Product Requirements Document

Real-Time Chat Application Rebuild

Version:	1.0
Last Updated:	November 2025
Status:	Development

1. Product Overview

1.1 Purpose

Rebuild a real-time chat application with modern architecture patterns, focusing on clean code, maintainability, and security.

1.2 Goals

- **Clean Architecture:** Separation of concerns, single responsibility
- **Maintainability:** Easy to understand, modify, and extend
- **Security:** Industry-standard authentication and data protection
- **Real-time Communication:** WebSocket-based instant messaging
- **Media Sharing:** Support for images and videos

1.3 Technology Stack

Component	Technology
Backend	Pure Python (no frameworks)
Database	MongoDB
Web Server	Nginx (reverse proxy, SSL, static files)
Frontend	Vanilla JavaScript
Deployment	Docker + Docker Compose

2. Component Breakdown

2.1 HTTP Core Layer

2.1.1 Request Parser

Purpose: Parse raw HTTP requests into structured objects

Requirements:

✓ Parse request line (method, path, HTTP version)
✓ Extract headers (case-insensitive lookup)
✓ Parse cookies from Cookie header
✓ Parse query parameters from URL
✓ Handle request body (raw bytes)
✓ Support JSON body parsing
✓ Support form-urlencoded parsing

2.1.2 Response Builder

Purpose: Construct HTTP responses programmatically

Requirements:

✓ Set status code and message
✓ Add/modify headers
✓ Set cookies with attributes (HttpOnly, Secure, Max-Age)
✓ Delete cookies (set expiration to past)
✓ Auto-set Content-Length
✓ Support text, HTML, JSON, and binary content
✓ Factory methods for common responses (redirect, error codes)
✓ Serialize to raw bytes for socket transmission

2.1.3 Router

Purpose: Map HTTP requests to handler functions

Requirements:

✓ Route registration via decorators (@router.get, @router.post)
✓ Support path parameters (/messages/{id})
✓ Extract path parameters and inject into request
✓ Method-based routing (GET, POST, PUT, DELETE)
✓ Route matching with regex patterns
✓ Middleware support (pre/post processing)

✓ 404 handling for unmatched routes

Example Usage:

```
@router.get('/messages/{id}')
def get_message(request):
message_id = request.path_params['id']
```

2.2 Authentication System

2.2.1 User Registration

Requirements:

- ✓ Accept username and password
- ✓ Validate password strength (min 8 chars, uppercase, lowercase, number, special char)
- ✓ Generate random salt (bcrypt)
- ✓ Hash password with salt
- ✓ Store username, salt, and hash in database
- ✓ HTML-escape username
- ✓ Reject duplicate usernames

Endpoint: POST /register

2.2.2 User Login

Requirements:

- ✓ Accept username and password
- ✓ Lookup user by username
- ✓ Verify password against stored hash
- ✓ Generate unique auth token (UUID)
- ✓ Hash auth token (SHA-256) and store in database
- ✓ Set secure cookies (auth_token: HttpOnly, Secure, 1 hour; auth: JavaScript-accessible)
- ✓ Associate token with username

Endpoint: POST /login

2.2.3 User Logout

Requirements:

- ✓ Retrieve auth token from cookie
- ✓ Delete token from database
- ✓ Expire cookies (set expiration to past)
- ✓ Redirect to home page

Endpoint: POST /logout

2.3 Chat System

2.3.1 Message Storage

Database Schema:

```
{
id: 'uuid',
username: 'string',
message: 'string' // HTML-escaped
}
```

2.3.2 REST API Endpoints

Method	Endpoint	Auth	Description
GET	/chat-messages	No	Retrieve all messages as JSON array
POST	/chat-messages	Optional	Create new message (requires XSRF token if auth)
DELETE	/chat-messages/{id}	Yes	Delete own message (403 if not owner)

2.3.3 XSRF Protection

Requirements:

✓ Generate unique token per authenticated user
✓ Store token in database (linked to username)
✓ Inject token into HTML (hidden input)
✓ Validate token on state-changing requests
✓ Reject requests with missing/invalid token (403)
✓ Skip validation for unauthenticated users

2.4 WebSocket System

2.4.1 Connection Management

Requirements:

✓ Handle WebSocket upgrade handshake
✓ Compute Sec-WebSocket-Accept key
✓ Maintain list of active connections
✓ Track authenticated vs guest users
✓ Clean up closed connections
✓ Support concurrent connections

2.4.2 Frame Parsing

Requirements:

✓ Parse frame headers (FIN, opcode, mask, length)
✓ Handle extended payload lengths (16-bit, 64-bit)
✓ Unmask payload using masking key
✓ Support fragmented messages (FIN=0)
✓ Buffer incomplete frames
✓ Handle multiple frames in single recv()
✓ Support text (opcode 1) and close (opcode 8) frames

2.4.3 Message Broadcasting

Message Types:

- **Chat Message:** {messageType: 'chatMessage', username: '...', message: '...', id: '...'}
- **Online Users:** {messageType: 'onlineUsers', list: [...]}
- **WebRTC Signals:** {messageType: 'webRTC-offer|answer|candidate', ...}

2.5 File Upload System

2.5.1 Multipart Parser

Requirements:

✓ Parse multipart/form-data body
✓ Extract boundary from Content-Type header
✓ Split body into parts
✓ Parse part headers (Content-Disposition, Content-Type)
✓ Extract field name and filename
✓ Return list of parts with headers and content

2.5.2 File Upload Handler

Requirements:

✓ Accept file via POST /form-path
✓ Detect file type from binary signature (not extension)
✓ Generate unique filename (UUID + extension)
✓ Save to public/user-content/ directory
✓ Create chat message with embedded media
✓ Associate with authenticated user
✓ Set file size limit (100MB via nginx)

Supported File Types (Binary Signatures):

- **JPG:** FF D8
- **PNG:** 89 50
- **GIF:** 47 49
- **MP4:** 00 00

3. Security Requirements

3.1 Authentication Security

✓ Never store passwords in plaintext
✓ Use bcrypt for password hashing (with salt)
✓ Hash auth tokens before database storage (SHA-256)
✓ Set HttpOnly flag on auth cookies
✓ Set Secure flag on auth cookies (HTTPS only)
✓ Token expiration (1 hour)

3.2 Input Validation

✓ HTML-escape all user input before display
✓ Validate file types by binary signature
✓ Validate password strength
✓ Sanitize filenames

3.3 XSRF Protection

✓ Generate unique token per user session
✓ Validate token on state-changing requests
✓ Reject requests with invalid tokens

3.4 Required Headers

✓ X-Content-Type-Options: nosniff
✓ Content-Type with charset
✓ Content-Length for all responses

4. Database Schema

MongoDB Collections:

users

```
{ username: String (unique), salt: Binary, hash: Binary }
```

tokens

```
{ username: String, hash: String, access_token: String }
```

chat

```
{ id: String (UUID), username: String, message: String }
```

xsrf_tokens

```
{ username: String (unique), xsrf_token: String (UUID) }
```

5. API Specification Summary

Method	Endpoint	Auth	Purpose
GET	/	No	Render home page
POST	/register	No	Create new user
POST	/login	No	Authenticate user
POST	/logout	Yes	End session
POST	/spotify-login	No	Initiate OAuth
GET	/spotify	No	OAuth callback
GET	/chat-messages	No	Retrieve all messages
POST	/chat-messages	Optional	Create message
DELETE	/chat-messages/{id}	Yes	Delete own message
POST	/form-path	Optional	Upload file
GET	/websocket	Optional	Upgrade to WebSocket

6. Success Criteria

6.1 Functional

✓ Users can register and login
✓ Users can send/receive chat messages in real-time
✓ Users can upload images and videos
✓ Users can delete their own messages
✓ OAuth login works with Spotify
✓ Video chat connects between two peers

6.2 Code Quality

✓ Single responsibility per class
✓ No function >50 lines
✓ No file >500 lines
✓ Clear separation of concerns
✓ Type hints where applicable
✓ Docstrings for public methods

6.3 Security

✓ No XSS vulnerabilities
✓ No SQL injection (MongoDB)
✓ XSRF protection on state-changing requests
✓ Secure password storage
✓ Secure cookie handling

6.4 Maintainability

✓ Easy to add new routes
✓ Easy to add new middleware
✓ Easy to modify business logic
✓ Clear error messages
✓ Centralized configuration

7. Development Phases

Phase	Timeline	Deliverables
Phase 1: Foundation	Week 1	HTTP core (Request, Response, Router) Basic routing, Static files, Database connection
Phase 2: Authentication	Week 1	User registration, Login/logout Auth middleware, Password validation, XSRF tokens
Phase 3: Chat API	Week 2	REST endpoints, Message CRUD HTML escaping, Database operations
Phase 4: Real-Time	Week 2	WebSocket handshake, Frame parsing Message broadcasting, Online users tracking
Phase 5: Media	Week 3	Multipart parsing, File upload File type detection, Media embedding
Phase 6: OAuth & WebRTC	Week 3	Spotify OAuth, WebRTC signaling Integration testing, Bug fixes

8. Out of Scope

The following features are explicitly excluded from this version:

- User profiles
- Message search
- User blocking
- Typing indicators
- File compression
- Rate limiting
- Message pagination
- Message editing
- Private messaging
- Message reactions
- Read receipts
- Image thumbnails
- Admin panel

9. Testing Strategy

9.1 Unit Tests

- Request parsing edge cases
- Response serialization
- Route matching
- WebSocket frame parsing
- Password validation
- HTML escaping

9.2 Integration Tests

- Full request/response cycle
- Authentication flows
- Chat operations
- File upload
- WebSocket connections

9.3 Manual Testing

- Browser compatibility
- WebRTC video chat
- Concurrent users
- File uploads (various types)
- OAuth flow

10. Deployment

10.1 Docker Compose Services

Service	Purpose	Details
nginx	Reverse proxy, SSL, static files	Ports: 80, 443 Handles WebSocket upgrade
server	Python application	Port: 8080 Waits for MongoDB
mongo	Database	MongoDB 4.2.5 No exposed ports

10.2 Environment Variables

```
CLIENT_ID=<spotify-client-id>
CLIENT_SECRET=<spotify-client-secret>
redirect_uri=http://localhost:8080/spotify
```

10.3 Deployment Commands

```
docker-compose up --build # Start all services
docker-compose down # Stop all services
```

End of Document

This PRD provides a comprehensive blueprint for rebuilding the chat application with clean architecture and modern design patterns.