Project information

Created on
February 23, 2026

Name	Last update
backend/API	6 hours ago
caddy	19 hours ago
frontend	19 hours ago
load-test	6 hours ago
.dockerignore	6 days ago
.gitignore	1 week ago
README.md	just now
docker-compose.yml	19 hours ago

README.md

0chat

0chat is a distributed real-time chat application built for the OST Distributed Systems challenge task, FS 2026. The system demonstrates failover: during the final presentation, one backend instance will be deliberately shut off and traffic must continue through the surviving instance.

A fresh clone plus one command must start the complete system:

```
docker compose up --build
```

Architecture

```
Browser
  -> Caddy (least-connections load balancer)
        -> .NET Instance 1 (ASP.NET Core, SignalR Hub) --+
        -> .NET Instance 2 (ASP.NET Core, SignalR Hub) --+
                                                         Redis (SignalR backplane)
        Both instances share:
                                                         PostgreSQL (persistent storage)
```

Components

Frontend: React + Vite

- Keep the UI simple and task-focused.
- Dark mode is required by default and is the only supported theme.
- Do not add a light-mode toggle or `prefers-color-scheme` light fallback.

Backend: ASP.NET Core (.NET 10), two identical stateless instances

- SignalR for real-time WebSocket communication.

- Redis backplane via `Microsoft.AspNetCore.SignalR.StackExchangeRedis`, so both instances can push messages to all connected clients.
- EF Core for database access and migrations.
- Backend instances run migrations on startup behind a PostgreSQL advisory lock so two instances do not migrate concurrently.
- Manual JWT authentication with a simple users table, PBKDF2 password hashes, normalized unique usernames, and rotating refresh tokens stored as server-side hashes.
- All non-auth endpoints are protected.

Load balancer: Caddy

- Uses least-connections load balancing because long-lived WebSocket connections should be distributed by current connection load instead of simple request order.

Database: PostgreSQL

- Single persistent database shared by both backend instances.

Containerization: Docker + Docker Compose

- Services: frontend, caddy, backend instance 1, backend instance 2, postgres, redis.

Real-Time Message Flow

1. Client sends a chat message.
2. One backend instance receives it.
3. The backend saves it to PostgreSQL.
4. Redis pub/sub notifies both backend instances.
5. Each backend instance pushes the message to its own connected clients.

Product Scope

The app is intentionally small for the current milestone while leaving room to expand later.

- There is one global chatroom.
- Every registered user is automatically a member of that chatroom.
- There are no separate rooms, direct messages, channels, invitations, or membership management yet.
- Users only have visible usernames.
- Do not add profile images, real names, bios, or user profile pages.
- Usernames must be unique, preferably enforced case-insensitively.
- Display usernames consistently.
- Chat messages belong to the global chatroom and should include author, content, and creation time.
- A `RoomId` is not needed while there is only one room.
- SignalR should broadcast new messages to all authenticated connected clients.
- No per-room group management is needed yet.

Frontend Routes

React Router should manage exactly two frontend routes for now:

1. `/login` - unauthenticated auth screen.
2. `/` - protected chat app. Redirect unauthenticated users to `/login`.

The `/login` screen should contain both login and registration as tabs or a segmented control. Only one form is visible at a time. Do not add a separate `/register` frontend route. The backend may still expose separate `/login` and `/register` API endpoints.

After successful login or registration, store the JWT client-side and redirect to `/`. Registration should log the user in immediately to keep the demo flow short.

The chat screen should stay focused: message history, message input, send action, visible current username, and logout.

Commands

```
# Start the full system
docker compose up --build

# Build/check the backend when local dotnet is unavailable
docker compose build backend-1

# Start in detached mode
docker compose up -d --build
```

```
# Stop everything
docker compose down

# Stop and remove volumes (wipe database)
docker compose down -v

# View logs for a specific service
docker compose logs -f <service-name>

# Run load test once the system is up, adapting URL/token as needed
k6 run load-test/script.js
# or: hey -n 10000 -c 100 -H "Authorization: Bearer <token>" http://localhost/api/...
# or: wrk -t4 -c100 -d30s -H "Authorization: Bearer <token>" http://localhost/api/...
```

Challenge Requirements

All requirements below must be met to pass the lecture:

1. Load balancing with multiple service instances and failover of one service instance.
2. Containerized deployment.
3. Simple frontend.
4. JWT-based authentication following OWASP guidance.
5. Persistent storage.
6. Latest stable releases of chosen libraries and frameworks.
7. Fresh clone plus a single command, for example `docker compose up`, brings up the entire working system including database migrations or seeding if needed. No manual setup steps are allowed except secrets management.
8. Load test against a JWT-protected endpoint, with documented results, bottleneck analysis, and requests per second.

Additional constraints:

- Login and registration are required to use the application.
- Every non-auth API endpoint must require a valid JWT.
- Existing libraries and code may be used only if they are open-source software.
- The frontend may stay simple, but it must be usable and dark mode only.
- The student must understand and be able to explain all code in the project, regardless of how it was produced.

Milestones

Date	Milestone
02.03.2026	Initial plan and project idea hand-in
30.03.2026	Initial commit with current progress
04.05.2026	Progress update, partial working state acceptable
18.05.2026	Final hand-in and presentation

Final presentation format:

- 5 minutes architecture, components, and design decisions.
- 5 minutes demo.
- 5 minutes Q&A.
- PDF presentation handed in after the presentation.