# **Comprehensive Documentation for Connection Logic System**

## **Overview**

This documentation covers the connection logic system that handles user authentication, WebSocket communication, and session management. The system consists of several interconnected modules that provide secure and scalable connection handling.

## **Core Components**

### **1. ConnectionAPI**

The central hub that manages:

* Database connections (PostgreSQL)
* Redis caching
* JWT token management
* Session handling
* Route registration

### **2. LoginModule**

Handles user authentication flows:

* Registration
* Login/logout
* Token refresh
* User deletion

### **3. WebSocketModule**

Manages real-time communication:

* Room-based messaging
* Presence tracking
* Connection lifecycle
* Permission management

### **4. WebSocketManager**

The engine behind WebSocket operations:

* Connection handling
* Room management
* Message broadcasting
* Rate limiting

### **5. JWTManager**

Handles JSON Web Token operations:

* Token creation/validation
* Token revocation
* Token refresh

### **6. RedisManager**

Provides secure Redis operations:

* Data encryption
* Connection pooling
* Room size tracking
* Presence management

## **Security Features**

1. **Authentication**:
   * JWT tokens with expiration
   * Token revocation capability
   * Refresh token rotation
   * Secure password hashing (bcrypt)
2. **Authorization**:
   * Room permission system (public/private/restricted/owner-only)
   * Role-based access control
   * Token validation for WebSocket connections
3. **Data Protection**:
   * Redis data encryption
   * Secure secret management
   * Query parameter sanitization
4. **Rate Limiting**:
   * Connection rate limits
   * Message rate limits
   * Per-client tracking
5. **Validation**:
   * Input validation for all endpoints
   * Message content sanitization
   * Room ID validation

## **Detailed Component Breakdown**

### **ConnectionAPI**

**Key Responsibilities**:

* Manages database connection pooling with retry logic
* Provides Redis caching with encryption
* Handles JWT token creation/validation
* Manages user sessions
* Provides route registration for Flask

**Notable Features**:

* Connection health checking
* Automatic query caching
* Session timeout handling
* Concurrent session limits

### **LoginModule**

**Authentication Flow**:

1. User provides credentials (email/password)
2. System verifies credentials against database
3. On success:
   * Creates access/refresh tokens
   * Caches user data in Redis
   * Returns tokens to client

**Security Measures**:

* Password hashing with bcrypt
* Token expiration enforcement
* Refresh token rotation
* Session tracking

### **WebSocketModule**

**Connection Flow**:

1. Client connects with valid JWT token
2. Server validates token and origin
3. On success:
   * Creates WebSocket session
   * Joins default rooms
   * Tracks presence

**Room Management**:

* Four permission levels
* Owner-based access control
* Room size limits
* Automatic cleanup

### **WebSocketManager**

**Core Operations**:

* Message validation/sanitization
* Room membership tracking
* Presence updates
* Rate limit enforcement
* Payload compression

**Security Features**:

* Origin validation
* Message size limits
* JSON depth/size validation
* Binary data validation

## **Process Examples**

### **1. User Registration Process**

**Flow**:

1. Client POSTs to /register with:
2. json
3. Copy

{

"username": "newuser",

"email": "user@example.com",

"password": "SecurePass123!"

1. }
2. Server:
   * Validates all fields are present
   * Checks email uniqueness
   * Hashes password with bcrypt
   * Creates user record in database
   * Returns success response

**Security Aspects**:

* Password never stored in plaintext
* Email uniqueness check prevents duplicate accounts
* Input validation prevents injection
* HTTPS required for transport

**Example Code Path**:

1. LoginModule.register\_user()
2. ConnectionAPI.create\_user()
3. LoginModule.hash\_password()
4. Database insertion with parameterized queries

### **2. User Login Process**

**Flow**:

1. Client POSTs to /login with:
2. json
3. Copy

{

"email": "user@example.com",

"password": "SecurePass123!"

1. }
2. Server:
   * Validates credentials
   * Creates access/refresh tokens
   * Caches user data in Redis
   * Returns tokens and user info
3. Client stores tokens securely

**Security Aspects**:

* Brute force protection via rate limiting
* Short-lived access tokens
* Long-lived refresh tokens (rotated on use)
* Token revocation capability
* Redis cache encrypted

**Example Code Path**:

1. LoginModule.login\_user()
2. ConnectionAPI.get\_user\_by\_email()
3. LoginModule.check\_password()
4. ConnectionAPI.create\_user\_tokens()
5. ConnectionAPI.cache\_user\_data()

### **3. WebSocket Connection & Messaging**

**Connection Flow**:

1. Client connects with token in query string:
2. Copy
3. ws://server/socket.io?token=ACCESS\_TOKEN
4. Server:
   * Validates token
   * Creates WebSocket session
   * Joins default rooms
   * Starts presence tracking

**Message Flow**:

1. Client emits message:
2. javascript
3. Copy

socket.emit('message', {

room\_id: "chat\_room",

message: "Hello world"

1. });
2. Server:
   * Validates message
   * Checks room permissions
   * Broadcasts to room members
   * Updates rate limits

**Security Aspects**:

* Token required for connection
* Origin validation
* Message content sanitization
* Room permission checks
* Rate limiting
* Payload size limits

**Example Code Path**:

1. WebSocketModule.\_handle\_connect()
2. JWTManager.verify\_token()
3. WebSocketManager.join\_room()
4. WebSocketManager.validate\_message()
5. WebSocketManager.broadcast\_to\_room()

## **Detailed Sequence Diagrams**

### **Registration Sequence**

Copy

Client -> LoginModule: POST /register (credentials)

LoginModule -> ConnectionAPI: Check email uniqueness

ConnectionAPI -> Database: SELECT user by email

Database -> ConnectionAPI: No results

LoginModule -> ConnectionAPI: Create user

ConnectionAPI -> Database: INSERT user (hashed password)

LoginModule -> Client: 200 OK

### **Login Sequence**

Copy

Client -> LoginModule: POST /login (credentials)

LoginModule -> ConnectionAPI: Get user by email

ConnectionAPI -> Database: SELECT user

Database -> ConnectionAPI: User record

LoginModule: Verify password hash

LoginModule -> ConnectionAPI: Create tokens

ConnectionAPI -> JWTManager: Generate tokens

JWTManager -> Redis: Store active tokens

ConnectionAPI -> Redis: Cache user data

LoginModule -> Client: Return tokens + user info

### **WebSocket Connection Sequence**

Copy

Client -> WebSocketManager: WS connect (with token)

WebSocketManager -> JWTManager: Verify token

JWTManager -> Redis: Check token validity

WebSocketManager: Validate origin

WebSocketManager: Check rate limits

WebSocketManager -> Redis: Store session data

WebSocketManager: Join default rooms

WebSocketManager -> Redis: Update presence

WebSocketManager -> Client: Connection acknowledged

## **Best Practices**

1. **Token Handling**:
   * Store refresh tokens securely (HttpOnly, Secure cookies)
   * Implement token rotation
   * Keep access token lifetime short (1 hour or less)
2. **WebSocket Security**:
   * Always validate origin headers
   * Enforce message size limits
   * Implement rate limiting
   * Use WSS (WebSocket Secure)
3. **Database Security**:
   * Always use parameterized queries
   * Limit connection pool size
   * Implement statement timeouts
4. **Error Handling**:
   * Use generic error messages for auth failures
   * Log detailed errors server-side only
   * Implement proper HTTP status codes

## **Monitoring Considerations**

1. Track:
   * Failed login attempts
   * Token validation failures
   * WebSocket connection drops
   * Room size violations
   * Rate limit hits
2. Metrics to monitor:
   * Concurrent connections
   * Message throughput
   * Authentication latency
   * Redis cache hit ratio