Developer Documentation Template

1. Overview

1.1 Project Description

LocalServe is a community service platform designed to connect local service providers (such as plumbers, electricians, tutors, and more) with users in their area. The platform enables users to discover services, make bookings, complete payments securely, and communicate with providers in real time. Service providers can manage their service listings, track bookings, and interact with customers, while admins oversee the platform to ensure smooth operation and quality of services.

1.2 Tech Stack

- **Frontend:** Vue.js, Vite, Pinia, TailwindCSS
- **Backend:** Node.js, Express.js, Java
- **Database:** MongoDB with Mongoose
- **Authentication:** JWT (Access + Refresh)
- **Integrations:** Socket.io, Razorpay, Cloudinary, Nodemailer

1.3 High-Level Architecture

The system follows a client-server architecture. The Vue.js client (web UI) communicates with an API Gateway built on Express (Node.js). The API Gateway exposes REST endpoints (and a Socket.io endpoint) and handles authentication, routing, and business logic. Persistent data is stored in MongoDB (accessed via Mongoose). External integrations—Socket.io for real-time messaging, Razorpay for payments, Cloudinary for media uploads, Twilio for SMS, and Nodemailer for email are connected to the API layer. Admin, provider, and user roles are enforced via middleware (JWT validation and role checks)

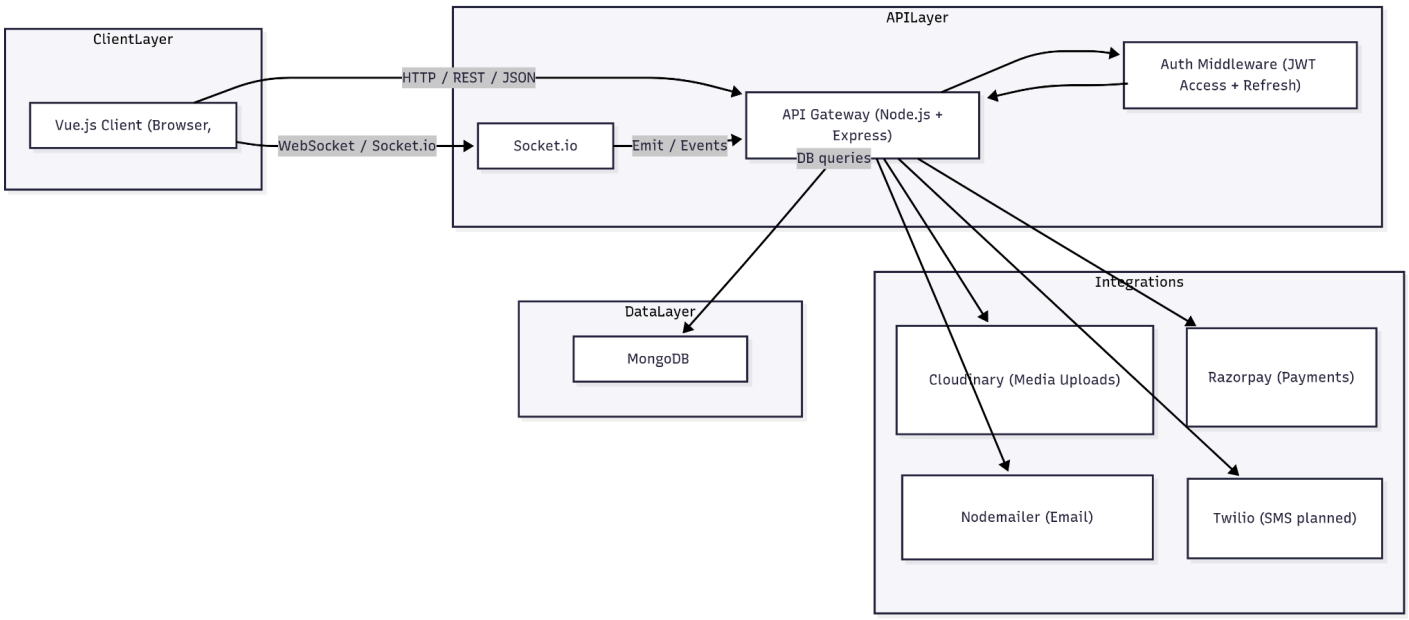

Figure 1.1 – System Architecture

2. Project Structure

2.1 Frontend Structure (Vue.js)

```
frontend/
 ├ src/
 |  ├ assets/          # Images, logos, and static assets
 |  ├ components/      # Reusable UI components
 |  ├ pages/           # Page-level views
 |  ├ store/           # Pinia stores (auth, services, bookings, etc.)
 |  ├ router/          # Vue Router setup and route definitions
 |  ├ utils/           # Helper functions and utilities
 |  ├ App.vue          # Root Vue component
 |  └ main.js          # Entry point for initializing Vue app
 ├ public/       # Public static files (favicon, manifest.json, etc.)
 ├ dist/         # Production build output (generated after build)
 ├ index.html          # Root HTML file for mounting Vue app
 └ package.json        # Project metadata and dependencies
```

2.2 Backend Structure (Node.js + Express)

```
backend/
  ├ src/
  |  ├ config          # Configuration files (DB connection)
  |  ├ models/              # Mongoose schemas (User.js, Booking.js
etc.)
  |  ├ routes/              # Express route definitions
  |  ├ middleware/     # Middleware functions (auth check
  |  ├ controllers/    # Route handlers (business logics)
  |  ├ utils/          # Utility functions (tokens, validators, helpers)
  |  ├ index.js        # App initialization (load routes, middleware)
  |  ├ server.js       # Server entry point
  ├ node_modules       # Installed dependencies
  ├ .env         # Environment variables (DB_URI, JWT_SECRET, API keys)
  └ package.json       # Project metadata, scripts, and dependencies
```

2.3 Key Files

- **server.js (backend)** – Starts the Express server and initializes Socket.io.
- **index.js (backend/src)** – Main backend app file; loads middleware, routes, and DB config.
- **index.js (frontend/src)** – Vue entry file; mounts the Vue app and sets up router + store.
- **api.js (frontend/src/utils)** – Centralized Fetch configuration for API calls.
- **.env (root/backend)** – Stores environment variables like DB_URI, JWT_SECRET, RAZORPAY_KEY etc.
- **package.json (frontend/backend)** – Defines dependencies, scripts, and project metadata.
- **middleware/authMiddleware.js (backend/src/middleware)** – Verifies JWT tokens for protected routes.

3. Installation & Setup

3.1 Prerequisites

- Node.js >= 18.x
- MongoDB >= 6.x (local or cloud e.g., MongoDB Atlas)
- Git
- Postman / Insomnia (for testing backend APIs)
- Modern Browser (Chrome, Firefox, Edge, or Safari for running the frontend)
- VS Code (or any preferred IDE)

3.2 Local Setup Instructions

```
# Clone the repository
git clone <repo-url>
cd project-folder

# Backend setup
cd backend
npm install     # Install backend dependencies
npm run dev     # Start backend server (http://localhost:3000 default)

# Frontend setup
cd frontend
npm install     # Install frontend dependencies
npm run dev     # Start frontend (http://localhost:5173 by default)
```

3.3 Environment Variables

Create a `.env` file inside the `backend/` folder with the following keys:

```
# MongoDB
DB_URI=mongodb+srv://<username>:<password>@cluster.mongodb.net/localse
rve

# JWT
JWT_SECRET=your_jwt_secret_key
JWT_REFRESH_SECRET=your_jwt_refresh_secret_key

# Payments (Razorpay)
RAZORPAY_KEY=your_razorpay_key
RAZORPAY_SECRET=your_razorpay_secret

# Cloudinary
CLOUDINARY_URL=cloudinary://<api_key>:<api_secret>@<cloud_name>

# Email (Nodemailer)
EMAIL_USER=your_email@example.com
EMAIL_PASS=your_email_password
```

Important: Never commit the `.env` file to version control (e.g., GitHub). It should always be kept private and excluded using a `.gitignore` entry.

4. Backend (Node.js + Express)

4.1 API Routes

Route	Method(s)	Description
`/auth`	`POST /login`, `POST /register`, `POST /refresh`	Handles user authentication, registration, and token refresh.
`/services`	`GET /`, `POST /`, `PUT /:id`, `DELETE /:id`	CRUD operations for services (list, add, update, soft-delete).
`/bookings`	`POST /`, `GET /my`, `PUT /:id`, `DELETE /:id`	Create, view, update, or cancel bookings.
`/admin`	`GET /users`, `DELETE /users/:id`, `GET /reports`	Admin-specific endpoints for user management, service monitoring, and reports.

4.2 Middleware

- **auth.js** – Handles both authentication and authorization:
    - **JWT Validation** – Verifies the access token for protected routes.
    - **Role Validation** – Checks the user's role (user, provider, or admin) before allowing access to role-specific endpoints.

4.3 Database Schema (Mongoose Models)

- **User**
    - `name` (String)
    - `email` (String, unique)
    - `password` (Hashed String)
    - `role` (Enum: user, provider, admin)
    - Address (String)
    - `createdAt`, `updatedAt`
- **Service**
    - `name` (String)
    - `category` (String)
    - `price` (Number)
    - `providerId` (ObjectId → User)
    - `description` (String)
    - `tags` (Array of Strings)
    - `available` (Boolean)

- **Booking**
    - `userId` (ObjectId → User)
    - `serviceId` (ObjectId → Service)
    - `status` (Enum: pending, confirmed, cancelled, completed)
    - `date` (Date)
    - `paymentId` (ObjectId → Payment)
    - `createdAt`, `updatedAt`

- **Payment**
    - `bookingId` (ObjectId → Booking)
    - `amount` (Number)
    - `status` (Enum: pending, paid, failed, refunded)
    - userId , providerId (ObjectId → User)
    - `transactionDetails` (Object)
    - `createdAt`

- **Review**
    - `userId` (ObjectId → User)
    - `serviceId` (ObjectId → Service)
    - `rating` (Number, min: 1, max: 5)
    - `comment` (String)

    - `images` (Array of up to 2 image objects → `{ url, public_id }`)
    - `createdAt`, `updatedAt`
    - **Note:** Unique compound index prevents multiple reviews from the same user for the same service.

- **Notification**
    - `userId` (ObjectId → User)
    - `senderId` (ObjectId → User, optional)
    - `title` (String)
    - `body` (String)
    - `type` (String, e.g., `booking.created`, `chat.message`)
    - `createdAt` (Date)

- **Message**
    - `conversationId` (ObjectId → Conversation)
    - `senderId` (ObjectId → User)
    - `text` (String)
    - `attachments` (Array of URLs)
    - `readBy` (Array of User ObjectIds)
    - `createdAt`, `updatedAt`

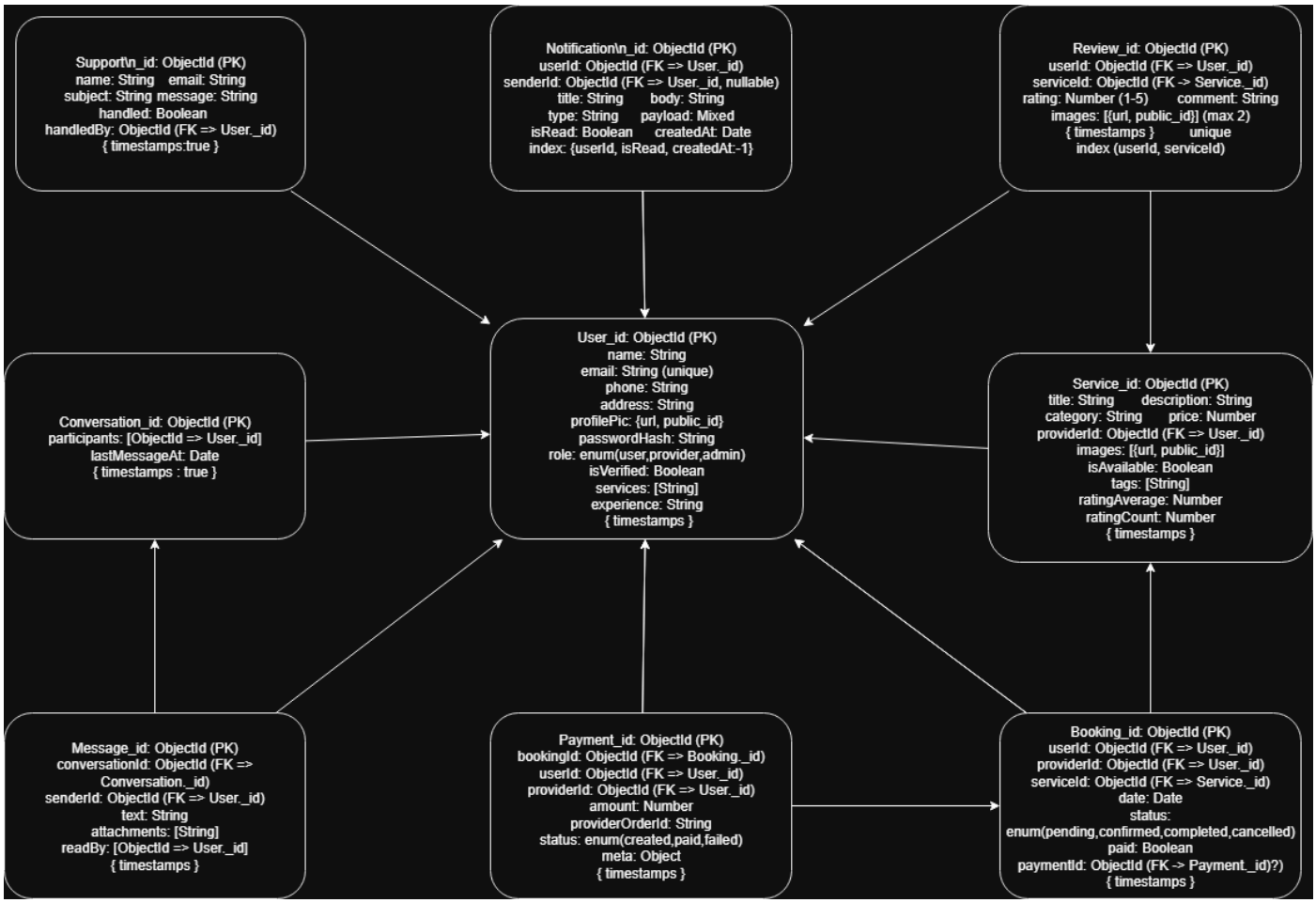

Figure 4.1 - ER Diagram

4.4 Example API Request/Response

Login API

```
POST /api/auth/login
Host: localhost:3000
Content-Type: application/json

{
  "email": "user@example.com",
  “password”: “mypassword”
}
```

Response (201 Created):


```
{
    "user": {
        "id": "68bb2e1988461f95a58c8eca",
        "name": "Shubhendu",
        "email": "shubhendu@test.com",
        "role": "admin"
    },
    "accessToken":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiI2OGJiMmUxOTg4NDYxZjk1YTU4YzhlY
2EiLCJyb2xlIjoiYWRtaW4iLCJlbWFpbCI6InNodWJoZW5kdUB0ZXN0LmNvbSIsImlhdCI6MTc1ODc5
OTc2NiwiZXhwIjoxNzU4ODAwNjY2fQ.Nhu4EN-zaQDs1JRCu2fotvbid8yysgPnaKctZoamMy4"
}
```


Error Example (401 Unauthorized):


```
{

    "error": "Invalid email or password"

}
```


API for Making a Service

Request:

```
POST /api/services HTTP/1.1
Host: localhost:3000
Content-Type: application/json
Authorization: Bearer <jwt_token>

{
  "name": "Plumbing Repair",
  "price": 500,
  "category": "Home Services",
```

```
  "description": "Fixing leaks and pipe issues",
  "tags": ["plumbing", "repair"]
}
```

Response (201 Created):

```
{
  "success": true,
  "serviceId": "64adf12345",
  "service": {
    "name": "Plumbing Repair",
    "price": 500,
    "category": "Home Services",
    "description": "Fixing leaks and pipe issues",
    "tags": ["plumbing", "repair"],
    "providerId": "64abf67890",
    "createdAt": "2025-09-25T10:15:00.123Z"
  }
}
```

Error Example (400 Bad Request):

```
{
  "success": false,
  "message": "Service name is required"
}
```

5. Frontend (Vue.js)

5.1 Routing Setup

The frontend uses *Vue Router* for client-side navigation. Routes are defined inside `src/router/index.js` and mapped to Vue components/pages.

Example Routes:

- `/login` → `Login.vue` (User authentication page)
- `/register` → `Register.vue` (User/Provider registration page)
- `/dashboard` → `Dashboard.vue` (Dashboard view for Providers)
- `/services` → `ServicesList.vue` (Browse available services)
- `/bookings` → `Bookings.vue` (View/manage user bookings)
- `/messages` → ConversationsList.vue (Real-time messaging with providers)
- `/admin` → `AdminDashboard.vue` (Admin management panel)

Route Guards:

- **Auth Guard:** Ensures protected routes are only accessible by logged-in users.
- **Role Guard:** Restricts access to role-based dashboards (`/admin` for admins, `/provider` for service providers, etc.).

5.2 Pinia Stores

The application uses ***Pinia*** for state management. Each store is responsible for a specific domain:

- **authStore.js** – Manages authentication state:
    - Stores JWT access & refresh tokens.
    - Keeps track of the logged-in user.
    - Provides actions for `login`, `logout`, `refreshToken`.
- **bookingStore.js** – Tracks user bookings:
    - Stores current and past bookings.
    - Provides actions to `fetchBookings`, `createBooking`, `cancelBooking`.
- **serviceStore.js** – Handles available services:
    - Stores the list of services fetched from the API.
    - Provides actions to `getServices`, `addService`, `updateService`, `deleteService`.
- **adminStore.js** – Provides state and actions for **admin functionalities**:
    - Stores user and provider lists.
    - Provides actions to `getAllUsers`, `verifyProvider`, `deleteUser`.
    - Tracks site-wide statistics for reports/analytics.

5.3 Components Overview

The frontend is built using reusable Vue components. Each component serves a specific purpose in the user interface:

- **Navbar.vue**
    - Provides navigation links (Home, Services, Bookings, Admin).
    - Displays login/register buttons if not authenticated.
    - Shows profile menu and logout option if logged in.

- **ServiceCard.vue**
    - Displays service details (name, category, price, provider).
    - Includes “Book Now” button for quick booking.
    - Reused across service listing and search pages.

- **BookingForm.vue**
    - Modal/form for users to book a selected service.
    - Collects booking details (date, time, address).
    - Validates input and submits booking requests.

- **ChatBox.vue**
    - Real-time messaging between users and providers.
    - Uses **Socket.io** for live communication.
    - Displays message history + supports new messages.

- **ReviewForm.vue**
    - Allows users to submit reviews after completing a booking.
    - Includes star rating (1–5), comment box, and optional image uploads (up to 2).
    - Sends review data to backend and updates service review list.
- **NotificationBell.vue**
    - Displays unread notifications and opens the notifications panel.
- **AdminDashboard.vue**
    - Displays all the details for the admin.
    - Admin can modify services, bookings and users as needed.

5.4 Socket.io Integration

The system uses **Socket.io** for real-time communication between the client (Vue.js) and backend (Node.js + Express). This enables live updates without needing page refreshes.

Use Cases

- **Real-time Notifications**
    - Users receive live updates about booking status (confirmed, cancelled, completed).
    - Admins and providers receive instant alerts for new bookings or cancellations.
- **User–Provider Chat**
    - Enables direct messaging between users and service providers.
    - Supports text messages and optional file/URL attachments.
    - Messages are persisted in MongoDB (`Message` collection) and delivered in real time via Socket.io.

6. Authentication & Authorization

The system uses **JWT (JSON Web Tokens)** for authentication and **role-based access control** to enforce permissions across different types of users.

JWT Tokens

- **Access Token**
    - Short-lived (e.g., 15–30 minutes).
    - Used in the `Authorization: Bearer <token>` header for API requests.
- **Refresh Token**
    - Long-lived (e.g., 7–30 days).
    - Used to generate new access tokens when the old one expires.
- Tokens are issued on login and stored securely in the client (e.g., HTTP-only cookies or local storage).

Role-Based Access Control (RBAC)

Different roles have different privileges within the system:

- **User (Customer):**
    - Can browse and book services.
    - Can manage their own bookings.
    - Can chat with providers and leave reviews.
- **Provider (Service Provider):**
    - Can add, update, and delete their services.
    - Can view and manage bookings related to their services.
    - Can chat with users and respond to reviews.

- **Admin:**
    - Has full access to the system.
    - Can manage users, providers, services, and bookings.
    - Can view reports, handle disputes, and oversee payments.

7. Integrations

The system integrates with several third-party services to extend its functionality:

- **Socket.io**
    - Provides real-time communication between users, providers, and admins.
    - Used for:
        - Live chat messaging.
        - Real-time booking updates.
        - Push-style notifications inside the app.

- **Razorpay**
    - Handles secure online payments.
    - Supports multiple payment methods (UPI, cards, net banking).
    - Returns a payment confirmation/transaction ID stored in the `Payment` schema.

- **Nodemailer**
    - Sends automated emails to users and providers.
    - Example use cases:

        - Booking confirmations.
        - Password reset emails (future enhancement).
        - Admin notifications.
- **Cloudinary**
    - Stores and serves service images, profile pictures, and review images.
    - Handles optimization and CDN delivery for faster loading.

8. Contribution Guidelines

8.1 Coding Standards

- Use **ESLint + Prettier** for code formatting and linting.
- Follow the **MVC pattern** for backend (Models, Routes/Controllers, Middleware).
- Use **PascalCase** for Vue components and classes.
- Use **camelCase** for variables and functions.
- Write meaningful comments for complex logic.

8.2 Running Tests

Developers can run automated tests to ensure system stability:

```
# Backend tests
cd backend
npm run test
```

```
# Frontend tests
cd frontend
npm run test
```

8.3 Pull Request Process

1. **Fork** the repository and create a new feature branch:

   ```
   git checkout -b feature/my-feature
   ```

2. Make changes and commit with meaningful messages:

   ```
   git commit -m "Add booking cancellation API"
   ```

3. Push to your fork and submit a Pull Request (PR).

4. At least one reviewer approval is required before merging into the main branch.

9. Future Improvements

We have noticed several enhancements that can be added to make the system more secure, user-friendly, and feature-rich:

- **Authentication Enhancements**
    - Add **email verification** during signup so new accounts must confirm ownership before accessing the system.
    - Implement **password reset** via email (with secure token + expiry link) and a **change password** option from the profile page.
    - Introduce **Two-Factor Authentication** (2FA) as an optional security step for users and providers.
- **Integrations**
    - Extend the **Twilio** SMS integration to send booking confirmations, OTPs, and alerts for important account activities.
    - Upgrade **Razorpay integration** to include advanced payment management features such as refunds, invoice generation, and transaction history.
- **Reviews & Feedback**
    - Allow providers to **view reviews and ratings** submitted by users on their services, giving them insights into performance and areas of improvement.
    - Admin moderation for reported reviews to maintain quality.
- **Admin User Experience**
    - Redesign the **Admin Dashboard UI** for better navigation, reporting, and management of users, providers, and services.
    - Add better visual analytics (graphs, charts) for quick monitoring of system activity.

“This concludes the Developer Documentation for LocalServe. The system is ready for deployment and future scaling.”