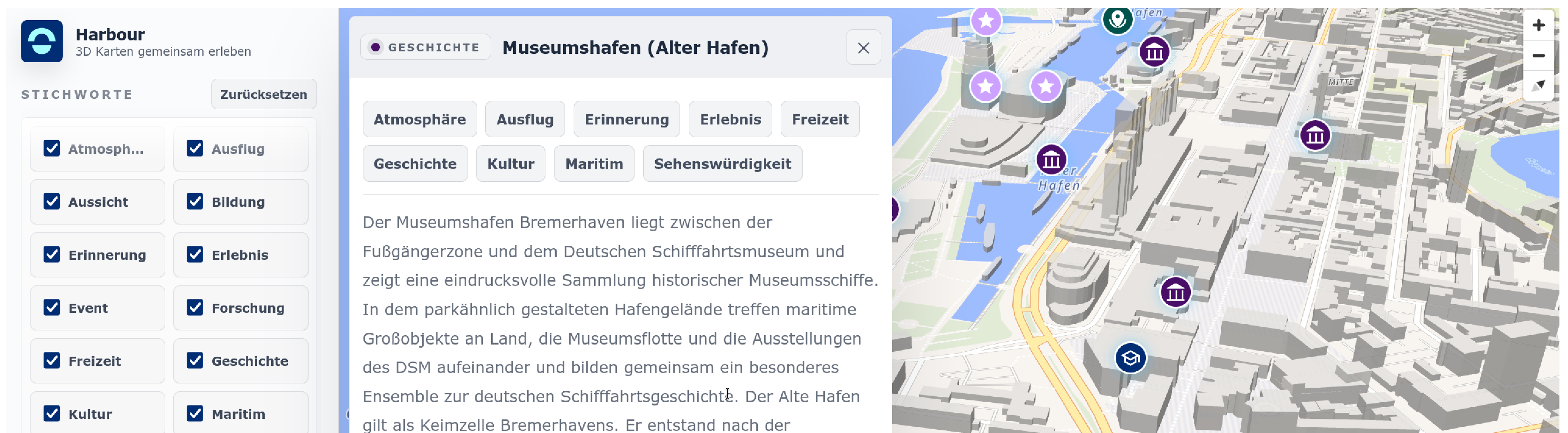
Harbour2025

Hochschule Bremerhaven

Bremerhaven interaktiv entdecken

Jonas Behling · Abass Camara · Kajiman Chongbang · Zidane Kenfack Fouape · Domenic Graca · Johannes Heidtmann · Lesly Matchio Kuete · Vanelle Happi · Henrik Mittelhäußer · Levi Range · Philipp Schur · Lennart Steffen · Oliver Radfelder



Einleitung

Harbour2025 ist eine interaktive 3D-Kartenanwendung aus dem Masterstudiengang Vertrauenswürdige Systeme, die es ermöglicht, Bremerhaven gemeinsam und synchron im Web zu erkunden. Interaktionen wie das Verschieben, Zoomen und Fokussieren der Kartenansicht werden in Echtzeit zwischen mehreren Teilnehmenden geteilt, sodass alle dieselbe Perspektive auf die Stadt erleben. Ergänzend bieten *Points of Interest* (POIs) anklickbare Orte mit zusätzlichen Informationen, wodurch Harbour2025 als Plattform für digitale Stadtführungen, Storytelling und weitere Anwendungsszenarien dient.

Motivation

Das Projekt *Harbour2025* entstand im Rahmen des Masterprojekts im Studiengang Vertrauenswürdige Systeme und knüpft an eine inhaltliche Vorgabe des betreuenden Professors an: Bremerhaven, insbesondere der Hafen, soll in einer interaktiven Webanwendung digital erlebbar gemacht werden. Dabei sollten moderne Technologien wie 3D und Augmented Reality (AR) berücksichtigt und in ein webbasiertes Nutzungskonzept integriert werden. Im Projektverlauf zeigte sich jedoch, dass AR-Funktionalitäten und anspruchsvolle 3D-Modelle im Web aktuell noch nicht auf allen gängigen Smartphones zuverlässig und einheitlich unterstützt werden. Aus diesem Grund wurde der ursprüngliche Fokus angepasst und zu einer praktikablen Lösung weiterentwickelt: einer kolaborativen, interaktiven 3D-Kartenanwendung, die Bremerhaven direkt im Browser zugänglich macht und eine gemeinsame Erkundung in Echtzeit ermöglicht.

Diese Entscheidung wurde auch durch die Rahmenbedingungen an der Hochschule gestützt: Kartentechnologien bieten bereits ein breites Fundament an vorhandenem Know-how und erlauben es, auf etablierte Werkzeuge und Methoden aufzubauen. Ergänzend schafft die Nutzung von OpenData eine realitätsnahe und skalierbare Grundlage, um die Stadt digital abzubilden und eigene Inhalte wie POIs sinnvoll zu integrieren. Gleichzeitig vereint Harbour2025 zahlreiche relevante Themenfelder unter einem Dach – von Frontend-Entwicklung und Backend-Architektur über WebSocket-basierte Echtzeitkommunikation und Datenbankanbindung bis hin zu 3D-Visualisierung und möglichen AR-Erweiterungen. Dadurch eignet sich das Projekt sowohl als technischer Demonstrator als auch als Plattform, die über das Modul hinaus weiterentwickelt werden kann.

Zielsetzung & Konzept

Harbour2025 verfolgt das Ziel, eine webbasierte, mobile-first 3D-Kartenplattform zu entwickeln, die eine gemeinsame und synchronisierte Erkundung von Orten ermöglicht. Im Mittelpunkt steht dabei nicht die individuelle Navigation, sondern eine geteilte Kartenperspektive, in der mehrere Personen gleichzeitig dieselbe Ansicht erleben und gemeinsam Inhalte entdecken können. Die Anwendung ist grundsätzlich nicht auf eine einzelne Stadt begrenzt, sondern als übertragbares Konzept für unterschiedliche Regionen und Szenarien gedacht. Bremerhaven dient im Projekt als konkretes Fallbeispiel zur Umsetzung und Evaluation.

Funktionale Anforderungen

Ein zentrales Ziel ist die Echtzeit-Synchronisation der Kartenansicht zwischen mehreren Nutzer:innen. Interaktionen wie Verschieben, Zoomen oder das Ausrichten der 3D-Ansicht sollen so übertragen werden, dass alle Teilnehmenden denselben Kartenausschnitt sehen und dadurch eine gemeinsame Orientierung entsteht. Ergänzend werden POIs als inhaltliche Ankerpunkte in die Karte integriert, die beim Anklicken zusätzliche Informationen über ein Overlay bereitstellen. Die Erstellung und Pflege dieser POIs ist zunächst nicht für Endnutzer:innen vorgesehen, sondern richtet sich an Betreiber wie Museen, Städte oder Veranstalter. Diese sollen Inhalte unkompliziert anlegen und verwalten können, ohne dafür tiefes technisches Wissen zu benötigen. Vorgesehen ist hierfür ein einfacher Workflow über CSV-basierte Datenimporte, wodurch POIs effizient ergänzt, aktualisiert und erweitert werden können.

Nicht-funktionale Anforderungen

- unterstützt alle gängigen Browser und Smartphones
- intuitive Benutzungsoberfläche
- skalierbar und hoch performant
- leicht zu erweitern
- einfach zu warten und pflegen (wenig Abhängigkeiten zu anderen Pakten/Frameworks/Bibliotheken)
- basiert vollständig auf Open-Source-Software.
- robust gegenüber typischen Angriffsszenarien

Abb. 1: Systemarchitektur

Systemarchitektur

Die Anwendung ist als containerisierte, serviceorientierte Architektur konzipiert und wird mithilfe von *docker compose* orchestriert. Ziel dieser Architektur ist eine klare Trennung der Verantwortlichkeiten einzelner Systemkomponenten, eine reproduzierbare Entwicklungs- und Produktivumgebung, sowie die Möglichkeit zur einfachen Erweiterung und Skalierung einzelner Dienste.

Container-Orchestrierung

Die Orchestrierung der einzelnen Dienste erfolgt mithilfe von *docker compose* auf Basis mehrerer Compose-Dateien, die unterschiedliche Einsatzszenarien abbilden. Durch diese Aufteilung kann dieselbe Systemarchitektur sowohl für die lokale Entwicklung als auch für produktionsnahe Umgebungen genutzt werden, ohne Konfigurationsduplikate oder manuelle Anpassungen vornehmen zu müssen.

Die Datei *compose.yaml* bildet die gemeinsame Basis der Systemarchitektur. Diese definiert alle zentralen Services, sowie grundlegende Abhängigkeiten zwischen den Containern. Diese Datei ist unabhängig von der jeweiligen Laufzeitumgebung einsetzbar.

Entwicklungsumgebung
Für die Entwicklungsumgebung wird die Basis-Konfiguration durch die Datei *compose.override.yaml* ergänzt. Diese enthält spezifische Anpassungen für die lokale Entwicklung, wie das Einbinden von Quellcode-Verzeichnissen als Volumes, sowie die Aktivierung von Hot-Reload-Mechanismen. Dadurch können Änderungen am Code unmittelbar in den laufenden Containern wirksam werden, was einen schnellen Entwicklungsprozess ermöglicht.

Zur Sicherstellung einer konsistenten Codequalität ist der Entwicklungsprozess durch den Einsatz von *Git-Hooks* ergänzt. Diese werden lokal vor jedem Commit ausgeführt und überprüfen den Code automatisch auf Formatierungsfehler. Durch diese frühe Qualitätssicherung werden Fehlerquellen reduziert und ein einheitlicher Code-Stil über das gesamte Projekt hinweg gewährleistet.

Testumgebung
Damit jedes Teammitglied die Webanwendung auch auf einer internetseitig erreichbaren Maschine testen konnte, wurde eine gemeinsame virtuelle Maschine eingerichtet, auf der jedem Teammitglied ein eigener Benutzeraccount zur Verfügung steht. Dies ermöglichte realitätsnahe Tests unter produktionsnahen Bedingungen, ohne lokale Entwicklungsumgebungen vorauszusetzen.

Für einen sicheren und wartungsarmen Betrieb wird *Docker* auf dieser Maschine im Rootless-Modus [1] eingesetzt. Dadurch können alle Teammitglieder Container eigenständig starten und verwalten, ohne Root-Rechte auf der virtuellen Maschine zu benötigen. Gleichzeitig wird verhindert, dass Nutzer potenziell auf sensible Systembereiche oder die Home-Verzeichnisse anderer Benutzer zugreifen können, was den parallelen Mehrbenutzerbetrieb deutlich absichert.

Produktivumgebung
Der produktive Betrieb wird über die Datei *compose.prod.yaml* abgebildet. In dieser Konfiguration werden ausschließlich die für den Betrieb notwendigen Abhängigkeiten berücksichtigt. Dies führt zu einer schlankeren und stabileren Laufzeitumgebung, die besser für den Einsatz in produktionsnahen Szenarien geeignet ist.

Frontend

Das User Interface für Harbour2025 ist die zentrale Schnittstelle zwischen dem System und Nutzenden. Während bei der Entwicklung technische Aspekte im Vordergrund standen, ist ein professionelles Erscheinungsbild ein Nebenbestandteil der Gesamtanwendung. So wurde das Farbkonzept sowie die Typografie an die Brand Design Guidelines der Hochschule Bremerhaven angelehnt.

Das zentrale Gestaltungselement bildet die Karte mit 3D Gebäuden sowie eine Seitenleiste. Auf dieser Karte werden die vom Backend bereitgestellten POIs dargestellt, die bei Interaktion entsprechende Details in einem Popup dargestellen. Über eine Auswahl an Stichworten können die dargestellten Marker thematisch gefiltert werden.
Das Kernfeature der Anwendung, das Synchronisieren der Kartenansicht, wird über Räume ermöglicht. Über ein Popup-Menü kann ein neuer Raum erstellt oder einem bestehendem beigetreten werden. Sobald der/die Nutzer:in in einem Raum ist, werden in der Seitenleiste Informationen wie der Raumname, Status, eigener Anzeigename und eine Liste der Teilnehmenden eingeblendet. Über einen Klick kann die eigene Sicht mit der eines anderen Teilnehmenden gekoppelt werden, wodurch die Kamera und die Interaktionen mit Markern synchronisiert werden.

Technisch umgesetzt wurde die Karte mit MapLibre GL JS, diese Kartenengine ermöglicht die Einbindung von 3D Basiskarten und 3D Objekten via Three.js. Als Basiskarte wurde die frei verfügbare und auf OpenStreetMap basierende Liberty Variante von OpenFreeMap.org implementiert. Zur Umsetzung der Interaktivität wurde das JavaScript Framework Alpine eingesetzt, für die Gestaltung der Interface Komponenten wurde auf TailwindCSS in Kombination mit einem eigenen Stylesheet eingesetzt.

Backend

Das Backend der Single-Page-Webanwendung wurde mittels der Sprache Python implementiert und bildet die Kernlogik der Anwendung. Nach einer längeren Entscheidungsphase wurde Flask als Webframework gewählt, da es gegenüber dem asynchron arbeitenden FastAPI weniger komplex ist. Flask wird zur Bereitstellung einer JSON-basierten API genutzt und stellt fünf Endpunkte bereit, welche die Datenabfrage im Frontend ermöglichen. Die Endpunkte ermöglichen folgende Abfragen: alle Kategorien, alle POIs, alle Tags, alle POIs gefiltert nach ausgewähltem Tag und ein POI mit allen seinen Tags.

Zur Verwaltung der Python-Abhängigkeiten wurde der Package-Manager *uv* eingesetzt. Dieser ermöglicht eine schnelle und deterministische Installation der benötigten Pakete und eignet sich insbesondere für den Einsatz in automatisierten Build- und Deployment-Prozessen. Die Definition der Abhängigkeiten sowie deren Versionen erfolgt zentral über die Datei *pyproject.toml*. Dadurch wird eine konsistente Laufzeitumgebung geschaffen und der Build-Prozess reproduzierbar gestaltet.

Um Daten persistent speichern zu können, nutzt das Backend eine relationale PostgreSQL-Datenbank. Die Datenbank wird beim Start des Backends mithilfe des zugrunde liegenden Datenbankschemas (siehe Abbildung 2) initialisiert. Das Schema modelliert dabei die Tabellen Points of Interest, Kategorien, Tags sowie deren Beziehungen in einer Hilfstabelle.
Um die Bearbeitung der Tabellen zu vereinfachen, wurde ein Bash-Skript (*csv2sql.sh*) hinterlegt, welches eine csv-Datei entgegen nimmt und diese in SQL-Befehle umwandelt, sodass die Datenbank mit Werten gefüllt werden kann.
Die Query-Logik nutzt einfache SQL-Abfragen sowie JOIN-Operationen um zusammengehörige Informationen sinnvoll zusammenzufassen und als JSON bereitzustellen.

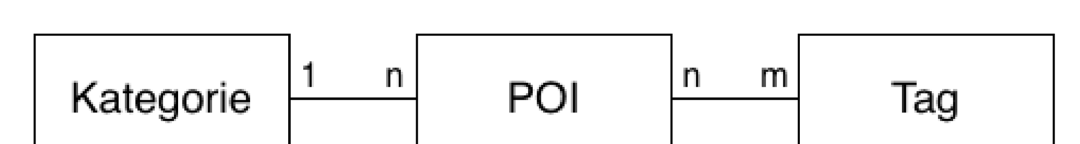


Abb. 2: Datenbankschema der Anwendung

Websockets

Der Websocket-Standard ermöglicht eine bidirektionale Echtzeitkommunikation zwischen allen gängigen Webbrowsern [2] und einer geeigneten Server-Komponente, und rückte daher frühzeitig als zentrale Technologie zwecks Umsetzung der Echtzeit-Synchronisation in den Fokus.

Als Serverkomponente stehen diverse freie Implementierungen zur Verfügung. Aus Gründen der Einfachheit wurde zunächst auf die durch das Python-Framework FastAPI bereitgestellte Abstraktion zurückgegriffen und die Funktionalität als Teil des Backends implementiert.

Da die von uns entwickelte Architektur jedoch keinerlei Überschneidungen mit dem Backend erfordert, wurde die Websocket-Komponente später in eine eigenständige Komponente ausgelagert, welche auf der Python-Bibliothek *websockets* basiert.

Die so erhaltene alleinstehende Komponente erlaubt zudem eine horizontale Skalierung durch den Einsatz mehrerer parallel laufender Instanzen hinter dem Load-Balancer *Caddy*. Die Kommunikation zwischen den Instanzen erfolgt über ein Redis-Pub/Sub-System, welches ebenfalls eine einfache horizontale Skalierung ermöglicht. Diese Architektur ist in Abbildung 3 dargestellt. Redis bezieht sich in diesem Kontext lediglich auf das Protokoll, zum Einsatz kommt die offene Implementierung *Valkey*.

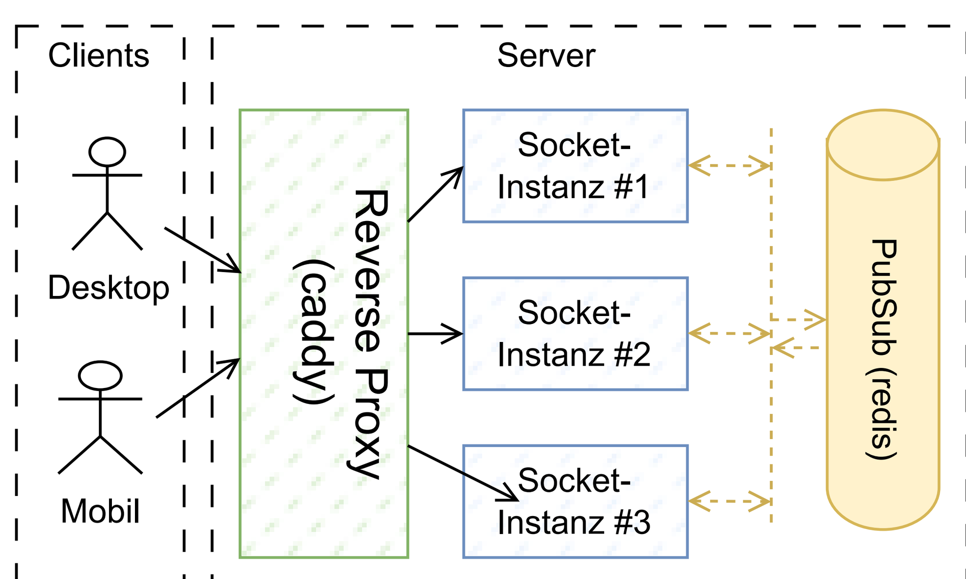


Abb. 3: Horizontale Skalierung des Websocket-Servers

Synchronisierte Kartenansicht

Die Synchronisierung der Kartenansicht erfolgt über eine persistente Websocket-Verbindung zwischen Client und Socket-Server und nutzt eine Reihe von definierten Nachrichten-Typen, die sowohl im Client als auch im Socket-Server implementiert sind.

Client-Perspektive

Der Client nutzt das in MapLibreGL integrierte Event-System, um Bewegungen der Kartenansicht zu erkennen, primär die Events *movestart*, *move* und *moveend*. Die so erhaltenen Positionsupdates werden jedoch nicht bei jeder Änderung gesendet. Stattdessen wird die jeweils aktuelle Position lokal vorgehalten und im regelmäßigen Abstand von 100 ms durch einen Timer verschickt (Throttling), was in unseren Tests einen angemessenen Kompromiss zwischen Nutzbarkeit und Nachhaltigkeit darstellte.

Eingehende Nachrichten werden im Client verarbeitet und der lokale Zustand der Kartenansicht entsprechend angepasst. Dabei wird die in MapLibreGL integrierte *easeTo*-Methode verwendet, welche eine sanfte Animation mit einer maximalen Dauer in der Länge des Update-Intervalls erzwingt, um die Bewegung zu glätten.

Server-Perspektive

Aufgebaute Websocket-Verbindungen werden serverseitig anhand eines durch den Backend-Server generierten kurzlebigen JSON *Web Token*[3] an eine eindeutige UUID gebunden, welche die Identität des jeweiligen Clients repräsentiert, und somit das Senden von Nachrichten im Namen anderer Benutzenden verhindert.

Der Server leitet eingehende Events asynchron an alle anderen Clients weiter, die diese benötigen. Mittels einer auf dem Socket-Server vorgehaltenen Lookup-Tabelle wird die Follower-Beziehung zwischen den Clients abgebildet, um sicherzustellen, dass nur jene Clients Positionsupdates erhalten, die dem sendenden Client folgen.

Da im Falle horizontaler Skalierung nicht zwingend alle Teilnehmenden eines Raumes mit derselben Socket-Server-Instanz verbunden sind, wird der Zustand des Raumes über ein zentrales Redis-Pub/Sub-System synchronisiert. Jeder Socket-Server abonniert die für seine Clients relevanten Räume und publiziert eingehende Events an diese.

Performance-Analyse

Die Evaluation der Performance verfolgt zwei zentrale Ziele: Zum einen soll sichergestellt werden, dass die Plattform eine ausreichende Anzahl an Nutzer:innen unterstützt, um den Anforderungen einer kollaborativen 3D-Kartenanwendung gerecht zu werden. Zum anderen soll durch effiziente Softwarearchitektur und einen bewussten Umgang mit Rechenressourcen der Bedarf an Hardware sowie der damit verbundene Energieverbrauch reduziert werden, um einen Beitrag zur ökologischen Nachhaltigkeit des Systems zu leisten.

Messung der Websockets

Für die Messungen des Ressourcenbedarfs ist vor allem der WebSocket-Server relevant. Es wird angenommen, dass das Ausliefern der statischen Dateien des Frontends sowie die wenigen Datenbankabfragen pro Verbindung im Vergleich zur konstanten Kamera-Synchronisierung über den WebSocket-Server kaum Einfluss auf die Gesamtlast haben.

Methodik

Um diese Komponente auf ihre Grenzen zu testen und zu optimieren, wurden verschiedene Versionen der Anwendung einem Lasttest unterzogen und relevante Metriken, wie die Dauer des Verbindungsaufbaus, die Anzahl der Nachrichten sowie die Verzögerung der Antworten verglichen. Zur Erzeugung der Last werden mithilfe des Open-Source-Tools *k6* beliebig viele virtuelle User erzeugt, die einem Raum beitreten und ihre Kamera bewegen oder anderen folgen.

Auswertung

Abbildung 4 zeigt, dass die erste Version des Websocket-Protokolls bis zu 100 Usern verzögerungsfrei, bei 150 aber schon kaum noch benutzbar ist. Die ausgetauschten Nachrichten wachsen quadratisch an, da die Kamera-Updates von allen Usern an alle anderen User gesendet werden. Ab 100 Usern können nicht mehr alle Nachrichten verarbeitet werden und die Verzögerung der einzelnen Verbindungen steigt stark an.

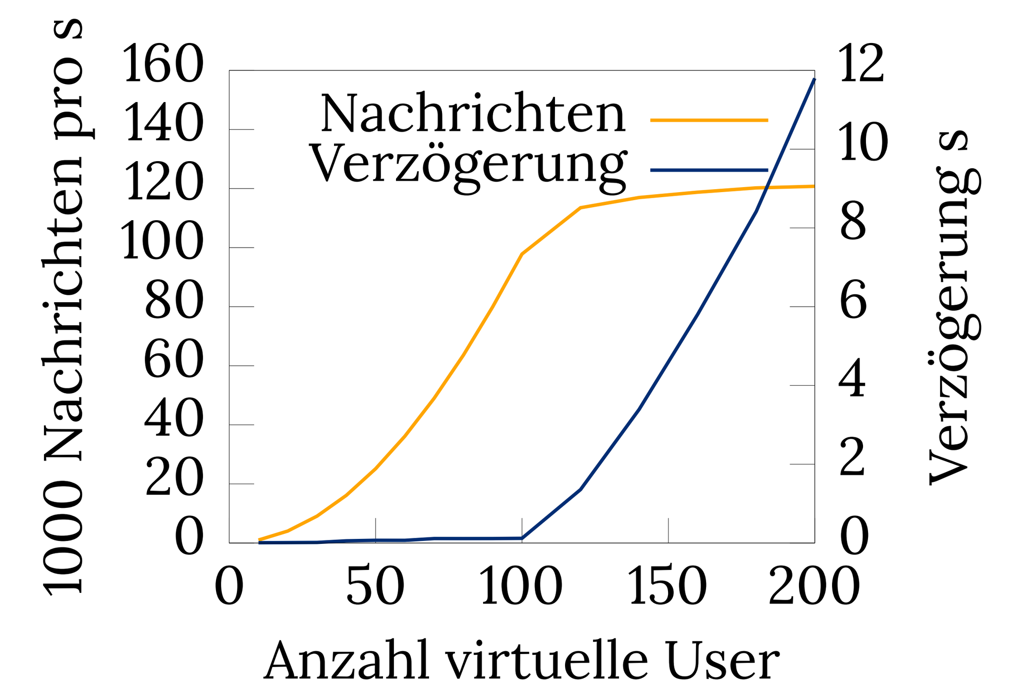


Abb. 4: Lasttest Websocket Protokoll v1

In der zweiten Version des Websocket Protokolls speichert der Server die Follower-Beziehungen und leitet Kamera-Updates nur an Clients weiter die diese Daten brauchen. Wie in Abbildung 5 zu sehen ist, reduziert sich dadurch die Anzahl der Nachrichten stark und es gibt bis 200 Usern kaum Verzögerungen. Erst ab ca. 350 Usern zeigen sich starke Verzögerungen und die Anzahl der Nachrichten die ausgetauscht werden können stößt an eine Grenze.

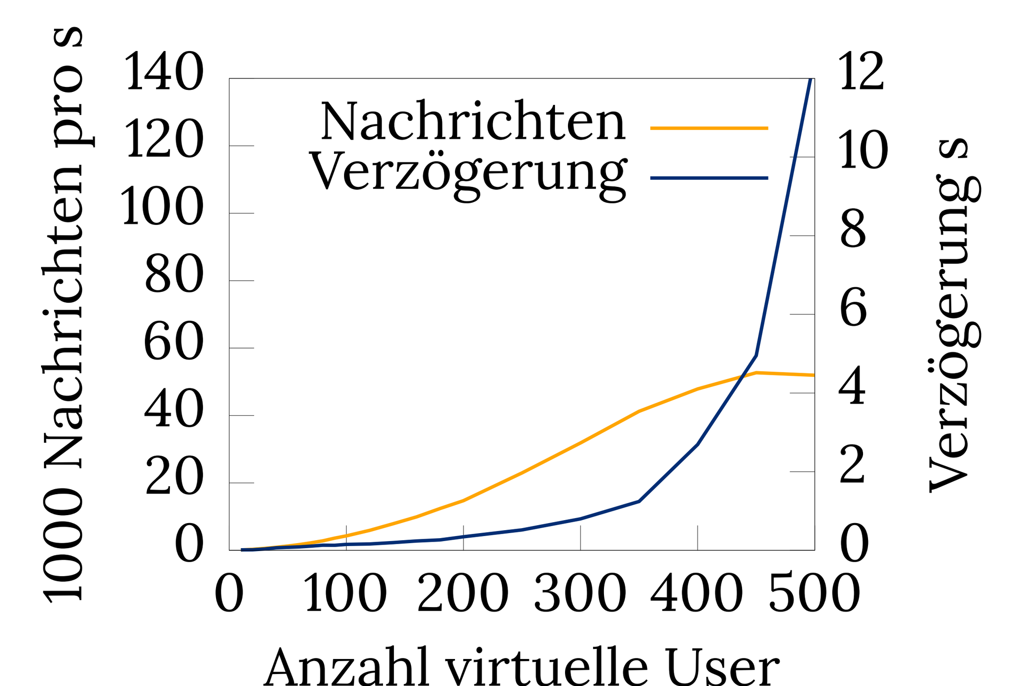


Abb. 5: Lasttest Websocket Protokoll v2

Durch die zuvor beschriebene horizontale Skalierung kann die Kapazität der Anwendung nochmals erhöht werden. Die Messung der Anwendung mit 16 Websocket-Containern, dargestellt in Abbildung 6, zeigt, dass es erst ab 500 Usern zu stärkeren Verzögerungen kommt und die Anwendung nicht mehr nutzbar ist.

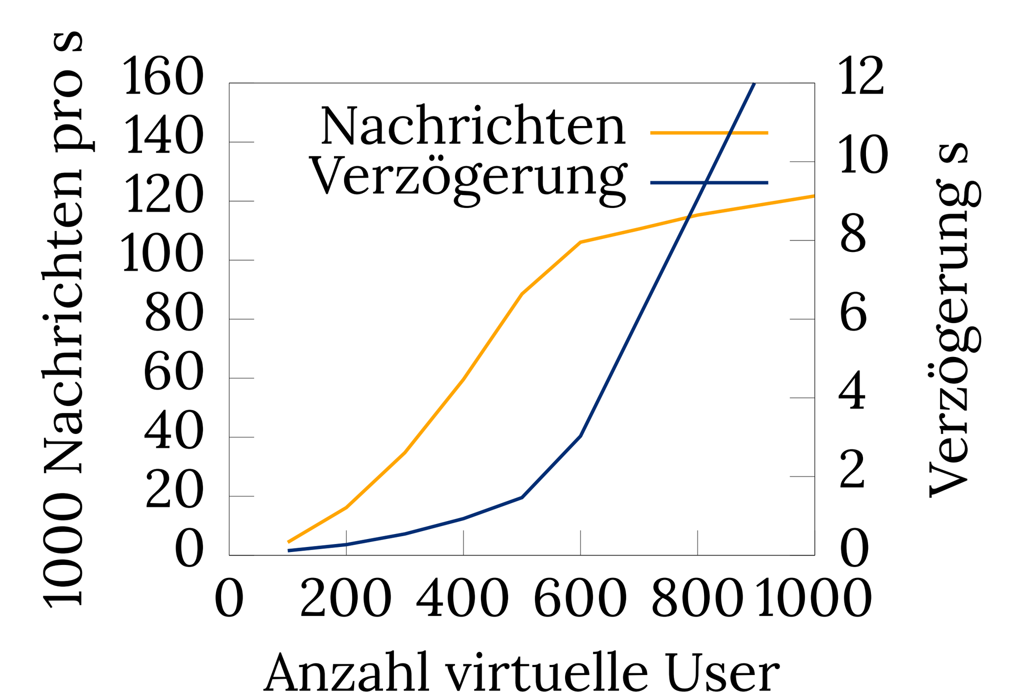


Abb. 6: Lasttest mit 16 Websocket Container

End-to-End Tests

Zur funktionalen Absicherung der Anwendung werden automatisierte End-to-End-Tests (E2E) eingesetzt. Diese Tests prüfen zentrale Nutzungsszenarien aus Sicht realer Nutzer:innen über die vollständige Systemgrenze hinweg.

Die E2E-Tests werden insbesondere nach funktionalen Änderungen an der Benutzeroberfläche, an der Raumverwaltung sowie vor Deployments ausgeführt. Ziel ist es, sicherzustellen, dass zentrale Interaktionsabläufe weiterhin korrekt funktionieren. Durch die frühzeitige Ausführung dieser Tests können funktionale Fehler erkannt werden, bevor sie sich unter Last oder im Produktivbetrieb manifestieren

Methodik

Die Tests werden mit Playwright in einem „Headless-Browser“ ausgeführt und sind über Kommandozeilenargumente parametrisierbar, wodurch eine reproduzierbare Ausführung in unterschiedlichen Umgebungen ermöglicht wird.

Ein zentrales E2E-Szenario ist das Erstellen und Starten eines Raums über die Benutzeroberfläche. Der folgende Codeausschnitt zeigt diesen Ablauf exemplarisch:

Auswertung

Die E2E-Tests ermöglichen eine qualitative Bewertung der funktionalen Stabilität der Anwendung. Fehler in der Benutzerführung, fehlerhafte UI-Zustände oder Probleme beim Initialisieren von WebSocket-Verbindungen können frühzeitig identifiziert werden. Die Tests bilden damit die Grundlage für weiterführende Last- und Performancemessungen, da sie sicherstellen, dass die getesteten Szenarien funktional korrekt sind.

Verteilter End-to-End-Lasttest

Neben der funktionalen Absicherung ist es erforderlich, das Verhalten der Gesamtanwendung unter realitätsnaher Last zu untersuchen. Hierfür werden die bestehenden E2E-Szenarien nicht nur einzeln ausgeführt, sondern auch auf mehrere virtuelle Maschinen verteilt.

Im Gegensatz zu synthetischen Lasttests erzeugen diese verteilten E2E-Tests reale Browser-Clients, die die Anwendung vollständig durchlaufen. Dadurch werden nicht nur Server-seitige Komponenten belastet, sondern auch Aspekte wie Session-Handling, WebSocket-Verbindungen und das Zusammenspiel mehrerer Subsysteme berücksichtigt [4].

Methodik

Die Last wird über mehrere virtuelle Maschinen bei Hetzner verteilt, wobei jede VM eine definierte Anzahl paralleler Playwright-Browser-Clients ausführt. Die VMs werden zentral in der Datei *hosts.txt* gepflegt und automatisiert per SSH angesteuert. Die Gesamtnutzerlast ergibt sich aus der Summe aller Clients über sämtliche VMs hinweg, wodurch eine realistische Simulation verteilter Nutzerzugriffe ermöglicht wird. Die virtuellen Maschinen werden dabei dynamisch über Shell-Skripte gestartet und gestoppt.

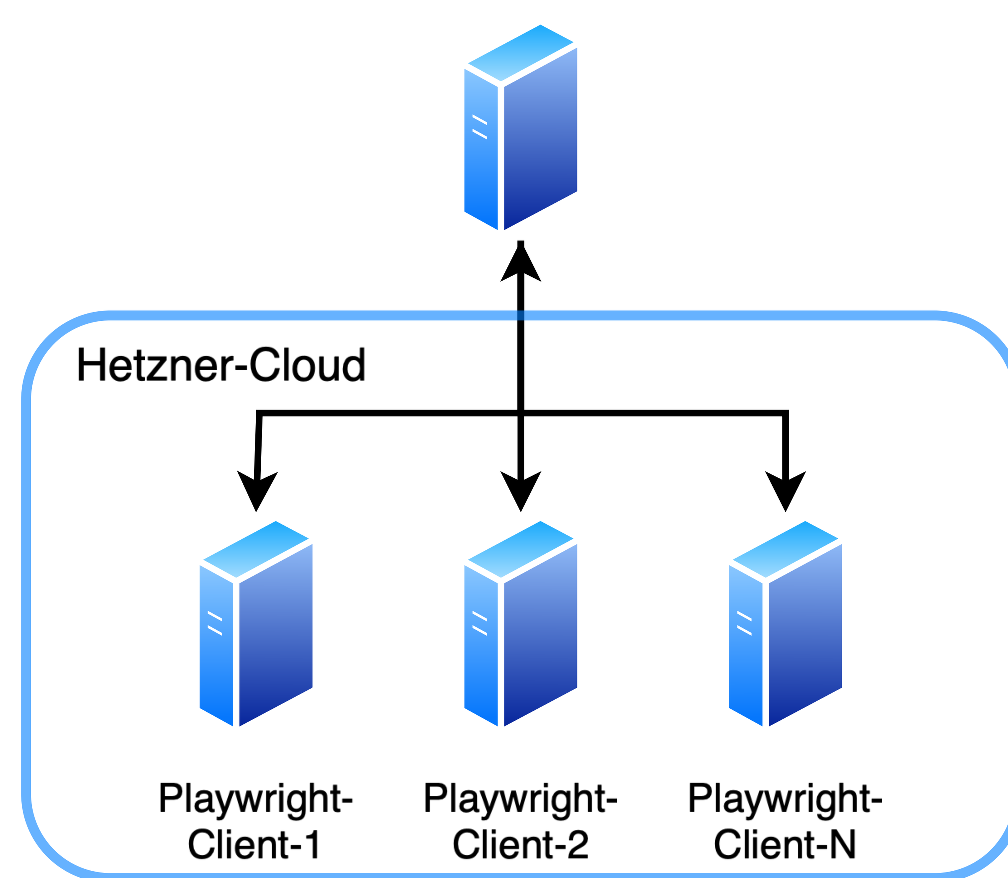


Abb. 7: Verteiltes Testen

Auswertung

Der verteilte E2E-Lasttest liefert eine ganzheitliche Bewertung der Stabilität der Gesamtanwendung unter realitätsnaher Last, da echte Browser-Clients den kompletten Nutzerfluss ausführen. Dadurch werden insbesondere Engpässe und Fehler sichtbar, die in isolierten Funktionstests oder reinen WebSocket-Simulationen nicht auftreten. Die Ergebnisse werden pro Host in Logs abgelegt und können anschließend zentral aggregiert werden, um Fehlerquoten, Laufzeiten und Abbruchraten zwischen Testläufen vergleichbar auszuwerten.

Résumé

In diesem Projekt sind wir auf viele unterschiedliche Probleme gestoßen, sowohl organisatorischer, als auch technischer Natur. Beispielhaft dafür steht die Terminfindung, die vor allem am Anfang des Projekts nicht immer ganz einfach war. Hinzu kamen unterschiedliche Wissensstände im Bereich der eingesetzten Technologien. Wir hatten uns im Vorhinein darauf geeinigt, dass wir alle den Anspruch haben, die Technologien, die wir einsetzen, auch verstehen und beherrschen zu können. Einige der Teilnehmer:innen des Projektes hatten bereits vorher mit bestimmten Technologien gearbeitet und konnten diese Ad hoc einsetzen, andere wiederum mussten mehrere Stunden in der Woche investieren, um ein gewisses Grundlevel zu erlangen, um an den wöchentlichen Diskussionen teilnehmen zu können.

Aufgrund der Größe unserer Gruppe haben wir uns dazu entschieden, in Kleingruppen zu arbeiten, sobald unser Grundkonzept stand. Die Ergebnisse wurden dann bei den wöchentlichen Treffen vorgestellt, diskutiert und das weitere Vorgehen besprochen. Diese Arbeitsweise hat uns in der zweiten Hälfte des Projekts erheblich geholfen, da wir uns vorher teilweise mit der gesamten Gruppe auf eine Aufgabe gestürzt haben und dementsprechend langsam vorangekommen sind. Trotzdem kann man dieser Art des Arbeitens nicht nur negatives abgewinnen, es hat uns beispielsweise auch geholfen technisch etwa auf einem Level zu bleiben und nicht nur Expert:innen in einem Teilbereich zu werden, sondern immer die Gesamtheit der Anwendung im Auge zu behalten. Zusätzlich konnte sich jedes Gruppenmitglied im Ergebnisprozess beteiligen und Argumente für Technologien seiner/ihrer Wahl vorbringen, ohne dass ein Teilbereich (z.B. Expert:innen) die alleinige Entscheidungsmacht über den von ihnen gewählten Bereich hatten. Die Diskussionskultur war dabei bis auf wenige Ausnahmen immer positiv und auf Augenhöhe und auch wenn es kleinere Meinungsverschiedenheiten gab, konnten diese spätestens beim gemeinsamen Kaffee in der Pause bereinigt werden.

Während des Projekts sind wir auch auf technische Probleme bzw. Limitationen gestoßen. So war der eigentlich Plan der Anwendung, eine durch Augmented Reality angereicherte Hafenerfahrung zu bieten, schnell wieder verworfen, da sowohl die Datengrundlage für ein solches Projekt, als auch die dafür benötigten Technologien entweder nicht aktuell, nicht plattformübergreifend oder nicht in einer Open-Source-Variante zur Verfügung standen. Trotz der vor allem anfangs häufig vorkommenden (technologischen) Sackgassen konnten stetig neue Lösungsansätze oder Wege gefunden werden, sodass die entwickelte Anwendung schlussendlich ein zufriedenstellendes Ergebnis für die gesamte Gruppe darstellt.

Retrospektiv können wir Erkenntnisse in mehreren Bereichen, zum Beispiel der gemeinsamen Gruppenarbeit mitnehmen. Hierzu zählen vor allem qualitätssichernde Maßnahmen, beispielsweise der Einsatz von Formatierungswerkzeugen wie *prettier* und *black* oder einem konsequentem, vorher abgestimmtem, Git-Workflow zu folgen. Auch auf technischer Seite konnten wir uns im Projekt weiterbilden, besonders hervorzuheben ist dabei die tiefere Einarbeitung in die Programmiersprache *Python* und die Beschäftigung mit FastAPI und seinen Vor- und Nachteilen. Abschließend haben wir also nicht nur das ausgesprochene Ziel einer kollaborativen Kartenanwendung erreicht, sondern konnten auch unser eigenes Können durch technisch versierte Diskussionen und Vertiefungen erweitern und wichtige Aspekte für weitere Projektarbeiten mitnehmen.

Literatur

[1] „Rootless mode,“ besucht am 26. Jan. 2026. Adresse: https://docs.docker.com/engine/security/rootless/.

[2] A. Deveria. „Web sockets,“ Can I use... browser support tables, Can I use..., besucht am 24. Jan. 2026. Adresse: https://caniuse.com/websockets.

[3] M. B. Jones, J. Bradley und N. Sakimura. „JSON web token (JWT),“ RFC 7519 (Proposed Standard), Internet Engineering Task Force (IETF), besucht am 25. Jan. 2026. Adresse: https://datatracker.ietf.org/doc/html/rfc7519.

[4] „Tests zur Benutzbarkeit,“ in *Handbuch zum Testen von Web-Applikationen: Testverfahren, Werkzeuge, Praxistipps*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, ISBN: 978-3-540-68185-4. Adresse: https://doi.org/10.1007/978-3-540-68185-4_9.