Projektbericht Trashpong

Aron Seidl, Maylis Grune, Carolin Enderlein

August 2024

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.

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.

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

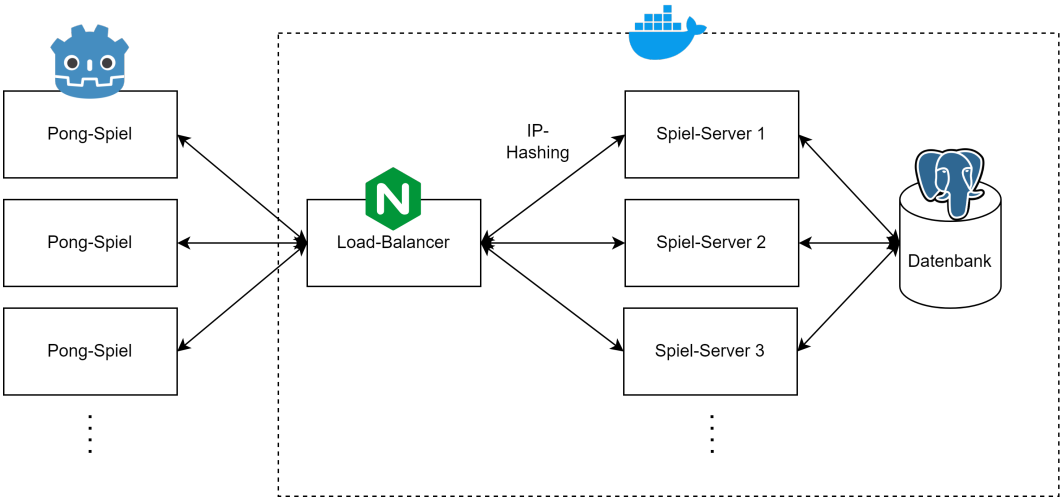

Abbildung 1: Kontext-Sicht der Client-Server-Architektur des Pong Spiels. Dabei zeigt die Kontextsicht die Interaktion zwischen den Komponenten. Der zentrale Punkt der Interaktion ist dabei der Nginx Load Balancer, welcher die Anfragen auf die verschiedenen Serverinstanzen verteilt. Die Server nutzen dabei die Postgres Datenbank zur Synchronisation und Speicherung der Daten.

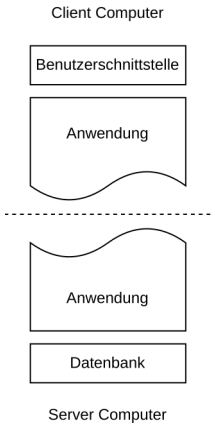

Abbildung 2: Aufteilung der Anwendungslogik in Client und Server nach Tannenbaum

Server Architektur

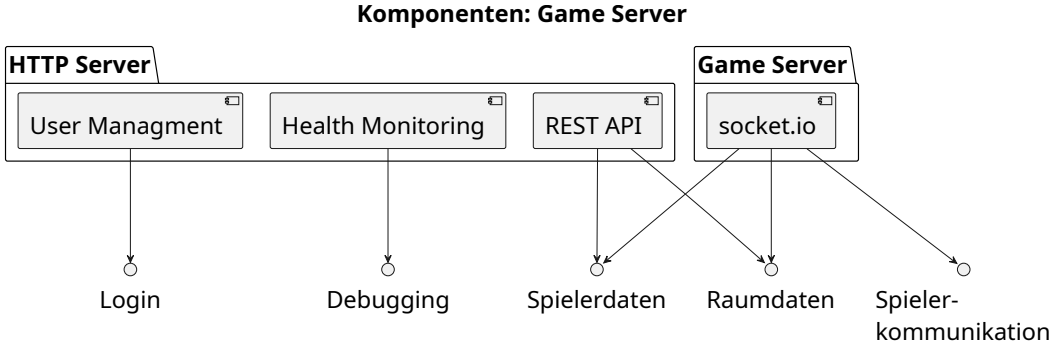

Abbildung 3: Komponentensicht der Serveranwendung

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 die Komponente, die Daten bereitstellen sollen die einfachste Form der Kommunikation darstellt.

Innerhalb des Spiels sind verbindungslose Protokolle nicht ausreichend, da hier eine Echtzeitkommunikation benötigt wird. Websockets wären hier eine Technologie, welche einen verbindungsorientierten Ansatz verfolgen. 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. 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 Websockets Sessions verwendet werden, ist es notwendig, dass die Verbindung eines Clients immer auf die gleiche Serverinstanz geleitet wird. Aufgrunddessen wird für die Verteilung von Anfragen an den Server IP-Hashing verwendet. Der Load Balancer speichert dabei die IP-Adresse des Clients und und hasht diese auf die entsprechne Serverinstanz, sodass die Anfragen des gleichen Nutzers immer auf die gleiche Serverinstanz weitergeleitet werden. Über einen Adapter wird der Websocketverkehr synchronisiert, sodass diese beliebig skaliert werden können.

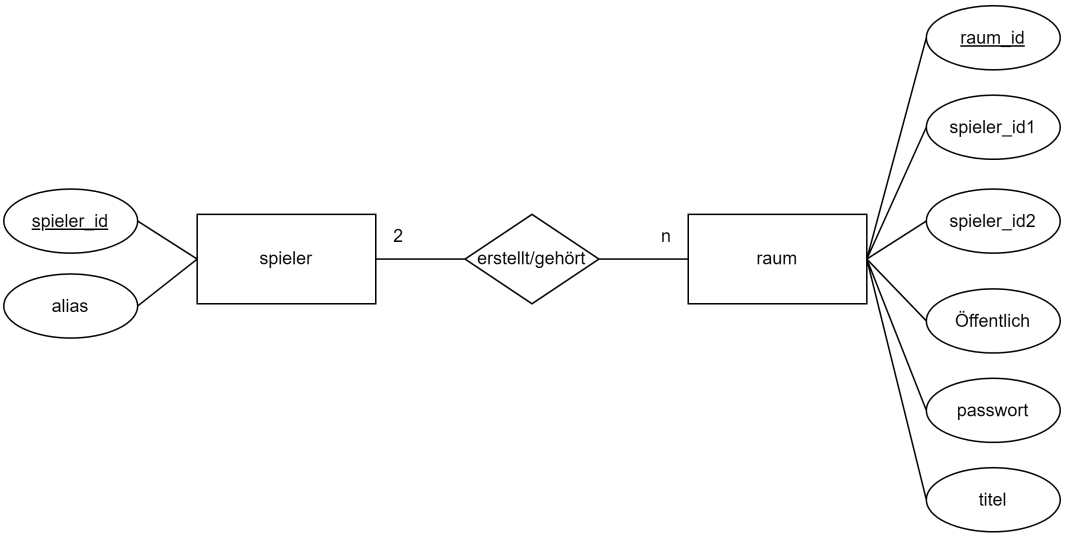

Abbildung 4: ER-Modell der Datenbankarchitektur. Beinhaltet die zwei Entitäten User und Raum. Der Raum verlinkt über spieler_id1 und spieler_id2 über einen Fremdschlüssel auf den Spieler. Dabei kann der Raum keine, einen oder zwei Spieler haben.

Client Architektur

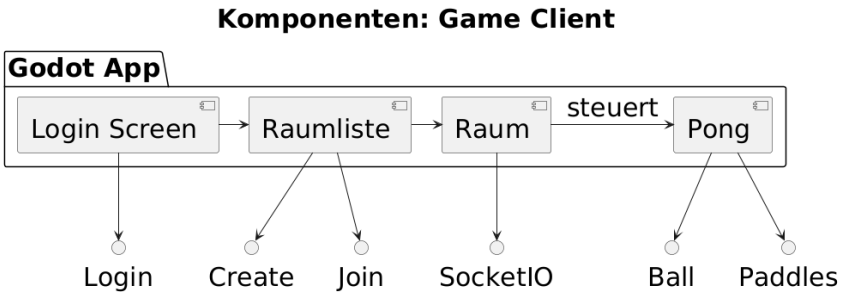

Abbildung 5: Komponentensicht des Clients.

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

Architektur Entscheidung

„Viele Client-Server-Anwendung sind grundsätzlich 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 unterstützt dabei HTTP-Anfragen, genauso wie Verbindungen über Websockets. 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 gesetzt. Einerseits wird auf das request-reply 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, vor allem 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.]

Deshalb kam für die nachrichtenbasierte Kommunikation socket.io zum Einsatz, welches die Funktionen von Websockets erweitert. Das unter dem socket.io liegenden Engine.IO ermöglicht dabei latenzarme Kommunikation mit geringem Overhead.[4]

socket.io bietet zudem eine einfache Möglichkeit zur Skalierung. Durch die Integration eines Adapters können nun verschiedene socket.io Server miteinander kommunizieren. Damit erhalten mehrere Server die Möglichkeit, ihre Zustände zu synchronisieren. Dies wird mithilfe des Postgres Adapters implementiert, welche auf die Listen und Notify Funktionen von Postgres zurückgreift [5].

Zur Authentifizierung werden JWT-Token verwendet. Dabei ist der Vorteil, dass der Server keine Session speichern muss. Der übertragende Token, kann dabei vom Client gespeichert werden. Die Informationen im Token sind dabei signiert und würden bei einer Manipulation des Tokens nicht mehr gültig sein. Übertragen wird ein Token beim Login und sobald der Start Game Button innerhalb des Spiels gedrückt wird, diesen empfangen beide Spieler.

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.[6, 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 anlegen
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

Die Umsetzung der Architektur erfolgte parallel zur Entwicklung des Spiels. Dabei wurden zuerst die Anforderungen an die Anwendungen definiert. Dies wurde als Grundlage für die Implementierung der Anwendungen verwendet. Während der Entwicklung entstanden immer neue Anforderungen, die parallel umgesetzt wurden.

Im ersten Schritt wurde eine Modellierung der Use Cases durchgeführt, woraufhin benötigte Daten definiert wurden. Daraufhin konnte eine Technologieauswahl getroffen werden. Mithilfe dieser wurde ein erstes Verteilungsdiagramm erstellt, welches mit der Entwicklung der Anwendung immer weiter verfeinert wurde. Dabei wurde auf die Trennung von Datenhaltung und Anwendungslogik geachtet. Diese wurde mithilfe von Sequenzdiagrammen dargestellt. Mithilfe dieser konnte die Funktionsweise der Anwendung durchdacht werden. Mithilfe dieser Diagramme wurde die Anwendung nun entwickelt.

Zu Beginn wurden die REST API, Nutzerverwaltung als auch Authentifizierung und Datenhaltung implementiert. Dabei wurde schnell festgestellt, dass durch die Zustandslosigkeit der REST API, keine Session Informationen gespeichert werden. Deshalb wurde auf JWT-Token gesetzt, um die Authentifizierung zu gewährleisten. Mithilfe dieses Tokens können Nutzerinformationen gespeichert werden, ohne dass der Server eine Session speichern muss.

In (Abbildung 6) wird der Ablauf der Nutzerregistrierung dargestellt. Der Nutzer sendet eine Anfrage an den Server, um sich zu registrieren. Der Server prüft, ob der Nutzer bereits in der Datenbank vorhanden ist. Ist dies nicht der Fall, wird der Nutzer in der Datenbank gespeichert und ein JWT-Token generiert. Existiert der Nutzer bereits, kann der Token als Antwort zurückgegeben werden, nur muss hier der Token in der Datenbank aktualisiert werden. Mithilfe dieses Tokens kann sich der Nutzer nun authentifizieren und auf die Ressourcen des Servers zugreifen.

Die REST API wurde mithilfe von Express.js implementiert, dabei wurde eine Middleware implementiert, die die Anfragen des Clients überprüft und den Nutzer authentifiziert. Die API akzeptiert dabei nur das JSON-Format.

Die Datenhaltung wurde mithilfe von Postgres realisiert. Dabei wurde ein ER-Modell erstellt, welches die Beziehungen zwischen den Tabellen darstellt. Die Abfragen werden dabei von Datenbank-Modellen durchgeführt, wodurch die Datenhaltung von der Anwendungslogik getrennt wird.

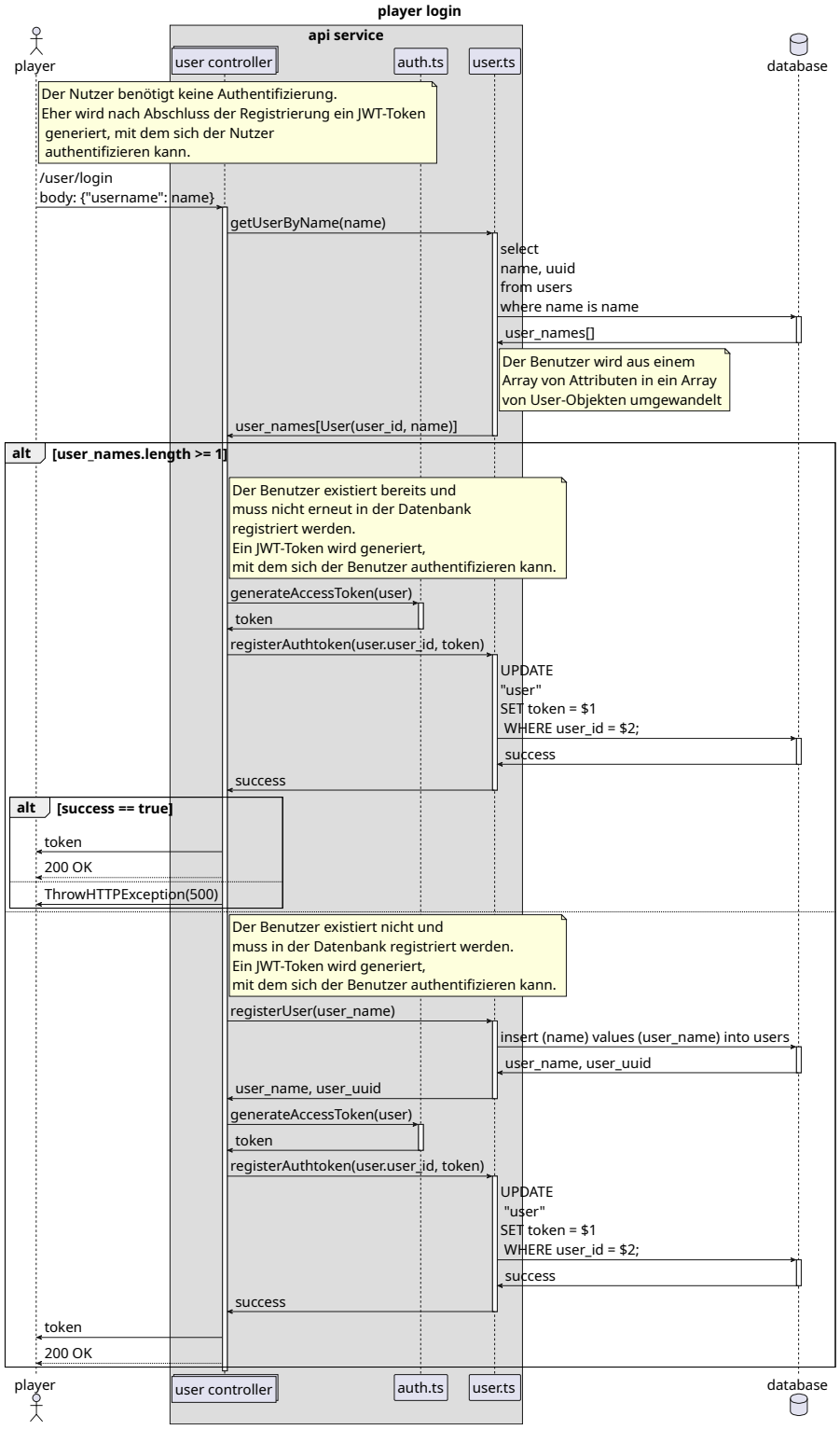

Abbildung 6: Zeigt den Ablauf einer Nutzerregistrierung.

Als nächstes muss es dem Nutzer möglich sein, sich Räume anzeigen zu lassen, diese zu erstellen als auch beizutreten. Die API-Route wurde dabei so implementiert, dass der Nutzer alle Räume abfragen kann, die in der Datenbank vorhanden sind. Dabei wird ein JWT-Token benötigt, um die Anfrage zu autorisieren. Der Ablauf dieses Features wird in (Abbildung 7) dargestellt.

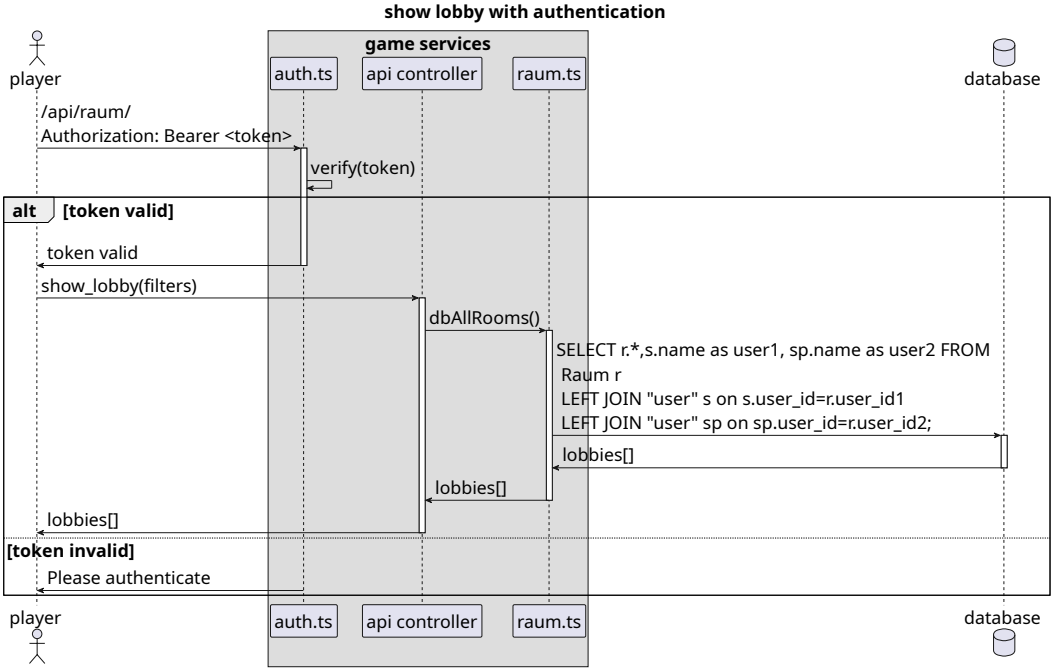

Abbildung 7: Zeigt den Ablauf der Abfrage aller Räume in der Datenbank, welche nach dem Login ausgeführt wird. Hier wir ein JWT-Token benötigt, um die Anfrage zu autorisieren.

Anschließend muss der Nutzer in der Lage sein, einen Raum zu erstellen. Dafür wird eine POST-Anfrage an den Server gesendet. Der Server prüft dabei, ob der Nutzer bereits in einem Raum ist. Ist dies nicht der Fall, wird ein Raum erstellt und der Nutzer hinzugefügt. Nachdem der Nutzer den Raum angelegt hat, kann dieser in den socket.io Raum beitreten. Der Ablauf dieses Features wird in (Abbildung 9) dargestellt. Hierbei sendet der Client über das Event joinRoom eine Anfrage an den Websocket. Dabei sendet dieser über den Body den gewünschten Raum, in den der Nutzer beitreten möchte. Der Server prüft nun zuerst, ob der Raum existiert und ob dieser bereits voll ist. Ist dies der Fall, wird der Nutzer in den Raum hinzugefügt. Ist der erste Nutzer bereits gesetzt und besitzt dieser nicht die gleiche ID wie der anfragende Nutzer, wird der zweite Nutzer mit der ID des anfragenden Nutzers besetzt. Das Setzten des Nutzers erfolgt dabei über die Datenbank.

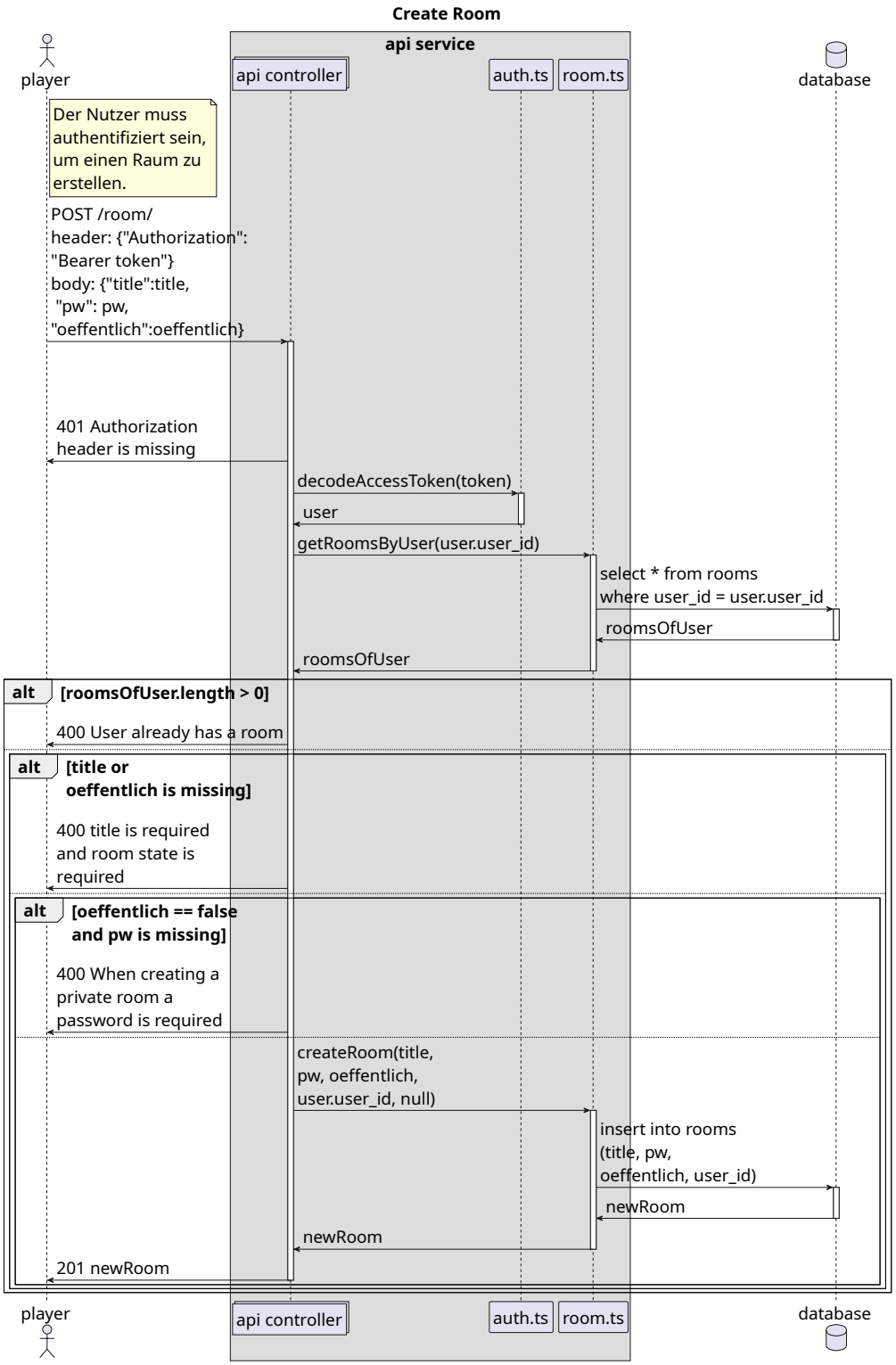

Abbildung 8: Zeigt die Erstellung eines Raumes. Der Nutzer sendet eine Anfrage an den Server, um einen Raum zu erstellen. Der Server prüft, ob der Nutzer bereits in einem Raum ist. Ist dies nicht der Fall, wird ein Raum erstellt und der Nutzer hinzugefügt.

Wenn die Aktualisierung der Räume in der Datenbank vorgenommen wurde, wird der anfragende Nutzer dem socket.io Raum hinzugefügt. Hierbei war die Schwierigkeit den richtigen Raumnamen zu wählen. Wenn man nun eine UUID verwendet, müsste man hier die UUID abfragen. Wenn man den Primary Key der Datenbank als Raumname verwendet, hat man einen eindeutigen Namen, welcher auch schon vorhanden ist. Diesen könnte man auch theoretisch raten, was zu einem potentiellen Sicherheitsproblem führen könnte. Es wurde sich dennoch für die Variante des Primary Key entschieden, da dieser intuitiver zu verwenden ist. Für eine zukünftige produktive Anwendung sollte hier jedoch eine UUID verwendet werden, welche über ein SHA-256 Hash generiert wird.

Nachdem der Nutzer dem Raum innerhalb des Game Servers beigetreten ist, kann dieser das Spiel starten. Dies geschieht über das startPressed Event, hierbei wird geprüft in welchen Raum der anfragende Nutzer ist. Dieser Prozess ist in (Abbildung 10) beschrieben. Wurde der Raum nicht mehr in der Datenbank gefunden, wird eine Fehlermeldung zurückgegeben. Ist dem Spiel kein zweiter Nutzer beigetreten, wird ebenfalls eine Fehlermeldung zurückgegeben. Passen alle Bedingungen, wird ein Room Token generiert und an beide Nutzer innerhalb des socket.io Raums zurückgegeben. Mithilfe dieses Tokens können die Nutzer nun Nachrichten innerhalb des Spiels austauschen.

Hier war die Schwierigkeit, dass wenn einer der Nutzer aus dem Spiel austritt, der Raum gelöscht wird, wodurch der Fehler „No room associated with this user“ auftritt. Dadurch kann das Spiel nicht weitergeführt werden.

Hier könnte eine elegantere Lösung gefunden werden. Möglich wäre eine weitere Websocket Nachricht, welche den Nutzer über das Verlassen des Raums informiert. Der Raum wird dann nicht gelöscht, sondern nur der Nutzer aus dem Raum entfernt. Wenn der Ersteller nun den Raum verlässt, wird der Raum gelöscht und alle Nutzer aus dem Raum entfernt.

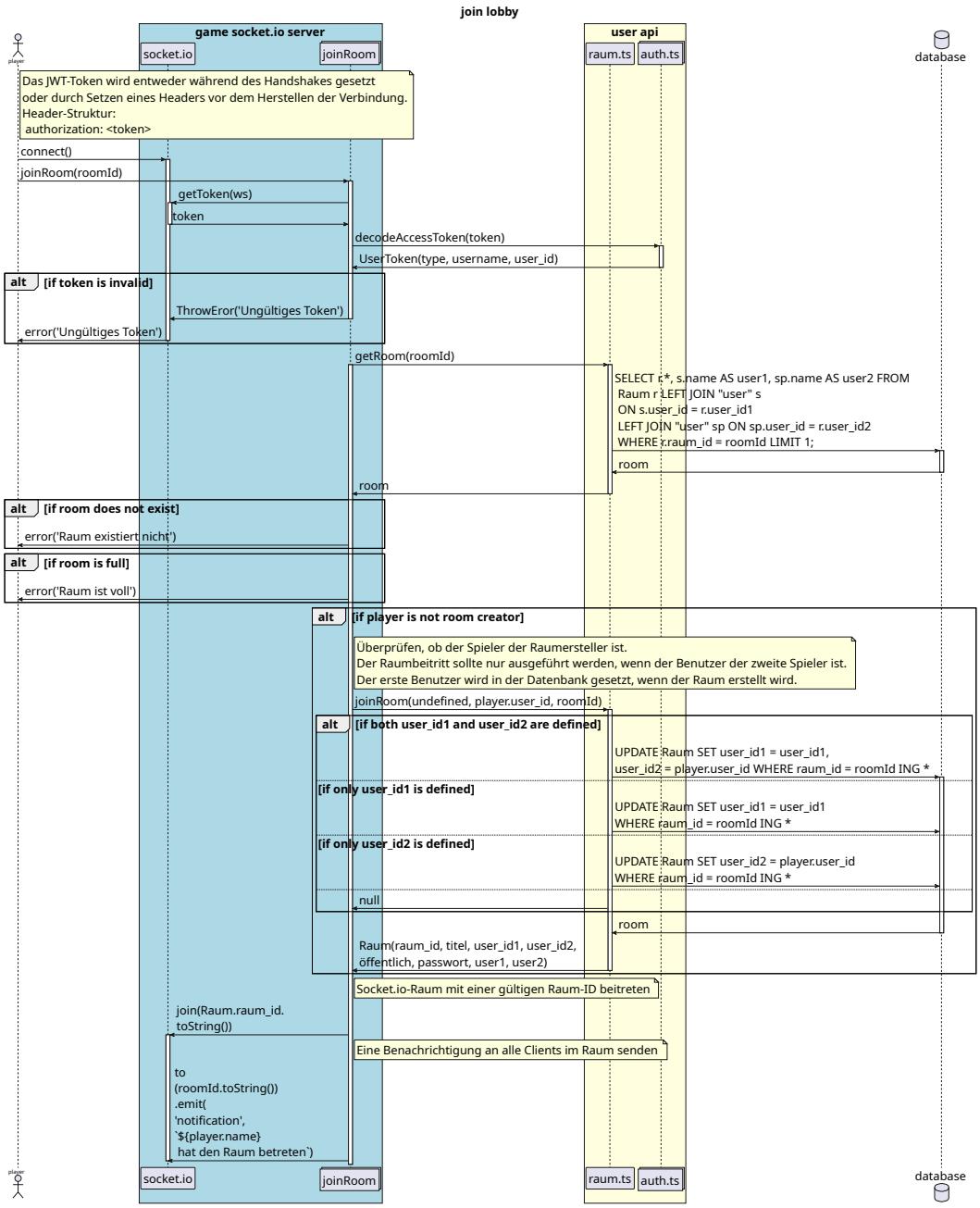

Abbildung 9: Der Nutzer tritt einem Raum bei. Hierbei wird ein JWT-Token benötigt, um die Anfrage zu autorisieren. Der Nutzer verbindet sich mit dem Websocket und sendet eine joinRoom Anfrage. Der Server prüft, ob der Nutzer die nötigen Berechtigungen hat, um den Raum beizutreten. Wenn der Nutzer alle Berechtigungen hat, wird er in den Raum hinzugefügt und empfängt nun Nachrichten von anderen Nutzern.

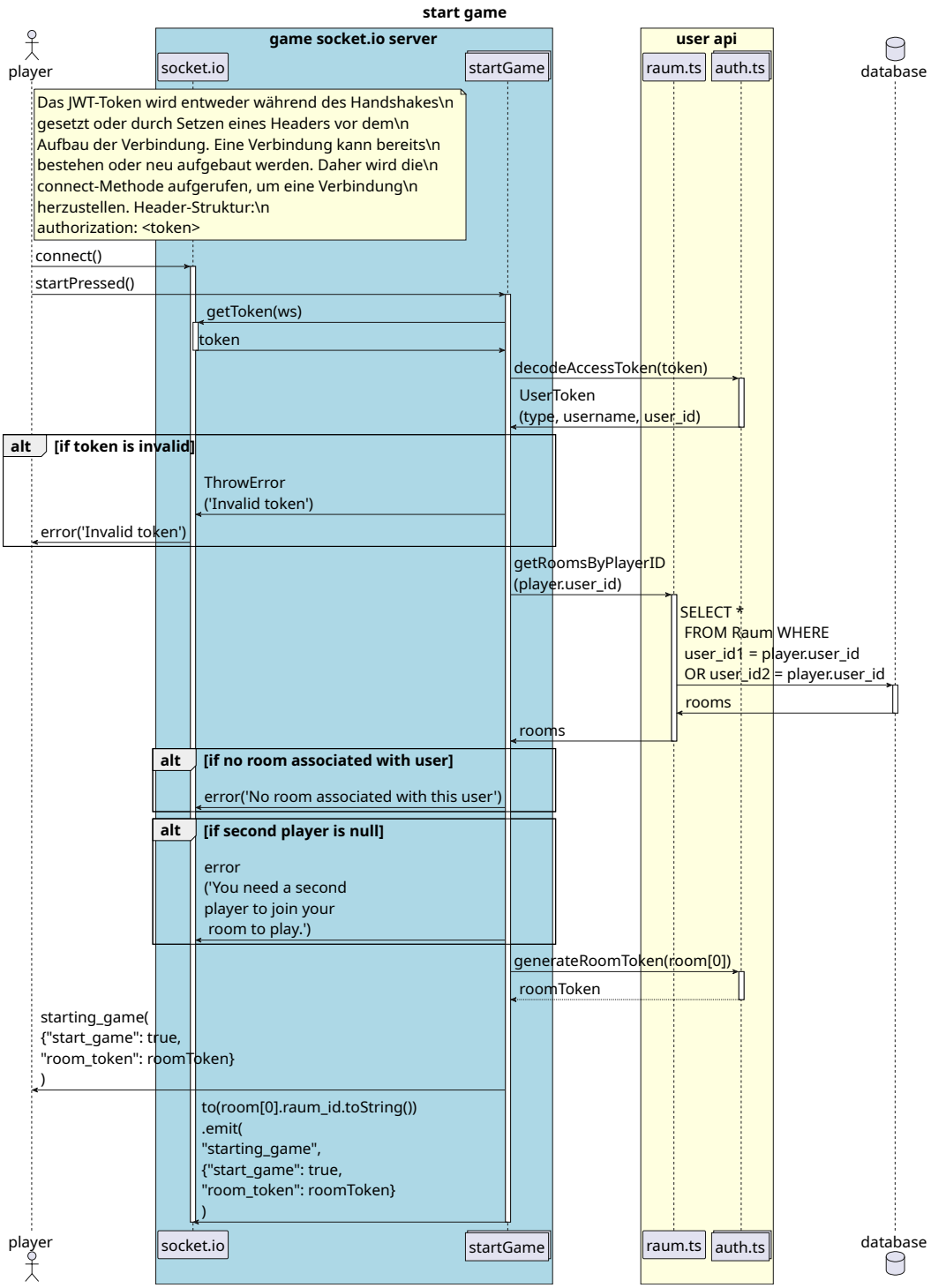

Abbildung 10: Drückt einer der Nutzer den Start-Button in der Benutzeroberfläche wird ein Event an den Server gesendet. Dieser prüft, ob der Nutzer innerhalb eines Raumes registriert wurde. Wenn der Nutzer alle Berechtigungen erfüllt, wird diesem ein Room Token übermittelt, mit dem der Nutzer weitere 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 abgeprallt ist. Genauso passiert das, wenn ein Tor geschossen wird (vgl. Abbildung 11).

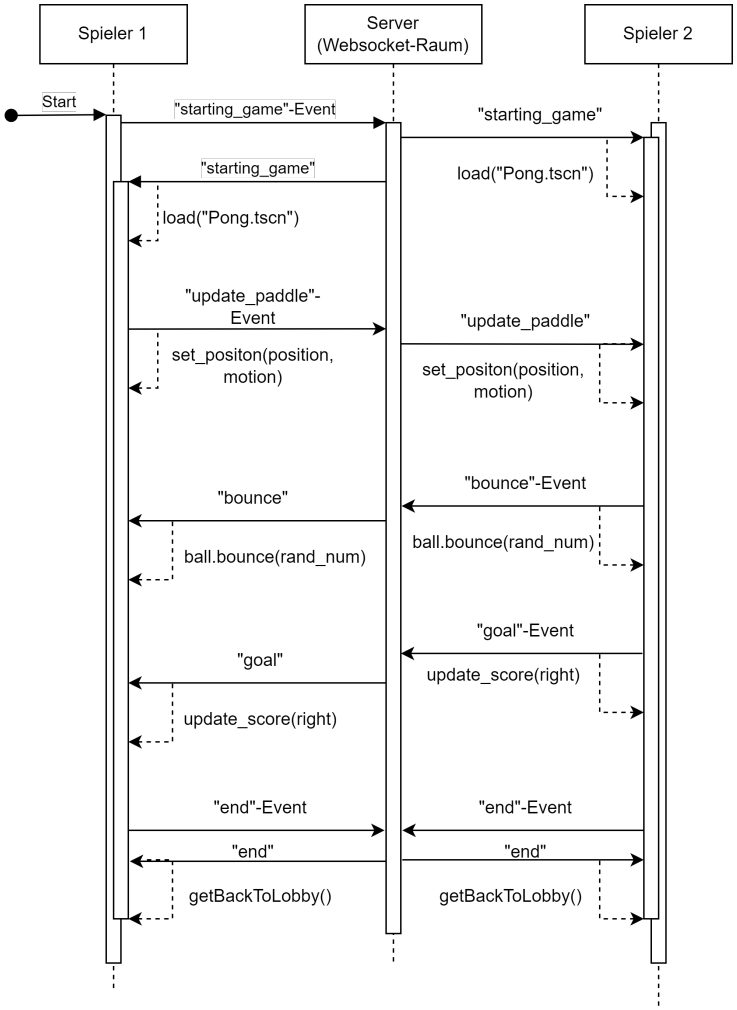

Abbildung 11: Eventbasierter Ablauf des Spiels über einen socket.io Websocket. Es handelt sich um eine vereinfachte Ansicht. Beide Seiten können die jeweiligen Events an den Server senden.

Einer der Schwierigkeiten waren stockende Bewegungen vom Ball während des 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 [7].

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 [8]. 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 [8]

Reflexion

Rückblick & Herausforderungen

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 für das Fixen von Bugs viel Zeit drauf gegangen, 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.

Für eine bessere Modularität hätte man hier die REST API und den Websocket in zwei separate Server aufteilen können. Somit wäre es möglich die Server nach Bedarf zu skalieren, sodass jeder Raum seinen eigenen Websocket Server hätte.

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] SocketIO. *The Engine.IO protocol*. Zuletzt Aufgerufen 19.09.2024. 2020. URL: https://socket.io/docs/v4/engine-io-protocol/.

[5] SocketIO. *Postgres adapter*. Zuletzt Aufgerufen 19.09.2024. 2020. URL: https://socket.io/docs/v4/postgres-adapter/.

[6] 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.

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

[8] 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.