Meilenstein 1: «Schiffe versenken»

Autorin: Jetsün Künsel Emchi

Repository: https://github.com/Kemiem/schiffe-versenken.git

Commit Hash: 48899de52f3c439fdc509a8b6363d848179fc4f1

Datum: 18.09.2025

1. Ziel und Zweck

Der Umfang des Projekts umfasst die Entwicklung eines Multiplayerspiels «Schiffe versenken», bei dem der Server als Source of Truth dient und zwei Spieler:innen gegeneinander antreten.

Die Spieler:innen platzieren ihre Flotte auf einem 8×8-Spielfeld; anschliessend folgen abwechselnde Schüsse.

Pro Lobby steht ein Echtzeit-Chat zur Verfügung, der über WebSockets realisiert ist.

Für die Datensicherung und die Persistenz der Spielzüge, des Action-Logs und des Chats wird SQLite eingesetzt.

Der Fokus dieses Projekts liegt auf dem Aufbau einer stabilen Realtime-Architektur mit klar definierten Regeln und einer einfachen Bedienbarkeit.

2. Regeln und Logiken

Das umgesetzte Spiel bildet eine leichte Adaption der Grundregeln des Spiels «Schiffe versenken».[1]

Die Regeln lauten wie folgt:

Name der Regel:	**Regelbeschrieb**:
Flotten	Die eingesetzten Flotten besteht aus je einem *4-er* und *3-er* sowie je zwei *2-er* und *1-er* Schiffen.
Platzierung	Die Platzierung erfolgt via *Klick*, *horizontal/vertikal*, innerhalb des *erlaubten Bereichs*, es sind *keine*

[1] Gesellschaftsspiele.Spielen.de. Schiffe versenken [PDF]. https://gesellschaftsspiele.spielen.de/uploads/files/3284/58e6634bd08c1.pdf (abgerufen am 15.09.2025)

	Überlappungen erlaubt und es gibt *keinen Mindestabstand*.
Schussauswertung	Es gibt die Schussauswertungen «Hit», «Miss» und «Sunk».
Sichtschutz	Die Spieler:in sieht nur die eigene Schiffe vollständig. Bei der Gegner:in sind nur die eigenen Schüsse und Resultate sichtbar.
Sieg	Wer alle gegnerischen Schiffe trifft, gewinnt.
Spielfeld	Das umgesetzte Feld ist *8x8*.
Züge	Die Züge sind alternierend.

3. Mockups

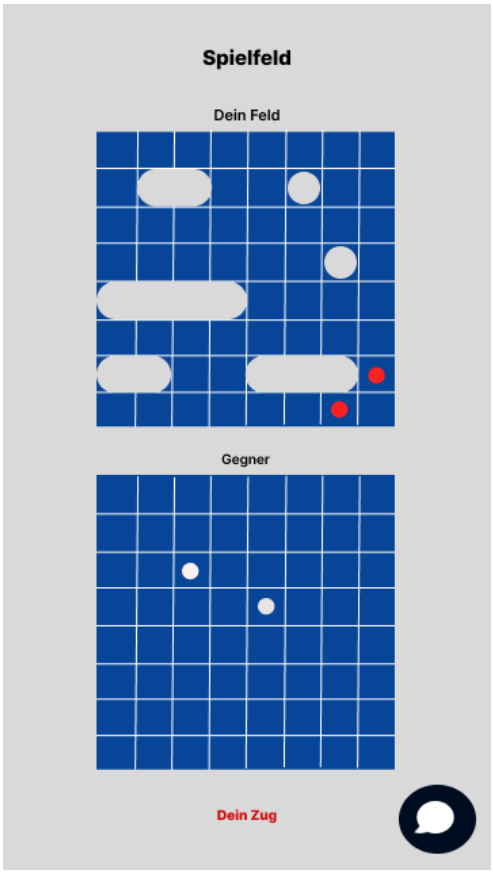

Abbildung 1: Android Mockup

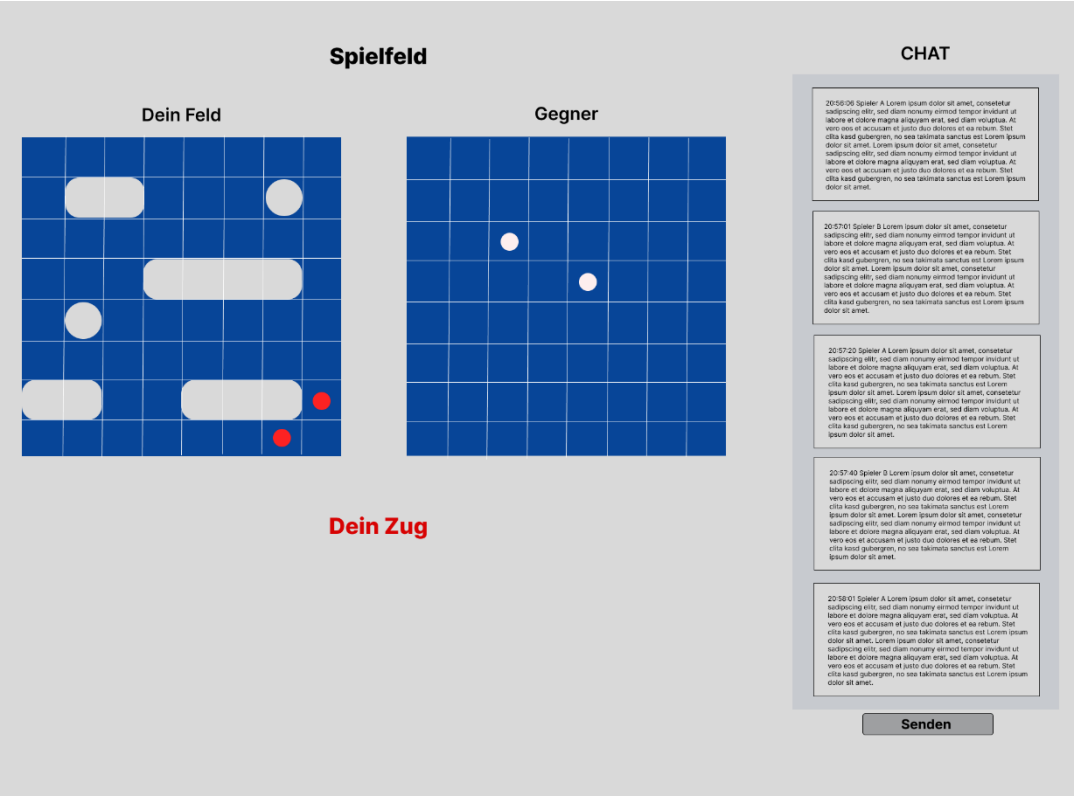

Abbildung 2: Desktop Mockup

4.1. Funktionale Anforderungen

Name:	Anforderungsart:
Dokumentiertes Kommunikationsprotokoll (Events und übertragene Daten)	Muss
Nachvollziehbarkeit: Jede Aktion muss mit «userId» und Zeitstempel im Action-Log ersichtlich sein.	Muss
Persistenz-Layer (SQLite via Prisma) • User:in • Lobby • Schiffe • Züge • Nachrichten • ActionLog.	Muss
Realtime-Chat via Websocket, pro Lobby	Muss
Rejoin mit Snapshot nach Reload	Kann
Responsives Frontend (Desktop und Mobil)	Muss

Server setzt die komplette Aktions-Logik durch (Platzierung, Zugrecht, Auswertung)	Muss
Statistik (Gewinne und Trefferquote)	Kann
Zug-Timer (Auto-Pass)	Kann

4.2. Nicht-Funktionale Anforderungen

Name:	Anforderungsart:
Performance: Server und Client sollen Benutzeraktionen (Login, Chat, Schuss) innerhalb von Max. 1 Sekunde verarbeiten.	Muss
Stabilität: Das System muss dauerhaft lauffähig sein und mehrere gleichzeitige Verbindungen ohne Fehler verarbeiten können.	Muss
Zugriffskontrolle: Nur eingeloggte Benutzer:innen dürfen Aktionen ausführen (z. B. Chat oder Schüsse).	Muss
Benutzerfreundlichkeit: Das UI soll intuitiv und klar strukturiert sein, sodass alle Nutzer:innen das Spiel bedienen können.	Muss
Responsives Design: Die Anwendung soll auf verschiedenen Geräten (Desktop, Mobile) korrekt dargestellt werden.	Muss
Fehlertoleranz und Wiederherstellung: Bei einem Seiten-Reload oder Verbindungsverlust sollen Benutzende wieder der letzten Spielinstanz beitreten können.	Muss
Nachvollziehbarkeit: Wichtige Ereignisse (Login, Chat, Spielaktionen) werden mit User-ID und Zeitstempel im ActionLog festgehalten.	Muss
Wartbarkeit: Der Code ist modular (Client/Server/DB getrennt) und in TypeScript klar strukturiert.	Kann

4.3. Abnahmekriterien (Prüfbare Punkte)

Folgende Punkte werden als Abnahmekriterium geprüft:

1. Alle Züge und Aktionen werden nach Spielende in der Datenbank gespeichert und sind dort nachvollziehbar.
2. Das Spiel lässt sich von zwei Benutzer:innen in separaten Browserfenstern vollständig spielen.
3. Nach einem Reload des Browsers wird der aktuelle Spielzustand korrekt wiederhergestellt.
4. Der Server reagiert auf Benutzeraktionen (Login, Chat, Schuss) innerhalb von höchstens 1 Sekunde.
5. Das Spiel kann über WebSockets stabil betrieben werden, auch wenn mehrere Clients gleichzeitig verbunden sind.
6. Nur eingeloggte Benutzer:innen können Spielaktionen (z. B. Chat oder Schüsse) ausführen.
7. Das Frontend ist responsiv gestaltet und funktioniert auf Desktop- und Mobilgeräten einwandfrei.
8. Die Codebasis (Server, Client, Datenbank) ist modular aufgebaut und kann bei Bedarf um zusätzliche Spielfunktionen (z. B. Statistik, Timer) erweitert werden.

5. Technologien und Bibliotheken

Für das **Frontend** nutze ich *TypeScript* in Kombination mit *React* und *Vite*. Das Styling erfolgt mit *Tailwind CSS,* was ein responsives Layout ermöglicht. Die Realtime-Kommunikation zum Server wird über Socket.IO-client realisiert.

Im **Backend** nutze ich *Node.js* in *TypeScript* mit *Express* als Server. *Socket.IO* verwende ich für die Echtzeit-Kommunikation zwischen Server und Clients. Für die Validierung der eingehenden Daten und Events verwende ich *Zod*.

Die Persistenz übernimmt eine *SQLite*-Datenbank. *Prisma ORM* wird genutzt, um einen direkten Zugriff auf die Datenbank aus *TypeScript* zu ermöglichen.

Zudem wird *Prettier* für die automatische Formatierung des Codes genutzt.

6. Kommunikation zwischen Client und Server

Die Kommunikation zwischen Client und Server erfolgt in Echtzeit über WebSockets mithilfe von Socket.IO.

Zusätzlich kann HTTP/REST optional für Healthchecks und Debugging genutzt werden.

6.1. Übertragene Daten

Folgende Daten werden zwischen Client und Server ausgetauscht:

Daten:	Beschrieb:
Benutzeridentifikation	• name • userId
Lobby-Informationen	• erstellen • beitreten • Status
Platzierungsdaten	• Schiffe • Positionen
Spielzüge	• Koordinaten der Schüsse
Chatnachrichten	• Text • User • Zeitstempel
Snapshots des Spielzustands	• Boards • Spielerstatus • Chatverlauf
Fehlermeldungen	• Validierungsfehler • Berechtigungsfehler

Alle eingehenden Event Daten werden serverseitig mit Zod-Schemas validiert. Der Server erzwingt ausserdem Turn-Ownership und Fog-of-War (Sichtschutz), sodass nur zulässige Daten übertragen werden.

6.2. Zweck des Austauschs

Die Datenübertragung dient der Anmeldung und Identifikation von Benutzer:innen, der Verwaltung von Lobbys (Erstellen, Beitreten und Statusänderungen) sowie der Übermittlung der Flottenplatzierung inklusive Validierung und Speicherung. Sie ermöglicht ausserdem das Abfeuern und Auswerten von Schüssen in Echtzeit und die Synchronisierung des aktuellen Spielzustands zwischen allen Clients. Darüber hinaus unterstützt sie die Realtime-Kommunikation im Chat innerhalb einer Lobby und erlaubt die Übermittlung von Fehlermeldungen, wenn ungültige Aktionen erkannt werden.

7. Übersicht der Events

7.1. Client zu Server (Eingaben)

Event	Payload-Beispiel	Zweck
auth.login	{name}	Benutzer anlegen/identifizieren
Lobby.create	{name}	Lobby erstellen
Lobby.join	{lobbyId}	Lobby beitreten
Lobby.ready	{lobbyId, ready}	Bereitschaft in der Platzierphase melden
Board.place	{ lobbyId, ships:[{type,cells:[{x,y}]}] }	Flotte senden (Validierung und Speicherung)
Shot.fire	{ lobbyId, x, y }	Schuss auslösen (nur wenn am Zug)
Chat.send	{ lobbyId, text }	Chatnachricht senden

7.2. Server zu Client (Resultate/State)

Event	Payload-Beispiel	Zweck
Lobby.snapshot	{ state, players, you:{seat}, yourBoardMask, enemyMask, chat }	Aktuellen Stand synchronisieren
Placing.updated	{ seatIndex, done }	Platzierstatus je Spieler
Game.start	{ firstTurnSeat }	Startsignal Spiel
Shot.result	{ bySeat: 1, x: 3, y: 5, result: "MISS" }	Der Server bestätigt allen Clients das Ergebnis eines Schusses
Turn.change	{ seatIndex }	Zugwechsel anzeigen
Game.over	{ winnerSeat }	Spielende kommunizieren
Chat.message	{ id,user,text,createdAt }	Chatnachricht verteilen
error	{ code,message }	Validierungs- oder Berechtigungsfehler melden

8. Datenmodell (Konzept)

Das Datenmodell umfasst die zentralen Entitäten *User, Lobby, LobbyPlayer, Ship, Shot, Message* und *ActionLog*.

Die Entität *User* steht für die Nickname-Identität der Spieler:innen.

Eine *Lobby* repräsentiert jeweils eine einzelne Spielinstanz und speichert unter anem den aktuellen Status sowie die Gewinner-ID.

LobbyPlayer verknüpft Benutzer:innen mit einer *Lobby* und hält Angaben wie Sitz, Bereitschaftsstatus und verbleibende Schiffszellen fest.

Ein *Ship* beschreibt Typ und Position der Flotte sowie bereits getroffene Segmente.

Shot speichert die Koordinaten der Schüsse, Ziel-Sitz, Resultat und Zeitpunkt.

Message bildet die Chatnachrichten pro Lobby ab.

Das *ActionLog* protokolliert alle Aktionen chronologisch (JOIN, PLACE, SHOT, TURN, START, END, CHAT) inklusive der übertragenen Daten und Zeitstempel.

9. Arbeitsplan /Balkendiagramm

Phase	Dokumente
M1 (Planung)	• Doku • Mockups • Protokoll • Datenmodell • Chat (Grundgerüst)
M2 (Grundgerüst)	• Login • Lobby • Erweiterung Chat (Realtime, in-memory) • Snapshot-Skeleton
M3 (Spielbar + Persistenz)	• Platzieren (Validierung und DB) • Schüsse (Auswertung und DB) • Turn-Handling • Snapshot
M4 (Feature-Complete)	• Zod überall • Rejoin/Snapshot stabil • Action-Log-UI • Feinschliff
Finale Abgabe	• README/Install • Screenshots • Präsentation.

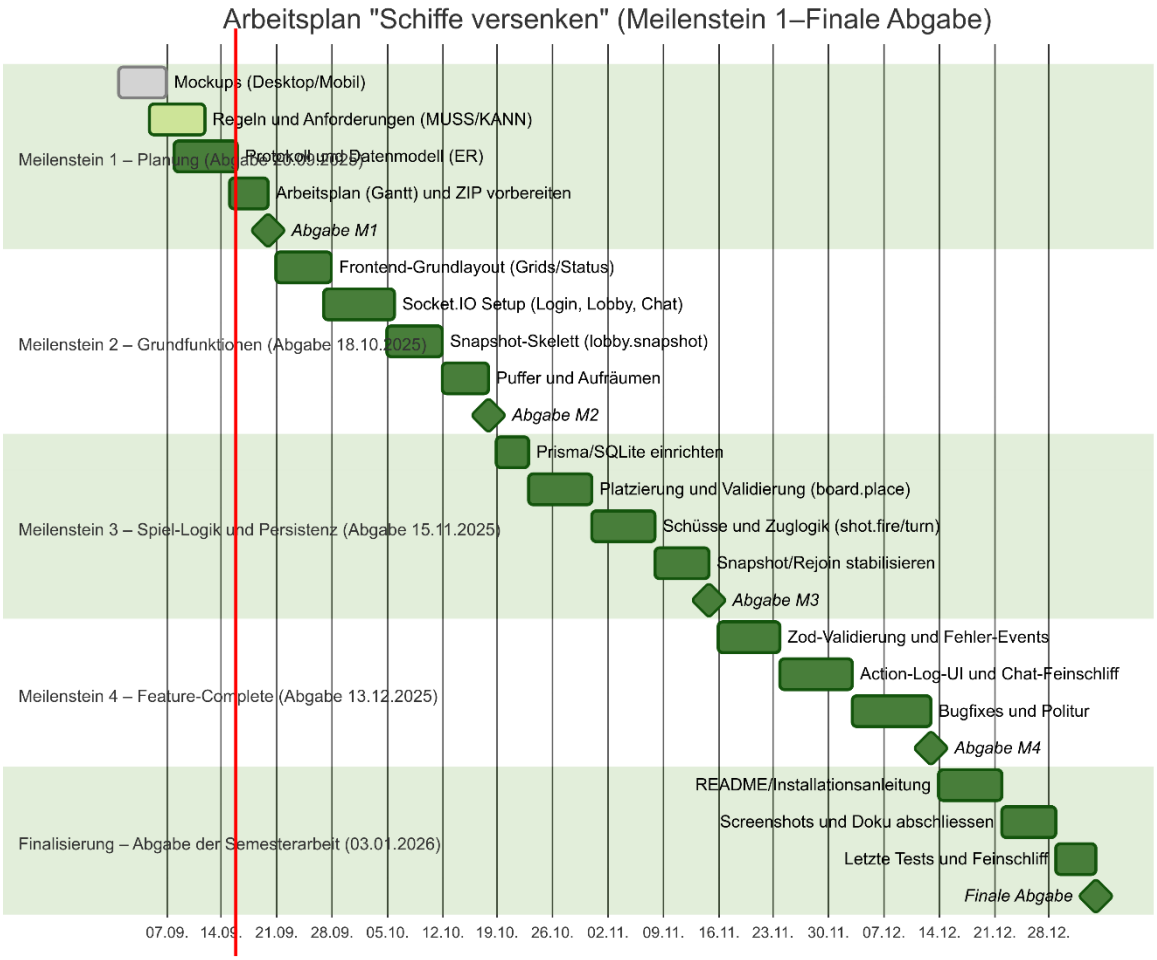

Abbildung 3: GANTT Chart

10. Grundgerüst Chat

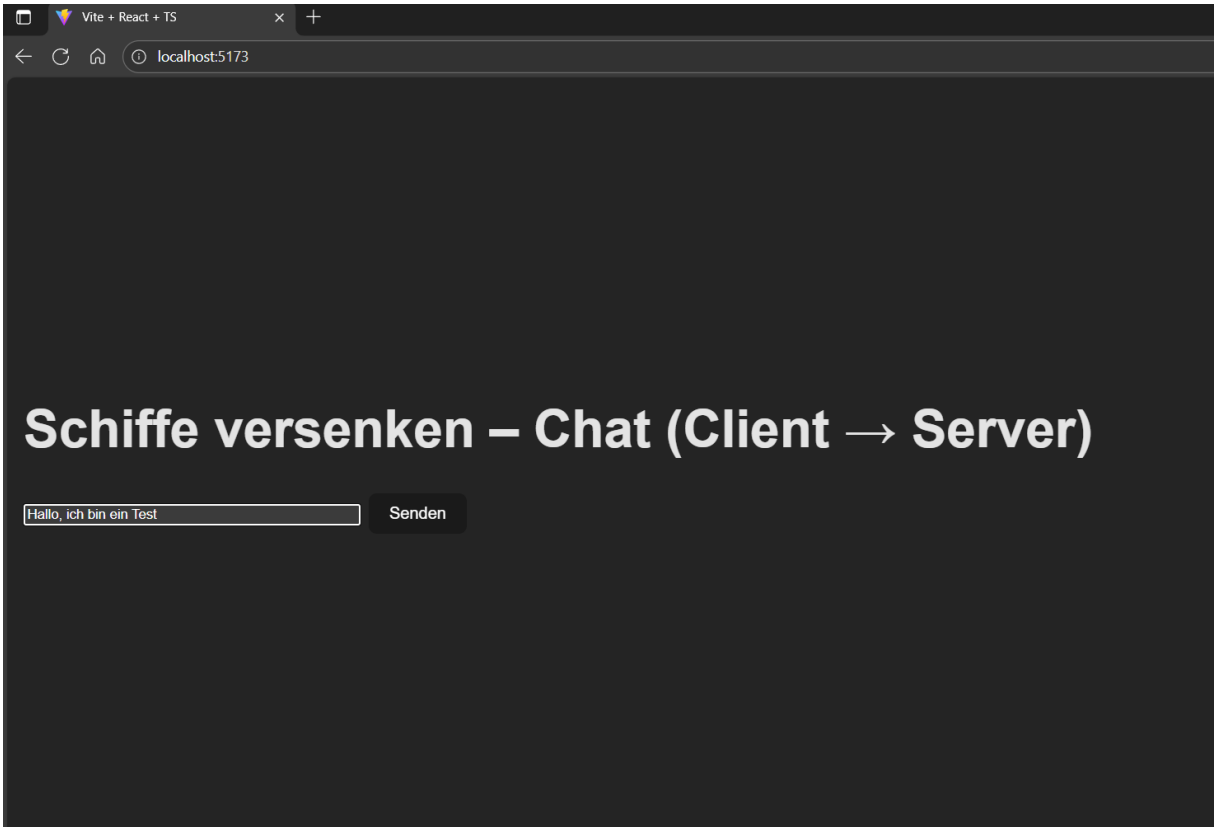

Abbildung 4: Textfeld mit Eingabe einer Chatnachricht

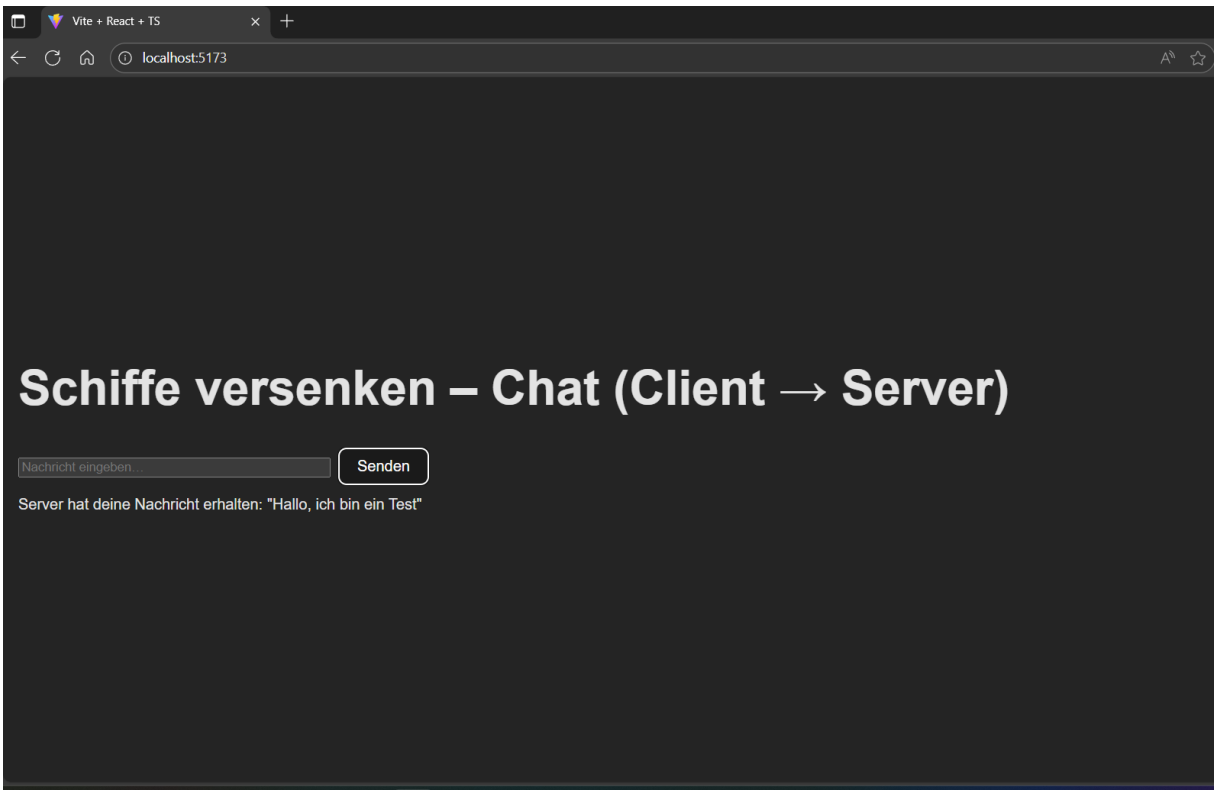

Abbildung 5: Bestätigung des Servers über der Erhalt der Nachricht