Project information

Created on
February 23, 2026

Name	Last update
backend	14 hours ago
docs/architecture	14 hours ago
frontend	2 days ago
scripts	14 hours ago
.env-example	1 week ago
.gitignore	14 hours ago
.gitlab-ci.yml	2 days ago
README.md	14 hours ago
docker-compose.yml	2 days ago

README.md

dsys-cahoot-light

A real-time, distributed quiz platform inspired by Kahoot. A **Host** (Teacher) creates and manages quizzes while **Players** (Students) join via a six-digit Game PIN and compete live – complete with real-time video streams powered by LiveKit.

The application is fully deployed and playable at:

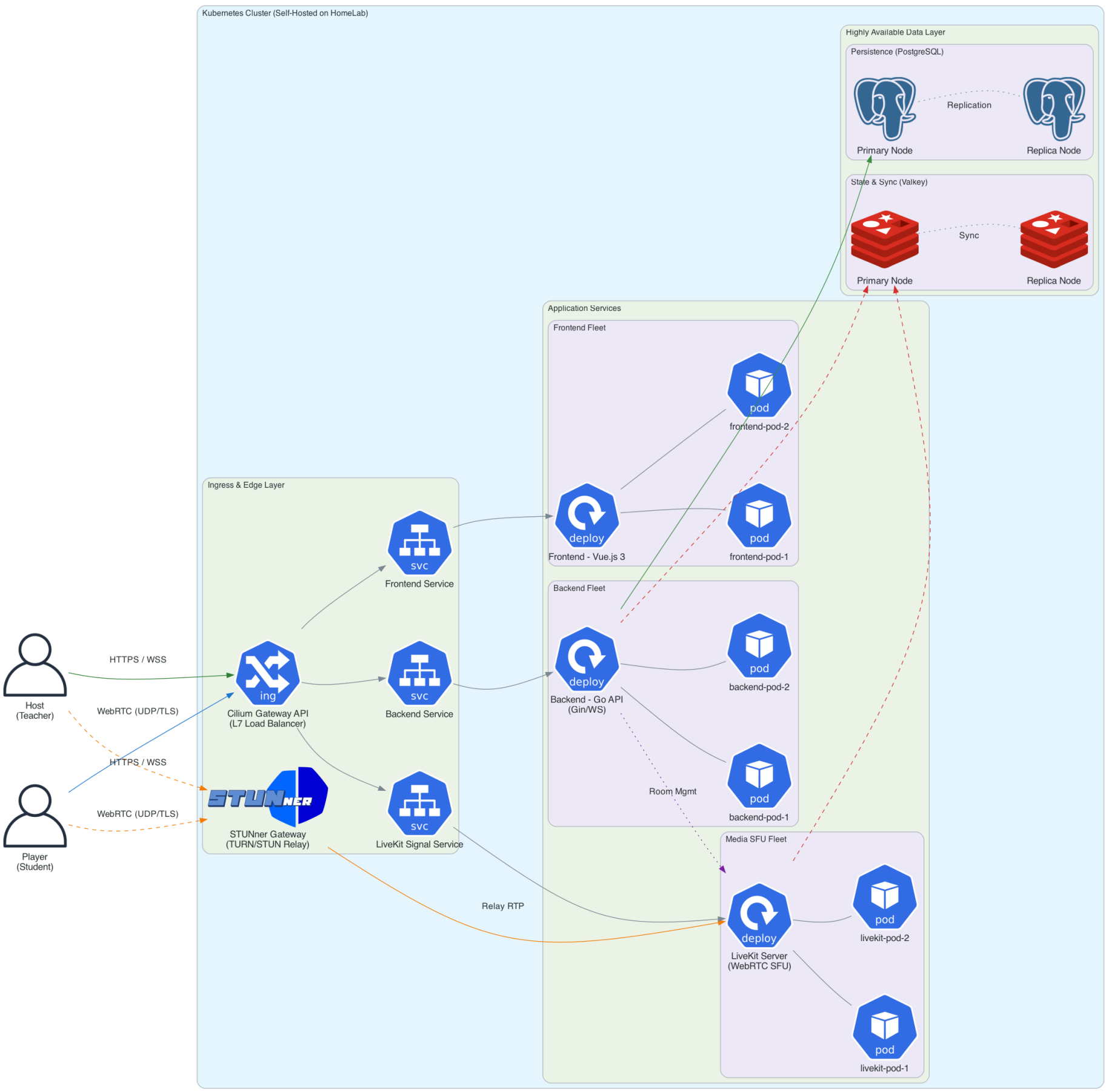
> 🌐 **Live URL:** https://dsys-cahoot-light.jorishaenseler.ch

Features

Feature	Description
Real-Time Gameplay	Low-latency Secure WebSocket connections deliver questions, collect answers, and push score updates to every participant simultaneously. Including ingame stats such as points, answerstreaks and relative leaderboard updates in real-time
Live Video Streams	Participants share their camera feeds via LiveKit (WebRTC). Opponents videofeeds are displayed around the game area throughout the session.
Role-Based Access	**Host** (JWT-protected): create quizzes, manage questions, control game flow. **Player**: join anonymously via a six-digit PIN.
Distributed State	Game state is not tied to a single server. Valkey Pub/Sub propagates events across all backend pods, so hosts and players can be connected to completely different backend instances.
Failover & Resilience	If any backend pod crashes during a game, the state is preserved in Valkey. The frontend uses exponential-backoff reconnection for both WebSocket and LiveKit connections, ensuring seamless recovery.

Feature	Description
Quiz Management	Hosts can create, edit, duplicate, and delete quizzes with multiple-choice questions and enable videostreaming functionality per quiz.
Persistent Data	User accounts, quizzes, and questions are stored in PostgreSQL with full replication.
Secure Auth	Access/Refresh token flow. Short-lived JWTs (15 min) for API calls; long-lived (7 days) `HttpOnly` cookie for refresh tokens with automatic rotation.
Observability	Every API response carries an `X-Backend-Server` header exposing the serving pod name, visible in the UI footer for instant infrastructure insight.
Load Testing	Bundled k6 and `hey` scripts for benchmarking REST, WebSocket, and quiz endpoints under high concurrency.

Architecture & Tech Stack



dsys-cahoot-light Architecture

> The diagram is generated by `docs/architecture/architecture.py` using the diagrams Python library. See docs/architecture/README.md for instructions on regenerating it locally.

Technology Stack

Layer	Technology
Backend	Go (Golang) · Gin · Gorilla WebSocket
Frontend	Vue.js 3 · Vite · Nginx
Real-Time Video	LiveKit (WebRTC SFU)
TURN / ICE Relay	STUNner (K8s-native TURN/STUN)
Distributed Cache	Valkey 9 via Hyperspike Valkey Operator
Database	PostgreSQL 18 via CloudNativePG
Orchestration	Kubernetes · Gateway API · Cilium
CI/CD	GitLab CI
Local Dev	Docker Compose

High Availability Design

Every layer of the production deployment is designed to eliminate single points of failure and is High Available (HA) by Design.

Component	HA Mechanism
Gateway API	Single Cilium-managed entry point; `HTTPRoute` objects get routed to the corresponding pods ensuring proper load balancing.
Frontend	Horizontally scalable; static assets served by Nginx containers. Any pod can serve any user.
Backend	Stateless Go pods. Game state lives in Valkey, so any pod can handle any request. Kubernetes restarts crashed pods automatically.
Valkey	Valkey Operator deploys a Primary + Replicas setup. Pub/Sub keeps all backend pods in sync. Automatic failover on primary loss.
PostgreSQL	CloudNativePG Operator provides synchronous replication and instant automatic failover to a replica.
LiveKit	Multiple LiveKit pods behind the Gateway API HTTPRoute. The SFU mesh distributes media load.
STUNner	Kubernetes-native TURN server (see section below). LoadBalancer service with a stable public IP.

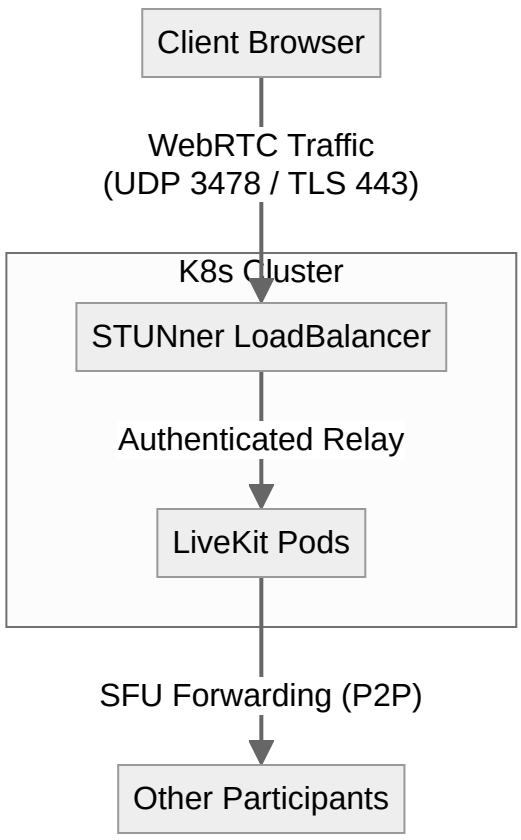
WebRTC Media: LiveKit + STUNner

Why a TURN Server?

WebRTC establishes peer-to-peer media paths using ICE (Interactive Connectivity Establishment). In many network environments – corporate firewalls, symmetric NAT, mobile carriers – direct peer-to-peer UDP is blocked. A **TURN (Traversal Using Relays around NAT) server** acts as a media relay of last resort: all audio/video packets flow through it when a direct path cannot be negotiated.

STUNner – Kubernetes-Native TURN

STUNner is a purpose-built TURN/STUN gateway designed to run natively inside Kubernetes. Instead of routing WebRTC media through an external relay, STUNner exposes a single `LoadBalancer` service (UDP + TLS/443) and forwards authenticated TURN sessions directly to LiveKit pods inside the cluster.



Key properties of the STUNner setup:

- **Protocol support**: UDP (port 3478) for low-latency paths; TURN-over-TLS (port 443) for firewalled clients.
- **Ephemeral credentials**: The backend generates short-lived TURN credentials via the LiveKit SDK (shared secret mechanism using JWT). Credentials are injected into Kubernetes `SealedSecret` manifests so they never appear in plaintext in the repository.
- **No external TURN cost**: Media stays inside the Kubernetes cluster, reducing egress costs and latency compared to cloud TURN services.
- **High availability**: The STUNner `GatewayConfig` and `UDPRoute` are Kubernetes CRDs managed by the STUNner operator. If the TURN pod restarts, Kubernetes reschedules it and the LoadBalancer IP remains stable.
- **Cilium integration**: Because Cilium is used as the CNI, proper `externalTrafficPolicy: Local` and SNAT configuration ensures that the client's public IP is preserved through the LoadBalancer, which is required for correct ICE candidate attribution.

LiveKit Configuration

LiveKit is deployed via its official Helm chart with:

- Multiple replicas behind the Gateway API HTTPRoute for signalling (WSS) aswell as room creation/management.
- Uses Valkey for as key value store and cluster sync between replicas.
- STUNner listed as the sole TURN endpoint in the LiveKit configuration so all ICE relay traffic flows through the in-cluster gateway.

Local Development with Docker Compose

> **Note**: The Docker Compose setup is intended for local development and demonstration only. It runs a single backend instance, single Valkey node, and LiveKit in `--dev` mode. It does **not** completely fulfil the distributed/HA requirements of the production Kubernetes deployment.

Prerequisites

- Docker and Docker Compose installed.

Quick Start

```
# 1. Clone the repository
git clone https://gitlab.ost.ch/joris.haenseler/dsys-cahoot-light.git
cd dsys-cahoot-light

# 2. Copy the example environment file (defaults work out-of-the-box)
cp .env-example .env

# 3. Start all services (PostgreSQL, Valkey, LiveKit, Go Backend, Vue Frontend)
docker-compose up --build -d
```

Services Started by Docker Compose

Service	Container	Port(s)
PostgreSQL 18	`cahoot-db`	`5432`

Service	Container	Port(s)
Valkey 9	`cahoot-valkey`	`6379`
LiveKit Server	`cahoot-livekit`	`7880` (HTTP), `7881` (RTC), `7882/udp` (RTC), `7801`
Go Backend	`cahoot-backend-1`	`8080`
Vue.js Frontend	`cahoot-frontend`	`80`

Access

Endpoint	URL
Frontend UI	http://localhost
Backend API	http://localhost:8080/api/v1
Health Check	http://localhost:8080/health

Environment Variables (`.env-example`)

```
POSTGRES_USER=postgres
POSTGRES_PASSWORD=password
POSTGRES_DB=cahoot

DB_HOST=db
DB_PORT=5432

VALKEY_HOST=valkey
VALKEY_PORT=6379

JWT_SECRET=local_dev_secret

VITE_BACKEND_URL=http://localhost:8080
VITE_WS_URL=ws://localhost:8080

LIVEKIT_URL=http://localhost:7880
LIVEKIT_API_KEY=devkey
LIVEKIT_API_SECRET=secret
```

Stopping

```
docker-compose down -v   # -v also removes the named volume (pgdata)
```

Kubernetes Deployment (Production)

The production stack runs on a self-hosted Kubernetes cluster with Cilium as the CNI and Cert-Manager for automatic TLS using letsencrypt. All manifests live in a separate GitOps repository.

Demonstrating High Availability (Failover)

To prove that the system withstands pod crashes without losing game state:

1. **Start a game**: log in as a Host, pick a quiz, and start the game.
2. **Join as a player**: open a second browser tab, join with the displayed Game PIN.
3. **List running pods**:

   ```
   kubectl get pods -n dsys-cahoot-light
   ```

4. **Kill a backend pod** (simulates crash):

```
kubectl delete pod <backend-pod-name> -n dsys-cahoot-light
```

The game continues – state is preserved in Valkey and the frontend reconnects automatically using exponential backoff.

5. **Kill a LiveKit pod** (simulates SFU loss):

```
kubectl delete pod <livekit-pod-name> -n livekit
```

Video feeds will briefly freeze/disconnect. The frontend's `useLiveKit` hook detects the disconnection and automatically reconnects with exponential backoff to a new pod.

6. **Kill a STUNner pod** (simulates TURN gateway loss):

```
kubectl delete pod <stunner-pod-name> -n stunner
```

Existing TURN allocations are dropped, but since STUNner runs in a high-availability LoadBalancer setup, new WebRTC connections (or ICE restarts) will immediately hit a healthy replica.

7. **Kill a Valkey pod** (simulates cache node loss):

```
kubectl delete pod <valkey-pod-name> -n dsys-cahoot-light
```

The Valkey Operator promotes a replica to primary within seconds.

8. **Kill a PostgreSQL pod** (simulates DB node loss):

```
kubectl delete pod <postgres-pod-name> -n dsys-cahoot-light
```

CloudNativePG performs automatic failover; the primary role is transferred to a replica.

9. **Verify**: the Host and Player UIs remain fully functional throughout all of the above.

Security & Authentication

- **Access Token**: Short-lived (15 min) JWT sent in `Authorization: Bearer` header.
- **Refresh Token**: Long-lived (7 days) JWT in a secure, `HttpOnly` cookie. Automatically used by Axios interceptors to obtain new access tokens without the user noticing.
- **Token Rotation**: Every successful token refresh issues a new pair, mitigating replay attacks.
- **WebSocket Auth**: Short-lived JWTs are passed during the WebSocket handshake.
- **LiveKit Tokens**: Server-signed room tokens generated per session by the backend.
- **TURN Credentials**: Ephemeral HMAC-SHA1 credentials (TTL 24 h) generated by the LiveKit SDK; never stored in plaintext.
- **CORS**: Strictly configured to the frontend origin; supports `Access-Control-Allow-Credentials`.

Load Testing

k6 – Full Scenario (WebSocket included)

Simulates up to 50 concurrent users over 50 seconds. Each virtual user:

1. Hits `/health` (public).
2. Registers a dummy teacher account (PostgreSQL write).
3. Logs in (DB read + JWT signing).
4. Generates a student join token (`/api/v1/game/join`).
5. Opens a WebSocket connection and stays connected 30 s – fake students appear live in the Teacher Lobby.

```
docker run --rm -i \
  -e GAME_PIN=123456 \
  -e BASE_URL=https://dsys-cahoot-light.jorishaenseler.ch/api/v1 \
  -e HEALTH_URL=https://dsys-cahoot-light.jorishaenseler.ch/health \
  grafana/k6 run - < scripts/load-test.js
```

Thresholds: 95 % of requests < 500 ms · error rate < 5 %.

Clean up test data

```
DELETE FROM users WHERE username LIKE 'teacher_%';
```

hey – Single-Endpoint Benchmark

```
# Install
brew install hey

# Remote endpoint
./scripts/hey-load-test.sh https://dsys-cahoot-light.jorishaenseler.ch/health 500 5000

# Local endpoint (interactive menu)
./scripts/hey-load-test-local.sh

# Local – public health check
./scripts/hey-load-test-local.sh http://localhost:8080/health 100 10000

# Local – protected quiz list (DB read + JWT auth)
TOKEN=$(curl -s -X POST http://localhost:8080/api/v1/login \
  -H "Content-Type: application/json" \
  -d '{"username":"loadtester","password":"password123"}' | grep -o '"token":"[^"]*' | grep -o '[^"]*$')

./scripts/hey-load-test-local.sh http://localhost:8080/api/v1/quizzes 100 10000 $TOKEN
```

Benchmark Results Local (Apple M1 Pro, Docker Compose)

Public health check (raw Go performance):

```
Requests/sec: 20 781
Average:      0.0047 s
Fastest:      0.0004 s
```

Protected quiz list (DB read + JWT + JSON serialisation):

```
Requests/sec:   1523
Average:        0.098 s
```

Testing WebSocket Security Manually

Without a token – should return `401 Unauthorized`:

```
curl -i -N \
  -H "Connection: Upgrade" -H "Upgrade: websocket" \
  -H "Host: localhost:8080" -H "Origin: http://localhost:8080" \
  -H "Sec-WebSocket-Key: MTIzNDU2Nzg5MDEyMzQ1Ng==" \
  -H "Sec-WebSocket-Version: 13" \
  http://localhost:8080/api/v1/game/ws
```

With a valid student token – should return `101 Switching Protocols`:

```
TOKEN=$(curl -s -X POST http://localhost:8080/api/v1/game/join \
  -H "Content-Type: application/json" \
  -d '{"game_id":"TEST1","username":"TestStudent"}' | grep -o '"token":"[^"]*' | grep -o '[^"]*$')

curl -i -N \
  -H "Connection: Upgrade" -H "Upgrade: websocket" \
  -H "Host: localhost:8080" -H "Origin: http://localhost:8080" \
  -H "Sec-WebSocket-Key: MTIzNDU2Nzg5MDEyMzQ1Ng==" \
  -H "Sec-WebSocket-Version: 13" \
  "http://localhost:8080/api/v1/game/ws?token=$TOKEN"
```

👥 Authors

- Joris Haenseler