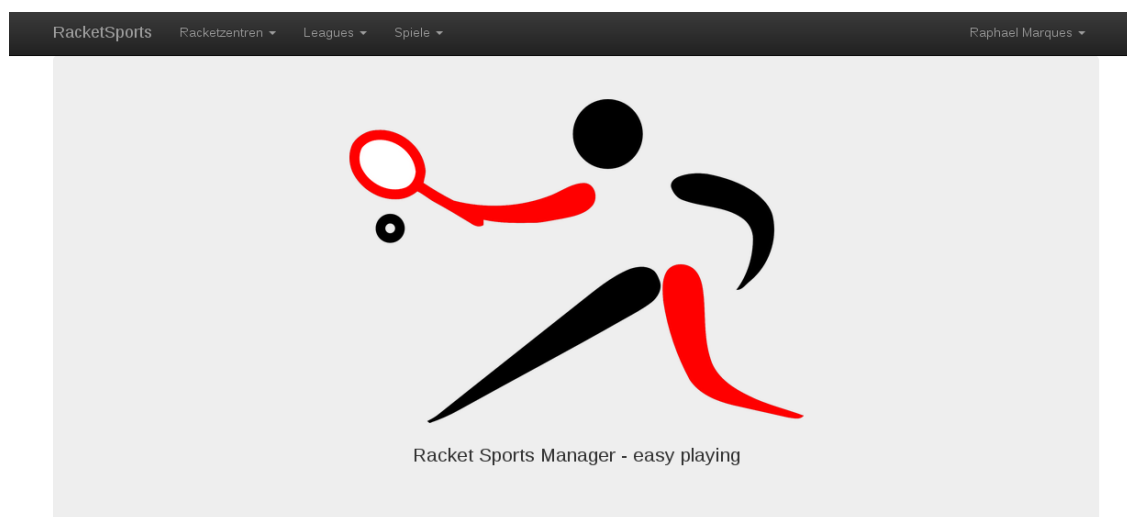
Von der Detail View kann man zusätzlich die Liga löschen, sofern man die Liga erstellt hat oder ein Admin ist.

#### 3.1.4.4 Match Section

Die Match Section beinhaltet alle Interaktionen im Match. Drei Views sind direkt aus dem Menu erreichbar. Auf der New Match View kann man ein neues Spiel erstellen. Man kann Court, Spieler in einem Formular auswählen. Über den Menupunkt New Broadcast Match View, wählt man ein Court sowie eine Zeit und alle Spieler, welche in diesem Court spielen werden angefragt für ein spontanes Spiel. In der List Match View werden alle Spiele aufgelistet. Von da kommt man in die View Match Details View, welche den Matchworkflow abdeckt.

#### 3.1.4.5 GUI Design

Als Design wurde Bootstrap benutzt. Das Design besteht aus einem Header sowie einem Hauptfenster.



#### Was ist der Racket Sport Manager?

Mit dem Racket Sport Manager ist es möglich einfach Sport Partner und Termine zu finden. Die Applikation motiviert dich zusätzlich öfters zu gehen.

- Vereinbare Spiele mit Freunden "Doodle-Style"
- Trete einer lokalen Liga bei
- Lass dir von der Liga Spielvorschläge unterbreiten und gewinne Sie
- Frage Personen in der Nähe nach einem spontanen Spiel
- Have fun...

Abbildung 3.5: GUI Design



## 4 Implementation

### 4.1 Projektaufbau

Um den Projektaufbau zu erklären wird zuerst die Ordnerstruktur angeschaut: Wie auf

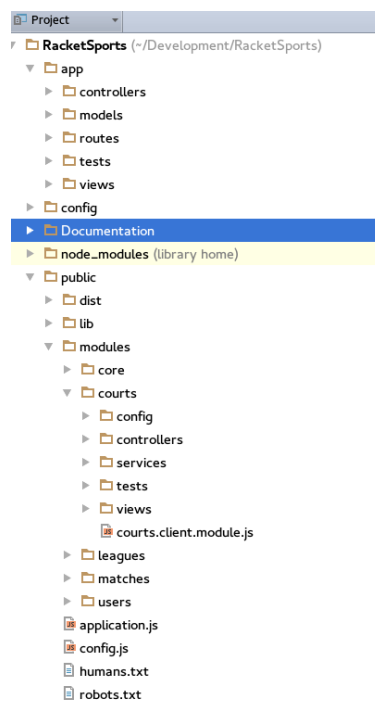


Abbildung 4.1: Projekt Ordner Struktur

der Graphik ersichtlich besteht das Projekt aus den Überordnern App, Documentation und public. Node\_modules ist ein System-Ordner und muss hier nicht betrachtet werden.

#### 4.1.1 app

Der Ordner App beinhaltet alle Serverseitige Logik für die Applikation. Es beinhaltet die ExpressJS Komponenten welche für das MVC Framework notwendig sind:

- Controller - Kontroller im MVC
- Models - Definition der Datenobjekte
- Routers - Routing definition durch das MVC

- Tests - Unittests für Controller und Models
- Views - Verschiedene Basisviews, welche noch nicht zum Client gesendet wurden

### 4.1.2 Documentation

Der Ordner Documentation beinhaltet diese Dokumentation sowie alle Präsentationen.

### 4.1.3 public

Public beinhaltet alle Files, welche an den Client gesendet werden.

- dist - Beinhaltet definitionen, welche Ordner gesendet werden, was für Module wo registriert werden sollen
- lib - beinhaltet statische Libraries, welche der Client braucht. Hier befinden sich nodeJS, AngularJS und Bootstrap als libraries.
- modules - Hier ist die Eigentliche Client Logik versteckt. Für jedes Modul gibt es folgende Ordner:
  - config - Konfiguration von Modul und Client Routing. Definiert z.b. was für Items über den Header ansprechbar sind
  - controller - AngularJS Controller
  - services - Definiert den Link zur API
  - tests - Unit Tests für die AngularJS Module
  - views - Definiert views für die einzelnen Seiten der Applikation.

## 4.2 REST API

Die REST API besteht aus fünf Endpunkten:

- /matches - Stellt alle Operationen für Matches zur Verfügung
- /leagues - Stellt alle Operationen für Liga Management zur Verfügung
- /courts - Stellt alle Operationen für die Verwaltung von Racketsportzentren zur Verfügung
- /users - Stellt alle Operationen für das Usermanagement zur Verfügung
- /core - Stellt Core-Funktionalitäten (Home Seite) zur Verfügung

Die Endpunkte /users und /core waren im MEANJS Stack schon vorhanden. Der User Endpunkt wurde jedoch modifiziert. Die Modifizierungen sind in dem Kapitel dokumentiert, die schon vorhandenen Endpunkte nicht.

### 4.2.1 Routing

Um die REST API anzusteuern gibt es ein zentrales Routing in der Applikation. Dieses Routing definiert, welche URL welche Funktion aufruft. Jedes Modul hat eine eigene Routing Definition um das die einzelnen Files übersichtlich zu halten.

Folgendes Codebeispiel zeigt das Routing des Courts Moduls. Es definiert URLs und HTTP Methoden wie die URL aufgerufen wird. Je nach Methode werden anschliessend verschiedene Parameter definiert, zum Beispiel wie der Endpunkt geschützt ist (requires Login, hat Autorisierung) und anschliessend welche Funktion aufgerufen wird (courts.update)

```
1 module.exports = function(app) {
2   var users = require('../.. /app/controllers /
3     users.server.controller ');
4   var courts = require('../.. /app/controllers /
5     courts.server.controller ');
6
7   // Courts Routes
8   app.route('/courts')
9     .get(courts.list)
10    .post(users.requiresLogin , courts.create);
11
12   app.route('/courts/:courtId')
13     .get(courts.read)
14     .put(users.requiresLogin , courts.hasAuthorization , courts.update)
15     .delete(users.requiresLogin , courts.hasAuthorization ,
16       courts.delete);
17   app.route('/courts/:courtId/join')
18     .get(users.requiresLogin , courts.join);
19   app.route('/courts/:courtId/leave')
20     .get(users.requiresLogin , courts.leave);
21   // Finish by binding the Court middleware
22   app.param('courtId' , courts.courtByID);
23 };
```

Am Anfang (Linie 2 und 4) der Routen werden die Controller inkludiert, damit die Funktionen auch gefunden werden.

### 4.2.2 Spiel Endpunkt /matches

Bei allen Match Endpunkten muss der User als Spieler registriert sein, um Informationen über das Spiel zu erhalten. Ausnahme ist, wenn er direkt die ID eingibt und direkt auf Spiel Details zugreift.

Mit einem Post fügt man der Datenbank ein Spiel hinzu, mit PUT aktualisiert man das Spiel mit neuen oder geänderten Daten. Hinter dem PUT interface gibt es gewisse Input

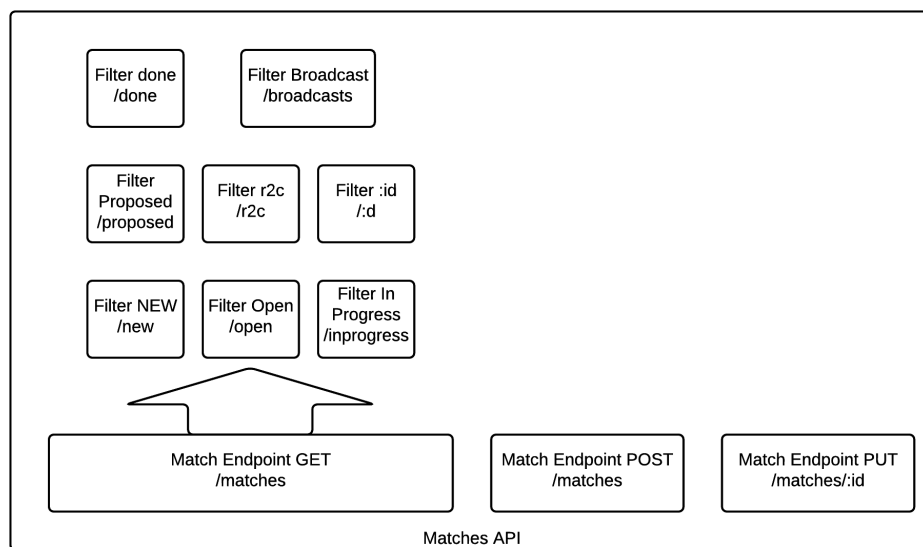


Abbildung 4.2: API für Spiele

Validations um Missbrauch zu verhindern. In dieser ersten Version sind die Validations jedoch relativ einfach gehalten.

Um eine gute Übersicht aller Spiele auf der List Matches View zu erstellen, gibt es für jeden Status eines Spieles einen eigenen API Call (`/matches/new`, `/matches/open`, `/matches/inprogress`, `/matches/proposed`, `/matches/r2c`, `/matches/done`).

Der API Endpunkt `/matches/broadcast` listet zusätzlich alle broadcasting Anfragen auf. Der Controller des Endpunkts korreliert, in welchen Racketsportzentren der User registriert ist und die Matches ohne zweiten Spieler und gibt das Resultat dem Client.

#### 4.2.2.1 Codebeispiel - Filtering und Population von Unterobjekten

Um bei den Spielen nur die offenen Spiele zu finden, wird in der Datenbank auf den Status des Spiels gefiltert. Dies wird über einen Select auf die JSON Eigenschaft gemacht:

```

1 exports.listOpen = function(req, res) {
2   var user = req.user;
3   Match.find({ 'spieler.user': req.user, state: 'open' })
4     .sort('-created').populate('spieler court')
5     .exec(function(err, matches) {
6       User.populate(matches, { path: 'spieler.user' },
7         function(err, user) {
8           if (err) {
9             return res.status(400).send({
10               message: errorHandler.getMessage(err)
11             });
12           } else {

```

```
13     res.jsonp(matches);
14   }
15   });
16 });
17 };
```

Auf der Zeile 3 ist der Select für die Datenbank zu finden. Ein Teil des zu suchenden Objektes wird definiert als erster Parameter der Match.find() Methode.

Das Match Objekt beinhaltet Spieler sowie Courts. Diese Felder sind jedoch nur IDs auf andere Objekte (Als Beispiel: ObjectID(AA4335GE0DE9EV88A) ). Um auf Datenfelder der Unterobjekte zugreifen zu können, müssen die Objekte popularisiert werden. Auf der Zeile 4 erkennt man, dass für das Objekt Match die Unterobjekte Spieler sowie Courts popularisiert werden. Nun hat das Objekt Spieler zusätzlich Unterobjekte. Hier muss man eine neue Population des Objektes User ausführen (siehe Zeile 6).

### 4.2.3 Liga Endpunkt

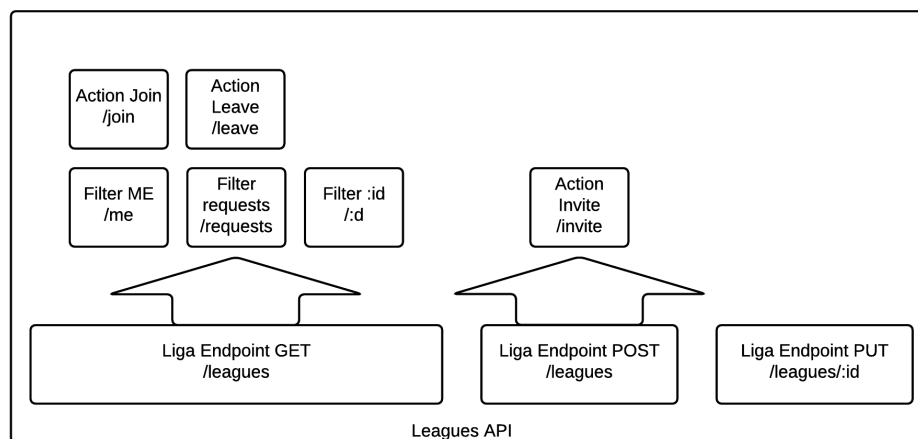


Abbildung 4.3: API für Ligen

Bei der Liga gibt es - wie bei allen Endpunkten - CRUD Endpunkte (/leagues für list all, /leagues/:id für Show Element, POST /leagues für Create League, PUT /leagues/:id für Update League). Zusätzlich gibt es eine Join Action, welche den authentisierten User einer Liga hinzufügt sowie ein Leave Endpunkt um die Registrierung zu löschen.

Ein zusätzlicher Endpunkt ist /leagues/invite, welcher ermöglicht einen User zu einer Liga einzuladen.

### 4.2.3.1 Codebeispiel - Join/Leave Funktionen

Die Join/Leave Funktionalität besteht aus zwei API Endpunkten. Wenn man nun die URL `/leagues/join` aufruft, wird folgender Code ausgeführt:

```
1 exports.join = function (req, res) {  
2  
3     var league = req.league;  
4     var user = req.user;  
5     var userToPoints = new UserToPoints();  
6     userToPoints.user = req.user;  
7     userToPoints.save();  
8  
9     league.users.push(userToPoints);  
10    user.leagues.push(league)  
11  
12    console.log(league);  
13    league.save(function (err) {  
14        ....  
15    })  
16 }
```

Auf der Zeile 3 und 4 werden User und Liga aus dem Request gelesen. Diese Daten werden implizit von AngularJS mitgeliefert.

Nun wird ein neues Liga-Spieler Objekt, ein Objekt welches den Spieler und seine Ranglistenpunkte beinhaltet erstellt und gespeichert (Zeile 5-7). Dieses Objekt wird nun in die Liste aller Liga-Spieler Objekte hinzugefügt und die Liga wird abgespeichert (Zeile 13).

```
1 exports.leave = function (req, res) {  
2     var league = req.league;  
3     var user = req.user;  
4     var i = league.users.indexOf(req.user);  
5     league.users.splice(i, 1);  
6     var j = user.leagues.indexOf(req.league);  
7     user.leagues.splice(j, 1);  
8     league.save(function (err) {  
9         ....  
10    })  
11 }
```

Will ein User die Liga verlassen, wird sein Objekt aus der Liste von Spielern gelöscht.

### 4.2.4 Racketsportzentrum Endpunkt /courts

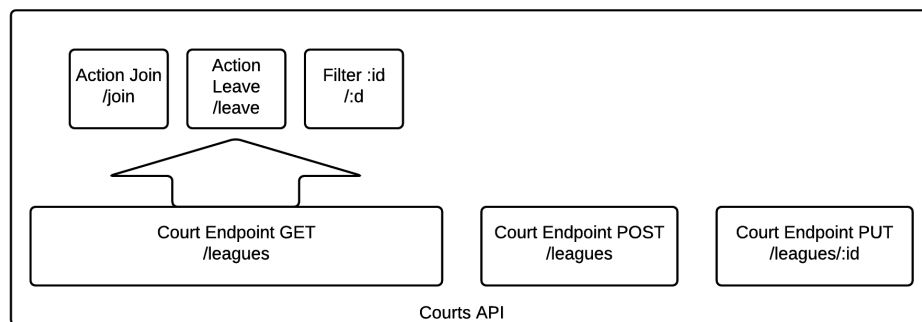


Abbildung 4.4: API für Racketsportzentren

Gleich wie beim Liga Endpunkt gibt es die CRUD Endpunkte sowie ein Join/Leave Endpunkt

### 4.2.5 Benutzer Endpunkt /users

Neben den üblichen Benutzerverwaltungs Endpunkten (/user/signin, /user/signout, /user/signup, /auth/forgot), welche hier nicht Dokumentiert werden, gibt es Endpunkte für das Freunde-System:

- GET /users/friend - Auflistung aller Freunde
- DELETE /users/friend - Löschen eines Freundes
- GET /users/request - Senden eines Freund Requests
- DELETE /users/request - Löschen eines Freund Requests

#### 4.2.5.1 Codebeispiel - OAuth Integration

Der MEAN Stack bietet ein OAuth Integrations Plugin an. Man muss nur noch das Plugin konfigurieren. Dafür wird für jeden Authentication Provider eine Strategie definiert:

```

1  ....
2  GoogleStrategy = require('passport-google-oauth').OAuth2Strategy,
3  ....
4  module.exports = function() {
5    // Use google strategy
6    passport.use(new GoogleStrategy({
7      clientID: config.google.clientID,
8      clientSecret: config.google.clientSecret,
9      callbackURL: config.google.callbackURL,
10     passReqToCallback: true
11   }),

```

```
12 function(req, accessToken, refreshToken, profile, done) {
13   // Set the provider data and include tokens
14   var providerData = profile._json;
15   providerData.accessToken = accessToken;
16   providerData.refreshToken = refreshToken;
17
18   // Create the user OAuth profile
19   var providerUserProfile = {
20     firstName: profile.name.givenName,
21     lastName: profile.name.familyName,
22     displayName: profile.displayName,
23     email: profile.emails[0].value,
24     username: profile.username,
25     provider: 'google',
26     providerIdentifierField: 'id',
27     providerData: providerData
28   };
29
30   // Save the user OAuth profile
31   users.saveOAuthUserProfile(req, providerUserProfile, done);
32 }
33 ));
34 };
```

Die Daten werden bei Registrierung über OAuth in das Userprofil abgelegt. (Siehe Zeile 19-27). Zusätzlich ist es nötig bei jedem Provider ein Access Token anzufordern und in die Produktionsumgebung der Applikation einzupflegen.

## 4.3 Web Applikation

Die WebApplikation ist wie bereits erwähnt mit Bootstrap und AngularJS programmiert. In AngularJS werden Direktiven erstellt, um einzelne API Endpunkte anzusprechen und diese Realtime in die View zu injecten. Hier ein Beispiel einer Direktive:

```
1 $scope.find = function() {
2   $scope.courts = Courts.query();
3 };
```

Wenn die Operation find() in der View aufgerufen wird, erstellt AngularJS eine Query zum Endpunkt /courts und bekommt alle Court Objekte zurück. Diese Courtobjekte werden nun in die globale Variable courts im \$scope abgefüllt. AngularJS updatet nun die View, welche die Variable courts anzeigt:

```
1 <section data-ng-controller="CourtsController" data-ng-init="find()">
```



```

2      ...
3      <table id="courtslist" class="table">
4          <tr>
5              <th>Name</th><th>Adresse</th><th>Verfügbare Sportarten</th>
6          </tr>
7
8          <tr data-ng-repeat="court in courts" id="{{court._id}}" ng-click="
go(court)" onmouseover="this.bgColor='#DDDDDD'" onmouseout="this.bgColor
='#FFFFFF'">
9              <td data-ng-bind="court.name"></td>
10             <td data-ng-bind="court.address"></td>
11             <td data-ng-bind="court.sports"></td>
12         </tr>
13     </table>
14     ...

```

Das ganze Webinterface ist auf solchen Direktiven aufgebaut.

### 4.3.1 Google Maps Integration

Eine Adresse wie z.B. 'Vitis' hat das Problem, dass man geographische Nähe suchen kann. Darum ist es wichtig, dass die Racketsport zentren ein Geographisch Valides Objekt haben. Um dies zu erreichen wurde die API von Google Maps integriert. Der Benutzer muss nun noch einem Objekt in Google Maps suchen, und dieses Selektieren um eine Valide Adresse zu bekommen. Folgendes Formularelement existiert in der View:

```

1 <div class="form-group">
2     <label for="address">Adresse</label>
3     <input type="text" onFocus="geolocate()" id="address" name="address" class=
"form-control" data-ng-model="address" required>
4 </div>
5

```

Wie auf Zeile 3 ersichtlich wird die direktive `geolocate()` aufgerufen. Diese Funktion ist Bestandteil der Google API, welche im Header von [https://maps.googleapis.com/maps/api/js?v=3.exp&signed\\_in=true&libraries=places](https://maps.googleapis.com/maps/api/js?v=3.exp&signed_in=true&libraries=places) gedownloadet wird. Die ID `address` wird von dem Court Controller aufgerufen und folgender Listener wird hinzugefügt:

```

1 var placeSearch, autocomplete;
2 if (document.getElementById('address')) {
3     autocomplete = new google.maps.places.Autocomplete(
4         /** @type {HTMLInputElement} */(document.getElementById('address')));
5     // When the user selects an address from the dropdown,
6     // populate the address fields in the form.
7

```

```
8     google.maps.event.addListener(autocomplete, 'place_changed', function ()
9     {
10         $scope.updateAddress();
11     });
12 }
```

Der autocomplete Listener ruft nun die AngularJS direktive updateAddress() auf:

```
1 $scope.updateAddress =function() {
2     var place = autocomplete.getPlace();
3     document.getElementById('address').value = place.formatted_address;
4     document.getElementById('lat').value = place.geometry.location.A;
5     document.getElementById('lng').value = place.geometry.location.F;
6     if ($scope.court) {
7         $scope.court.address = place.formatted_address;
8         $scope.court.lat = place.geometry.location.A;
9         $scope.court.lng = place.geometry.location.F;
10    }else{
11        this.address = place.formatted_address;
12        this.lat = place.geometry.location.A;
13        this.lng = place.geometry.location.F;
14    }
15 };
```

UpdateAddress sucht das Google Maps Objekt und speichert die Values - Adresse sowie Koordinaten - in die View. Bei dem Abschicken des Formulars wird nun geprüft ob die Koordinaten existieren, falls nicht, ist es keine valide Adresse, wie man im Court-Model sieht (Zeile 3 und 7):

```
1 lat: {
2     type: String,
3     required: "Please fill in a correct address (select it from the dropdown)"
4 },
5 lng: {
6     type: String,
7     required: "Please fill in a correct address (select it from the dropdown)"
8 }
```

## 4.4 Android Applikation

Als Grundlage für die Android Applikation wurde eine Applikation von gonative.io generiert. Im Laufe des Projektes - nach erheblicher Überschreitung des vorgeschriebenen Aufwandes

- wurde entschieden keine vollständig Native Webapplikation zu erstellen. Stattdessen wird eine WebView erstellt, welche die Mobile Webseite darstellt. Um alle Funktionalität zu behalten, wird über die WebView und Interception Algorithmen Push-Nachrichten ermöglicht. Der einzige Setback ist, das die Website offline nicht verfügbar ist.

#### 4.4.0.1 Codebeispiel - Push Interception

TBD!!!!

## 4.5 Workflows

## 4.6 Allgemeine Workflows

### 4.6.1 CRUD für Datenobjekte

Alle Datenobjekte haben einen Endpunkt. Jeder Endpunkt stellt CRUD Operationen zur Verfügung:

- C - Neues Objekt erstellen
- R - Ein Objekt anzeigen
- U - Ein Objekt aktualisieren
- D - Ein Objekt löschen

Zusätzlich wird noch einen Endpunkt zur Auflistung aller Objekte angeboten.

## 4.7 Court

### 4.7.1 Court Registrierung

Wenn der User den Knopf im User Interface zur Registrierung des Racketsportzentrums drückt, wird im Hintergrund der `/courts/join` API Call ausgeführt. Dieser Call fügt der User der Anfrage in ein Array - bestehend aus allen registrierten Usern - ein.

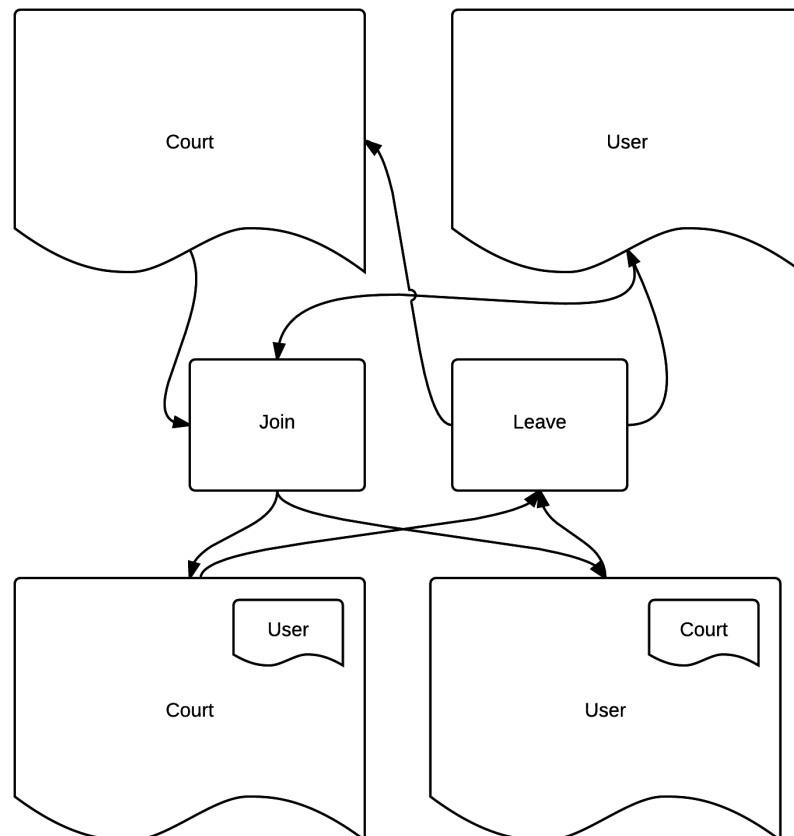


Abbildung 4.5: Racketsportzentrum Workflow

## 4.8 Liga

### 4.8.1 Liga Registrierung

Identisch zu der Court Registrierung funktioniert die Liga Registrierung

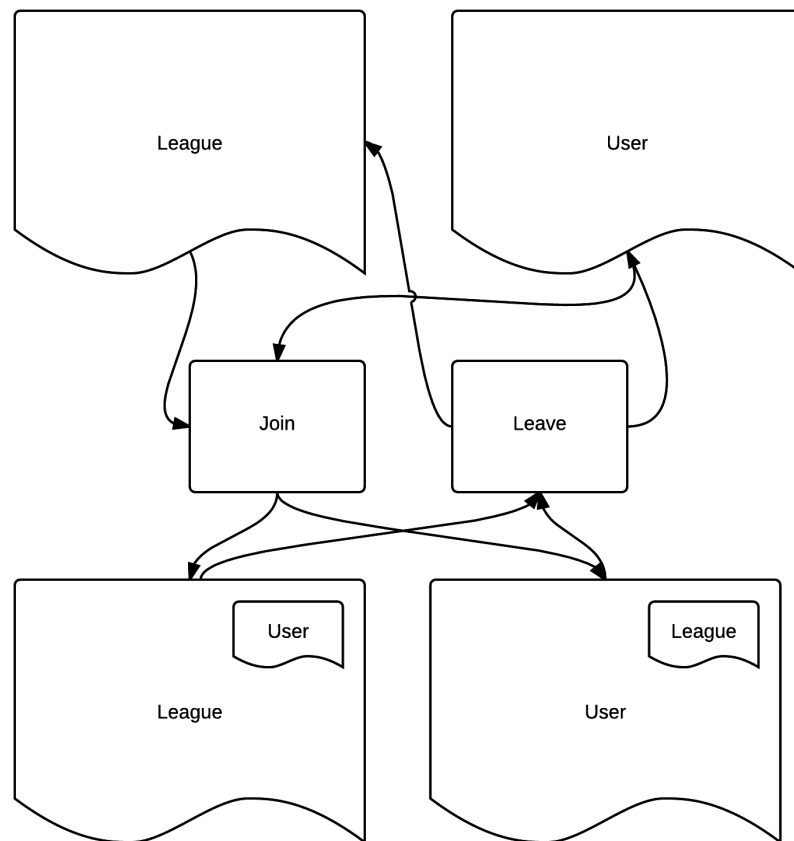


Abbildung 4.6: Liga Workflow

### 4.8.2 Automatische Herausforderung Liga

Bei der Erstellung einer Liga kann ausgewählt werden, ob automatische Herausforderungen aktiviert werden sollten. Aktuell gibt es vier verschiedene auswählbare Modi:

- Weeklyall: Wöchentliche Herausforderung, jeder gegen jeder, zufälliger Gegner
- Biweeklyall: Herausforderung alle zwei Wochen, jeder gegen jeder, zufälliger Gegner
- WeeklyTopTwo: Herausforderung jede Woche, immer die zwei Nächsten in der Rangliste
- BiweeklyTopTwo: Herausforderung alle zwei Wochen, immer die zwei Nächsten in der Rangliste

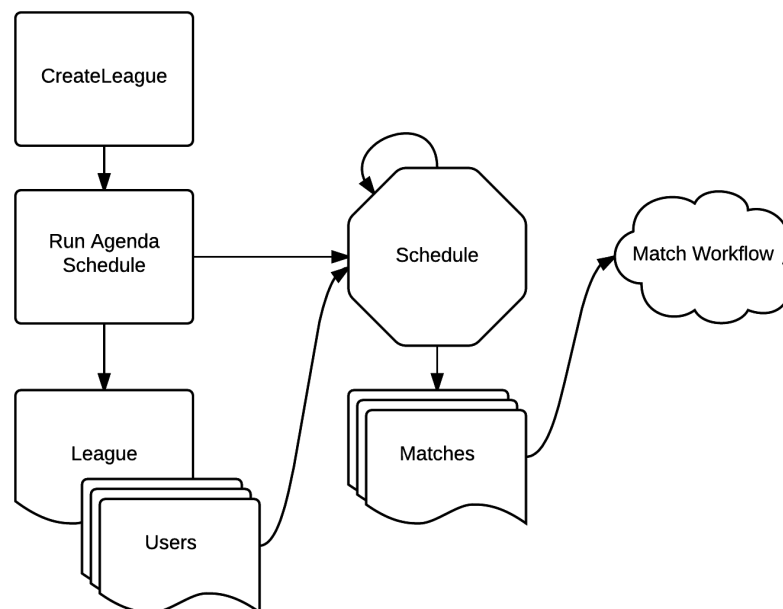


Abbildung 4.7: Schedule Workflow

### 4.8.2.1 Codebeispiel - Scheduling

Für das Scheduling wird eine Externe Scheduling Library für NodeJS gebraucht. Diese Library wird in den Startup der Applikation eingebunden:

```
1 var agenda = new Agenda({db: {address: config.dbagenda}});
2 console.log("starting agenda");
3 app.use('/agenda-ui', agendaUI(agenda, {poll: 1000}));
4 agenda.start();
```

Die Scheduling Library - weiterführen Referenziert mit dem Namen Agenda - benützt nun MongoDB um die Schedules zu persistieren. Wenn nun eine Liga erstellt wird, wird angegeben ob man Automatische Herausforderungen wünscht. Falls dies gewünscht ist, wird in der Funktion bei der Erstellung der Liga das Scheduling eingerichtet:

```
1 if (league.requestShedule){
2     var agenda = new Agenda({db: {address: config.dbagenda}})
3     console.log("execute schedulermatches sending ID: " + league);
4     agenda.define(league._id + ' schedule', League.scheduleMatches);
5     var job = agenda.create(league._id + ' schedule', league.id);
6     if (league.requestShedule === 'weeklyAll'){
7         job.repeatAt('in 1 weeks');
8     }
9     if (league.requestShedule === 'biweeklyAll'){
10        job.repeatAt('in 2 weeks');
11    }
12    if (league.requestShedule === 'biweeklyTwoTop'){
13        job.repeatAt('in 2 weeks');
14    }
15    if (league.requestShedule === 'weeklyTwoTop'){
16        job.repeatAt('in 1 weeks');
17    }
18
19    //agenda.every('10 minutes', league._id + ' schedule', league.id);
20
21    job.save()
22    agenda.start();
23 }
24
```

### 4.8.2.2 Random Herausforderungen / Rang Herausforderungen

Das Scheduling hat nun keinen Kontext der Applikation. Es beinhaltet nur das Model sowie alle Funktionen des Models. Aus diesem Grund wurde eine Funktion für das Starten der Herausforderungen innerhalb des Models erstellt:

```
1 LeagueSchema.statics.scheduleMatches = function(job){
2   var League = mongoose.model('League'),
3     UserToPoints = mongoose.model('UserToPoints'),
4     Matches = mongoose.model('Match');
5   var leagueid = job.attrs.data;
6   League.findById(leagueid).populate("users").exec(function(err, league) {
7     UserToPoints.populate(league.users, {path: 'user', select: 'username' }, function (err, user) {
8
9       if (league) {
10        var l = league.users.length;
11        var player1, player2;
12        var k = league.users.length;
13        while (true){
14          if(league.requestShedule == 'biweeklyTwoTop' || league.requestShedule == 'weeklyTwoTop' ) {
15
16            if (league.users.length > 1) {
17              player1 = league.users[0].user;
18              player2 = league.users[1].user;
19              league.users.splice(0,2);
20            }
21            else {
22              break;
23            }
24
25          }else {
26
27            if (league.users.length > 1) {
28              var p1 = Math.floor(Math.random() * l);
29              player1 = league.users[p1].user;
30              league.users.splice(p1, 1);
31              l--;
32              var p2 = Math.floor(Math.random() * l);
33              player2 = league.users[p2].user;
34              league.users.splice(p2, 1);
35              l--;
36
37            }
38            else {
39              break;
40            }
41
42          }
43          var match = new Matches();
44          match.spieler.push({user: player1});
45          match.spieler.push({user: player2});
46          match.sport = league.sport;
```



```
47         match.league = league;
48         match.state = 'new';
49         match.save(function (err) {
50             if (err) {
51                 console.log(err);
52             }
53         });
54         if (1 == 0) {
55             break;
56         }
57     }
58
59     } else {
60         console.log("no league with ID " + leagid + " found");
61     }
62 });
63 }
64 );
65
66 }
```

Bis zur Zeile 9 sucht sich nun diese statische Funktion der Kontext aus der Datenbank zusammen. Es sucht sich selber, popularisiert die User und startet ab Zeile 14 das erstellen von Herausforderungen.

Je nach Modus (Random Herausforderungen, oder nach Rangliste) wird nun der Rangliste nach Spieler herausgefordert und diese aus der Rangliste gelöscht. Da das Liga Objekt nicht gespeichert wird, sind die Änderungen der Rangliste nur temporär.

Ist nun der Modus Random, wird das ganze etwas Komplexer. Zuerst wird aus der Rangliste ein Spieler zufällig ausgewählt (Zeile 28). Dieser Spieler wird nun von der Liste gelöscht und der Zeite Spieler wird ausgewählt (Zeile 32). Dieser wird nun auch gelöscht und der Match zwischen den zwei Spielern wird vereinbart. Das ganze wiederholt sich, bis keine Spieler mehr in der Rangliste sind.

## 4.9 Match

### 4.9.1 Match Workflow

Der Matchworkflow ist das Hauptelement der Applikation. Der Workflow regelt, wie der Match als Business Prozess durchgeführt wird.

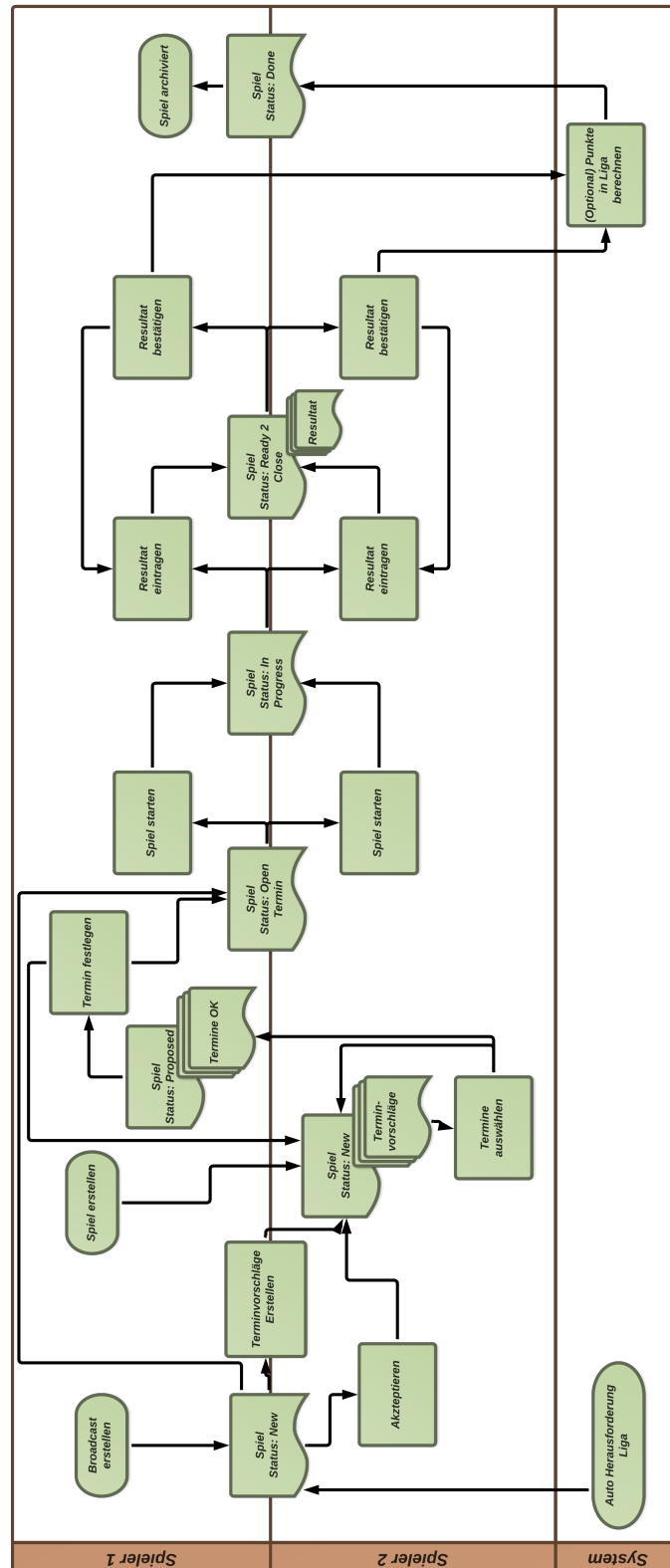


Abbildung 4.8: Spiel Workflow

Der Workflow wird über drei verschiedene Fälle gestartet:

- Das System erstellt Auto-Herausforderungen für die Liga
- Der User erstellt ein Broadcast Spiel
- Der User erstellt ein reguläres spiel.

Wenn der User ein **reguläres Spiel** erstellt, sind beide Spieler, sowie Terminvorschläge schon definiert. Es folgt die Aktion “Termin auswählen”.

Erstellt der User ein **broadcast Spiel**, hat das Spiel den Status “New“, jedoch noch keinen zweiten Spieler definiert. Zusätzlich werden keine Terminvorschläge ausgefüllt, sondern einen fixen Termin. Akzeptiert jemand den Broadcast wird der zweite Spieler eingetragen und der Status ändert sich direkt auf Open.

Sind User in einer Liga, erstellt die **Liga eine Herausforderung**. Das Spiel enthält kein Court und keine Terminvorschläge. Der User muss nun Terminvorschläge ausfüllen und ein Court definieren.

Anschliessend haben alle Use Cases den gleichen Workflow. Ist das Spiel und der Termin definiert. Geht der Status des Spiels zu “Open“. Danach kann von beiden Spielern der Status auf “In progress“ gesetzt werden. Beide können ein Resultat eintragen. Der jeweil andere Spieler bestätigt anschliessend das Resultat. Bei der Bestätigung des Resultats wird das Spiel archiviert und optional die Rangliste der Liga aktualisiert.

## 4.10 User

### 4.10.1 Freunde System

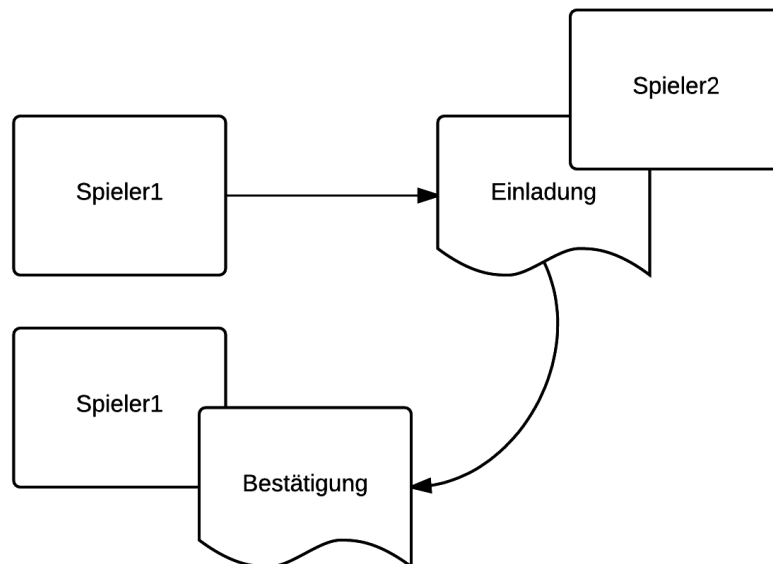


Abbildung 4.9: Freunde Workflow

Spieler1 sendet eine Einladung zur Freundschaft an Spieler 2. Diese Einladung ist eine Liste bei Spieler2 sowie bei Spieler1:

```
1 friendrequests: [{  
2     type: Schema.ObjectId ,  
3     ref: "User"  
4 }],
```

Bei der Bestätigung der Einladung trägt sich Spieler2 sowie Spieler1 das User-Objekt von dem friendrequests Feld in das friends Feld:

```
1 friends: [{  
2     type: Schema.ObjectId ,  
3     ref: "User"  
4 }],
```

# **5 Test**

## **5.1 Unit Tests**

## **5.2 System Tests**

## **5.3 User Acceptance Tests**

## **6 Reflektion**

# Abbildungsverzeichnis

1.1	Grobplanung für Applikation . . . . .	6
3.1	Grobkonzept für Applikation . . . . .	30
3.2	Detaillierte Applikations Architektur . . . . .	37
3.3	Datenbank Modell . . . . .	39
3.4	GUI Interaktions Modell . . . . .	40
3.5	GUI Design . . . . .	41
4.1	Projekt Ordner Struktur . . . . .	42
4.2	API für Spiele . . . . .	45
4.3	API für Ligen . . . . .	46
4.4	API für Racketsportzentren . . . . .	48
4.5	Racketsportzentrum Workflow . . . . .	53
4.6	Liga Workflow . . . . .	54
4.7	Schedule Workflow . . . . .	55
4.8	Spiel Workflow . . . . .	59
4.9	Spiel Workflow . . . . .	61