Projektbericht Trashpong

Januar 2023

Architektur

Im Rahmen der Architektur wird eine zentralisierte Architektur verwendet, wobei es sich um eine Client-Server-Architektur handelt (vgl. Abbildung 1). Die Programmteile werden in zwei separate Einheiten unterteilt, um eine klare Trennung zu gewährleisten. Der Server stellt einen bestimmten Service bereit und leitet die Anfragen an die Clients weiter. Die Clients empfangen diese Anfragen.[1] Der Client ist eine kompilierte Executable, geschrieben in der Open-Source Godot Engine [2]. Dort befindet sich die Gesamte Spiel Logik über die der Benutzer das Spiel spielen kann. Der Server befindet sich in einem Docker Compose bestehend aus einer Posgres Datenbank, einem Load Balancer und beliebig skalierbare Spieleserver welche die Anfragen der Clients verarbeiten.

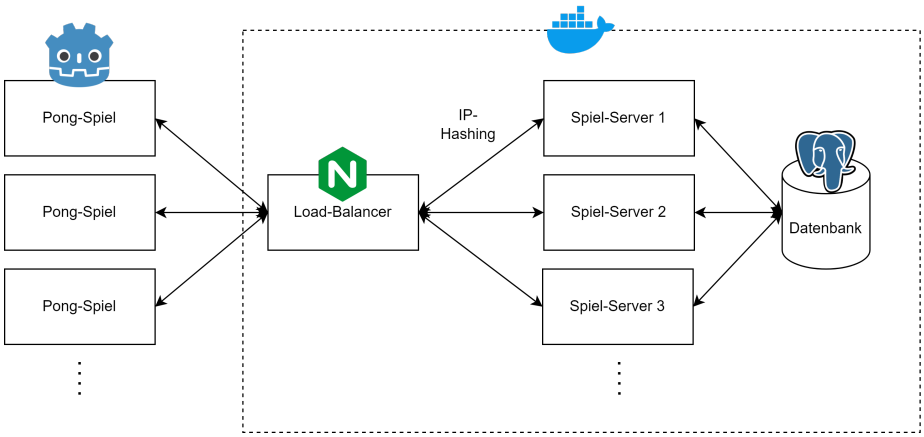

Abbildung 1: Kontext-Sicht der Client-Server-Architektur des Pong Spiels

Die Datenbank speichert die Räume in der die Spieler ihr Spiel spielen. Jeder Spieler kann Räume erstellen und ihnen beitreten und gehört somit diesen Räumen an (vgl. Abbildung 2). Ein Spieler kann zwar mehrere Räume erstellen, zum Zeitpunkt des Spiels befindet sich dieser nur in einem einzigen Raum.

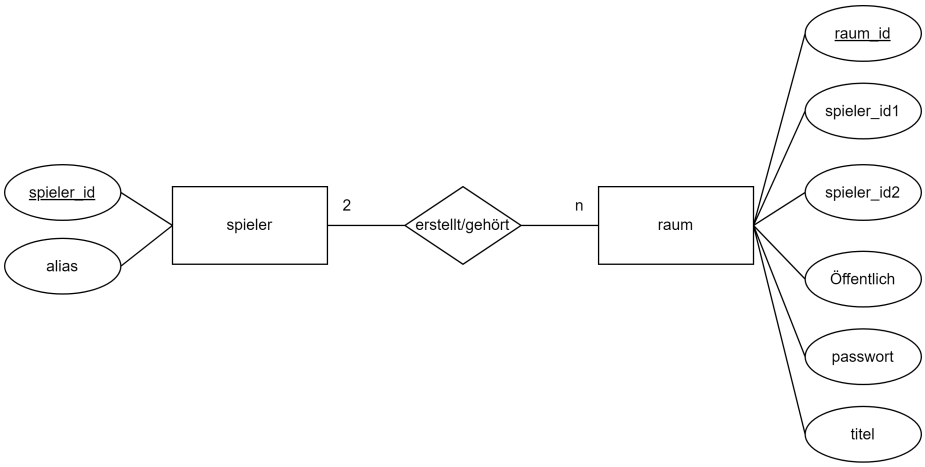

Abbildung 2: ER-Modell der Datenbankarchitektur

Dieser verwaltet die Daten der Spieler und die Spielstände. Außerdem werden die Daten für die Räume gespeichert. Die REST API ist in einem Docker Container gehostet und kann beliebig skaliert werden. Ein NGINX Load Balancer verteilt die Anfragen auf die einzelnen Server. „Viele Client-Server-Anwendung sind grundsetzlich aus drei unterschiedlichen Komponenten zusammengesetzt[...]“ [1,

S.57 ff.]. So interagieren auch die Clients mit einer Benutzerschnittstelle, der REST API. Diese wird verwendet um Daten über Nutzer oder Räume zu erhalten. Daten werden dabei über das HTTP Protokoll ausgetauscht. Die REST API kann dabei verwendet werde um Räume zu erstellen. Die REST Schnittstelle interagiert dabei mit Controllern, die die Logik der Anwendung implementieren und welche dann mit der Datenhaltung interagieren. Damit kann eine klare Trennung von Datenhaltung und Logik erreicht werden, welches die Anwendung modular macht. Es können dadurch andere Datenbanksysteme verwendet werden, ohne dass die Logik der Anwendung angepasst werden muss.

Die Abbildung 5 zeigt den Ablauf einer REST Anfrage um die Räume zu erhalten. Hierbei interagiert der Client mit der REST API, welche die Anfrage an den Controller weiterleitet. Der Controller interagiert dann mit der Datenbank um die Räume zu erhalten. Anschließend werden die Daten an den Client zurückgegeben.

Die Verteilung der Anwendungslogik auf Client und Server wird durch die Abbildung 3 verdeutlicht. Diese ist ein ausschnitt aus der Abbildung 2.5(c) aus [1, S.41]

Die Benutzerschnittstelle ist hierbei die kompilierte Godot Anwendung, welche die Spiellogik implementiert. Um Daten zu erhalten, sendet die Anwendung Anfragen an die REST API, welche Anwendungslogik implementiert hat um mit der Datenbank zu interagieren. Die Benutzeranwendung muss zudem eigene Logik implementieren, da diese aufgrund von Echtzeit Anforderungen nicht auf einen Server warten kann. Komponenten mit Echtzeit Anforderungen, die das Rendering, die übernahme von Benutzer eingaben müssen auf dem Client Computer ausgeführt werden.

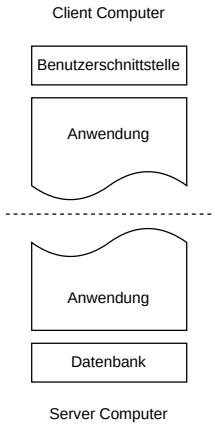

Abbildung 3: Aufteilung der Anwendungslogik in Client und Server

Architektur Entscheidung

Erklärung, warum diese Architektur gewählt wurde (z. B. Skalierbarkeit, Modularität, Sicherheit).

Innerhalb der Server Applikation setzen wir auf zwei Paradigmen. Einerseits haben wir uns bei der Datenhaltung für ein zentrlisiertes Client Server Modell entschieden. Dabei ist der Server „ein Prozess der einen bestimmt Dienst implementiert“ [1, S.55 ff.]. In Fall von Trash Pong ist dies eine REST API, die für die Datenhaltung zuständig ist.

Die zentralisierte Form hilft dabei

Systemkomponenten

Anforderungen

Spieler-Registrierung und -Login (FA1)
Beschreibung: Benutzer müssen sich mit ihren Namen anmelden können, um am Spiel teilzunehmen. Die Nutzung sollte niederschwelling sein. Ein Benutzer muss deshalb kein Konto anlegenc

Echtzeit-Multiplayer-Funktionalität (FA2)
Beschreibung: Das Spiel muss in der Lage sein, mehrere Spieler in Echtzeit zu verbinden und ein synchronisiertes Spiel zu ermöglichen. Die Bewegungen der Schläger und der Ball müssen in Echtzeit zwischen den Spielern synchronisiert werden.

Punkteverwaltung (FA3)
Beschreibung: Das System muss die Punkte der Spieler während des Spiels erfassen und verwalten können. Nach jedem Spiel sollte ein Punktestand angezeigt werden, der den Gewinner ermittelt.

Leistung und Skalierbarkeit (NF1)
Beschreibung: Das Spiel sollte auch bei hoher Benutzerzahl flüssig und ohne Verzögerungen laufen. Das System muss skalierbar sein, um eine große Anzahl von gleichzeitigen Spielern zu unterstützen.

Sicherheit (NF2)
Beschreibung: Die Benutzerdaten, einschließlich Anmeldedaten und Spielstatistiken, müssen sicher gespeichert und übertragen werden. Das System sollte gegen häufige Sicherheitsbedrohungen wie SQL-Injektionen und Cross-Site-Scripting geschützt sein.

Datensparsamkeit (NF3)
Beschreibung: Die Datenerhebung und -speicherung wird nur im notwendigen Maß durchgeführt. Ziel ist es, nur die Daten zu erfassen und zu speichern, die für den Betrieb und die Funktionalität des Systems unbedingt erforderlich sind. Dies trägt zum Schutz der Privatsphäre der Nutzer bei und reduziert das Risiko von Datenmissbrauch und -verlust. Deshalb sollen nur der Nutzername und die Punkte gespeichert werden.

Benutzerfreundlichkeit (NF4)
Beschreibung: Die Benutzeroberfläche des Spiels sollte intuitiv und leicht zu bedienen sein. Neue Spieler sollten sich schnell zurechtfinden und das Spiel ohne umfangreiche Anleitungen verstehen können.

Umsetzung

Implementierung

- **Umsetzung der Architektur**: Beschreibung der Implementierung der einzelnen Komponenten und ihrer Interaktionen.
- **Schwierigkeiten und Lösungen**: Was waren die technischen Herausforderungen und wie wurden sie bewältigt?

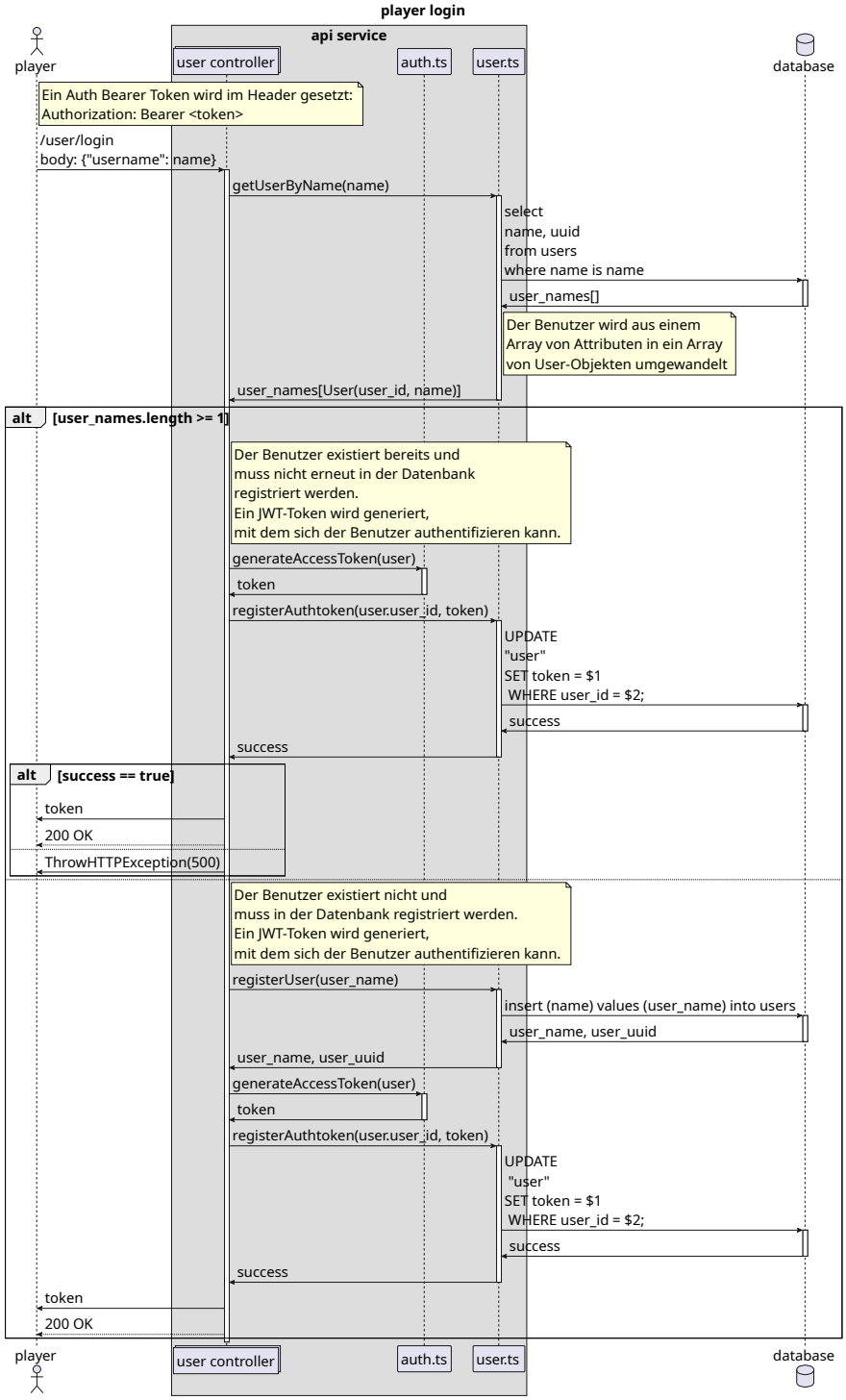

Abbildung 4: Dein TExt hier

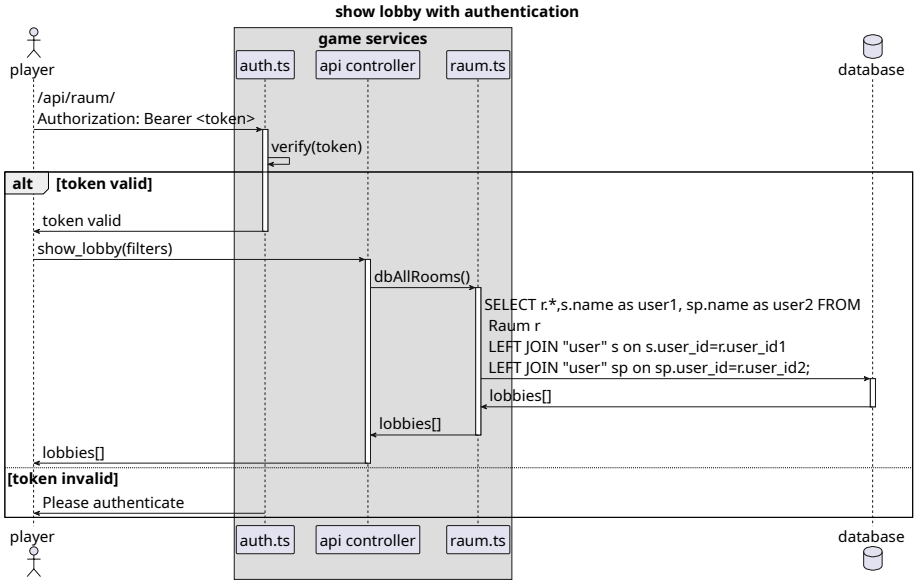

Abbildung 5: Dein TExt hier

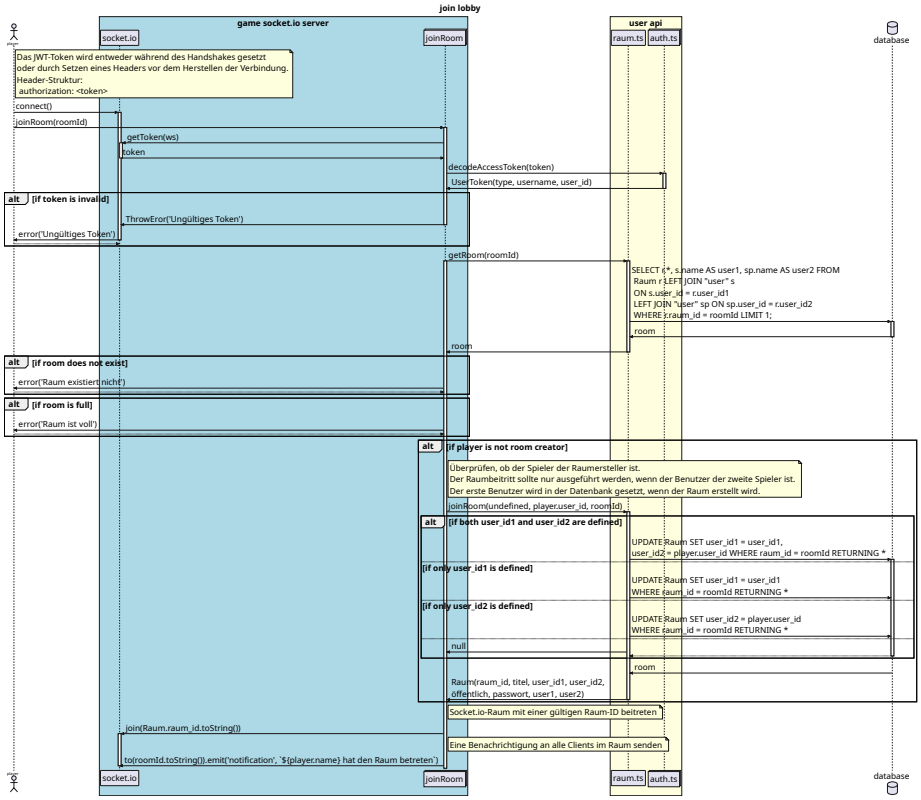

Abbildung 6: Dein TExt hier

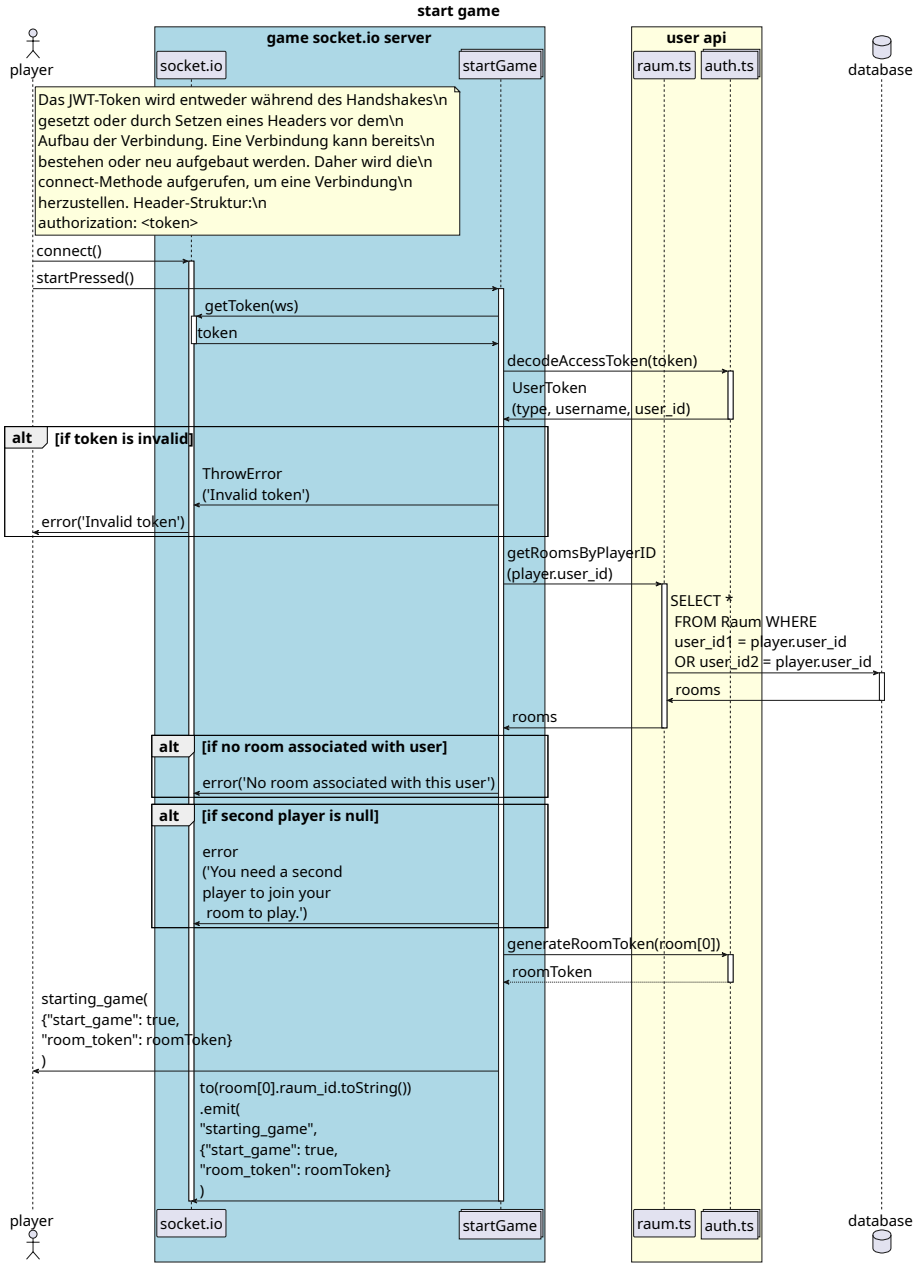

Abbildung 7: Dein TExt hier

Wurde das Spiel gestartet wird bei jedem Client die Pong Instanz geladen. Die Kommunikation passiert über einen Socket-IO Websocket. Über den Socket

senden die Clients dann verschiedene Events (ähnlich wie bei einem Publisher/Subscriber Prinzip [1]) welche dann auf der anderen Seite interpretiert werden. Die Clients senden nun die Positionen ihrer Paddles an den Server welcher diese an die Clients weitergibt. Prallt ein Ball an einem Paddle ab meldet der Client an dem der Ball abgeprallt ist dass dieser abprallt ist. Genauso passiert das wenn ein Tor geschossen wird (vgl. Abbildung 8).

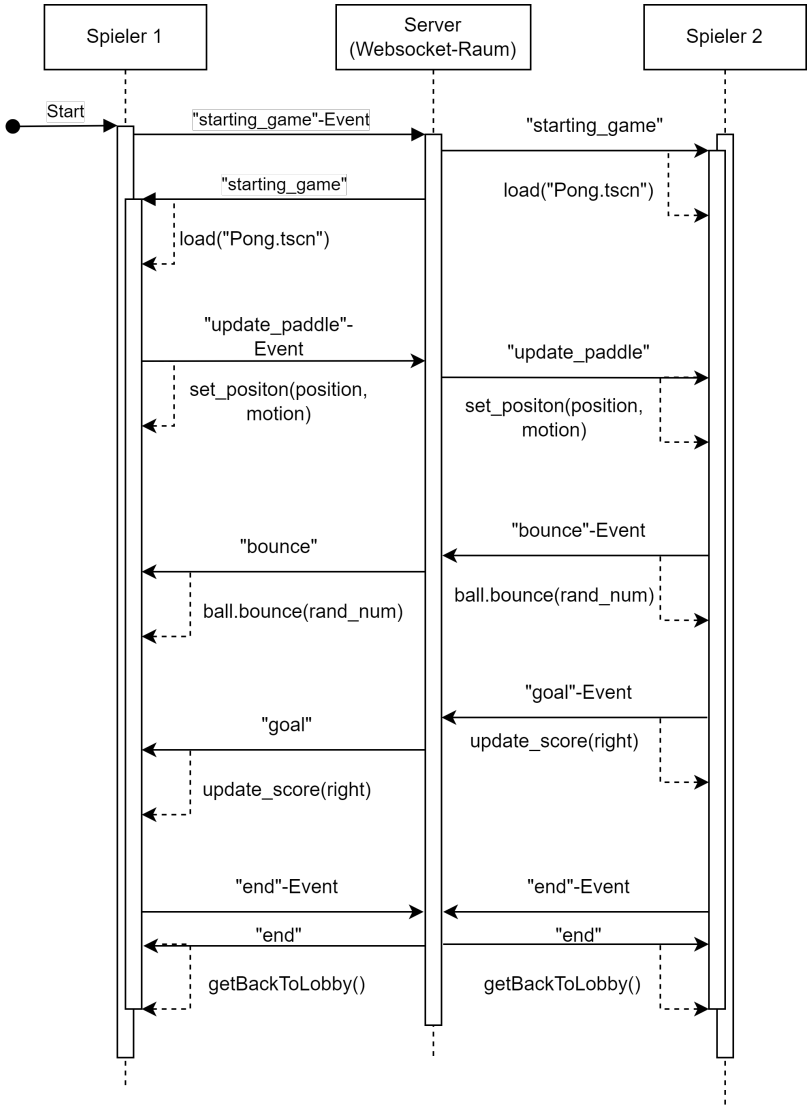

Abbildung 8: Eventbasierter Ablauf des Spiels über einen Socket-IO Websocket. Es handelt sich um eine Vereinfachte Ansicht. Beide Seiten können die jeweiligen Events an den Server senden.

Einer der Schwierigkeiten waren „stockende“ Bewegungen beim Spielablauf. Um dem Problem entgegenzuwirken Rendert jeder Client den Ball für sich selbst und nur die Änderungen, z.B. beim Aufprall werden übertragen. Außerdem musste sichergestellt werden, dass jeder Raum über einen sepearaten Channel kommuniziert. Hierfür eignete sich das Raum-Management Feature von Socket-IO , da die Verbindung & Synchronisation schon übernommen wurde und man einfach die Raum ID's aus der Datenbank sehr einfach einbinden konnte. So konnte ein höherer Programmieraufwand und damit Potenzielle Fehler vermieden werden [3].

Mögliche Alternativen

Anstelle einer zentralisierten Architektur wäre ein dezentralisierter Ansatz möglich. Es könnte ein Peer-To-Peer System Implementiert werden. Anstelle von einem zentralen System ist jeder Client miteinander direkt verbunden [4]. Im folgenden werden die Vor- und Nachteile eines Peer-To-Peer Systems erläutert:

Vorteile

- **Geringere Latenz** wenn die zwei Spieler eine direkte Verbindung zueinander herstellen
- **Load Balancer** wird nicht benötigt da jeder Client seine eigenen Ressourcen mitbringt und die Spiellogik dann von Peer to Peer abläuft

Nachteile

- **Synchronisation** ist bei einem P2P-Netzwerk komplexer herzustellen, anstatt mit einem zentralisiertem Server
- **Firewalls** „erschweren“ den Datenaustausch wenn ein Client keine Verbindung zulässt [4]

Reflexion

Rückblick

- **Änderungen nach dem Projekt**: Was würde man im Nachhinein anders machen, um das System zu verbessern?

Herausforderungen

- **Größte Herausforderungen**: Rückblick auf die bedeutendsten Schwierigkeiten und wie sie gelöst wurden.

Quellen

[1] A.S. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms*. Pearson Prentice Hall, 2007. ISBN: 9780136135531. URL: https://books.google.de/books?id=UKDjLQAACAAJ.

[2] *Godot Engine*. https://godotengine.org/. Zuletzt Aufgerufen 19.09.2024.

[3] *SocketIO Rooms*. https://socket.io/docs/v3/rooms/. Zuletzt Aufgerufen 19.09.2024.

[4] George Couloris et al. *Distributed Systems: Concepts and Design*. Pearson Prentice Hall, 2001. ISBN: 9780132143011. URL: https://github.com/lijasonvip/Books_Reading/blob/master/George-Coulouris-Distributed-Systems-Concepts-and-Design-5th-Edition.pdf.