🏗️ PeerFusion - Detailed Project Architecture

📋 Table of Contents

🎯 Architecture Overview

PeerFusion is a full-stack **MERN** (MongoDB, Express, React, Node.js) application designed as a professional networking platform for developers - essentially "Tinder for Developers". The application follows a **client-server architecture** with real-time communication capabilities.

Key Architecture Patterns:

- **Monorepo Structure** - Backend and Frontend in separate folders
- **REST API** - For standard CRUD operations
- **WebSocket (Socket.io)** - For real-time chat functionality
- **JWT Authentication** - Token-based stateless authentication
- **Cookie-based Sessions** - Secure session management
- **MVC Pattern** - Model-View-Controller on backend
- **Context API Pattern** - State management on frontend

🛠️ Technology Stack

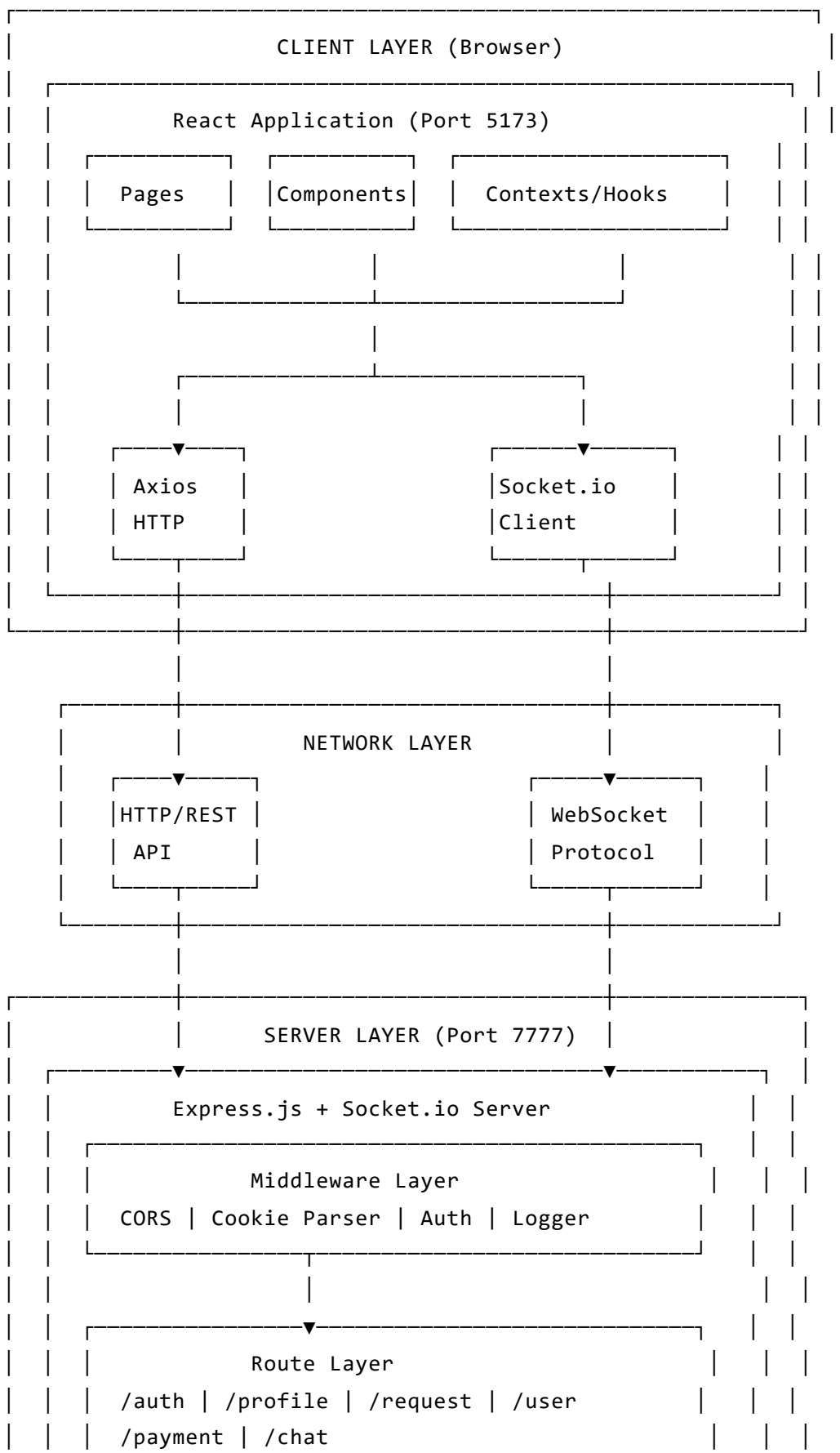
Backend Technologies

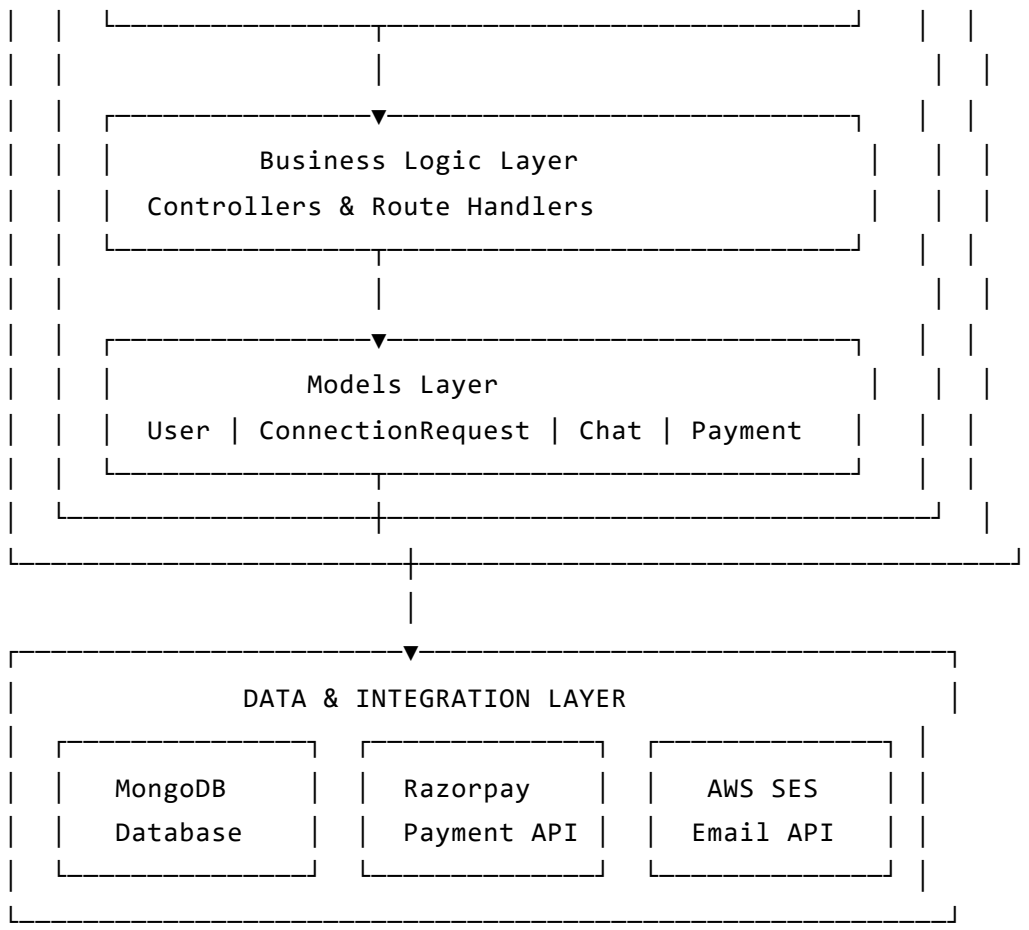
```
Node.js v16+
├── Express.js v4.19.2          → Web framework
├── MongoDB (Mongoose v8.6.1)   → Database & ODM
├── Socket.io v4.8.1            → Real-time bidirectional communication
├── JWT (jsonwebtoken v9.0.2)   → Authentication tokens
├── bcrypt v5.1.1               → Password hashing
├── Razorpay v2.9.5             → Payment gateway
├── AWS SES Client v3.716.0     → Email service
├── Node-cron v3.0.3            → Scheduled tasks
├── Validator v13.12.0          → Data validation
├── CORS v2.8.5                 → Cross-origin resource sharing
├── Cookie-parser v1.4.6        → Cookie handling
└── Date-fns v4.1.0             → Date utilities
```

Frontend Technologies

```
React v19.1.0
├── React Router DOM v7.7.1     → Client-side routing
├── Vite v7.0.4                 → Build tool & dev server
├── Axios v1.11.0               → HTTP client
├── Socket.io-client v4.8.1     → WebSocket client
├── Lucide-react v0.535.0       → Icon library
└── ESLint v9.30.1              → Code linting
```

🏛️ System Architecture




Business Logic Layer
Controllers & Route Handlers
Models Layer
User | ConnectionRequest | Chat | Payment
DATA & INTEGRATION LAYER
MongoDB Database
Razorpay Payment API
AWS SES Email API

🔧 Backend Architecture

Directory Structure

```
Backend/
├── app.js                    → Entry point & server configuration
├── config/
│   └── database.js           → MongoDB connection setup
├── middlewares/
│   └── auth.js               → JWT authentication middleware
├── models/
│   ├── user.js               → User schema & methods
│   ├── connectionRequest.js  → Connection request schema
│   ├── chat.js               → Chat & message schemas
│   └── payment.js            → Payment transaction schema
├── routes/
│   ├── auth.js               → Authentication endpoints
│   ├── profile.js            → User profile management
│   ├── request.js            → Connection request handling
│   ├── user.js               → User discovery & search
│   ├── chat.js               → Chat history endpoints
│   └── payment.js            → Payment processing
└── utils/
    ├── constants.js          → App constants
    ├── cronjob.js            → Scheduled tasks (membership expiry)
    ├── razorpay.js           → Razorpay integration
    ├── sendEmail.js          → Email service wrapper
    ├── sesClient.js          → AWS SES configuration
    ├── socket.js             → Socket.io setup & handlers
    └── validation.js         → Request validation functions
```

Key Backend Components

1. Server Setup (app.js)

- Express server with HTTP server wrapper for Socket.io
- CORS configuration for multiple frontend ports
- Cookie-based authentication
- Centralized error handling
- Database connection management
- Socket.io initialization

2. Middleware Pipeline

```
Request → CORS → JSON Parser → Cookie Parser → Logger → Auth Middleware → Route Handler
```

3. Authentication Flow

```
// JWT Token Generation
User.methods.getJWT = function() {
  return jwt.sign({ _id: this._id }, JWT_SECRET, { expiresIn: "7d" });
}

// Middleware Verification
1. Extract token from cookies
2. Verify JWT signature
3. Fetch user from database
4. Attach user to req.user
5. Call next() or return 401
```

🎨 Frontend Architecture

Directory Structure

```
Frontend/src/
├── main.jsx                 → App entry point
├── App.jsx                  → Root component & routing
├── App.css                  → Global styles
├── components/
│   ├── Navbar.jsx           → Navigation component
│   ├── Navbar.css
│   └── ProtectedRoute.jsx   → Route protection wrapper
├── contexts/
│   ├── AuthContext.jsx      → Auth state provider
│   └── AuthContextDefinition.js → Context definition
├── hooks/
│   └── useAuth.js           → Auth custom hook
├── pages/
│   ├── Landing.jsx          → Public landing page
│   ├── Login.jsx            → Login page
│   ├── Signup.jsx           → Registration page
│   ├── Dashboard.jsx        → User dashboard
│   ├── Feed.jsx             → Swipe-based discovery
│   ├── Connections.jsx      → Connection management
│   ├── Chat.jsx             → Real-time messaging
│   ├── Profile.jsx          → Profile management
│   └── Premium.jsx          → Subscription page
├── services/
│   ├── api.js               → Axios API service
│   └── socket.js            → Socket.io client setup
└── utils/
    └── helpers.js           → Utility functions
```

Component Architecture

```
App (Router Provider)
├── AuthProvider (Context)
│   ├── AppContent
│   │   ├── Navbar (if authenticated)
│   │   └── Routes
│   │       ├── Public Routes
│   │       │   ├── Landing
│   │       │   ├── Login
│   │       │   └── Signup
│   │       └── Protected Routes (ProtectedRoute wrapper)
│   │           ├── Dashboard
│   │           ├── Feed
│   │           ├── Connections
│   │           ├── Chat
│   │           ├── Profile
│   │           └── Premium
```

State Management

```
AuthContext (Global State)
├── user          → Current authenticated user
├── loading       → Authentication check status
├── login()       → Login function
├── signup()      → Signup function
├── logout()      → Logout function
└── updateUser()  → Update user profile
```

💾 Database Schema

1. User Model

```
{
  _id: ObjectId,
  firstName: String (required, 4-50 chars),
  lastName: String,
  emailId: String (required, unique, validated),
  password: String (required, strong password, hashed),
  age: Number (min: 18),
  gender: Enum ['male', 'female', 'other'],
  isPremium: Boolean (default: false),
  membershipType: String,
  photoUrl: String (validated URL),
  about: String,
  skills: [String],
  timestamps: { createdAt, updatedAt }
}
```

2. ConnectionRequest Model

```
{
  _id: ObjectId,
  fromUserId: ObjectId (ref: User, required),
  toUserId: ObjectId (ref: User, required),
  status: Enum ['ignored', 'interested', 'accepted', 'rejected'],
  timestamps: { createdAt, updatedAt },

  // Indexes
  index: { fromUserId: 1, toUserId: 1 }
}
```

3. Chat Model

```
{
  _id: ObjectId,
  participants: [ObjectId] (ref: User, required),
  messages: [
    {
      _id: ObjectId,
      senderId: ObjectId (ref: User, required),
      text: String (required),
      timestamps: { createdAt, updatedAt }
    }
  ]
}
```

4. Payment Model

```
{
  _id: ObjectId,
  userId: ObjectId (ref: User, required),
  paymentId: String,
  orderId: String (required),
  status: String (required),
  amount: Number (required),
  currency: String (required),
  receipt: String (required),
  notes: {
    firstName: String,
    lastName: String,
    membershipType: String
  },
  timestamps: { createdAt, updatedAt }
}
```

🔌 API Architecture

Authentication Endpoints

```
POST   /signup              → Register new user
POST   /login               → User login
POST   /logout              → User logout
```

Profile Endpoints

```
GET    /profile/view        → Get current user profile
PATCH  /profile/edit        → Update user profile
PATCH  /profile/password    → Change password
```

Connection Request Endpoints

```
POST   /request/send/:status/:toUserId  → Send connection request
POST   /request/review/:status/:requestId → Review incoming request
GET    /request/received    → Get received requests
```

User Discovery Endpoints

```
GET    /user/feed           → Get users for discovery (not interacted)
GET    /user/connections    → Get accepted connections
```

Chat Endpoints

```
GET    /chat/:targetUserId  → Get chat history with user
```

Payment Endpoints

```
POST   /payment/create-order → Create Razorpay order
POST   /payment/verify       → Verify payment signature
GET    /payment/history      → Get payment history
```

💬 Real-time Communication

Socket.io Architecture

```
Server-side (utils/socket.js)
├── initializeSocket(server)
│   ├── CORS configuration
│   └── Event Handlers:
│       ├── 'connection'     → New client connected
│       ├── 'joinChat'       → User joins room
│       │   └── Creates secret room ID using SHA256 hash
│       ├── 'sendMessage'    → Send/broadcast message
│       │   ├── Validate connection exists
│       │   ├── Save message to MongoDB
│       │   └── Emit to all room participants
│       └── 'disconnect'     → Client disconnected
```

Room Management

```
// Secret Room ID Generation
getSecretRoomId(userId, targetUserId) {
  return SHA256([userId, targetUserId].sort().join('$'))
}

// Ensures same room for both users regardless of who initiates
```

Message Flow

1. User A opens chat with User B
2. Client emits 'joinChat' with userId & targetUserId
3. Server calculates room ID and adds client to room
4. User A sends message
5. Server validates connection, saves to DB
6. Server broadcasts to room (both users receive)
7. Both clients update UI in real-time

🔐 Authentication & Security

Authentication Flow

Signup Process

1. User submits signup form
2. Validate input data (email, password strength)
3. Check if user exists
4. Hash password with bcrypt (10 rounds)
5. Create user in database
6. Generate JWT token
7. Set HTTP-only cookie
8. Return user data (without password)

Login Process

1. User submits credentials
2. Find user by email
3. Compare password with bcrypt
4. Generate JWT token
5. Set HTTP-only cookie (8 hours expiry)
6. Return user data

Protected Route Access

```
Client Request → Cookie with JWT
     ↓
Auth Middleware (userAuth)
     ↓
Extract & verify token
     ↓
Fetch user from DB
     ↓
Attach to req.user
     ↓
Route Handler
```

Security Measures

- ✓ JWT tokens (7-day expiry)
- ✓ HTTP-only cookies (prevents XSS)
- ✓ bcrypt password hashing (10 rounds)
- ✓ Strong password validation
- ✓ Email validation
- ✓ CORS configuration (specific origins)
- ✓ Secure cookies in production
- ✓ Request validation on all inputs
- ✓ MongoDB schema validation
- ✓ Pre-save hooks (prevent self-connections)

💳 Payment Integration

Razorpay Flow

```
1. User selects premium plan
   ↓
2. Frontend: POST /payment/create-order
   {
     amount: 30000,  // in paise (₹300)
     membershipType: "silver"
   }
   ↓
3. Backend: Create Razorpay order
   ↓
4. Backend: Save order to Payment model
   ↓
5. Frontend: Open Razorpay checkout modal
   ↓
6. User completes payment on Razorpay
   ↓
7. Razorpay callback with payment details
   ↓
8. Frontend: POST /payment/verify
   {
     razorpay_order_id,
     razorpay_payment_id,
     razorpay_signature
   }
   ↓
9. Backend: Verify signature with Razorpay secret
   ↓
10. Backend: Update payment status
    ↓
11. Backend: Update user (isPremium = true)
    ↓
12. Return success response
```

Membership Tiers

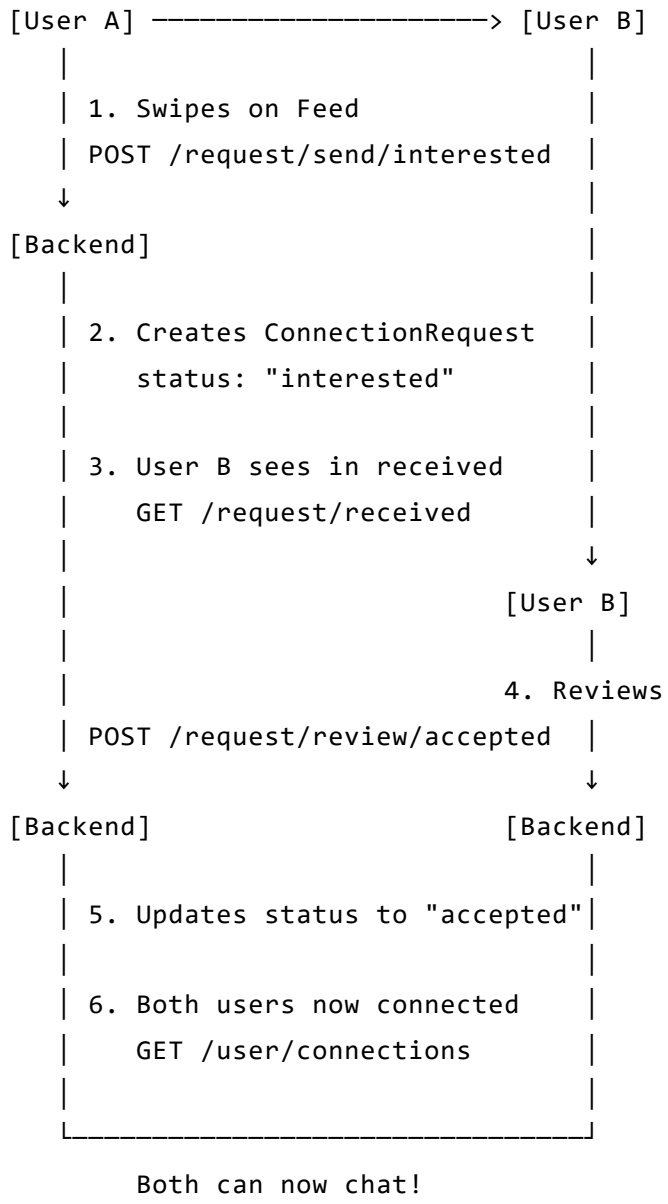
```
Silver: ₹300/month (30000 paise)
Gold:   ₹700/3 months (70000 paise)
```

Cron Job (Membership Expiry)

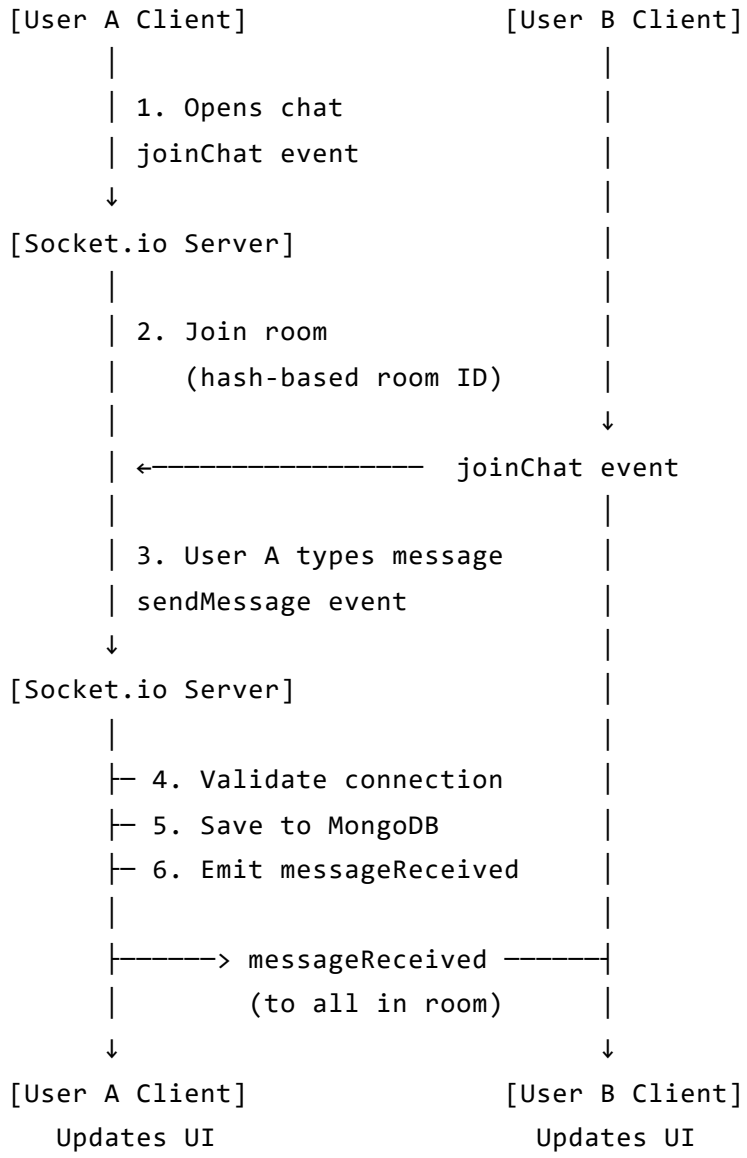
```
// Runs periodically via node-cron
- Check payment records for expired memberships
- Update user.isPremium = false
- Send expiry notification emails via AWS SES
```

🔄 Data Flow Diagrams

Connection Request Flow



Chat Message Flow



🎯 Key Features Implementation

1. Swipe-based Discovery (Feed)

```
Algorithm:
1. Get current user
2. Find all connection requests involving user
3. Extract all interacted user IDs
4. Query users NOT in interacted list
5. Exclude current user
6. Return paginated results

MongoDB Query:
User.find({
  $and: [
    { _id: { $nin: [...interactedUserIds, currentUserId] } }
  ]
}).limit(10)
```

2. Connection Status States

```
ignored    → User swiped left (passed)
interested → User swiped right (pending)
accepted   → Both users interested
rejected   → Receiver declined request
```

3. Premium Feature Gates

```
// Check before premium-only actions
if (!req.user.isPremium) {
  return res.status(403).json({
    error: "Premium membership required",
  });
}
```

🚀 Deployment Architecture

Development Environment

```
Backend:  localhost:7777
Frontend: localhost:5173, 5174, 5176
Database: localhost:27017/peerfusion
```

Production Considerations

✓ Environment variables (.env)
✓ Secure cookies (secure: true)
✓ HTTPS enforcement
✓ Production MongoDB (Atlas)
✓ Static file serving
✓ Rate limiting
✓ Request logging
✓ Error monitoring
✓ Load balancing
✓ CDN for assets

📊 Performance Optimizations

Database Indexes

- ConnectionRequest: { fromUserId: 1, toUserId: 1 }
- User: { emailId: 1 } (unique)
- Chat: Compound index on participants

Query Optimization

- Projection to exclude sensitive fields
- Pagination on feed/connections
- Population for referenced documents
- Lean queries where appropriate

Frontend Optimization

- Vite for fast bundling
- Code splitting with React Router
- Lazy loading of routes
- Context API for minimal re-renders
- Axios interceptors for error handling

🧪 Testing Structure

```
test/
├── scripts/
│   ├── cleanDatabase.js        → Clear all collections
│   ├── seedDatabase.js         → Populate test data
│   ├── seedDummyUsers.js       → Create fake users
│   ├── seedInteractions.js     → Create test connections
│   ├── testChatPersistence.js  → Test message storage
│   ├── testDashboard.js        → Test dashboard data
│   ├── testFeed.js             → Test feed algorithm
│   └── verifyData.js           → Data validation
```

🔑 Environment Variables

```
# Database
DB_CONNECTION_SECRET=mongodb://localhost:27017/peerfusion

# Authentication
JWT_SECRET=PEER@Fusion$790

# Server
PORT=7777
NODE_ENV=development

# Razorpay
RAZORPAY_KEY_ID=your_key_id
RAZORPAY_KEY_SECRET=your_secret

# AWS SES
AWS_REGION=your_region
AWS_ACCESS_KEY_ID=your_access_key
AWS_SECRET_ACCESS_KEY=your_secret_key
```

This is a **comprehensive, production-ready architecture** for a modern networking platform with real-time capabilities, payment integration, and robust security measures. The system is designed to be scalable, maintainable, and follows industry best practices for full-stack development.