



Projektbericht Trashpong

Aron Seidl, Maylis Grune, Carolin Enderlein

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 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 Anwendung, geschrieben in der Open-Source Godot Engine [2], welcher die Services des Servers nutzt. Die Verteilung der Anwendungslogik, auf Client und Server, wird durch die (Abbildung 2) verdeutlicht. Der Client besitzt dabei die gesamte Spiellogik, über die der Benutzer das Spiel spielen kann. Die Benutzerschnittstelle ist hierbei die kompilierte Godot Anwendung, welche die Spiellogik implementiert.

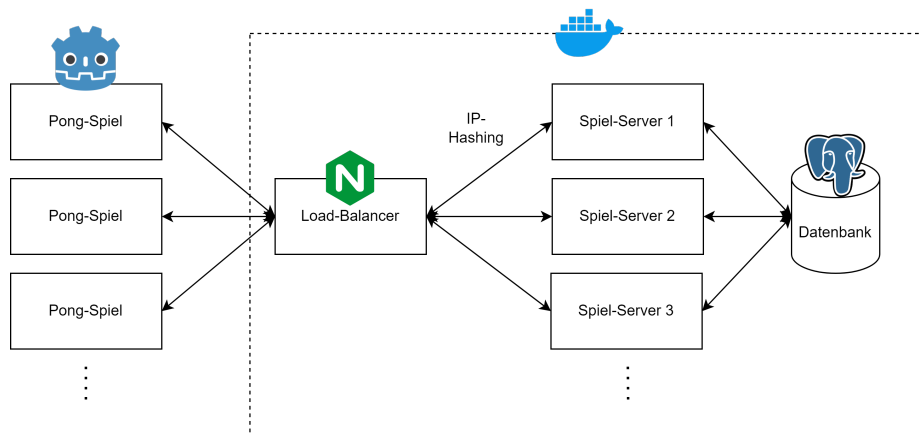


Abbildung 1: Kontext-Sicht der Client-Server-Architektur des Pong Spiels

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 Benutzereingaben müssen auf dem Client Computer ausgeführt werden. Server wie auch Client besitzen demnach ihre eigene Anwendungslogik, die sich auf die jeweiligen Anforderungen spezialisiert haben.

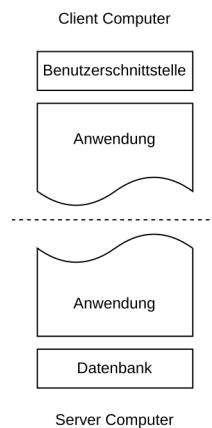


Abbildung 2: Aufteilung der Anwendungslogik in Client und Server

Der Server kann über eine Docker Compose bereitgestellt werden. Innerhalb dieses Deployments wird eine Postgres-Datenbank, ein Nginx-Load-Balancer und ein beliebig skalierbarer Spieleserver, welcher die Anfragen der Clients verarbeitet, bereitgestellt.

Server Architektur

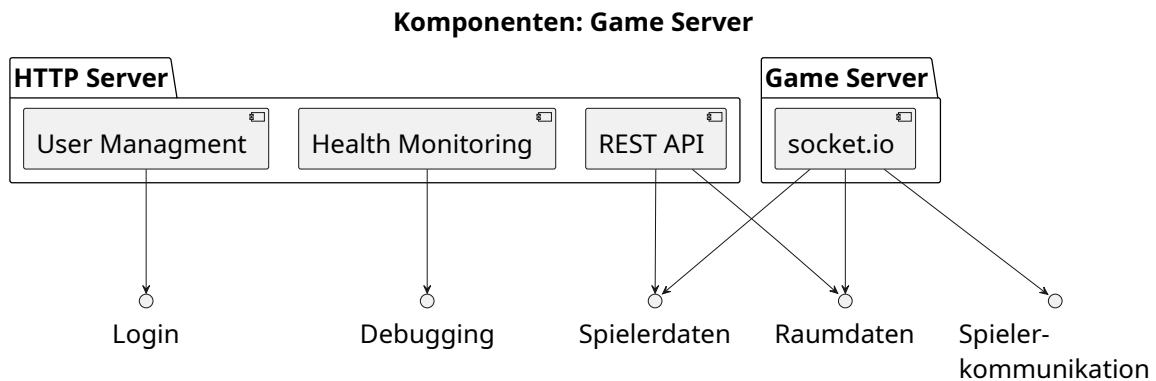


Abbildung 3: Komponentensicht der Serveranwendung

Der Server besteht aus zwei Komponenten, dem HTTP Server und dem Game Server. Der HTTP Server verfolgt dabei eine „[...] client-server Interaktion, auch bekannt als request-reply Verhalten [...]“ [1, S.37 ff.]. Diese Komponente, wie sie in (Abbildung 3) dargestellt ist, bietet dem Client Benutzerschnittstellen an um sich anzumelden, Räume zu erstellen und diesen beizutreten. Der HTTP Server ist dabei verbindungslos, das bedeutet, dass der Server nachdem er die Anfrage bearbeitet hat, die Verbindung schließt. Der Server hat dabei keine Referenz zu dem Client, der die Anfrage gestellt hat. Was für Anwendung, die Daten bereitstellen sollen die einfachste Form der Kommunikation dargestellt.

Innerhalb des Spiels, sind verbindungslose Protokolle nicht ausreichend, da hier eine Echtzeitkommunikation benötigt wird. Websockets wären hier ein Protokoll, welches einen verbindungsorientierten Ansatz verfolgt. Diese „[...] verwenden eine einzelne TCP-Verbindung für den Datenverkehr in beide Richtungen.“ [3, Kapitel 1.1]. Somit kann mit Websockets ein bidirektionaler Kommunikationskanal aufgebaut werden mit dem die Clients und der Server in Echtzeit kommunizieren können. Spieleinformationen können so zwischen den Clients ausgetauscht werden. Der Server wird dabei als Vermittler verwendet. Um dies zu ermöglichen wird die Socket.io Bibliothek verwendet. Socket.io ergänzt das WebSocket Protokoll und ermöglicht es Räume zu erstellen und zu verwalten. Damit können mehrere Clients innerhalb eines Raums miteinander kommunizieren.

Die Datenbank speichert die Räume, in denen die Spieler ihr Spiel spielen. Jeder Spieler kann Räume erstellen und ihnen beitreten und gehört somit diesen Räumen an (vgl. Abbildung 4). Ein Spieler kann dabei mehrere Räume erstellen, so ist es diesem auch möglich, in mehreren Räumen gleichzeitig zu sein. Zum Zeitpunkt des Spiels kann der Spieler nur in einem Raum sein.

Zum Ausgleichen von Anfragen wird ein Nginx Load Balancer verwendet. Dieser verteilt die Anfragen auf die verschiedenen Serverinstanzen. Da für die Echtzeitkommunikation Websockets verwendet werden, ist es notwendig, dass der Verbindung eines Clients immer auf die gleiche Serverinstanz leiten. Dafür wird für die Verteilung der Anfragen IP-Hashing verwendet. Der Load Balancer speichert dabei die IP-Adresse des Clients und leitet die Anfragen immer an die gleiche Serverinstanz weiter.

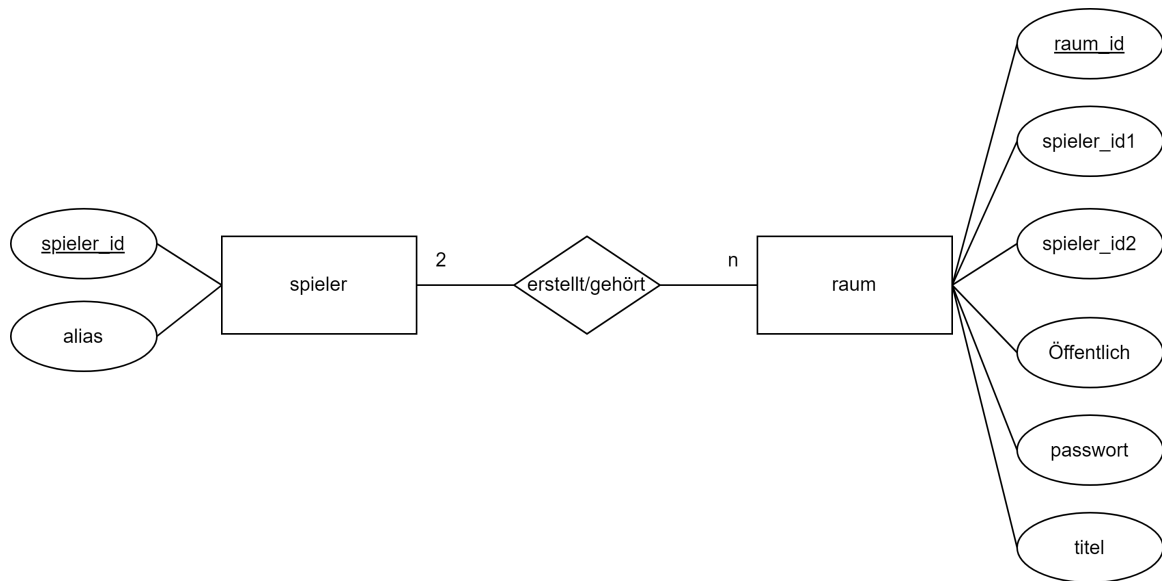


Abbildung 4: ER-Modell der Datenbankarchitektur

Client Architektur

Komponenten: Game Client

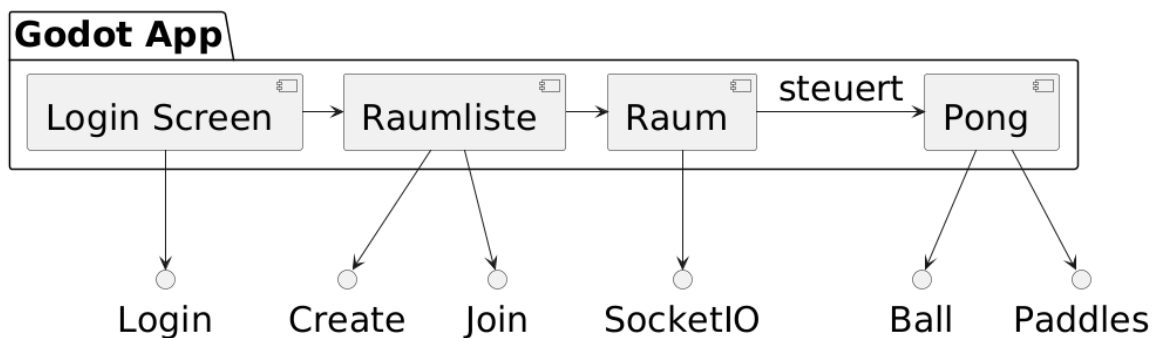


Abbildung 5: Komponentensicht des Clients.

Der Client besteht aus 4 Komponenten. Im Login Screen authentifiziert sich der Client mit dem Server verbindungslos. Auch das erstellen und betreten der Räume ist nur über REST-Anfragen gehandelt. Erst wenn ein Spieler den Raum beitrifft wird mit Websockets ein bi-direktionaler Kommunikationskanal erstellt (vgl. Server Architektur). Über diesen Kanal werden dann die Spielelemente des Pong Spiels gesteuert.

Architektur Entscheidung

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

„Viele Client-Server-Anwendung sind grundsetzlich aus drei unterschiedlichen Komponenten zusammengesetzt[...]“ [1, S.57 ff.]. Die Clients interagieren daher auch mit einer Benutzerschnittstelle. Dabei handelt es sich um ein HTTP Server, der die Anfragen der Clients entgegennimmt. Der HTTP Server unterstützen dabei HTTP Anfragen, genauso wie Websocketverbindungen. Diese wiederum 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.

Innerhalb der Server Applikation wird auf zwei Paradigmen. Einerseits wird auf das request-replay Paradigma gesetzt, um Nutzer zu autorisieren, Räume zu erstellen und zu verwalten. Hierbei ist keine Echtzeitkommunikation notwendig, da die Anfragen nur einmalig durchgeführt werden. Auf der Game Server Seite wurde sich dabei für eine nachrichtenbasierte Kommunikation entschieden, um die Anforderungen der Echtzeitkommunikation zu erfüllen. Diese Entscheidung wurde getroffen, weil synchrone Kommunikationsmethoden wie RPCs (Remote Procedure Calls) nicht immer geeignet sind, insbesondere in Szenarien, in denen die empfangende Seite möglicherweise nicht sofort verfügbar ist.

„Insbesondere, wenn nicht davon ausgegangen werden kann, dass die empfangende Seite zum Zeitpunkt der Anforderung ausgeführt wird, sind alternative Kommunikationsdienste erforderlich. Ebenso muss die inhärent synchrone Natur von RPCs, bei der ein Client blockiert wird, bis seine Anforderung verarbeitet wurde, manchmal durch etwas anderes ersetzt werden.“ [1, S. 140 ff.]

Nach Tannenbaum eignen sich relationale Datenbanken um Anwendungslogik von den Daten zu trennen. „Die Verwendung relationaler Datenbanken im Client-Server-Modell hilft dabei, die Verarbeitungsebene von der Datenebene zu trennen, da Verarbeitung und Daten als unabhängig betrachtet werden.“ [1, S. 40 ff.] Zudem ermöglichen relationale Datenbanken eine einfache Skalierung, der Serveranwendungen. Ein weiterer Vorteil ist die eingebaute Replikationsfunktion, welche die Datenbanken synchronisiert und hohe Verfügbarkeit gewährleistet.[4, Chapter 27]

(Nicht-)Funktionale Anforderungen

Spieler-Registrierung und -Login (FA1)
Beschreibung: Benutzer müssen sich mit ihren Namen anmelden können, um am Spiel teilzunehmen. Die Nutzung sollte niederschwellig 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.

<p>Leistung und Skalierbarkeit (NF1)</p> <p>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.</p>
<p>Sicherheit (NF2)</p> <p>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.</p>
<p>Datensparsamkeit (NF3)</p> <p>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 Nutzernamen und die Punkte gespeichert werden.</p>
<p>Benutzerfreundlichkeit (NF4)</p> <p>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.</p>

Umsetzung

- **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?

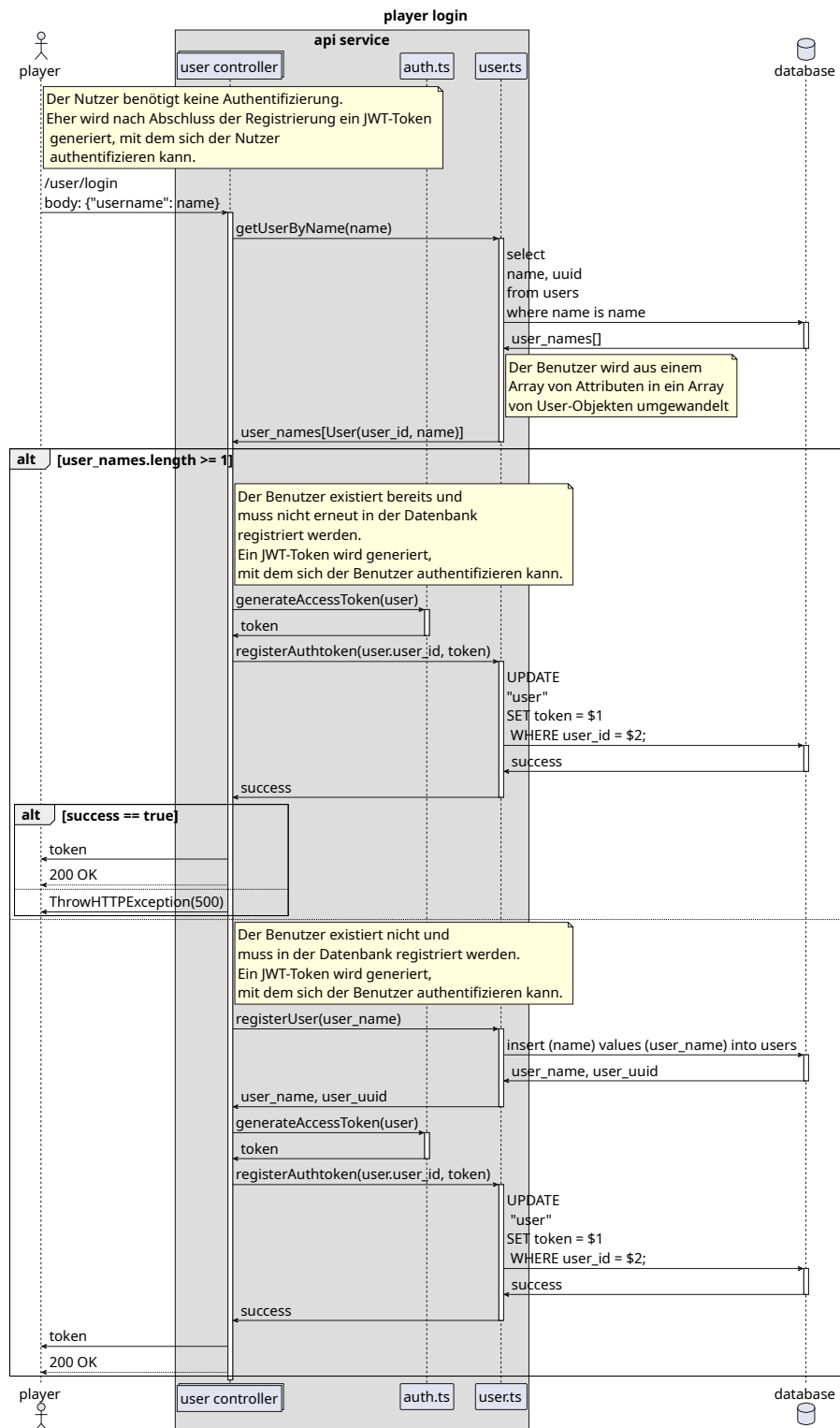


Abbildung 6: Zeigt den Ablauf einer Nutzerregistrierung.

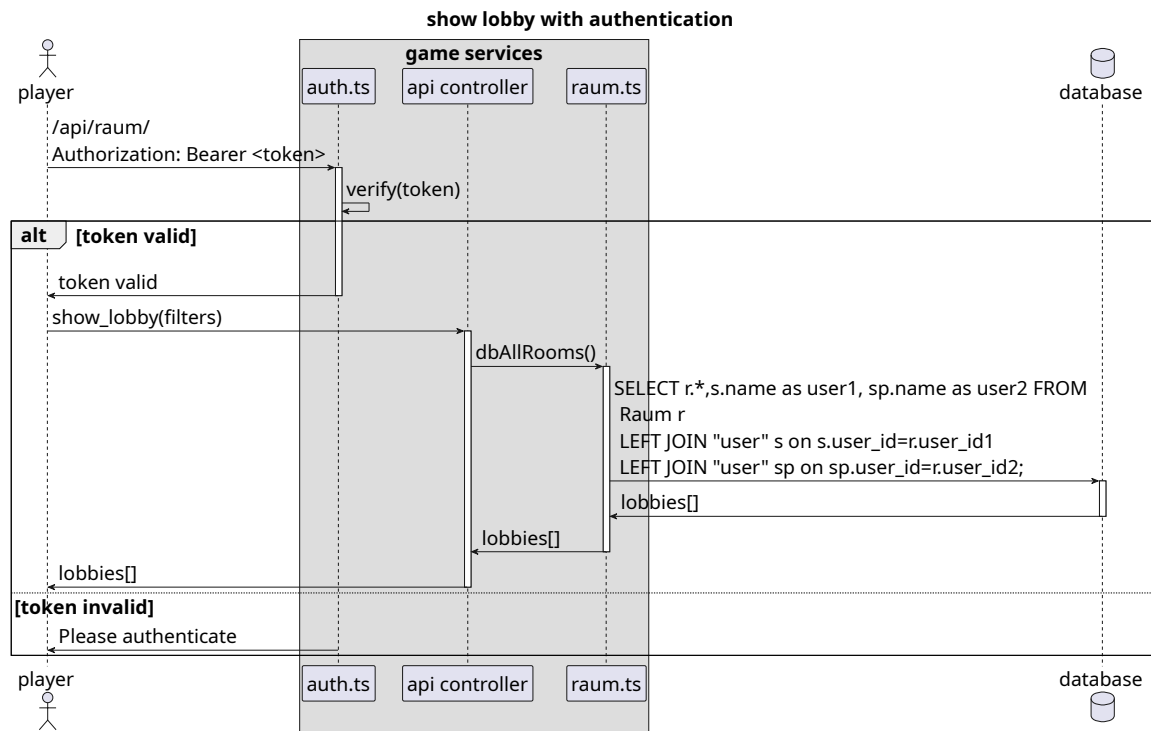


Abbildung 7: Zeigt den Ablauf der Abfrage aller Räume in der Datenbank, welche nach dem Login ausgeführt wird. Hier wird ein JWT-Token benötigt, um die Anfrage zu autorisieren.

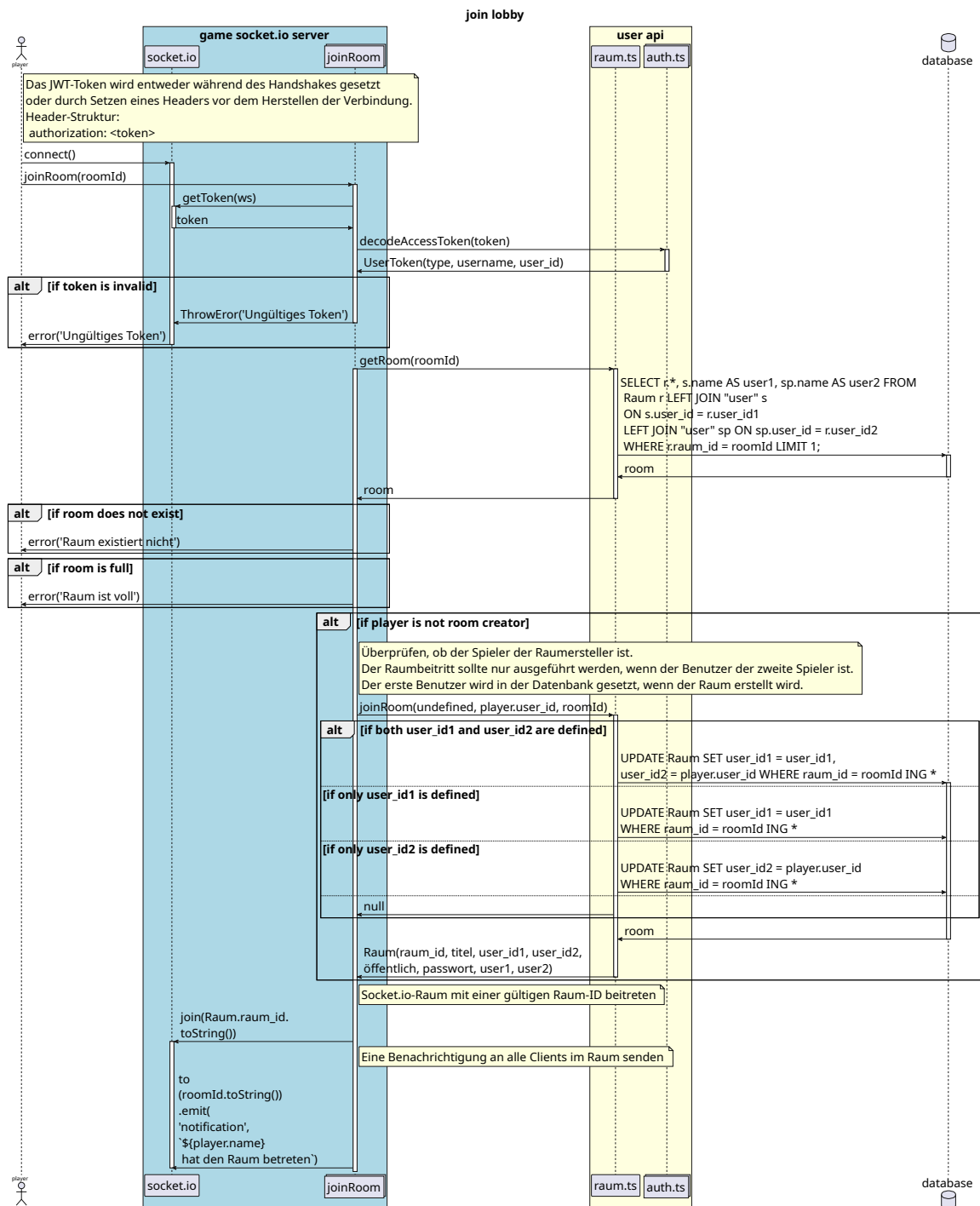


Abbildung 8: Der Nutzer tritt einem Raum bei. Hierbei wird ein JWT-Token benötigt um die Anfrage zu autorisieren. Der Nutzer verbindet sich mit dem Websocket und sendet eine joinRoom Anfrage. Der Server prüft ob der Nutzer die nötigen Berechtigungen hat um in den Raum beizutreten. Wenn der Nutzer alle Berechtigungen hat wird er in den Raum hinzugefügt und empfängt nun Nachrichten von anderen Nutzer.

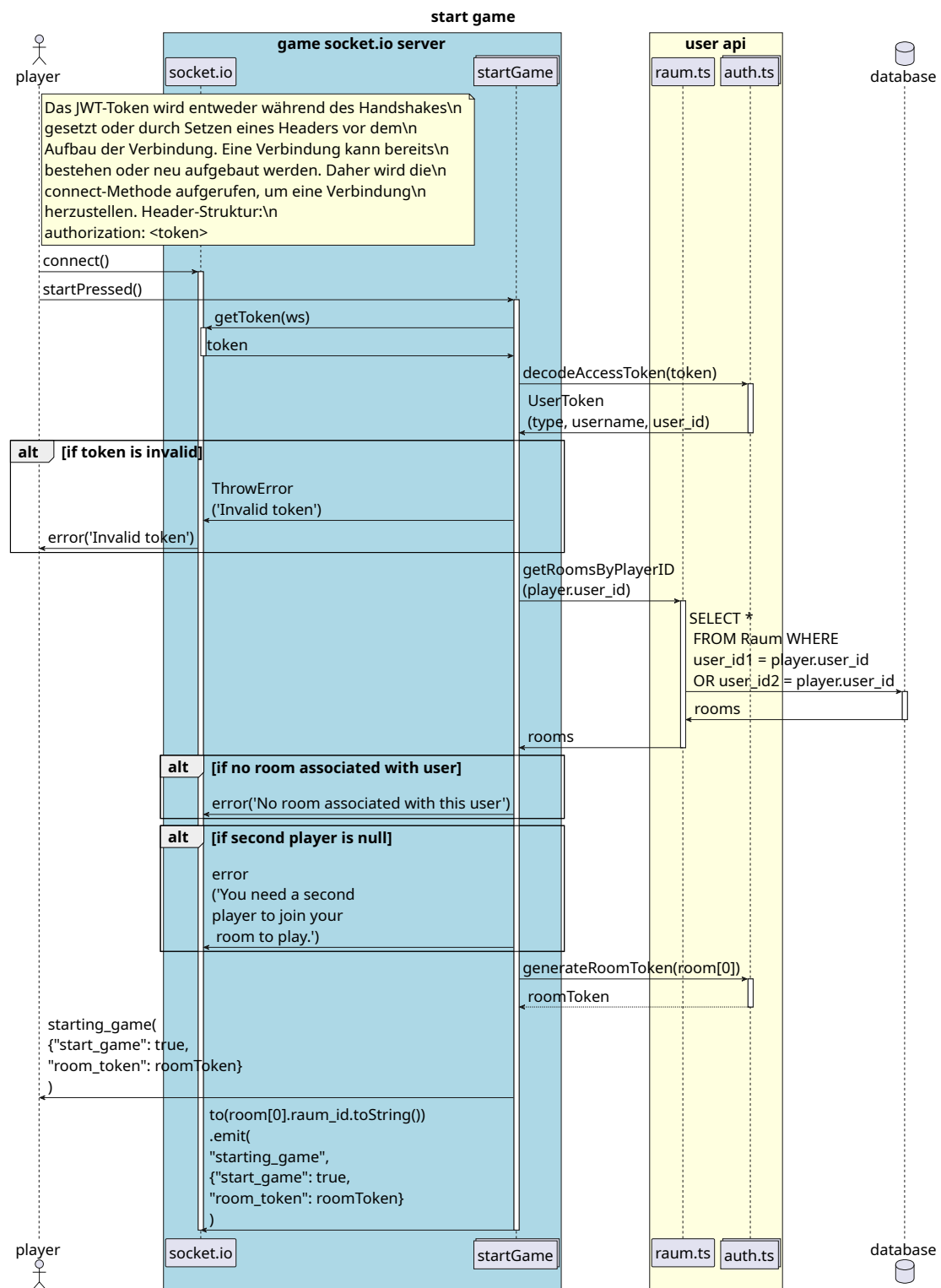


Abbildung 9: Drückt einer der Nutzer den Start-Button in der Benutzeroberfläche wird ein Event an den Server gesendet. Dieser Prüft ob der Nutzer innerhalb eines Raumes registriert wurde. Wenn der Nutzer alle Berechtigungen erfüllt wird diesem ein Room Token übermittelt mit dem der Nutzer weitere Informationen authentifizieren kann.

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 10).

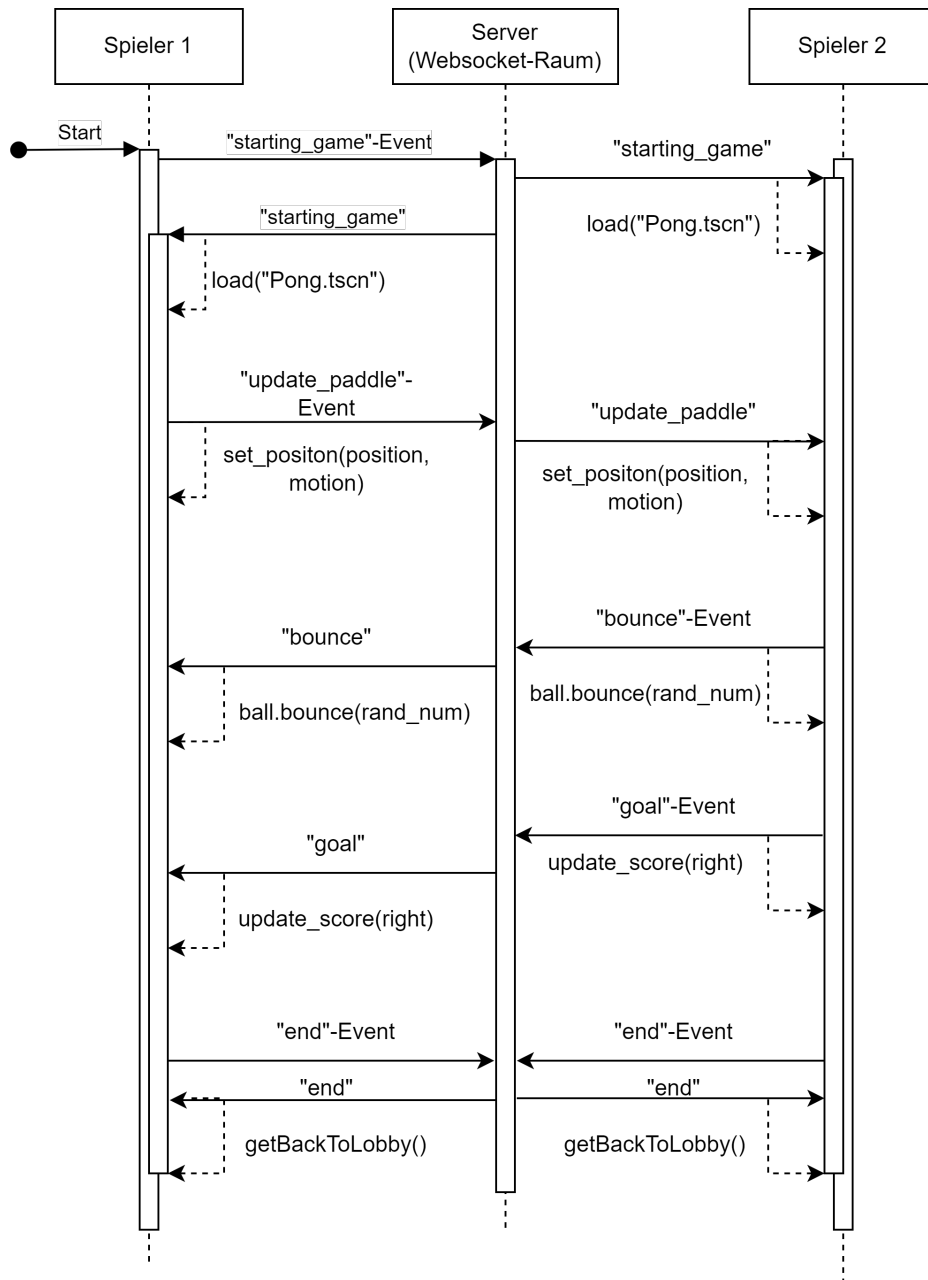


Abbildung 10: Eventbasierter Ablauf des Spiels über einen Socket-IO Websocket. Es handelt sich um eine Vereinfachte Ansicht. Beide Seiten können die jeweiligen Events an den Server senden.

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

separaten 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 [5].

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 [6]. 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
- Ein 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 zentralisierten Server
- Firewalls „erschweren“ den Datenaustausch wenn ein Client keine Verbindung zulässt [6]

Reflexion

Rückblick & Herausforderungen

- **Änderungen nach dem Projekt:** Was würde man im Nachhinein anders machen, um das System zu verbessern?
- **Größte Herausforderungen:** Rückblick auf die bedeutendsten Schwierigkeiten und wie sie gelöst wurden.

Rückblickend gesehen wäre es besser gewesen die gesamte Logik der Raumliste mit Websockets aufzubauen. Anstatt die Räume mit einer REST-API abzufragen, könnten Aktualisierungen der Räume in Echtzeit übertragen werden. So müsste der Benutzer nicht Manuell die Räume abfragen und potenzielle Race-Conditions, wie beim Raumbeitritt könnten vermieden werden. Außerdem müssten damit nicht die gesamten Räume vom Server gefetcht werden sondern es könnte zu Beginn einmal alles gefetcht werden und nur noch Änderungen müssten übertragen werden. Eine große Herausforderung war außerdem die Verwendung der Programmiersprache Typescript. Da alle Teilnehmer des Projektes nahezu keine Erfahrung mit Typescript hatten, ist viel Zeit drauf gegangen um Bugs zu fixen welche aufgrund vom Datentypen handling der Programmiersprache entstanden sind. In Zukunft wird für das Backend eine andere Sprache verwendet.

Für die Verwaltung von Daten wäre eine objektorientierte Datenbank wie z.B. MongoDB besser geeignet gewesen, da Trash Pong die Komplexität für eine relationale Datenbank nicht mitbringt.

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] IETF. *RFC 6455 - The WebSocket Protocol*. Zuletzt Aufgerufen 19.09.2024. 2011. URL: <https://tools.ietf.org/html/rfc6455>.
- [4] postgresql. *Chapter 27. High Availability, Load Balancing, and Replication*. Zuletzt Aufgerufen 19.09.2024. 2024. URL: <https://www.postgresql.org/docs/current/warm-standby.html#SYNCHRONOUS-REPLICATION>.
- [5] *SocketIO Rooms*. <https://socket.io/docs/v3/rooms/>. Zuletzt Aufgerufen 19.09.2024.
- [6] George Coulouris 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.