MILESTONE-4

ChatterBox: Real-Time WebSocket Chat Application

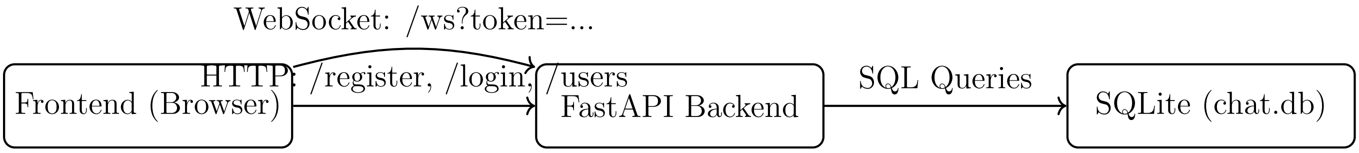
Maria Biju

February 17, 2026

Project Overview

ChatterBox is a real-time chat application built using FastAPI, WebSockets, and SQLite. It supports user registration, login, token-based session validation, live online user tracking, message persistence, and message management (send/edit/delete). An admin dashboard displays registered users and currently online users.

System Architecture Diagram



Explanation: The browser communicates with FastAPI through HTTP for authentication and admin data, and through WebSocket for real-time chat events. FastAPI reads/writes user and message data in SQLite.

Database Design (ER Diagram)

Explanation: The `users` table stores authentication details. The `messages` table stores chat history with sender username and timestamp.

Database Initialization (init_db.py)

Code Snippet

```
conn = sqlite3.connect("chat.db")
c = conn.cursor()

c.execute("""
CREATE␣TABLE␣IF␣NOT␣EXISTS␣users␣(
␣␣␣␣id␣INTEGER␣PRIMARY␣KEY␣AUTOINCREMENT,
␣␣␣␣username␣TEXT␣UNIQUE␣NOT␣NULL,
␣␣␣␣email␣TEXT␣UNIQUE␣NOT␣NULL,
␣␣␣␣password_hash␣TEXT␣NOT␣NULL
)
""")

c.execute("""
CREATE␣TABLE␣IF␣NOT␣EXISTS␣messages␣(
␣␣␣␣id␣INTEGER␣PRIMARY␣KEY␣AUTOINCREMENT,
␣␣␣␣username␣TEXT␣NOT␣NULL,
␣␣␣␣message␣TEXT␣NOT␣NULL,
␣␣␣␣timestamp␣TEXT␣NOT␣NULL,
␣␣␣␣seen␣INTEGER␣DEFAULT␣0
)
""")
```

Explanation

This script creates the SQLite database schema. It creates two tables: `users` for authentication and `messages` for chat persistence.

Application Entry Point (main.py)

Code Snippet

```
app = FastAPI()

app.include_router(user_router)
app.include_router(websocket_router)

app.mount("/ui", StaticFiles(directory="frontend", html=True),
   name="frontend")
```

Explanation

`main.py` initializes the FastAPI application, registers route modules, and serves the frontend HTML pages from `/ui`. The backend is modularized into authentication routes and WebSocket routes.

Data Validation Models (user_models.py)

Code Snippet

```python
class UserRegister(BaseModel):
    username: str
    email: EmailStr
    password: str

class UserLogin(BaseModel):
    username: str
    password: str

class UserResponse(BaseModel):
    id: int
    username: str
    email: EmailStr
```

Explanation

Pydantic models validate incoming request payloads and define response structure. Email validation uses `EmailStr` to ensure correct email format.

Authentication and Users API (user_routes.py)

Code Snippet: Password Hashing

```python
sessions = {}  # token -> username

def hash_password(password: str):
    return hashlib.sha256(password.encode()).hexdigest()
```

Explanation

Passwords are never stored in plain text. Instead, a SHA256 hash is stored in the database. A simple in-memory session store maps generated tokens to usernames.

Code Snippet: Register Endpoint

```
@router.post("/register", response_model=UserResponse)
def register(user: UserRegister):
    hashed = hash_password(user.password)
    cursor.execute(
        "INSERT␣INTO␣users␣(username,␣email,␣password_hash)␣
           VALUES␣(?,␣?,␣?)",
        (user.username, user.email, hashed)
    )
```

Explanation

Registration hashes the password and inserts the new user into SQLite. If username/email already exists, FastAPI raises an HTTP 400 error.

Code Snippet: Login Endpoint

```
@router.post("/login")
def login(user: UserLogin):
    hashed = hash_password(user.password)
    cursor.execute(
        "SELECT␣username␣FROM␣users␣WHERE␣username␣=␣?␣AND␣
           password_hash␣=␣?",
        (user.username, hashed)
    )
    token = str(uuid.uuid4())
    sessions[token] = user.username
```

Explanation

Login verifies the username and hashed password. On success, a UUID token is generated and stored in `sessions`. This token is later used to validate WebSocket connections.

Code Snippet: List Users Endpoint

```
@router.get("/users", response_model=List[UserResponse])
def list_users():
    cursor.execute("SELECT␣id,␣username,␣email␣FROM␣users")
```

Explanation

This endpoint returns the list of registered users, used by the admin dashboard.

Real-Time Chat Service (websocket_routes.py)

Code Snippet: Token Validation

```
@router.websocket("/ws")
async def websocket_endpoint(ws: WebSocket):
    token = ws.query_params.get("token")
    if token not in sessions:
        await ws.close()
        return
    username = sessions[token]
```

Explanation

The WebSocket endpoint requires a valid session token. If the token is invalid, the connection is closed to prevent unauthorized access.

Code Snippet: Connection Manager

```
class Manager:
    def __init__(self):
        self.connections: list[WebSocket] = []

    async def connect(self, ws: WebSocket, username: str):
        await ws.accept()
        self.connections.append(ws)

    async def broadcast(self, data: dict):
        for c in list(self.connections):
            await c.send_json(data)
```

Explanation

The Manager maintains all active WebSocket connections and supports broadcasting JSON events to all clients.

Code Snippet: Sending Chat History

```
cursor.execute("SELECT␣id,␣username,␣message,␣timestamp,␣seen␣
   FROM␣messages")
for row in cursor.fetchall():
    await ws.send_json({
        "type": "message",
        "id": row[0],
        "user": row[1],
        "text": row[2],
        "time": row[3],
        "seen": bool(row[4])
    })
```

Explanation

When a user connects, the backend fetches message history from SQLite and sends it to the client, allowing chat persistence.

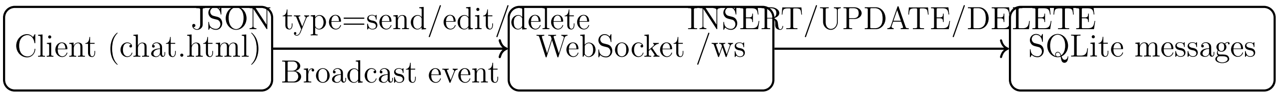
Code Snippet: Send/Edit/Delete Message Types

```
if msg_type == "send":
    cursor.execute(
        "INSERT␣INTO␣messages(username,␣message,␣timestamp)␣
           VALUES␣(?,␣?,␣?)",
        (username, text, time)
    )
    await manager.broadcast({"type":"message", ...})

elif msg_type == "edit":
    cursor.execute("UPDATE␣messages␣SET␣message=?␣WHERE␣id=?",
                   (data["text"], data["id"]))
    await manager.broadcast({"type":"edit", "id": data["id"], "
       text": data["text"]})

elif msg_type == "delete":
    cursor.execute("DELETE␣FROM␣messages␣WHERE␣id=?", (data["id"
       ],))
    await manager.broadcast({"type":"delete", "id": data["id"]})
```

Explanation

Clients send JSON with a `type` field: `send` stores and broadcasts a message, `edit` updates a message and broadcasts the change, `delete` removes a message and broadcasts the deletion event.

WebSocket Event Flow Diagram



Frontend: Login (login.html)

Code Snippet

```
const res = await fetch("http://127.0.0.1:8000/login", {
  method: "POST",
  headers: {"Content-Type":"application/json"},
  body: JSON.stringify({username, password})
});

if (res.ok) {
  localStorage.setItem("username", data.username);
  localStorage.setItem("token", data.token);
  window.location.href = "/ui/chat.html";
}
```

Explanation

The login page sends an HTTP POST request to `/login`. On success, the token and username are stored in localStorage and the user is redirected to chat.

Frontend: Register (register.html)

Code Snippet

```
const res = await fetch("http://127.0.0.1:8000/register", {
  method: "POST",
  headers: {"Content-Type":"application/json"},
  body: JSON.stringify({username, email, password})
});
```

Explanation

The register page sends an HTTP POST request to `/register` to create a new user account.

Frontend: Chat Client (chat.html)

Code Snippet: WebSocket Connection

```
const ws = new WebSocket("ws://127.0.0.1:8000/ws?token=" + token)
   ;
```

Explanation

The chat page creates a WebSocket connection to `/ws` using the token for authentication.

Code Snippet: Sending Events

```
ws.send(JSON.stringify({ type: "send", text: msg }));
ws.send(JSON.stringify({ type: "edit", id: id, text: trimmed }));
ws.send(JSON.stringify({ type: "delete", id: id }));
```

Explanation

All operations are sent as JSON messages with a `type`. The backend processes these types and broadcasts corresponding events.

Code Snippet: Receiving Events

```
ws.onmessage = function(event) {
  const data = JSON.parse(event.data);
  if (data.type === "message") renderMessage(data);
  else if (data.type === "users") updateOnlineUsers(data.users);
  else if (data.type === "edit") applyEdit(data);
  else if (data.type === "delete") applyDelete(data.id);
};
```

Explanation

The frontend listens for server events and updates the UI based on event type.

Frontend: Admin Dashboard (admin.html)

Code Snippet: Fetch Registered Users

```
const res = await fetch("http://127.0.0.1:8000/users");
const data = await res.json();
```

Explanation

The admin dashboard loads the registered users from `/users` and displays them.

Code Snippet: WebSocket Online Users

```
const ws = new WebSocket("ws://127.0.0.1:8000/ws?token=" + token)
    ;

ws.onmessage = function(event) {
  const data = JSON.parse(event.data);
  if (data.type === "users") updateOnlineList(data.users);
};
```

Explanation

The admin dashboard connects to the same WebSocket endpoint and receives `users` events to show who is online.

Conclusion

ChatterBox integrates REST APIs and WebSockets to provide authentication, persistent chat history, and real-time message updates. The modular structure improves maintainability and supports future enhancements.