Chat Application with User Authentication

This project is a full-stack real-time chat application .It implements secure user authentication, public chat rooms, and private messaging, using a modern Java backend and a simple, responsive web UI.

1. Project Overview

The application allows users to:

- Register and log in with a unique username and email.
- Join public chat rooms and exchange messages in real time.
- Send private messages to specific users.
- View chat history for rooms and private conversations.

The focus is on demonstrating full-stack skills: frontend (HTML/CSS/JS), backend (Spring Boot), database design (MySQL + JPA/Hibernate), REST APIs, real-time communication with WebSockets, and secure authentication using JWT + Spring Security.

2. Tech Stack

Backend

- Java 17+
- Spring Boot 3 (Web, Security, WebSocket, Spring Data JPA)
- MySQL as relational database
- Hibernate / JPA for ORM
- JWT for stateless authentication and authorization

Frontend

- HTML5

- CSS3 (custom, responsive with Flexbox)
- Vanilla JavaScript (no framework)
- SockJS + STOMP for WebSocket communication

3. Features vs Assignment Requirements

Below is how this project maps to the internship assignment points.

3.1 User Interface (HTML/CSS/JS)

- **Login & Registration Page (index.html)**
  - Contains separate sections for registration and login, with simple tab-like buttons to switch between them.
  - Uses clean HTML forms, semantic labels, and a shared stylesheet (`style.css`) for layout and styling.
- **Chat Page (chat.html)**
  - Layout split into sidebar (room list + private message section) and main chat area (current room title, history, message input).
  - Responsive design using CSS flexbox and media queries; adjusts to smaller screens by stacking sections vertically.

3.2 Responsive and Accessible Design

- Uses a centered card-style container with clear typography and contrast.
- Layout degrades gracefully on smaller screens using media queries (sidebar moves above chat area).
- Basic accessibility practices: label–input pairs, consistent font size, and no complex custom widgets.

3.3 Real-Time Messaging (WebSockets)

- Real-time communication implemented using Spring WebSocket with STOMP.
- Server exposes a STOMP endpoint at `/ws`.
- Clients connect using SockJS + STOMP and:

    - Send messages to `/app/chat.send`.
    - Subscribe to `/topic/room.{roomId}` for public room updates.
    - Subscribe to `/queue/user.{userId}` for private messages.

3.4 Server-Side Logic (Java + Spring Boot)

- Backend built on Spring Boot with layered architecture:
    - `entity` for JPA entities (`User`, `ChatRoom`, `Message`).
    - `repository` for Spring Data JPA repositories.
    - `service` for business logic (auth, chat, message handling).
    - `controller` for REST endpoints and WebSocket controllers.

3.5 RESTful APIs

Key REST endpoints:

- **Auth**
    - `POST /api/auth/register` – user registration.
    - `POST /api/auth/login` – user login, returns JWT token and user info.
- **User**
    - `GET /api/users/me` – current user info (JWT-protected).
    - `GET /api/users/online` – returns IDs of users marked as online (simple tracker).
- **Rooms & Messages**
    - `GET /api/rooms` – list of chat rooms (public and, internally, private if used).
    - `GET /api/rooms/{roomId}/messages` – history for a specific room.
    - `POST /api/rooms/messages` – send a message via REST (stored and then can be fetched).
    - `GET /api/rooms/private/{otherUserId}/messages` – 1-to-1 conversation history between current user and another user.

3.6 Authentication & Authorization (JWT + Spring Security)

- User passwords are hashed using BCrypt (`PasswordEncoder`).

- On login, a JWT token is generated containing user id and username, with a finite expiration time.
- Spring Security is configured with a stateless `SecurityFilterChain` and a custom `JwtAuthenticationFilter` that:
  - Extracts token from `Authorization: Bearer <token>`.
  - Validates token and sets the authentication in the security context.
- REST APIs under `/api/**` (except `/api/auth/**` and static assets) require a valid JWT.

3.7 Relational Database Design (MySQL)

Entities and schema:

- `User`
  - `id`, `username`, `email`, `password`, `createdAt`.
- `ChatRoom`
  - `id`, `name`, `type` (`PUBLIC`, `PRIVATE`), `createdAt`.
- `Message`
  - `id`, `sender` (FK to `User`), `receiver` (FK to `User`, nullable for public), `room` (FK to `ChatRoom` or nullable depending on configuration), `content`, `timestamp`.

The schema is created/updated via JPA/Hibernate (`ddl-auto=update`) during development.

3.8 JPA / Hibernate Usage

- Repositories use Spring Data JPA interfaces, e.g.:
  - `UserRepository` with `findByUsername`, `existsByEmail`, etc.
  - `MessageRepository` with methods for fetching room history and user-to-user history (ordered by timestamp).
- Entities use standard JPA annotations (`@Entity`, `@Id`, `@GeneratedValue`, `@ManyToOne`, etc.) and Lombok to reduce boilerplate.

3.9 Real-Time Notifications / In-App Alerts

- Real-time “notification” behavior is provided via WebSocket subscriptions:
  - When a new message arrives in a room, all clients subscribed to that room’s topic immediately see the update in the chat area.

    - When a new private message is sent, both the sender and receiver get it in their personal queues (subscribed at `/queue/user.{userId}`).

3.10 Data Security

- Passwords are hashed with BCrypt; plain passwords are never stored.
- All authenticated API calls use JWT, and access to protected endpoints is controlled by Spring Security.
- For deployment, the app is intended to run behind HTTPS so WebSocket and REST traffic are encrypted in transit (this is mentioned as a recommendation in documentation; local dev uses HTTP).

4. Setup & Running Locally

4.1 Prerequisites

- Java 17+
- Maven
- MySQL running locally (or accessible connection)

4.2 Database Setup

1. Create a database:

```
CREATE DATABASE chat_app;
```

2. Update `application.yml` (or `application.properties`) to match your MySQL credentials:

```
spring:
  datasource:
    url: jdbc:mysql://localhost:3306/chat_app
    username: your_mysql_user
    password: your_mysql_password
    driver-class-name: com.mysql.cj.jdbc.Driver

  jpa:
    hibernate:
      ddl-auto: update
    show-sql: true
    properties:
      hibernate:
        format_sql: true
```

4.3 Run the Application

```
mvn spring-boot:run
```

The app will start on `http://localhost:8080`.

4.4 Using the App

- Open `http://localhost:8080/index.html` in a browser.
- Register a new user, then log in.
- On successful login, you are redirected to `chat.html`.
- On `chat.html`:
    - Select a room from the room list (pre-seeded rooms can be inserted via SQL or created via code).
    - Type messages in the main input to send public messages to that room.
    - Use the “Private (debug)” section: enter another user’s `userId` and send a private message (visible only to the sender and receiver).

5. Assumptions & Simplifications

To keep the project focused and complete within the assignment time frame, a few assumptions and simplifications were made. These are documented so expectations are clear.

1. **HTTPS & Production Security**
   - Local development uses HTTP; in a real deployment, the app should run behind HTTPS to encrypt all network traffic, including WebSocket messages and JWT tokens.
   - JWT secret is kept in application configuration for this demo; in production it should be stored securely (e.g., environment variables or a secret manager).
2. **Basic Error Handling**
   - Error responses are simple (e.g., with generic messages) and not heavily localized or structured.
   - This is sufficient for the assignment but could be improved with a global exception handler and standardized error format in a production system.
3. **Private Messaging and Rooms**
   - Private messages are stored with `sender` and `receiver` set; `room` may be null or mapped to a dedicated private room depending on database configuration. The main requirement is that user-to-user conversation history and real-time delivery are working, which they are.
   - Private messaging UI is implemented in a simple “debug” style (enter receiver userId manually) to prove the backend supports it. In a real product, this would be replaced with a user list or search.
4. **Online Users Tracking**
   - A basic in-memory `OnlineUserTracker` is provided, along with an endpoint to read online user IDs.
   - Full integration with WebSocket connect/disconnect events is possible and can be added later; for the assignment, the current approach is enough to demonstrate the concept of in-app awareness/notifications.
5. **Minimal Frontend Framework Use**
   - The frontend uses plain HTML/CSS/JavaScript without frameworks (React, Angular, etc.) intentionally, to keep focus on the required technologies and reduce complexity.

6. Possible Improvements (Future Work)

If more time is available, the following improvements could be added on top of the assignment requirements:

- Replace the debug private-message UI with a proper user list and clickable private chats.
- Integrate WebSocket connect/disconnect events with `OnlineUserTracker` automatically.
- Add message read indicators, typing indicators, and room creation UI.
- Improve validation and show user-friendly error messages in the frontend.
- Add integration tests and more robust unit tests for services and controllers.