A **mini chat website** is a perfect project to practice the basics of **Node.js**, **MongoDB**, and **Vue.js**. Here's a suggested roadmap to help you achieve your goal with a small but complete real-time chat app:

## 🧠 ****Core Features for Your Mini Chat Project****

### ✅ Backend (Node.js + Express + MongoDB):

* User registration and login (simple auth)
* Store chat messages in MongoDB
* Use **Socket.IO** for real-time messaging
* API to get previous messages (chat history)

### ✅ Frontend (Vue.js):

* Register/Login UI
* Chat interface (message input + chat history)
* Real-time message updates using **Socket.IO client**

## 📁 Suggested Folder Structure

chat-app/

├── backend/ # Node.js backend

│ ├── server.js

│ ├── routes/

│ ├── models/

│ └── controllers/

├── frontend/ # Vue.js frontend (Vite or Vue CLI)

│ ├── src/

│ └── public/

└── README.md

## ⚙️ Key Technologies

| **Layer** | **Stack** |
| --- | --- |
| Frontend | Vue 3, Socket.IO client, Axios |
| Backend | Node.js, Express, Socket.IO, Mongoose |
| Database | MongoDB (local or MongoDB Atlas) |

## 🗓️ Project Plan (Milestones)

### Day 1–2: Setup Project

* Setup backend (Node + Express)
* Setup MongoDB and test connection
* Setup Vue project

### Day 3–4: Basic User System

* User model (username only or simple password)
* Registration/Login routes
* JWT or session auth (optional for simple project)

### Day 5–6: Real-time Messaging

* Integrate Socket.IO (send/receive messages)
* Save messages in MongoDB
* Fetch message history on page load

### Day 7: Final Touches

* Improve UI (basic styles)
* Display usernames and timestamps
* Test and deploy (optional)

Would you like a starter template or code snippet for the backend or frontend setup?

## ✅ ****Day 1–2: Setup Project****

### 📁 Folder Structure

chat-app/

├── backend/ # Node.js + Express

├── frontend/ # Vue 3 app

## 1. ✅ Setup Backend (Node + Express)

### Step-by-step:

mkdir chat-app

cd chat-app

mkdir backend

cd backend

npm init -y

npm install express mongoose cors dotenv

### Create basic files:

touch server.js .env

mkdir routes models

### server.js (basic setup):

const express = require('express');

const mongoose = require('mongoose');

const cors = require('cors');

require('dotenv').config();

const app = express();

app.use(cors());

app.use(express.json());

// Sample route

app.get('/', (req, res) => {

res.send('Chat backend is running!');

});

// MongoDB connection

mongoose.connect(process.env.MONGO\_URI, { useNewUrlParser: true, useUnifiedTopology: true })

.then(() => console.log('MongoDB connected'))

.catch((err) => console.log('MongoDB error:', err));

const PORT = process.env.PORT || 5000;

app.listen(PORT, () => {

console.log(`Server running on port ${PORT}`);

});

### .env file:

MONGO\_URI=mongodb://localhost:27017/chatdb

PORT=5000

Replace the URI with your MongoDB Atlas URI if you're not using local MongoDB.

## 2. ✅ Setup MongoDB

* If local: make sure MongoDB is running (e.g., mongod)
* If remote: use [MongoDB Atlas](https://www.mongodb.com/cloud/atlas/register)
* Use mongoose.connect() as shown above to test

## 3. ✅ Setup Vue 3 Frontend

### From root chat-app folder:

npm create vue@latest frontend

* Choose:
  + **Vue 3**
  + **No TypeScript** (or yes, your choice)
  + **No router** (you can add later)
  + **No Pinia** (or yes if you want)
  + **No testing** for now

cd frontend

npm install

npm run dev

## ✅ Test Everything

* Backend: visit http://localhost:5000 → should show "Chat backend is running!"
* Frontend: visit http://localhost:5173 (or the port Vite shows) → should show Vue starter page
* MongoDB: should log "MongoDB connected"

**🔐 Advanced User Features**

* **User profile with avatar**
* **Password reset via email**
* **Online/offline indicator**
* **Typing indicator**

**💬 Chat Enhancements**

* **Private (1-on-1) messaging**
* **Chat rooms / channels**
* **Emoji support**
* **Edit/Delete messages**
* **File or image attachments**

**📱 UX/UI Improvements**

* **Dark mode toggle**
* **Responsive mobile view**
* **Scroll to latest message on new chat**
* **Sound notification for new messages**

**🛠️ Technical Add-ons**

* **Pinia/Vuex** for managing state
* **Vue Router** for login/chat/profile pages
* **Token-based auto-login (JWT stored in cookies/localStorage)**
* **Socket.IO reconnection handling**

Perfect! You're moving into 🔐 **Advanced User Features**, which will make your chat app feel much more dynamic and user-friendly. Let’s go step by step — here’s a breakdown and brief plan for each:

**🔐 Advanced User Features**

**✅ 1. User Profile with Avatar**

* Update user schema to include an avatar field.
* Store avatar as a URL or use multer for image uploads.

**Update User.js schema:**

const userSchema = new mongoose.Schema({

username: { type: String, required: true, unique: true },

password: { type: String, required: true },

avatar: { type: String, default: '/default-avatar.png' }, // link to image

createdAt: { type: Date, default: Date.now }

});

Would you like to support image uploads or use just avatar URLs?

**✅ 2. Password Reset via Email**

**Steps:**

1. Add a resetToken and resetExpires field to the user schema.
2. Create a route to request password reset (/forgot-password).
3. Send email with a unique token link (use nodemailer).
4. Create a route to reset password using the token.

**Example additions to schema:**

resetToken: String,

resetExpires: Date,

Would you like me to provide the full flow with nodemailer?

**✅ 3. Online/Offline Indicator**

Use Socket.IO to track user presence in real-time.

**Basic idea:**

* When a user connects via Socket.IO, mark them online.
* When they disconnect, mark them offline after a delay.

**In memory (or Redis for scale):**

const onlineUsers = new Set();

io.on('connection', socket => {

const userId = socket.handshake.query.userId;

onlineUsers.add(userId);

socket.on('disconnect', () => {

onlineUsers.delete(userId);

});

});

You can then show green/red status on the frontend.

**✅ 4. Typing Indicator**

Also using Socket.IO:

* Client emits typing and stop typing events.
* Server broadcasts to other users in the same chat room.

socket.on('typing', (roomId) => {

socket.to(roomId).emit('display-typing', { userId });

});

socket.on('stop-typing', (roomId) => {

socket.to(roomId).emit('hide-typing', { userId });

});

Frontend shows “User is typing...” temporarily.

Which feature would you like to build first? (Avatar upload, password reset, online status, or typing indicator?)

You're welcome! Let's start by creating a **User Registration Page** in Vue.js.

**📝 1. User Registration Page**

This page will include:

* **Username** input
* **Email** input
* **Password** input
* **Confirm Password** input
* **Register** button

**🧱 2. Register.vue Component**

<template>

<div class="register-page">

<h2>Register</h2>

<form @submit.prevent="registerUser">

<label>Username</label>

<input type="text" v-model="user.username" required />

<label>Email</label>

<input type="email" v-model="user.email" required />

<label>Password</label>

<input type="password" v-model="user.password" required />

<label>Confirm Password</label>

<input type="password" v-model="user.confirmPassword" required />

<button type="submit">Register</button>

</form>

<div v-if="errorMessage" class="error-message">

<p>{{ errorMessage }}</p>

</div>

<p>Already have an account? <router-link to="/login">Login</router-link></p>

</div>

</template>

<script>

export default {

data() {

return {

user: {

username: '',

email: '',

password: '',

confirmPassword: ''

},

errorMessage: ''

};

},

methods: {

async registerUser() {

if (this.user.password !== this.user.confirmPassword) {

this.errorMessage = 'Passwords do not match!';

return;

}

const newUser = {

username: this.user.username,

email: this.user.email,

password: this.user.password

};

try {

const res = await fetch('http://localhost:3000/api/users/register', {

method: 'POST',

headers: { 'Content-Type': 'application/json' },

body: JSON.stringify(newUser)

});

const data = await res.json();

if (res.ok) {

this.$router.push('/login'); // Redirect to login page after successful registration

} else {

this.errorMessage = data.error || 'Registration failed!';

}

} catch (error) {

this.errorMessage = 'Server error: ' + error.message;

}

}

}

};

</script>

<style scoped>

.register-page {

padding: 20px;

max-width: 400px;

margin: auto;

}

.register-page form {

display: flex;

flex-direction: column;

}

.register-page input {

margin-bottom: 10px;

padding: 8px;

}

.register-page button {

padding: 10px;

background-color: #4CAF50;

color: white;

border: none;

cursor: pointer;

}

.error-message {

color: red;

margin-top: 10px;

}

</style>

**✅ 2. Backend API (Express.js)**

Now let's create the backend route to handle registration. We'll use bcrypt to hash the password.

// POST /api/users/register

const bcrypt = require('bcrypt');

const User = require('../models/User');

router.post('/register', async (req, res) => {

const { username, email, password } = req.body;

// Basic validation

if (!username || !email || !password) {

return res.status(400).json({ error: 'All fields are required!' });

}

// Check if user already exists

const existingUser = await User.findOne({ email });

if (existingUser) {

return res.status(400).json({ error: 'Email already registered!' });

}

// Hash password

const hashedPassword = await bcrypt.hash(password, 10);

// Create new user

const newUser = new User({

username,

email,

password: hashedPassword

});

try {

await newUser.save();

res.status(201).json({ message: 'User registered successfully!' });

} catch (err) {

res.status(500).json({ error: 'Error registering user!' });

}

});

**✅ 3. Add Route for User Registration in Backend**

In app.js or server.js, make sure to use the new route for registration.

const userRoutes = require('./routes/users');

app.use('/api/users', userRoutes);

**🚀 4. Run the Application**

* Run the **backend server** to handle the registration API.
* Use the **Vue.js frontend** to test the user registration page.

**👨‍💻 Additional Improvements**

* Add form validation (e.g., check if email format is valid).
* Implement **email verification** for extra security.
* Redirect users to the login page after successful registration.

Would you like help with adding login functionality next?

Sure! Let's create a **Login Page** in Vue.js that includes:

**✅ Features:**

* Email + Password input
* Frontend validation
* API request to backend /api/users/login
* Error message display
* Redirect on success

**🧱 Login.vue Component**

<template>

<div class="login-page">

<h2>Login</h2>

<form @submit.prevent="loginUser">

<label>Email</label>

<input type="email" v-model="email" />

<span class="error" v-if="errors.email">{{ errors.email }}</span>

<label>Password</label>

<input type="password" v-model="password" />

<span class="error" v-if="errors.password">{{ errors.password }}</span>

<button type="submit">Login</button>

</form>

<div v-if="serverError" class="error-message">

<p>{{ serverError }}</p>

</div>

<p>Don’t have an account? <router-link to="/register">Register</router-link></p>

</div>

</template>

<script>

export default {

data() {

return {

email: '',

password: '',

errors: {},

serverError: ''

};

},

methods: {

validateForm() {

this.errors = {};

let isValid = true;

if (!this.email) {

this.errors.email = 'Email is required';

isValid = false;

} else if (!/^[^\s@]+@[^\s@]+\.[^\s@]+$/.test(this.email)) {

this.errors.email = 'Invalid email format';

isValid = false;

}

if (!this.password) {

this.errors.password = 'Password is required';

isValid = false;

}

return isValid;

},

async loginUser() {

this.serverError = '';

if (!this.validateForm()) return;

try {

const res = await fetch('http://localhost:3000/api/users/login', {

method: 'POST',

headers: { 'Content-Type': 'application/json' },

body: JSON.stringify({

email: this.email,

password: this.password

})

});

const data = await res.json();

if (res.ok) {

// Save token or session info

localStorage.setItem('token', data.token);

this.$router.push('/dashboard'); // Redirect to dashboard or chat

} else {

this.serverError = data.error || 'Login failed';

}

} catch (err) {

this.serverError = 'Server error: ' + err.message;

}

}

}

};

</script>

<style scoped>

.login-page {

max-width: 400px;

margin: auto;

padding: 20px;

}

input {

width: 100%;

padding: 8px;

margin-bottom: 8px;

}

.error {

color: red;

font-size: 13px;

margin-bottom: 8px;

}

.error-message {

color: red;

margin-top: 15px;

}

</style>

**✅ Backend API Endpoint (Example in Express.js)**

// POST /api/users/login

const bcrypt = require('bcrypt');

const jwt = require('jsonwebtoken');

const User = require('../models/User');

router.post('/login', async (req, res) => {

const { email, password } = req.body;

if (!email || !password)

return res.status(400).json({ error: 'Email and password are required' });

const user = await User.findOne({ email });

if (!user) return res.status(401).json({ error: 'User not found' });

const isMatch = await bcrypt.compare(password, user.password);

if (!isMatch) return res.status(401).json({ error: 'Invalid credentials' });

const token = jwt.sign({ userId: user.\_id }, 'your-secret-key', { expiresIn: '1d' });

res.status(200).json({ token });

});

Would you like to add **“Remember me”**, **Google login**, or go straight to the **chat/dashboard page** next?