Projektdokumentation Virus Clicker

Dennis Appel & Luca Malisan

Anforderungen

Requirements Client

Funktional:

- **MUSS**
    - Spieler müssen private Sessions erstellen können, der andere Spieler beitreten können
    - Spielergruppen müssen sich für die Spieldauer vor Beginn einer Runde entscheiden können
    - Spieler müssen per Klick auf einen Button Viren generieren können
    - Die Spieler müssen Informationen über den aktuellen Spielstand durch ein Leaderboard sowie Angaben zur eigenen Virenproduktion erlangen können
    - Spieler müssen Debuffs erwerben und temporär anderen Spielern schaden können.
    - Spieler müssen Buffs erwerben und temporär ihre Virenproduktion verstärken können.
    - Spieler müssen PowerUps erwerben und damit ihre Viren-Produktion dauerhaft verstärken können.
    - Spieler müssen permanente Waffen erwerben und damit ihren Gegnern schaden können.
    - Spieler müssen sich gegenseitig mit einer Chat-Funktion austauschen können.

- **SOLL**
    - Das Spiel soll durchsetzen, dass Spieler über den Spielverlauf hinweg immer wieder gewisse Mindestwerte der Produktionsrate vorweisen müssen, um nicht eliminiert zu werden
    - Die Bewertungsmethode soll eingestellt werden können, beispielsweise wer die meisten Viren gesammelt hat oder wer die höchste Produktionsrate aufweist.
    - Das Spiel soll einen FPS-Counter anzeigen

Nicht funktional:

- **MUSS**
    - Accessibility: Das GUI muss so gestaltet sein, dass das Spiel auch mit visuellen Einschränkungen problemlos gespielt werden kann
    - Der Code muss an den zentralen Stellen dokumentiert sein, Kommentare müssen den Best Practices folgen. Da das Spiel Open-Source ist, muss sichergestellt sein, dass der Code für andere Entwickler leicht verständlich ist.
    - Der Client muss mit dem Server lose gekoppelt sein, Anwendungslogik muss im Client auf ein Minimum beschränkt sein.
- **SOLL**
    - Flüssige Darstellung: Das GUI soll sich auf 95% der Client-Geräte mit min. 60 FPS aktualisieren

Requirements Server

Funktional:

- **MUSS**
    - Clients müssen in Echtzeit miteinander synchronisiert werden, sodass Spieler gegeneinander antreten können
    - Der Server muss alle nötigen REST-API Schnittstellen und die Verarbeitungslogik für sämtliche Client-Funktionen bereitstellen
    - Der Server muss eine Registrierungsfunktion für neue Spieler bzw. eine Login-Funktion für bestehende Spieler bereitstellen.
    - Der Server muss mehrere unabhängige Spielsessions verwalten. Insbesondere muss der Server sicherstellen, dass ein Spieler nicht auf mehrere Spielsessions gleichzeitig zugreifen kann.

Nicht funktional:

- **MUSS**
    - Die Synchronisation von bis zu 100 Clients muss innerhalb von einer halben Sekunde stattfinden.
    - Sicherstellung von Partitionstoleranz: Der Ausfall eines Clients darf die anderen Clients nicht negativ beeinflussen
    - Sicherheit: Der Server muss die rechtlichen Bedingungen für Datenschutz umsetzen. Zudem muss der Server konsequent sicherstellen, dass nur authentifizierte User auf die Serverfunktionalität zugreifen können
    - Die Datenintegrität muss jederzeit sichergestellt sein, um ein reibunglsoses Spielerlebnis zu garantieren und die Nachvollziehbarkeit getätigter Aktionen zu erhalten.
    - Der Code muss vollständig dokumentiert sein, Kommentare müssen den Best Practices folgen. Da das Spiel Open-Source ist, muss sichergestellt sein, dass der Code für andere Entwickler leicht verständlich ist.
    - Verfügbarkeit: 99.9% der Zeit muss der Server erreichbar sein
- **SOLL**
    - Möglichkeiten des Cheatings sollen so weit wie möglich verhindert werden

Journal

Projektwoche	Änderung Client	Änderung Server	Allgemeines
10.02-25 - 16.02.25	-	- Github Repository aufgesetzt - Postgres-Datenbank mit Docker aufgesetzt - Registrierungs-Funktion per REST-API umgesetzt	
17.02.25 – 23.02.25	-	- Datenbanktabellen definiert - Model-Klassen erstellt	- Spielregeln definiert - Anforderungen definiert - Netzwerk-protokoll entworfen - Mockups erstellt
24.02.25 - 02.03.25	- Github-Repository aufgesetzt - Websocket-Connection mit Server hergestellt - Chat-Feature implementiert - Register-/Login umgesetzt - Übeprüfung, ob User authentifiziert ist, umgesetzt	- Chat-Funktion mit Websockets umgesetzt - Registrierungs-/ Login-Prozess auf Websockets migriert - Refresh-Token implementiert	
03.03.25 – 09.03.25	- Game-Loading Screen umgesetzt - Session Creation und Joining Screen umgesetzt	- Speicherung von Chat-Nachrichten in der Datenbank	
10.03.25 – 16.03.25	- Implementiert, dass joined Spieler auf dem Game-Loading Screen angezeigt werden - Re-Styling des Frontends - Session-Timer umgesetzt	- Funktionalität für Session-Erstellung und Beitreten umgesetzt - Websocket-Route umgesetzt, die Informationen zur Session gibt - Start von Game Sessions und Session-Timer umgesetzt	
17.03.25 – 23.03.25	- Start und Beendigung der Game Session implementiert - Button-Click-Feature umgesetzt - Leaderboard umgesetzt	- Fehlerbehandlung verbessert, allgemeines Refactoring - Funktionalität für Button-Click implementiert - Leaderboard implementiert	- Netzwerk-protokoll überarbeitet
24.03.25 – 30.03.25	- Frontend-Architektur für Effekte aufgebaut	- Architektur für die einfache Implementierung neuer Effekte aufgebaut	

		- Verwaltung der Effekte auf der Datenbank - Auto-Click-Effekt umgesetzt	
31.03.25 – 06.04.25	-	- Berechnung der Effizienz und Dauer der Effekte umgesetzt - Implementiert, dass Session Timer nach Reload noch mit Server synchronisiert ist	
07.04.25 – 13.04.25	- Frontend-Style überarbeitet - Cooldown der Effekte umgesetzt	- Critical-Hit Effekt umgesetzt - Pub-Sub-Architektur für Effekte implementiert - Erworbene Effekte werden auf der Datenbank gespeichert - Implementiert, dass Effekte einen gewissen Cooldown haben nach Aktivierung	
14.04.25-20.04.25	- Effect Log implementiert	- Reverse Engineered Effekt umgesetzt - Effect-Log implementiert, das aktive Effekte dokumentiert	
21.04.25 – 27.04.25	- Frontend-Style überarbeitet	- Beendigung von Game Sessions implementiert	Effekte definiert und konzipiert
28.04.25 – 04.05.25	-	-	
05.05.25 – 11.05.25	- Chat in Game-Screen integriert	- Implementiert, dass Server erkennt, welche Spieler on-/offline sind - Popupinator-Effekt implementiert - Implementiert, dass Chat nach Reload wieder alle Nachrichten lädt - Fehlerbehebungen	
12.05.25 – 18.05.25	- Navigation verbessert - Listener implementiert, um offline Spieler zu erkennen - Berechtigungen für Routen durchgesetzt - Frontend-Caching des Session Keys	- Implementiert, dass offline Spieler nicht berücksichtigt werden - Fehlerbehebungen	
19.05.25 – 25.05.25	- Berechtigungen für Routen durchgesetzt - End-Leaderboard umgesetzt	- Deployment des Servers als Docker-Container umgesetzt - Code kommentiert - Fehlerbehebungen	

	- Deployment des Clients als Docker-Container umgesetzt - Code kommentiert - Fehlerbehebungen		
26.05.25 – 01.06.25	- Fehlerbehebungen - Auswahl der Evaluation-Method umgesetzt	- Verschiedene Evaluation-Methods implementiert - Fehlerbehebungen	
02.06.25 – 08.06.25	- README.md geschrieben - Deployment auf Google Cloud	- README.md geschrieben - Deployment auf Google Cloud	

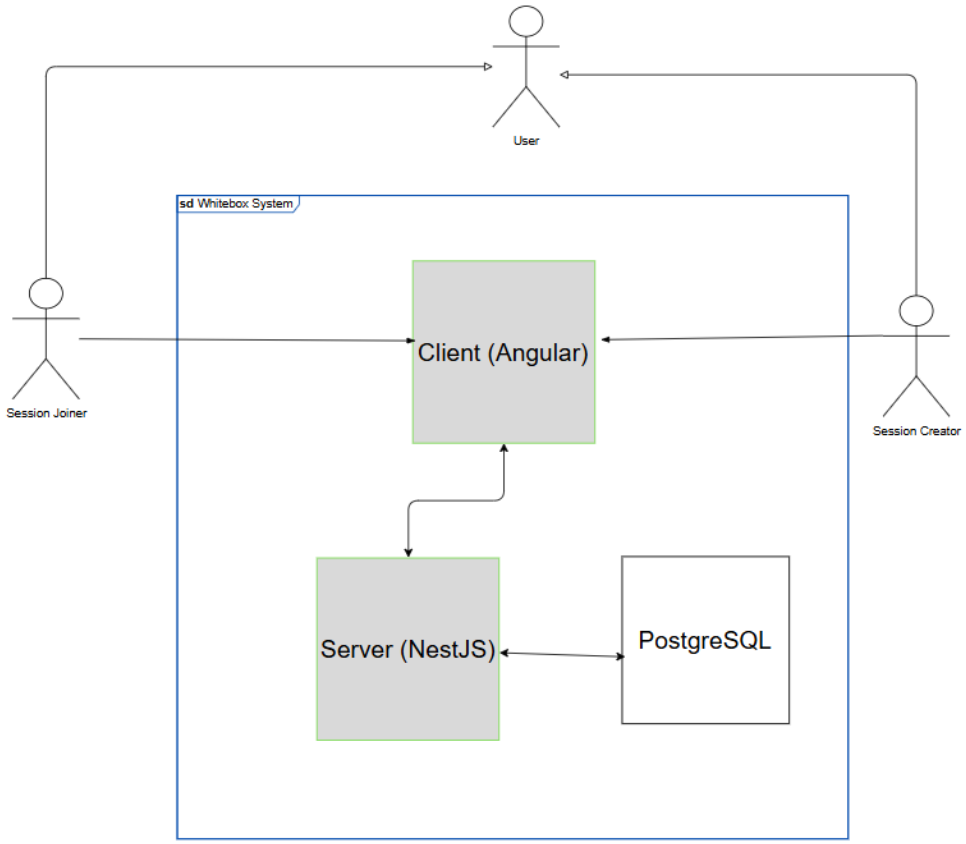
Dokumentation der Applikation

Bausteinsicht

Whitebox Gesamtsystem

Das System besteht aus drei eigenständigen Modulen: dem Client, Server und er PostgreSQL-Datenbank. Der Server ist an die Datenbank angebunden und kommuniziert mit dem Client.

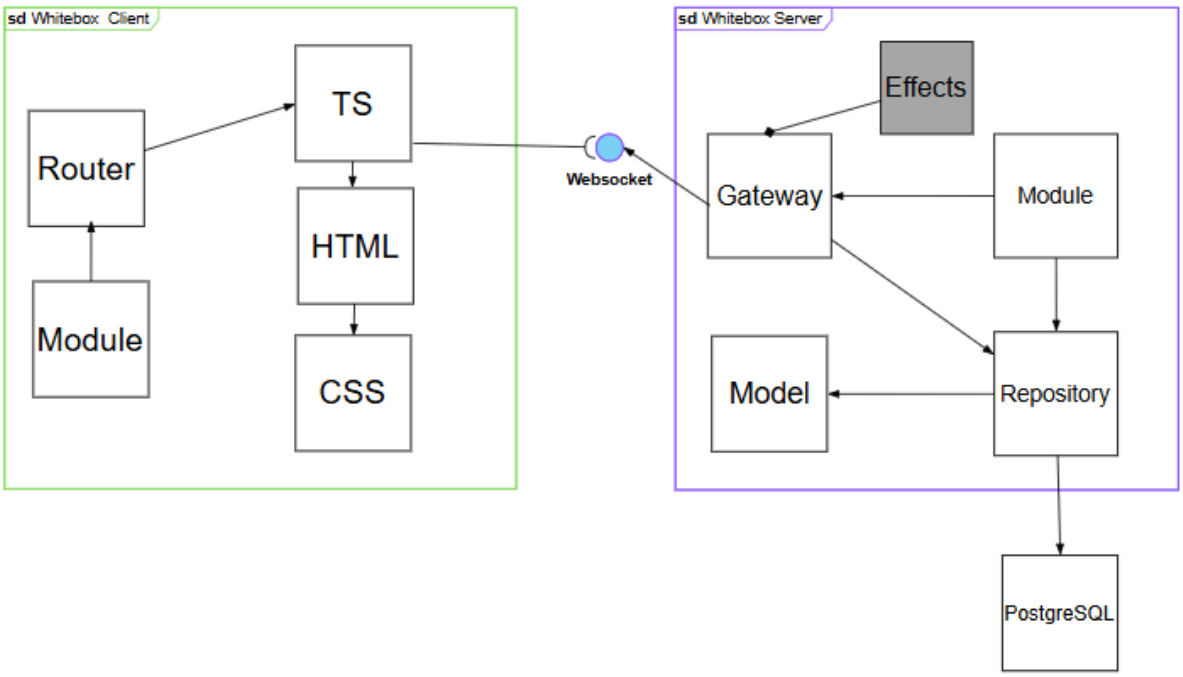
Bei den Rollen wird unterschieden zwischen dem Session Creator, der neue Game-Sessions erstellt und dem Session Joiner, der den erstellten Sessions beitritt.



Whitebox Client / Server

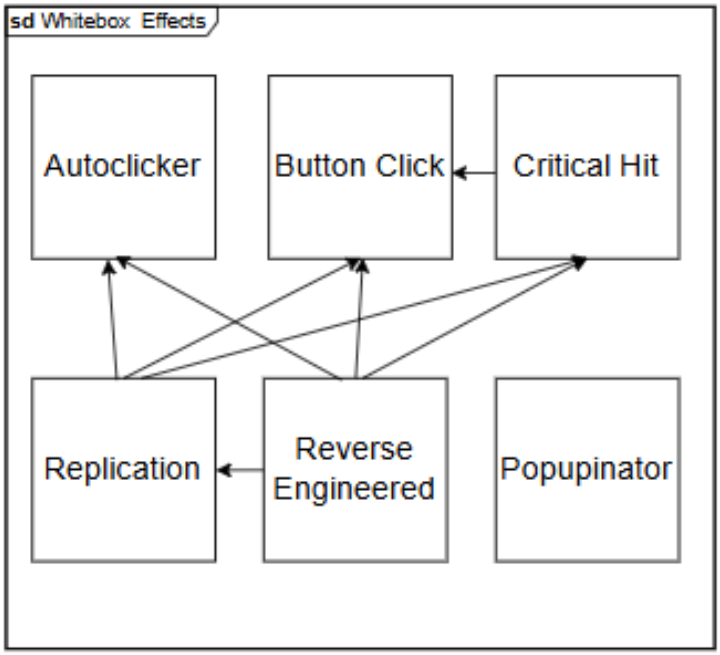
Der Aufbau des Clients folgt einer für Angular typischen Struktur. Jede Frontend-Komponente besteht aus HTML, das mit einer CSS-Datei gestaltet und von einem Typescript-Controller dynamisiert wird. Komponenten bilden einen Teil der Webpage ab, die Root-Komponenten setzen sie zu einer vollständigen Seite zusammen. Die URLs dieser Root-Komponenten werden mit einem Router definiert, der ins Hauptmodul eingebunden wird.

Der Server kommuniziert sowohl mit der Datenbank als auch mit dem Client. Für ersteres werden hierbei die Model-Klassen und Repositories verwendet, um die Datenbanktabellen abzubilden und darauf Queries auszuführen. Für letzteres sind die Gateways zuständig, die bestimmte Websocket-Schnittstellen zur Verfügung stellen, die vom Client aufgerufen werden können.
Pro Komponente wird ein Modul definiert, das alle Gateways und Repositories einbindet.



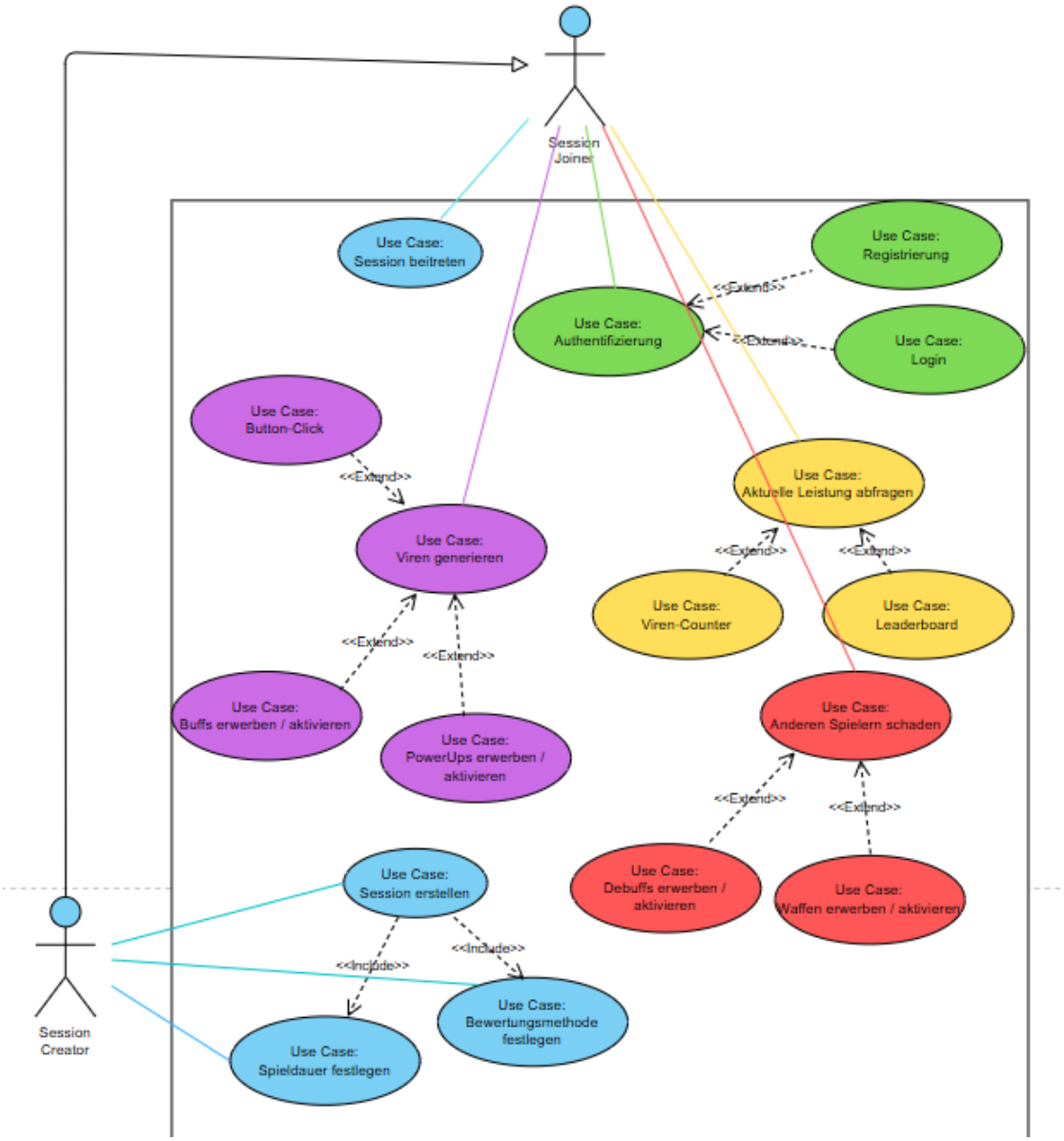
Whitebox Effekte

Ein Teil der Gateways bilden jene der Effekte. Diese bilden teilweise Kettenreaktionen mit anderen Effekten, weshalb intern eine Pub-Sub-Architektur verwendet wird. In der Abbildung ist dargestellt, zwischen welchen Subscribe-Abhängigkeiten bestehen.



Use-Case-Diagramm

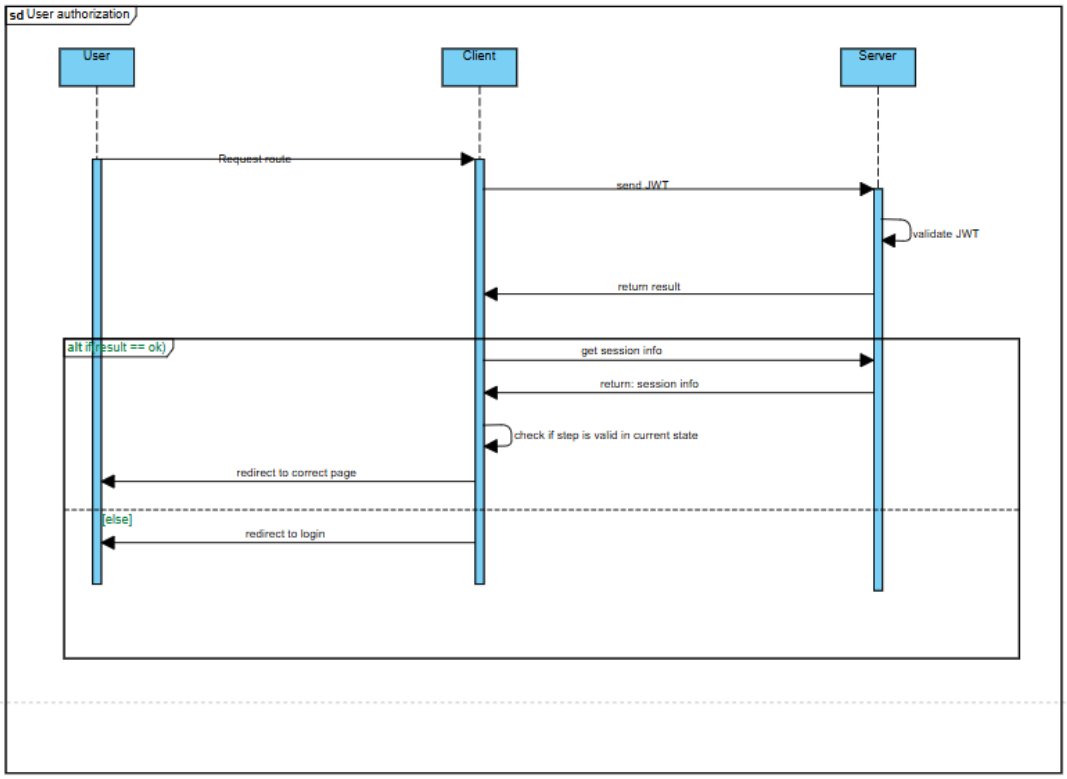
Die Rollen «Session Creator» und «Session Joiner» unterscheiden sich nur darin, wer die Session erstellt und damit die Rundenparameter festlegen kann. Da aber auch der Session Creator der erstellten Session beitritt, stehen die beiden Rollen in einer contains-Beziehung.



Laufzeitsicht

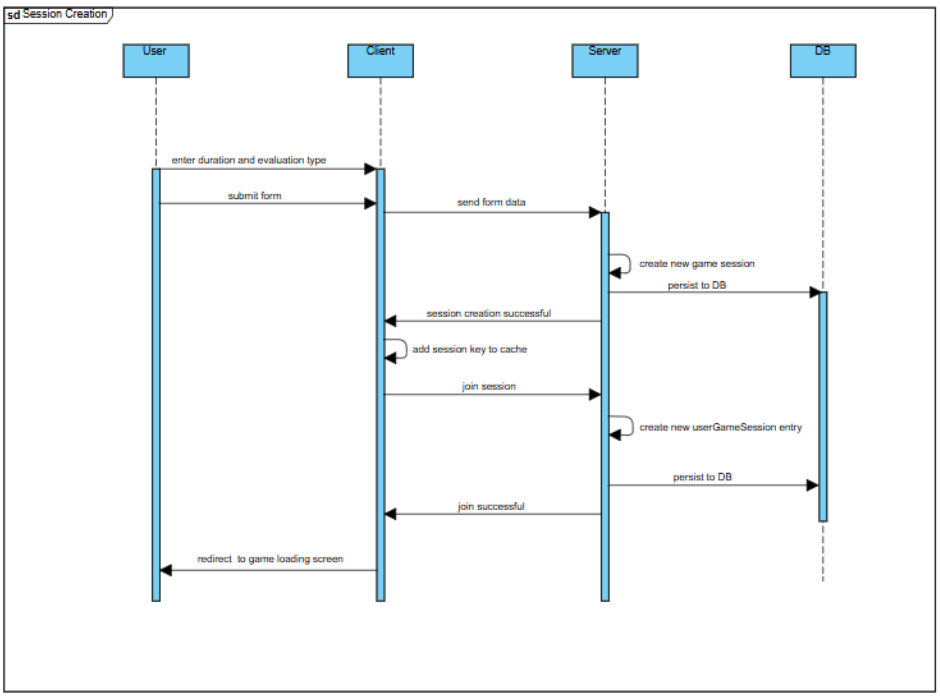
Authentifizierung von Usern

Bei jedem Zugriff auf eine Client-Seite wird zuerst das JWT aus dem Cache validiert. Wenn der User nicht authentifiziert ist, wird auf die Login-Seite redirected. War die Authentifizierung dagegen erfolgreich, wird eine Autorisierung durchgeführt. Der aktuelle Stand der Session wird abgefragt, um zu beurteilen, ob die aktuelle Seite aufgerufen werden darf. Beispielsweise darf das End-Leaderboard nicht aufgerufen werden, solange die Session noch läuft.



Session-Erstellung

Bei der Erstellung einer Session füllt der User ein Formular mit Dauer und Evaluationsmethode aus und schickt es ab. Aus diesen Daten wird im Server eine neue Session erstellt und dem Client eine Erfolgsmeldung zurückgegeben. Dieser veranlasst daraufhin, dass der User der erstellten Session beitritt, was im Server durch einen neuen userGameSession-Eintrag umgesetzt wird. Der Client redirected anschliessend auf den Loading-Screen.

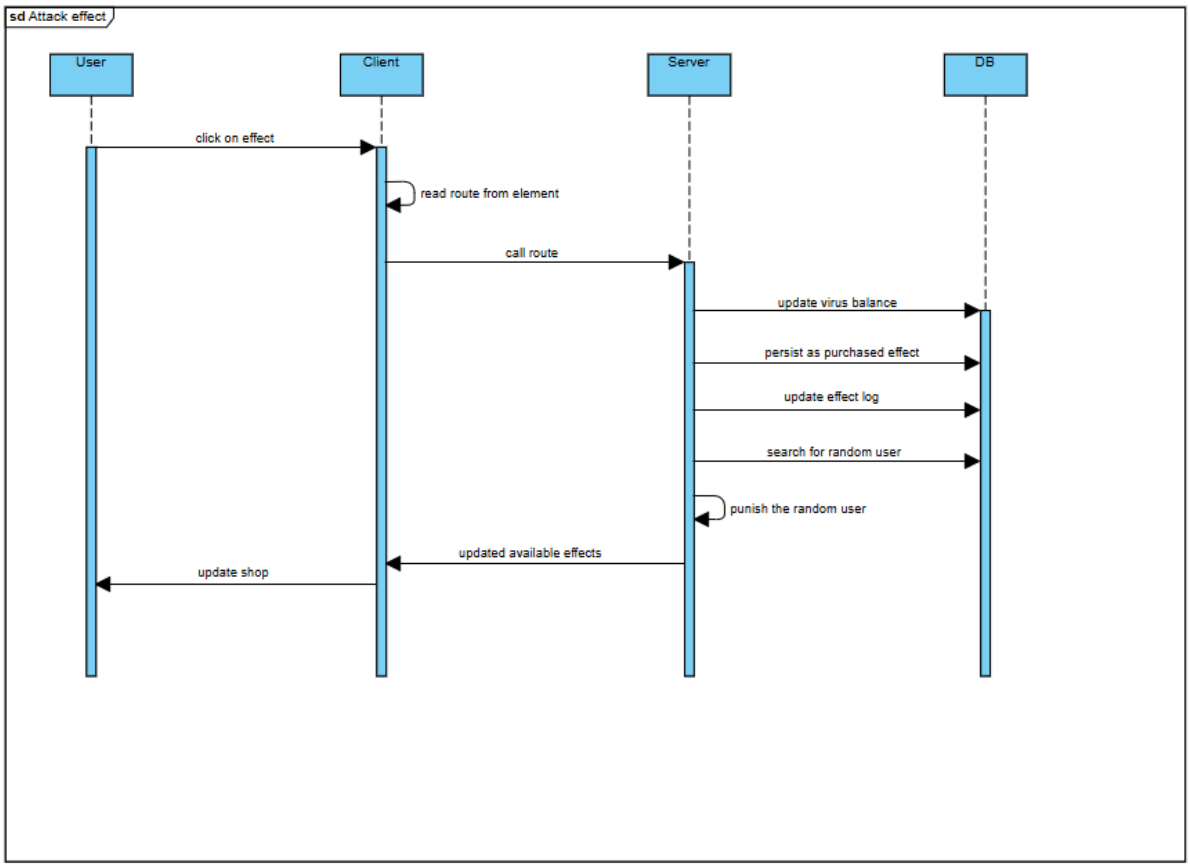


Angriffseffekte

Die grundlegende Logik gilt für alle Effekte. Der User aktiviert einen Effekt durch Klick auf das entsprechende Element. Dieses hat die entsprechende Websocket Route hinterlegt, die der Client aufrufen kann. Der Server zieht anschliessend den Kaufpreis von der Balance ab, markiert den Effekt als gekauft und macht einen Eintrag im effect log.

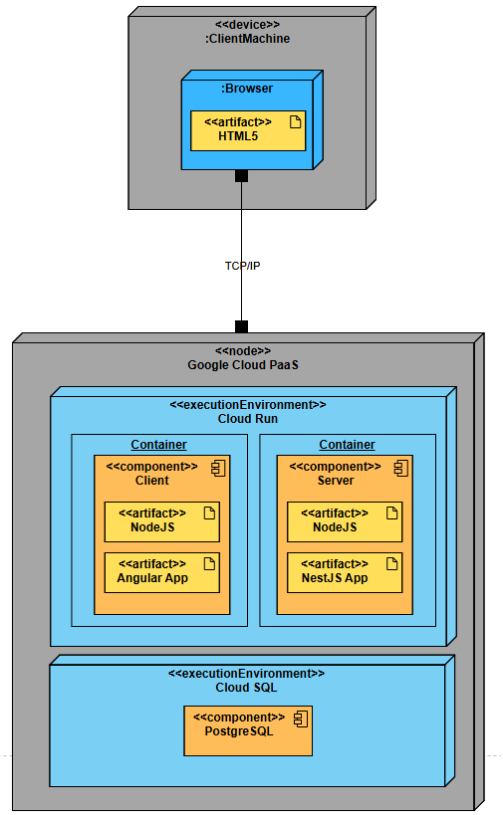
Speziell bei Angriffseffekten ist, dass nun zusätzlich nach einem random User gesucht wird, der dann entsprechend bestraft wird. Je nach Implementierung erfolgt die Bestrafung nicht direkt, sondern in einem Intervall oder Timeout.

Schliesslich werden die verfügbaren Effekte im Client aktualisiert.

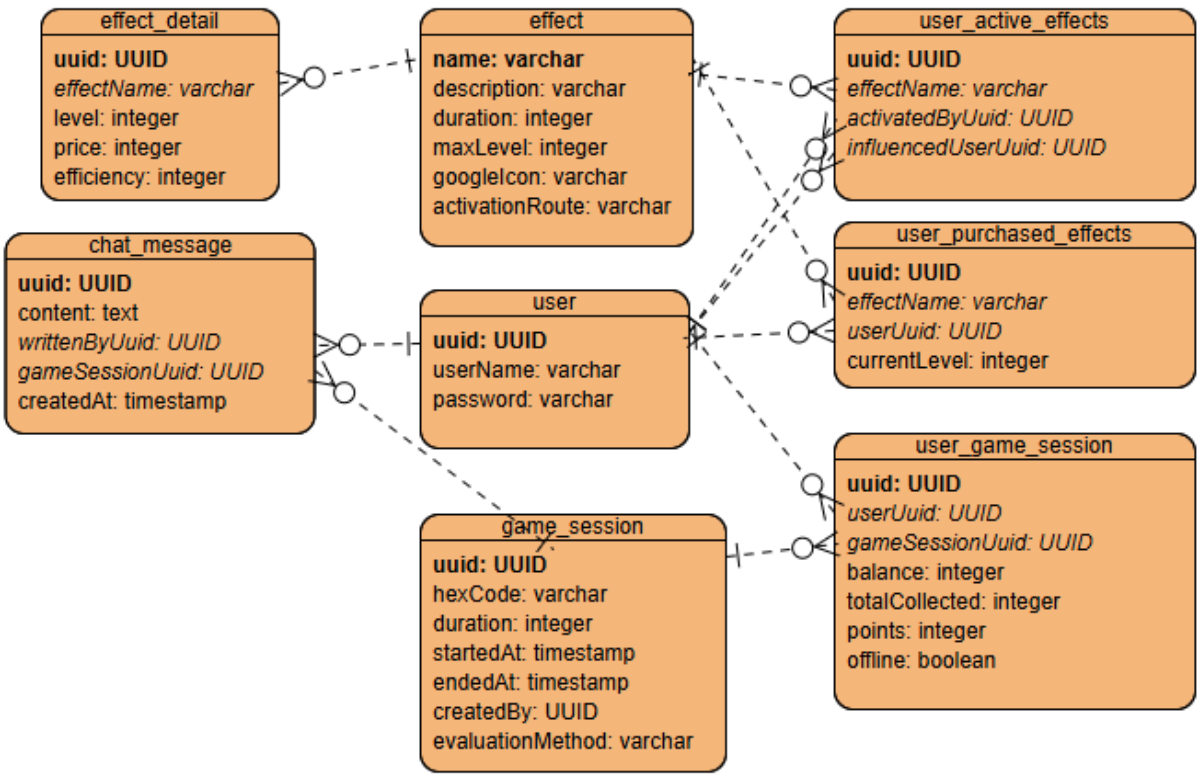


Verteilungssicht

Die gesamte Applikation läuft auf Google Cloud. Für die Datenbank wird hierbei Cloud SQL verwendet, Client- und Server werden als Docker Container in Cloud Run ausgeführt.



ERD-Diagramm



Querschnittliche Konzepte

Verwendung von Interfaces

Im Projekt wurden durchgängig Interfaces für die versendeten JSON verwendet, um sicherzustellen, dass diese von Client und Server korrekt gelesen werden können.

User Experience

Das Design der gesamten Software wurde einheitlich gestaltet und folgt klaren Regeln. Dieses soll die Benutzerfreundlichkeit erhöhen, aber auch thematisch zum Spielkonzept passen. Neue Client-Seiten müssen an das bestehende Style-Konzept angepasst werden. Wo möglich wurden zudem eindeutige, verständliche Icons eingesetzt. Dazu wurde auf eine bestehende Lösung (Google Material Symbols) zurückgegriffen.

Für den User soll zudem die darunterliegende Architektur transparent sein. Das Spiel soll trotz separatem Client und Server wie ein einziger Monolith wirken.

Sicherheit

Zur Authentifizierung von Usern kommen JWT sowie Referesh-Token zum Einsatz. Der Vorteil liegt darin, dass eine hohe Sicherheit gewährleistet werden kann, ohne dass sich der User jedes Mal neu einloggen muss.

Bei jedem versuchten Zugriff auf eine Domain wird das JWT-Token und der aktuelle Stand der letzten Game-Session überprüft. Wenn ein User unzureichende Berechtigungen besitzt, wird er auf die Login Seite weitergeleitet. Diese Überprüfung findet zentral im Client statt, um das Risiko für Sicherheitslücken zu minimieren.

Deployment, Betrieb und Migration

Client und Server können separat deployed und physisch verteilt betrieben werden. Diese Technologieunabhängigkeit ermöglicht eine unkomplizierte Migration. Zudem ermöglicht dies die Verwendung horizontaler Skalierung, um flexibel auf Belastungsspitzen zu reagieren.

Um maximale Flexibilität zu garantieren, wird zwischen den Containern ein eigenes Netzwerk etabliert. So können diese unter Verwendung von DNS über deren Namen angesprochen werden und erlauben eine flexible Zuweisung von Ports.