Nautical Nonsense: Because Sometimes You Just Need to Sink Something

Jakob Götz
j.goetz2@oth-aw.de

Uwe Kölbel
u.koelbel@oth-aw.de

Maximilian Schlosser
m.schlosser@oth-aw.de

Oliver Schmidts
o.schmidts@oth-aw.de

Jan Schuster
j.schuster@oth-aw.de

Philipp Seufert
p.seufert@oth-aw.de

Fabian Wagner
f.wagner@oth-aw.de

***Abstract*—Dieser Technical Report beschreibt die Architektur von dem Cloud Native Browser-Game *Nautical Nonsense*. Insbesondere gibt der Report Information über die Planung, Implementierung und welche Werkzeuge für die Umsetzung benötigt wurden.**

I. Einleitung

Schiffeversenken ist seit je her ein beliebtes Spiel. Hauptsächlich wird dieses Spiel jedoch als Brettspiel in Präsenz gespielt. Als 2019 die Corona-Pandemie startete, mussten zwangsweise die Kontakte reduziert werden. Dies ruf eine massive Veränderung in der Lebensweise der Menschen hervor. Besonders im Bereich der Unterhaltung entwickelten sich in kurzer Zeit viele Lösungen, um Aktivitäten, welche zuvor in Präsenz waren, in den Online-Raum zu migrieren. Inspiriert von dieser Entwicklung entstand die Idee, ein Cloud-Nativ Schiffeversenken zu erstellen.
In den weiteren Abschnitten dieses Dokuments wird auf die technischen Details des Vorhabens eingegangen. Abschnitt II enthält dabei die Vorgehensweise, welche sicherstellt, dass die Projektziele erreicht werden. Es folgt Abschnitt III, welches einen Überblick über die einzelnen Bausteine gibt und Abschnitt IV, worin die Entwicklungswerkzeuge erläutert werden. Abschließend wird in Abschnitt V auf das Fazit und den Ausblick eingegangen.

II. Vorgehensweise

Nautical Nonsense wurde als Cloud-Nativ Browser-Game Anwendung konzipiert, weshalb sich bei der Vorgehensweise einige Punkte ergaben, welche die grundlegende Struktur des Projektes und die Umsetzung beeinflussten..

Container Das Frontend und das Backend ist jeweils in einen extra Container. Bei der Datenbank wurde sich gegen einen separaten Container entschieden, da die verwendete Datenbank bereits eine Option zur Skalierung bereitstellt und eine Ausfallsicherheit garantiert.

keine Persistenz im Backend Die Lösungsstrategie im Backend war es eine Kommunikation zu ermöglichen ohne größere Datenmengen darin zu speichern. Da *Nautical Nonsense* als Cloud-Native Anwendung konzipiert wurde, durfte die Persitenz der Daten nicht im Backend gehalten werden. Dies wurde gelöst, indem die Verarbeitung der Daten im Backend erfolgt, jedoch direkt im Anschluss eine Speicherung in der Datenbank vorgenommen wird.

Kodierung des Spielzustandes Der Spielzustand soll als eine Liste von Zahlen kodiert werden. Dies ermöglicht eine einfache Speicherung und Übertragung des Spielfeldes. Zusätzlich werden die Schiffe der Spieler als Liste von Listen bereitgestellt, welche jeweils die Koordinaten des jeweiligen Schiffes enthalten. Ist ein Schiff an einer Stelle getroffen, wird auf diese 100 aufaddiert.

TABLE I
Kodierung des Spielfeldes

Zustand	Beschreibung
0	Wasser
1	Wasser getroffen
2	Schiff
3	Schiff getroffen
4	Schiff versenkt

Synchronisation im Frontend Wenn zwei menschliche Spieler gegeneinander spielen, ist es wichtig, den aktuellen Spielstand zwischen beiden Spielern synchron zu halten. Sobald ein Spieler eine Aktion ausführt und diese an das Backend sendet, muss dieses den zweiten Spieler über die Änderung informieren. Deshalb wird für die Kommunikation im Spiel zwischen Front- und Backend eine Websocket Verbindung genutzt.

Herausforderungen in der Datenbank ein Text welcher die Vorgehensweise, Herausforderungen und Lösungswege beschreibt

Provisionierung der Infrastruktur Um die Anwendung Cloud-Nativ zu gestalten, wurde auf die Verwendung eines Kubernetes-Clusters zurückgegriffen, welcher auf der Cloud-Plattform *Amazon Web Services* als Managed Service bereitgestellt wird. Die Infrastruktur wurde dabei mit *Terraform* als Code definiert. Dies ermöglicht eine einfache und schnelle Bereitstellung und bei hoher Auslastung die Möglichkeit der Skalierung.

III. Bausteinsicht

A. Datenbank

Zur dauerhaften Speicherung aller Daten, die während des Spiels anfallen, wird die NoSQL-Datenbank *MongoDB* **mongodb** verwendet. Um den Implementierungsaufwand zu reduzieren, wird dabei auf das Cloudangebot *MongoDB Atlas* **mongodb-atlas** zurückgegriffen. In der Datenbank werden JSON-artige Dokumente in einzelnen Collections abgelegt, die für je eine bestimmte Funktion zuständig sind. Tabelle II zeigt eine Übersicht über die vorhandenen Collections.

TABLE II
COLLECTIONS

collection	Beschreibung
`games`	Aktuelle Spielzustände
`leaderboard`	Bestenliste
`stats`	Spielstatistiken

In `games` werden die aktuellen Spielstände jedes Spiels gespeichert. Hierzu gehören neben einer eindeutigen Game-ID Informationen zu den Spielern, deren momentane Spielfelder sowie getroffene und versenkte Schiffe.

Ist ein Spiel zu Ende, wird der Gewinner zusammen mit seiner benötigten Zuganzahl in `leaderboard` gespeichert. Außerdem werden Daten zu Kapitulation und Gegner (Mensch oder Computer) ergänzt, sodass im Frontend zwei getrennte Bestenlisten angezeigt werden können.

Die Spielstatistiken werden ebenso nach jedem beendeten Spiel aktualisiert. Ein Dokument in `stats` enthält dabei Daten zu bereits abgeschlossenen Spielen, wie Spielanzahl und durchschnittliche Schusszahl - getrennt nach Mensch- und Computergegner. Zusätzlich werden häufige Schiffs- und Schusspositionen sowie die Abschussquote der einzelnen Schiffstypen protokolliert.

B. Backend

Das Backend ist mit Python und dem Framework FastAPI **fastapi** umgesetzt. Das Frontend kommuniziert mit dem Backend über zwei Schnittstellen. Zum einen über die RESTful-API, zum anderen über eine Websocket Verbindung. Die RESTful-API übernimmt alle Anfragen des Frontends, welche nicht die Kommunikation zwischen zwei Spieler betrifft. Somit dient die RESTful-API hauptsächlich als Knotenpunkt zur Datenbank. Die Websocket Schnittstelle dient zur Kommunikation zwischen zwei Frontends. Die Kommunikationsdaten der Websocket-Verbindung werden im Backend aufbereitet, validiert und in der Datenbank gespeichert. Darauf folgt eine sofortige Antwort an beide Frontends, um diese über den neuen Spielschritt zu informieren.

Folgende HTTP-Endpunkte in Tabelle III, stellt die API im Backend für das Frontend zur Verfügung:

Die `/`-Route gibt eine Universally Unique Identifier (UUID) als String zurück, welche das Frontend nutzen kann, um den Spieler gegenüber des Backends bei nachfolgenden Anfragen eindeutig zu bestimmen. Ein Spiel wird mit der

TABLE III
RESTFUL-API

Endpunkt	Beschreibung
`GET /`	Startseite
`POST /play`	Starten eines neuen Spiels
`WEBSOCKET /ws/{client_id}`	Kommunikation der Spieler
`GET /leaderboard`	Daten für das Leaderboard
`GET /stats`	Daten für Spielstatistiken

`/play`-Route gestartet. Diese Route nimmt ein JSON-Objekt entgegen, in welchen spezifiziert wird, welchen Spielmodus der Spieler spielen möchte. Als Rückgabe bekommt das jeweilige Frontend eine Ready-Flag und die UUID der beiden Kontrahenten. Nachdem ein Spiel bereit ist, wird die Websocket Verbindung über die `/ws/`-Route aufgebaut. Über diese Route läuft die komplette Kommunikation bis zum Spielende.

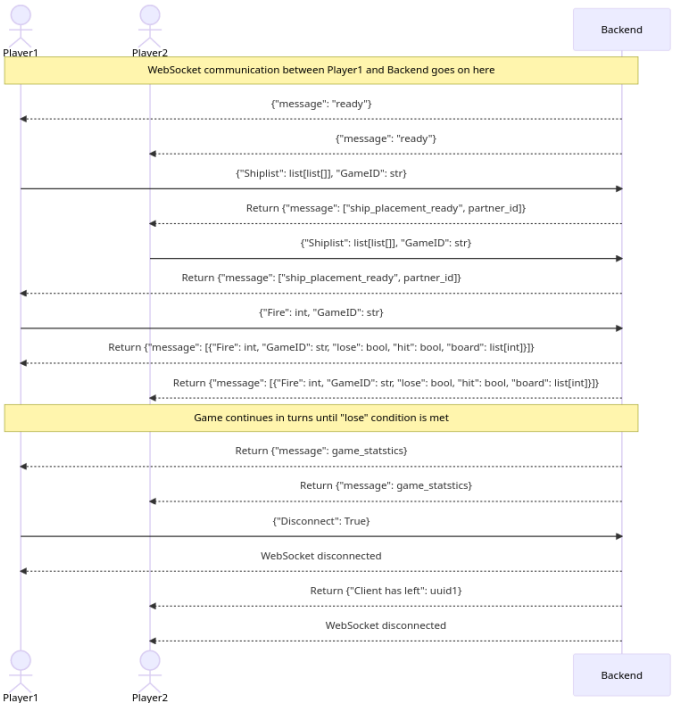

Fig. 1. Websocket Kommunikation

In Grafik 1 wird die Kommunikation zwischen zwei Spielern und dem Backend während des Spielverlaufs ersichtlich. Im ersten Schritt wird bei erfolgreichen Verbindungsaufbau der beiden Spieler ein `ready`-Flag vom Backend gesendet. Daraufhin können die Clients ihre Schiffe platzieren und die Position dem Backend übermitteln. Erst wenn beide Spieler ihre Schiffe platziert haben, wird das `ship_placement_ready`-Flag vom Backend auf True gesetzt und der Spielablauf beginnt. Hier übermitteln die Spieler jeweils eine Reihe an Parametern, damit das Backend den Spielfluss steuern kann. Das Spiel endet, wenn ein Spieler alle Schiffe des gegenüber versenkt hat. Darauf folgt, dass der Rückgabewert für lose gesetzt wird. Es folgt lediglich vom

Backend die Spielstatistik über das gespielte Spiel. Danach wird die Websocket Verbindung von den Spielern aufgelöst.

C. Frontend

Zu Beginn des Spieles befindet sich der Spieler auf der Startseite, die es ihm ermöglicht, einen Spielernamen und die Art des Spiels auszuwählen. Zusätzlich kann der Nutzer auf die Optionen-Seite wechseln. Diese dient als "Hub" für weitere Szenen. Über die Optionen sind das Leaderboard, die Statistiken, die Spielanleitung und die Credits zu erreichen.

Sind alle nötigen Eingaben getroffen kann mit der Partie begonnen werden.

Die nächste Szene erlaubt es dem Spieler, die Schiffe auf dem Spielfeld zu platzieren. Dazu kann er jedes Schiff einzeln durch anklicken oder alle auf einmal über den Random-Button platzieren. Die einzelnen Fahrzeuge können nach dem Anwählen durch einen Klick rotiert und verschoben werden. Der Reset-Knopf setzt die Schiffe an ihre ursprüngliche Position zurück. Sind alle Schiffe platziert, können die Positionen mit Confirm bestätigt werden.

In der eigentlichen Spielszene wird das Spielfeld mit den zuvor platzierten Schiffen und eine Wahlmöglichkeit zum Platzieren von Schüssen auf die gegnerische Flotte angezeigt. Die beiden Spieler sind dabei abwechselnd am Zug. Die Treffer und Fehlschüsse werden entsprechend auf dem eigenen Spielfeld und dem Feld des Gegners angezeigt. Über den Capitulate-Button kann die Partie durch Aufgabe beendet werden.

Ist ein Spieldurchlauf beendet, entweder durch Kapitulation eines Spielers oder durch Versenken aller Schiffe eines Teilnehmers, wird in eine entsprechende Gewinner- oder Verlierer Szene gewechselt.

1) Kommunikation mit Backend: Beim Aufrufen der Webseite sendet der Client einen HTTP-GET-Request an die Startseite und erhält ein JSON-Objekt mit der Client-ID. Diese ID wird der globalen Variable *sharedData.clientID* zugewiesen.

Die Websocket-Verbindungs-URL wird durch das Hinzufügen der Client-ID zu *sharedData.websocket_url* erstellt.

Wenn der Client in der Start-Szene „Spiel gegen Random“ auswählt, erfolgt ein HTTP-POST-Request an die `/play`-Route mit Informationen wie der Client-ID, dem ausgewählten Modus und dem Spielernamen. Die empfangene Game-ID wird in *sharedData.game_id* gespeichert.

Die Websocket-Verbindung wird erst nach erfolgreicher Ausführung dieses Befehls initialisiert, bevor zur Szene „Waiting1“ gewechselt wird. Dieser Websocket existiert ebenfalls als globale Variable namens *sharedData.socket*.

In allen Szenen ist die Methode *onmessage* implementiert, die es ermöglicht, über Websockets eingehende Nachrichten vom Backend zu empfangen. Diese Methode ist Bestandteil des JavaScript-Objekts *WebSocket*.

Wenn sich ein zweiter Client auf der Webseite verbindet und die Option „Spiele gegen Random“ auswählt, sendet das Backend über die Websocket-Verbindung ein „ready“-Flag an beide Clients. In Folge dessen wird die Variable *sharedData.ready* auf den Wert „true“ gesetzt und es erfolgt ein nahtloser Wechsel zur Szene „Shipplacement“.

In dieser Szene wird die Platzierung der Schiffe (Positionen) und die Game-ID an das Backend über die Websocket-Verbindung übermittelt. Für das Senden wird zuerst überprüft, ob die Websocket-Verbindung steht und anschließend die Methode *JSON.stringify* verwendet, um ein JSON-Objekt in einen JSON-String zu konvertieren. Dieser wird anschließend mittels *sharedData.socket.send(JSON-String)* an das Backend gesendet.

Die grundlegende Vorgehensweise beim Empfangen und Senden von Nachrichten via Websockets ändert sich in den restlichen Szenen nicht. Es wurde lediglich die Verarbeitung der Daten auf den jeweiligen Use-Case angepasst. In Abbildung 1 ist ein Schaubild der Websocket Kommunikation ersichtlich.

D. Infrastruktur

Für die Bereitstellung der Anwendung in der Cloud wird ein Kubernetes-Cluster **k8s** verwendet. Um eine nachvollziehbare und deklarative Konfiguration zu gewährleisten, wurde Terraform **terraform** als Infrastructure-as-Code-Tool verwendet. Alle Bestandteile des Setups können somit über eine CI/CD-Pipeline, welche per Weboberfläche in Gitlab ausgelöst wird, direkt deployt werden. Als Cloud-Anbieter wurde Amazon Web Services (AWS) **aws** gewählt. Die einzelnen Komponenten werden im Folgenden näher erläutert und sind in Abbildung 2 dargestellt. Der erste Schritt beinhaltet die Erstellung einer Virtual Private Cloud (VPC) mit privaten und öffentlichen Subnetzen in verschiedenen Availability Zones (AZs). Anschließend werden die privaten Subnetze für die Provisionierung des Kubernetes-Clusters verwendet, während in eines der öffentlichen Subnetze ein NAT-Gateway für den Internetzugriff der Pods erstellt wird. Zusätlich werden die benötigten IAM-Rollen und Policies für den Zugriff auf die AWS-Ressourcen erstellt. Zuletzt erfolgt die Erstellung eines AWS Load Balancer Controllers als Ingress Controller für den Zugriff auf die Anwendung von außen. Nach dem Aufsetzen der Infrastruktur wird über die CI/CD-Pipeline ein Test-Deployment durchgeführt, um die Erreichbarkeit des Clusters zu überprüfen und den erfolgreichen Ablauf aller Operationen zu gewährleisten. Des Weiteren wird das Kubernetes Dashboard installiert, um die einzelnen Komponenten des Clusters zu überwachen. Ebenfalls ist es möglich, über eine zweite Pipeline ein manuelles Deployment der Anwendung durchzuführen. Hierfür werden Docker-Container verwendet, die erstellt werden, wenn ein Tag auf dem default-Branch erstellt wird.

IV. Entwicklungswerkzeuge

A. Backend

Das Backend wird hauptsächlich mit FastAPI implementiert, einem Framework, das auf Starlette **starlette** und Pydantic **pydantic** aufbaut. Starlette stellt Funktionen wie

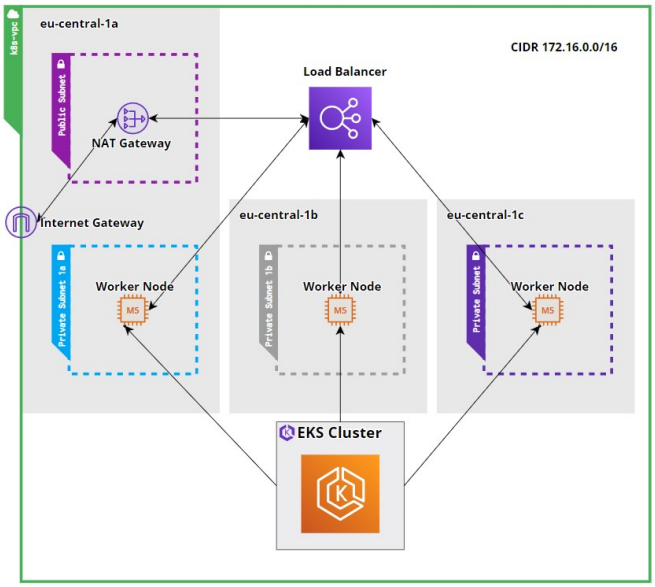

Fig. 2. Überblick Kubernetes-Cluster

WebSockets zur Verfügung, während Pydantic die Validierung von Datenmodellen ermöglicht, die von FastAPI verwendet werden. Dank Starlette kann auch das Test-Framework Pytest **pytest** direkt für Unit-Tests eingesetzt werden. Der Asynchronous Server Gateway Interface (ASGI) Server Uvicorn **uvicorn** wird für die Bereitstellung genutzt. Für den Zugriff auf die Mongo-Datenbank wird das Framework PyMongo **pymongo** genutzt.

B. Frontend

Um das Frontend zu realisieren, wird Phaser3 **phaser** verwendet. Dabei handelt es sich um ein Framework zur Realisierung von Browser-Spielen auf Desktop und mobilen Geräten. Phaser bietet dazu mehrere Funktionen vom Erzeugen und Importieren verschiedener Canvas-Elemente und Assets bis hin zu einfacher Physiksimulation und Szenenverwaltung.

Als Entwicklungsumgebung wird der Phaser Editor 2D eingesetzt. Dieser erlaubt das Erzeugen einer Projektgrundstruktur, die im Verlauf der Arbeit immer weiter angepasst und ausgebaut wird.

Für die Tests im Frontend werden eine Vielzahl an Werkzeugen genutzt. Die Verwaltung dieser erfolgt über npm **npm**. Phaser3 benötigt zur problemfreien Funktion ein Document-Object-Modell (DOM). Das Frontend-Test Framework Cypress **cypress** bietet mit seinem End-to-End Modus ein solches DOM und erlaubt damit das automatisierte testen des Frontends. Um die Testabdeckung zu messen kommt istanbul **istanbul** zum Einsatz.

V. Fazit und Ausblick