Real Estate by React

Project Name: Estately

Supervised by:

Eng. Basma Abdel Halim

Team Members:

- **Antonyos Milad** (MERN Stack Developer)
- **Felopateer Shokry** (Frontend Developer)
- **Karim Bassem** (Frontend Developer)
- **Abdelrahman Shaban** (Frontend Developer)
- **Ahmed Nader** (Frontend Developer)

Acknowledgement

We would like to extend our sincere appreciation to **Eng. Basma Abdel Halim** for her distinguished guidance and unwavering support throughout our participation in the Digital Egypt Pioneers Initiative. Her mentorship has been integral to enhancing our knowledge and strengthening our capabilities.

Her extensive technical expertise, insightful feedback, and steadfast commitment to fostering innovation have significantly contributed to the advancement and success of our work. She has consistently provided clear direction and effective solutions, enabling us to address challenges with confidence and precision.

We are deeply grateful for the opportunity to learn and grow under her exemplary leadership. Her professionalism and dedication have left a profound impact on our journey, and we truly value her continued efforts in guiding and supporting us at every stage.

Abstract

The increasing reliance on digital platforms for property buying and selling has emphasized the need for secure, efficient, and user-friendly real estate solutions. This project presents **Estately**, an online real estate platform designed to facilitate seamless interaction between sellers and buyers, offering a safe and organized environment similar to modern classified platforms.

The system utilizes a **React-based front-end** to ensure a responsive, intuitive, and accessible user experience. Through a dedicated and well-structured **Node.js/Express back-end**, the platform supports secure user authentication, property listing management, and reliable communication between parties via **Socket.io**. The architecture is built to maintain data integrity using **MongoDB/Prisma**, enhance user trust, and provide a scalable foundation capable of supporting a wide range of real estate activities.

By integrating modern web technologies and focusing on security and usability, this project contributes to the digital transformation of the real estate market. It delivers a practical solution that simplifies property advertising and discovery, ensuring transparency, reliability, and efficiency in connecting buyers and sellers.

Table of Contents

Chapter 1: Project Overview

1.1. Introduction

The real estate sector is one of the most dynamic industries globally. However, the process of buying, selling, or renting properties is often plagued by inefficiencies, lack of transparency, and communication barriers. **Estately** is a web-based platform developed to solve these issues by creating a direct bridge between property owners and seekers.

1.2. Project Scope

The project encompasses the full software development lifecycle (SDLC) of a web application:

- **Frontend:** A React Single Page Application (SPA) for users to browse, filter, and manage listings.
- **Backend:** A Node.js REST API to handle business logic and database operations.
- **Real-Time Server:** A Socket.io server to handle instant messaging.
- **Database:** A structured MongoDB database managed via Prisma ORM.

Chapter 2: Project Planning & Management

2.1. Project Proposal

Estately aims to be the "OLX of Real Estate," but with specialized features tailored to the housing market, such as bedroom filters, amenity checklists, and map-based searching.

2.2. Development Timeline

The project was executed over a 9-week period:

- **Week 1-2:** Requirement Gathering, Feasibility Study, and UI Prototyping.
- **Week 3-4:** Database Schema Design (Prisma) and Backend API Setup.
- **Week 5-6:** Frontend Component Development (React) and API Integration.
- **Week 7:** Real-Time Chat implementation using Socket.io.
- **Week 8:** Map Integration (Leaflet), Testing, and Bug Fixing.
- **Week 9:** Final Deployment, Documentation, and Presentation.

2.3. Task Assignment & Roles

Team Member	Role	Responsibilities
Antonyos Milad	Team Lead / Backend	System Architecture, Database Design, API Development, Socket.io Server.
Felopateer Shokry	Frontend Developer	State Management (Context API), Authentication Flow, Chat UI.
Karim Bassem	Frontend Developer	Responsive Styling (SCSS), Home Page, Layouts, Theme Toggling.
Abdelrahman Shaban	Frontend Developer	Map Integration, Search Filters, Property Cards.
Ahmed Nader	Frontend Developer	User Profile, Dashboard, Form Validation, Testing.

2.4. Tools & Technologies

- **Frontend:** React.js, Vite, SASS (SCSS), React Router Dom, Leaflet (Maps), React Quill (Rich Text).
- **Backend:** Node.js, Express.js, Bcrypt (Security), JSON Web Token (Auth), Cookie Parser.
- **Database:** MongoDB Atlas (Cloud DB), Prisma ORM (Data Modeling).
- **Real-Time:** Socket.io (WebSockets).
- **Version Control:** Git & GitHub.

Chapter 3: Literature Review

3.1. Market Analysis

The modern real estate market demands speed. According to recent studies, homes listed with high-quality images and instant agent availability sell 30% faster. Users are moving away from desktop-only platforms to mobile-responsive web apps that offer "app-like" experiences.

3.2. Limitations of Existing Systems

Many existing local competitors suffer from:

1. **Static Forms:** Users fill out a "Contact Agent" form and wait 24-48 hours for a reply.
2. **Cluttered UI:** Interfaces are often overloaded with ads and irrelevant links.
3. **No Map Search:** Listings are just lists; users cannot see "what is near this house" easily.
4. **Session Insecurity:** Many older sites do not use secure HTTP-only cookies for session management.

3.3. The Estately Solution

Estately differentiates itself by solving the "Latency" problem.

- **Immediate Feedback:** If an agent is online, the buyer sees a green dot. They can chat instantly.
- **Geospatial Context:** The screen is split 50/50 between the list and the map, giving equal weight to "What the house looks like" and "Where it is."
- **Modern Tech Stack:** Using the MERN stack ensures the site is fast (SPA), SEO-friendly, and easily scalable.

Chapter 4: Requirements Gathering

4.1. Stakeholder Analysis

- **Property Buyers/Renters:** Need powerful search filters (e.g., "Under $2000", "3 Bedrooms") and safety.
- **Property Owners/Agents:** Need an easy way to upload photos and manage listing status.
- **Admins:** Need to ensure the platform remains spam-free.

4.2. User Stories

Authentication & Profile

- **US-01:** As a user, I want to register via email so I can save my favorite houses.
- **US-02:** As a user, I want to upload a profile picture so agents know who they are talking to.

Discovery & Search

- **US-03:** As a buyer, I want to filter by "Buy" vs "Rent" to match my needs.
- **US-04:** As a buyer, I want to see the distance to the nearest school and bus stop.
- **US-05:** As a buyer, I want to view the location on a map.

Communication

- **US-06:** As a user, I want to receive a notification badge when I get a new message.
- **US-07:** As an agent, I want to mark chats as "read" after I reply.

4.3. Functional Requirements

1. **Authentication System:** The system must generate a JWT upon login and store it in an HTTP-only cookie.
2. **Property CRUD:** Users must be able to Create, Read, Update, and Delete their own posts.
3. **Search Engine:** The backend must support query parameters for city, type, property, minPrice, maxPrice, and bedroom.
4. **Image Upload:** The system must integrate with **Cloudinary** (via Upload Widget) to handle image storage.
5. **Real-Time Messaging:** The system must use WebSockets to push messages to online users instantly.

4.4. Non-Functional Requirements

1. **Performance:** The application should load the initial view (Home) in under 1.5 seconds.
2. **Reliability:** The chat server should automatically reconnect if the internet connection blips.
3. **Scalability:** The database schema (Prisma) should be normalized to handle thousands of listings.
4. **Security:** All passwords must be hashed using bcrypt before storage. API endpoints must be protected by verifyToken middleware.

Chapter 5: System Analysis & Architecture

5.1. System Architecture

Estately is a **distributed web application**:

1. **Client Layer:** React.js running in the browser. Handles routing, UI rendering, and API calls using axios.
2. **Service Layer:** Node.js/Express API. Receives HTTP requests, validates data, and queries the database.
3. **Data Layer:** MongoDB. Stores JSON-like documents for Users, Posts, and Chats.
4. **WebSocket Layer:** A standalone Socket.io server that maintains persistent connections for chat.

5.2. Problem Statement

Managing real estate transactions manually or through outdated legacy systems presents several challenges:

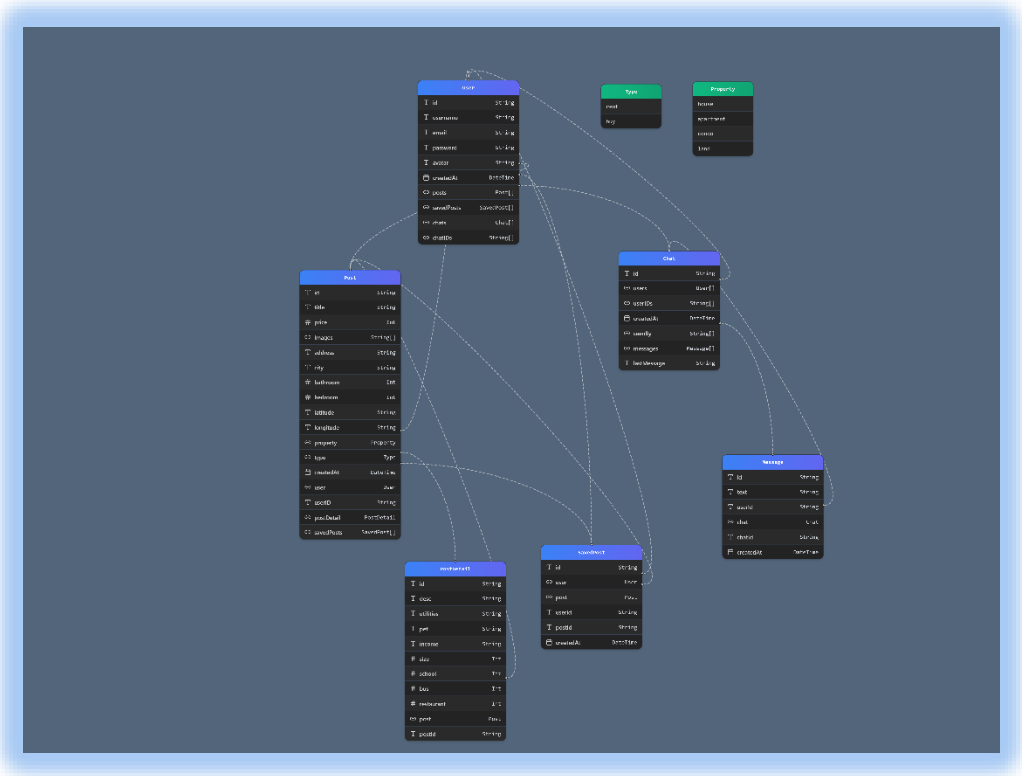
1. **Fragmented Information:** Property details are often scattered across social media, newspapers, and word-of-mouth, making it hard to compare options.
2. **Lack of Trust:** Without secure profiles, users hesitate to interact with strangers.
3. **Communication Latency:** Relying on phone calls or emails leads to missed opportunities.
4. **Poor Visualization:** Text-based listings fail to convey the geographical context of a property (e.g., proximity to schools or transport).

Objectives

- **Centralization:** Consolidate property listings into a single, searchable database.
- **Interactivity:** Provide interactive maps and image sliders for better property visualization.
- **Real-Time Connection:** Enable instant chat between buyers and sellers to speed up negotiations.
- **Security:** Ensure all users are authenticated and data is protected using industry-standard encryption (JWT).
- **User Experience:** Deliver a "Mobile-First" design that works seamlessly on all devices.

5.3. Database Design (UML Design & Prisma Schema)

UML Design (Class diagram):



The data model is defined in schema prisma

Core Models:

1. User

- id: ObjectId (Primary Key)
- email: String (Unique)
- username: String (Unique)
- password: String (Hashed)
- avatar: String (URL)
- *Relations:* One-to-Many with Posts, Many-to-Many with Chats.

2. Post

- id: ObjectId
- title: String
- price: Integer
- images: String[]
- address: String
- city: String
- bedroom: Integer
- bathroom: Integer
- type: Enum (buy, rent)
- property: Enum (apartment, house, condo, land)
- *Relations:* Belongs to User, Has one PostDetail.

- **3. PostDetail**
- id: ObjectId
- desc: String
- utilities: String
- pet: String
- income: String
- size: Integer
- school: Integer (Distance)
- bus: Integer (Distance)
- restaurant: Integer (Distance)

4. Chat & Message

- **Chat:** Connects two users (userIDs array). Tracks seenBy users.
- **Message:** Contains text, userId (Sender), chatId, and createdAt.

5.4. Real-Time Communication Architecture

The chat system does not rely on HTTP polling.

1. **Connection:** When a user logs in, the React Client connects to ws://localhost:4000.
2. **Identification:** The client emits a newUser event with the userId. The server maps the socket.id to the userId.
3. **Messaging:** When User A sends a message:

   - Client saves message to DB via API (Persistence).
   - Client emits sendMessage event to Socket Server.
   - Socket Server checks if User B is online.
   - If online, Socket Server emits getMessage event to User B.
   - User B's React Client updates the UI instantly.

Chapter 6: Implementation Details

6.1. Backend Implementation

The backend is structured using the **MVC (Model-View-Controller)** pattern, although in Node.js/Express it is often Route-Controller-Service.

- **app.js:** The entry point. Initializes Express, sets up CORS (Cross-Origin Resource Sharing) to allow requests from the frontend, and parses cookies.
- **controllers/auth.controller.js:**
  - register(): Hashes password using bcrypt.hash(). Creates user in Prisma.
  - login(): Verifies password using bcrypt.compare(). Generates a JWT token. Sets the token as a cookie with httpOnly: true and maxAge.
- **controllers/post.controller.js:**
  - getPosts(): Uses prisma.post.findMany(). It dynamically builds the query object based on req.query (e.g., filtering by city or price).
- **middleware/verifyToken.js:**
  - Intercepts requests. Checks req.cookies.token. Uses jwt.verify(). If valid, attaches req.userId to the request and calls next(). If invalid, returns 401 Not Authenticated.

6.2. Frontend Implementation

The frontend uses **React Functional Components** and **Hooks**.

- **lib/apiRequest.js:** A configured Axios instance that automatically sends credentials (cookies) with every request.
- **context/AuthContext.jsx:** Uses createContext. It wraps the entire app. It checks LocalStorage on load to see if a user is logged in and updates the state. This allows any component (Navbar, Profile) to access the currentUser object.

- **routes/listPage/listPage.jsx:**
  - Uses useLoaderData() (from React Router) to fetch data *before* the page renders, ensuring no "loading spinners" are needed for the initial content.
  - Displays the Filter component (top), Card list (left), and Map (right).

6.3. Real-Time Chat (Socket.io)

The `socket` logic ensures immediate delivery of messages. The client listens for the `getMessage` event and updates the React state array `[messages, setMessages]`, triggering a re-render of the chat window.

6.4. Map Integration (Leaflet)

We utilize react-leaflet for rendering maps.

- **components/map/Map.jsx:**
  - Receives an array of items (posts).
  - Renders a <MapContainer> centered on the first item or a default location.
  - Iterates through items to render <Pin> components.
  - Each <Pin> contains a <Popup> with the property image, title, and price.

6.5. Deployment & Execution

To ensure high availability, scalability, and robust real-time performance, **Estately** utilizes a **hybrid cloud deployment strategy**. We decoupled the hosting environments to leverage the specific strengths of **Vercel** and **Replit**.

6.5.1. Deployment Architecture

- **Frontend (Client) & Backend (API):** Deployed on **Vercel**.

Reasoning: Vercel provides a world-class CDN for serving static React assets and scalable Serverless Functions for the API.

Real-Time Server (Socket): Deployed on **Replit**.

Reasoning: Serverless platforms like Vercel are "stateless" and kill connections quickly, making them unsuitable for WebSockets. **Replit** allows for continuous, stateful server execution, ideal for maintaining active chat connections.

6.5.2. Frontend & API Deployment (Vercel)

The core application was deployed using Vercel's automated CI/CD pipeline.

1. **CI/CD Integration:** Linked GitHub repository triggers a new build on every push to `main`.
2. **Serverless Configuration:** Utilized `vercel.json` to route `/api/*` requests to the Node.js backend and all other requests to the React frontend.
3. **Environment Security:** `DATABASE_URL` and `JWT_SECRET_KEY` are stored in Vercel's encrypted variables.

6.5.3. Socket Server Deployment (Replit)

The Socket.io server requires a **stateful environment** to track the `onlineUser` array.

1. **Standalone Deployment:** The `socket` directory runs continuously on Replit.
2. **CORS Configuration:** Configured to accept connections *only* from `https://estately-app.vercel.app`.
3. **Uptime Management:** Implemented a keep-alive mechanism to prevent the Replit container from sleeping.

6.5.4. Execution Flow

1. **User Access:** User visits Vercel URL -> Vercel CDN delivers React UI.
2. **Data Fetching:** User searches -> HTTP request to Vercel API -> Query MongoDB.
3. **Real-Time Connection:** User logs in -> WebSocket connection established to Replit URL.

Chapter 7: UI/UX Design

7.1. Design Philosophy

- **Clean & Minimalist:** High use of whitespace to make property photos pop.
- **Dark/Light Mode:** A toggle in the navbar allows users to switch themes. This is handled by darkModeContext.scss changing global CSS variables.
- **Responsive:** Using SCSS media queries (@include mobile, @include tablet), the layout changes from a multi-column grid on desktop to a single-column stack on mobile.

7.2. Page Layouts

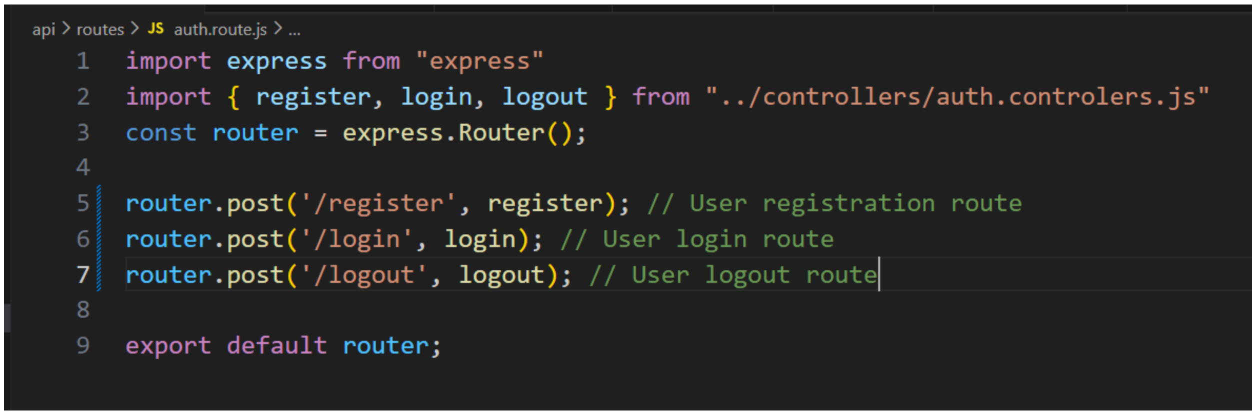
- **Home Page:** Features a Hero Section with a "Search Bar" overlay.
- **List Page:** A specialized "Split Layout". The left side scrolls (listings), while the right side remains sticky (Map). This is crucial for UX, allowing users to explore the map without losing their place in the list.
- **Single Page:** Features a grid layout for images (1 large, 3 small) and a sidebar for the Agent's contact info.

Chapter 8: API Documentation

8.1. API organization

8.1.1. Authentication

- POST /api/auth/register – Registers a new user account in the system.
- POST /api/auth/login – Authenticates a user and returns a secure session cookie.
- POST /api/auth/logout – Ends the user session and clears authentication cookies.

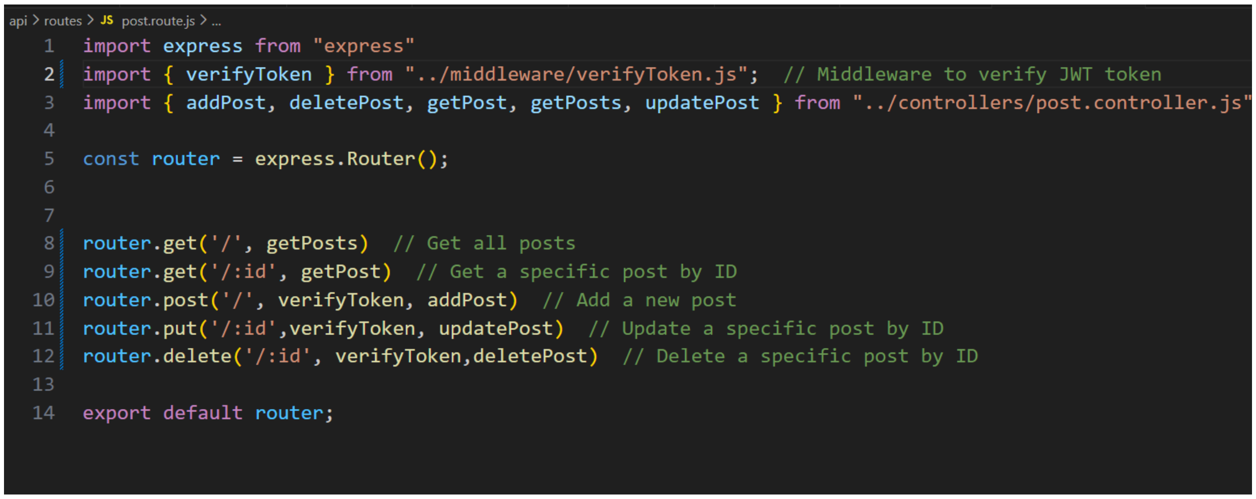


api > routes > JS auth.route.js > ...

```javascript
import express from "express"
import { register, login, logout } from "../controllers/auth.controlers.js"
const router = express.Router();

router.post('/register', register); // User registration route
router.post('/login', login); // User login route
router.post('/logout', logout); // User logout route

export default router;
```

8.1.2 Posts (Properties)

- GET /api/posts – Retrieves a list of all property listings (supports filtering by city, price, etc.).
- GET /api/posts/:id – Retrieves full details of a single property listing.
- POST /api/posts – Creates a new property listing with details and images.
- PUT /api/posts/:id – Updates details of an existing property listing.
- DELETE /api/posts/:id – Deletes a specific property listing.



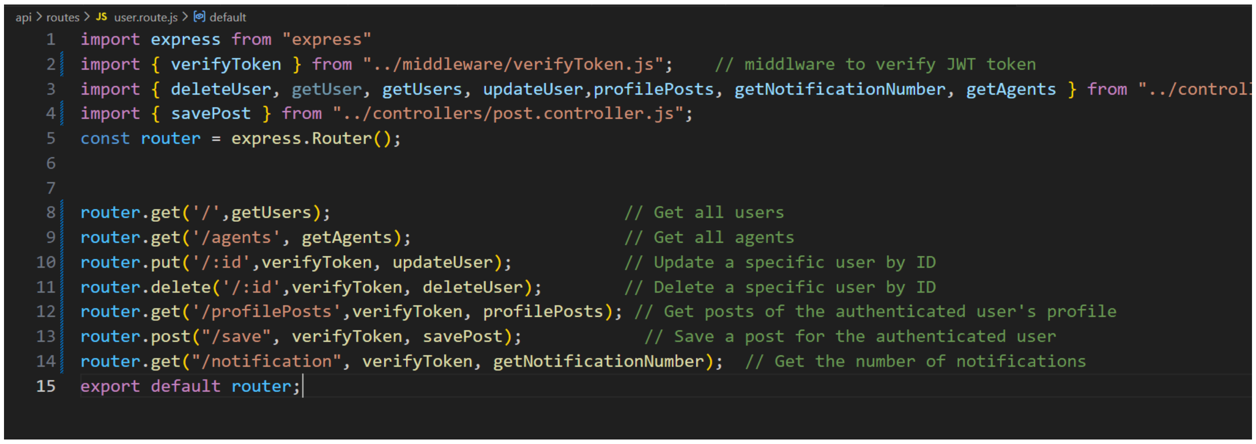
api > routes > JS post.route.js > ...

```javascript
import express from "express"
import { verifyToken } from "../middleware/verifyToken.js";  // Middleware to verify JWT token
import { addPost, deletePost, getPost, getPosts, updatePost } from "../controllers/post.controller.js"

const router = express.Router();


router.get('/', getPosts)  // Get all posts
router.get('/:id', getPost)  // Get a specific post by ID
router.post('/', verifyToken, addPost)  // Add a new post
router.put('/:id',verifyToken, updatePost)  // Update a specific post by ID
router.delete('/:id', verifyToken,deletePost)  // Delete a specific post by ID

export default router;
```

8.1.3 Users

- GET /api/users – Retrieves a list of all users (for admin or public view).
- GET /api/users/:id – Retrieves profile details of a specific user.

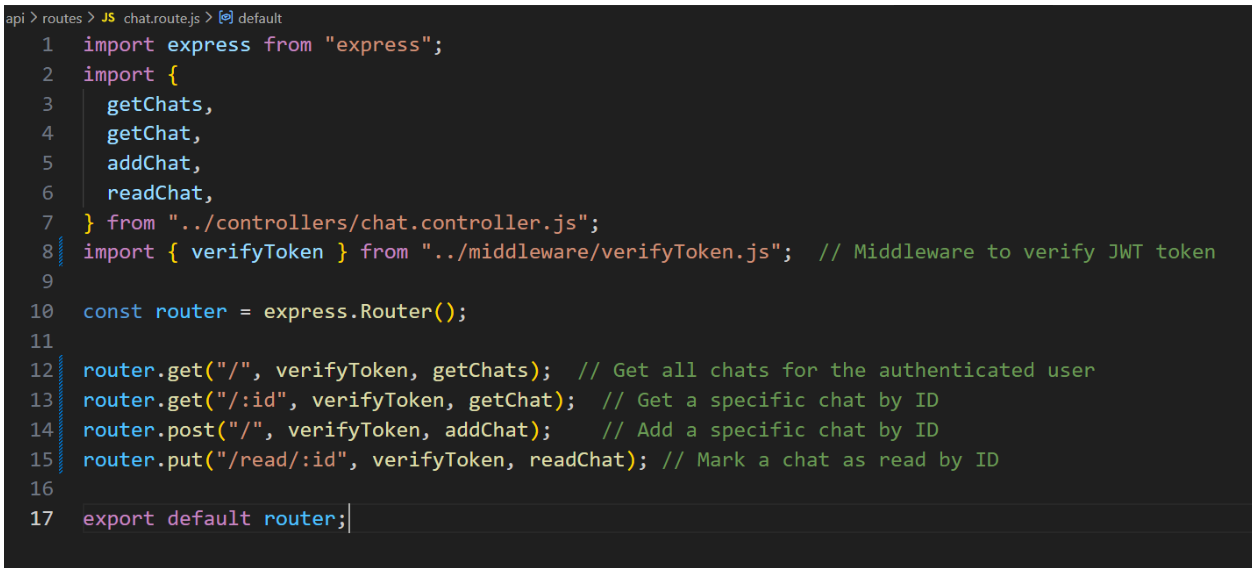
- PUT /api/users/:id – Updates an existing user's profile information (e.g., avatar, password).
- DELETE /api/users/:id – Deletes a specific user account.
- GET /api/users/profilePosts – Retrieves all posts created or saved by the logged-in user.
- GET /api/users/notification – Retrieves the count of unread message notifications.
- GET /api/users/agents – Retrieves a specific list of users registered as agents.



```
import express from "express"
import { verifyToken } from "../middleware/verifyToken.js";    // middlware to verify JWT token
import { deleteUser, getUser, getUsers, updateUser,profilePosts, getNotificationNumber, getAgents } from "../controll
import { savePost } from "../controllers/post.controller.js";
const router = express.Router();


router.get('/',getUsers);                              // Get all users
router.get('/agents', getAgents);                      // Get all agents
router.put('/:id',verifyToken, updateUser);            // Update a specific user by ID
router.delete('/:id',verifyToken, deleteUser);         // Delete a specific user by ID
router.get('/profilePosts',verifyToken, profilePosts); // Get posts of the authenticated user's profile
router.post("/save", verifyToken, savePost);             // Save a post for the authenticated user
router.get("/notification", verifyToken, getNotificationNumber);  // Get the number of notifications
export default router;
```

8.1.4 Chat

- GET /api/chats – Retrieves a list of all active chat conversations for the logged-in user.
- GET /api/chats/:id – Retrieves the full message history of a single chat session.
- POST /api/chats – Initiates a new chat room with another user.
- PUT /api/chats/read/:id – Marks a specific chat conversation as "read".

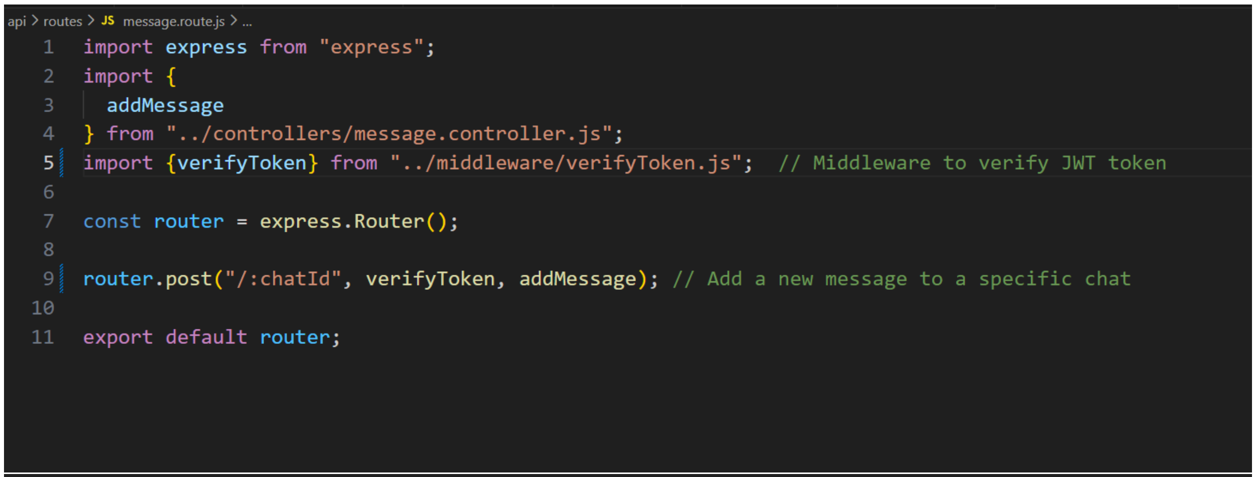


api > routes > JS chat.route.js > default

```javascript
import express from "express";
import {
  getChats,
  getChat,
  addChat,
  readChat,
} from "../controllers/chat.controller.js";
import { verifyToken } from "../middleware/verifyToken.js";  // Middleware to verify JWT token

const router = express.Router();

router.get("/", verifyToken, getChats);  // Get all chats for the authenticated user
router.get("/:id", verifyToken, getChat);  // Get a specific chat by ID
router.post("/", verifyToken, addChat);    // Add a specific chat by ID
router.put("/read/:id", verifyToken, readChat); // Mark a chat as read by ID

export default router;
```

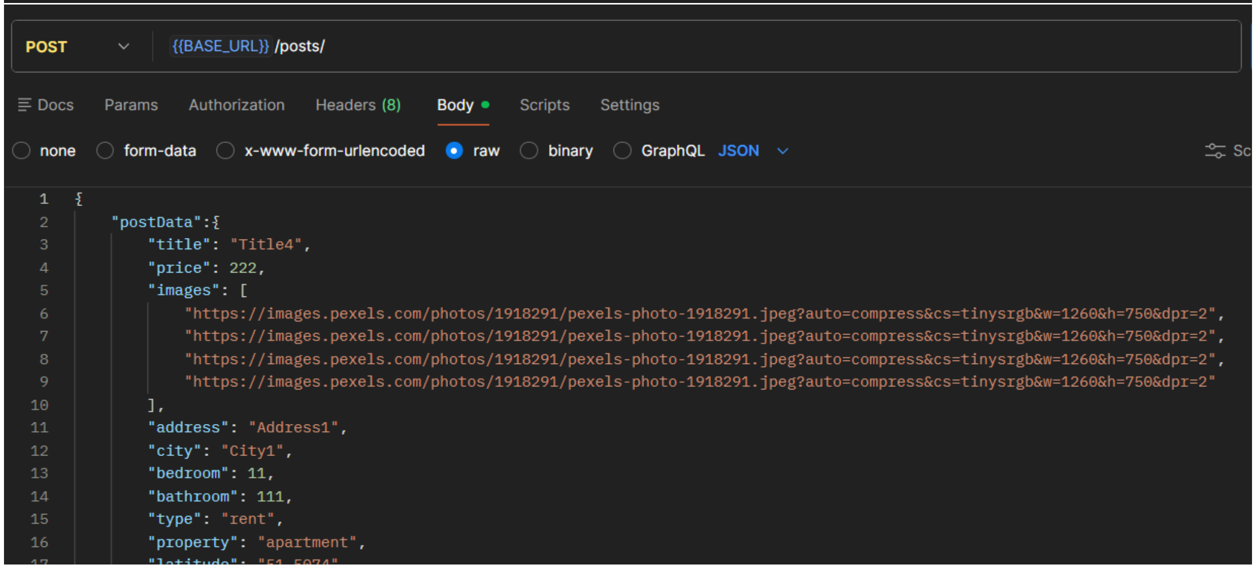
8.1.5 messages

- POST /api/messages/:chatId – Sends a new text message within a specific chat room.



api > routes > JS message.route.js > ...

```javascript
import express from "express";
import {
  addMessage
} from "../controllers/message.controller.js";
import {verifyToken} from "../middleware/verifyToken.js";  // Middleware to verify JWT token

const router = express.Router();

router.post("/:chatId", verifyToken, addMessage); // Add a new message to a specific chat

export default router;
```



POST {{BASE_URL}} /posts/

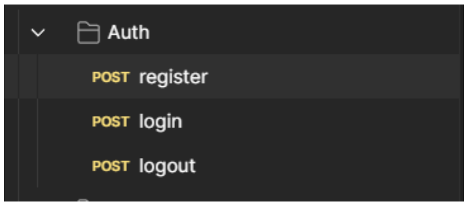
Docs Params Authorization Headers (8) Body ● Scripts Settings

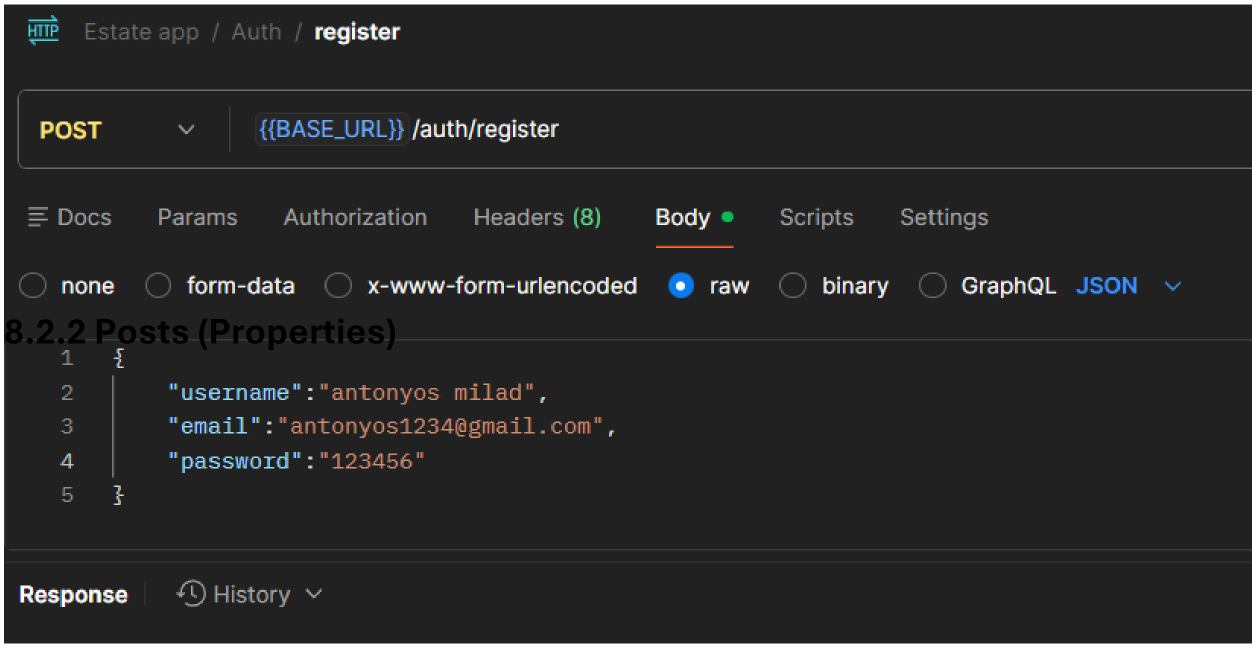
none form-data x-www-form-urlencoded [✓ raw] binary GraphQL JSON

```json
{
    "postData":{
        "title": "Title4",
        "price": 222,
        "images": [
            "https://images.pexels.com/photos/1918291/pexels-photo-1918291.jpeg?auto=compress&cs=tinysrgb&w=1260&h=750&dpr=2",
            "https://images.pexels.com/photos/1918291/pexels-photo-1918291.jpeg?auto=compress&cs=tinysrgb&w=1260&h=750&dpr=2",
            "https://images.pexels.com/photos/1918291/pexels-photo-1918291.jpeg?auto=compress&cs=tinysrgb&w=1260&h=750&dpr=2",
            "https://images.pexels.com/photos/1918291/pexels-photo-1918291.jpeg?auto=compress&cs=tinysrgb&w=1260&h=750&dpr=2"
        ],
        "address": "Address1",
        "city": "City1",
        "bedroom": 11,
        "bathroom": 111,
        "type": "rent",
        "property": "apartment",
```

8.2. Postman collection

8.2.1. Authentication



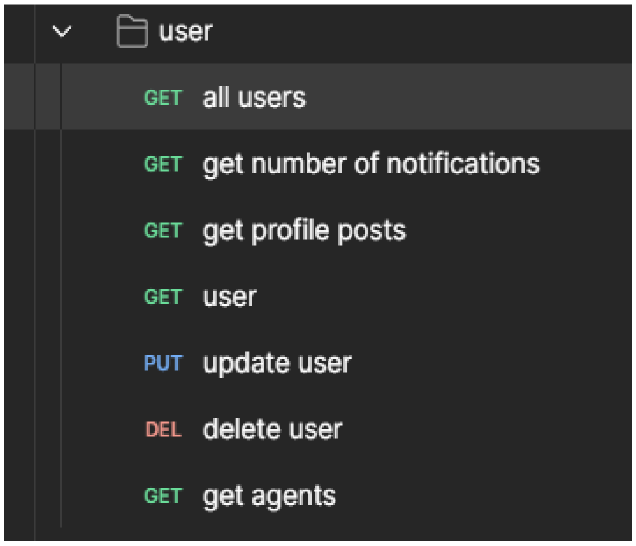


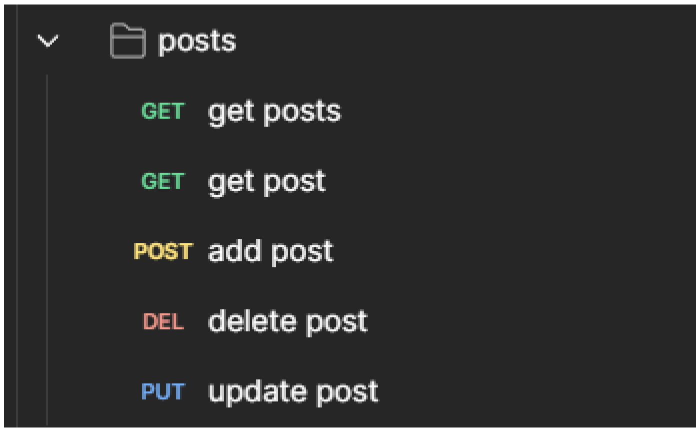
```
{
    "username":"antonyos milad",
    "email":"antonyos1234@gmail.com",
    "password":"123456"
}
```

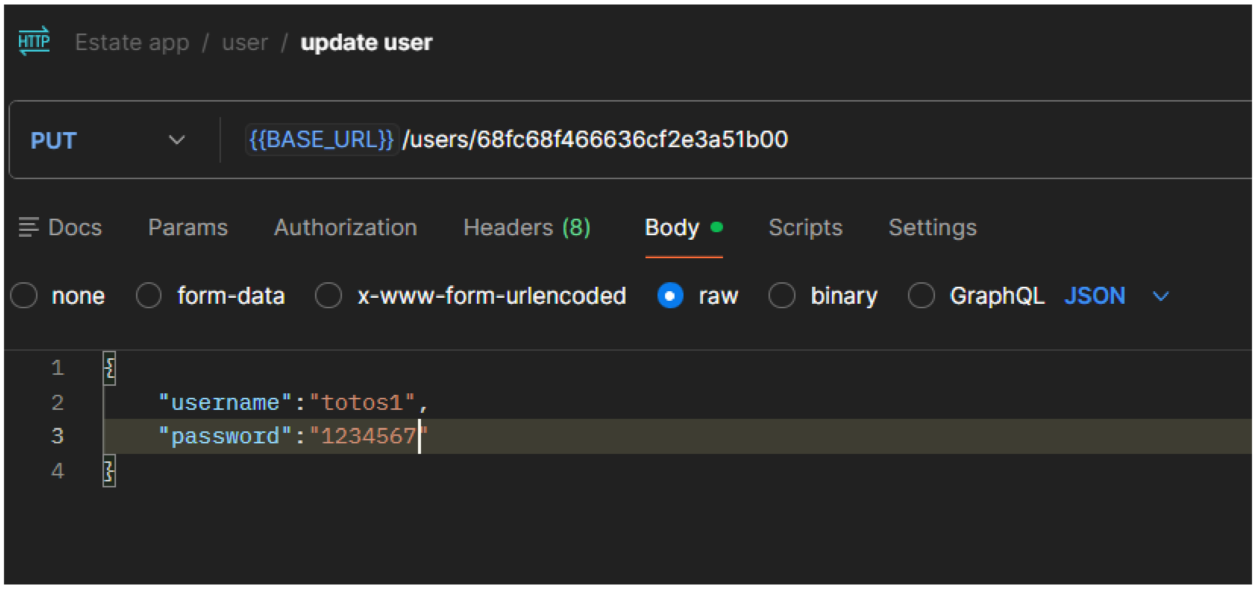
8.2.2 Posts (Properties)

8.2.3 Users

- **GET /api/users/profilePosts**







Estate app / user / **update user**

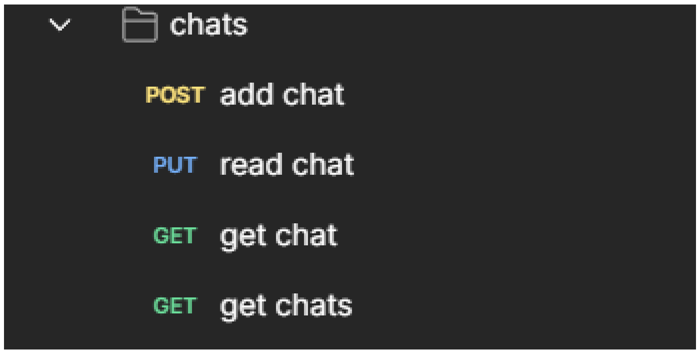
PUT {{BASE_URL}} /users/68fc68f466636cf2e3a51b00

Docs Params Authorization Headers (8) Body Scripts Settings

none form-data x-www-form-urlencoded [✓ raw] binary GraphQL JSON

```
{
    "username":"totos1",
    "password":"1234567"
}
```

8.2.4 Chat



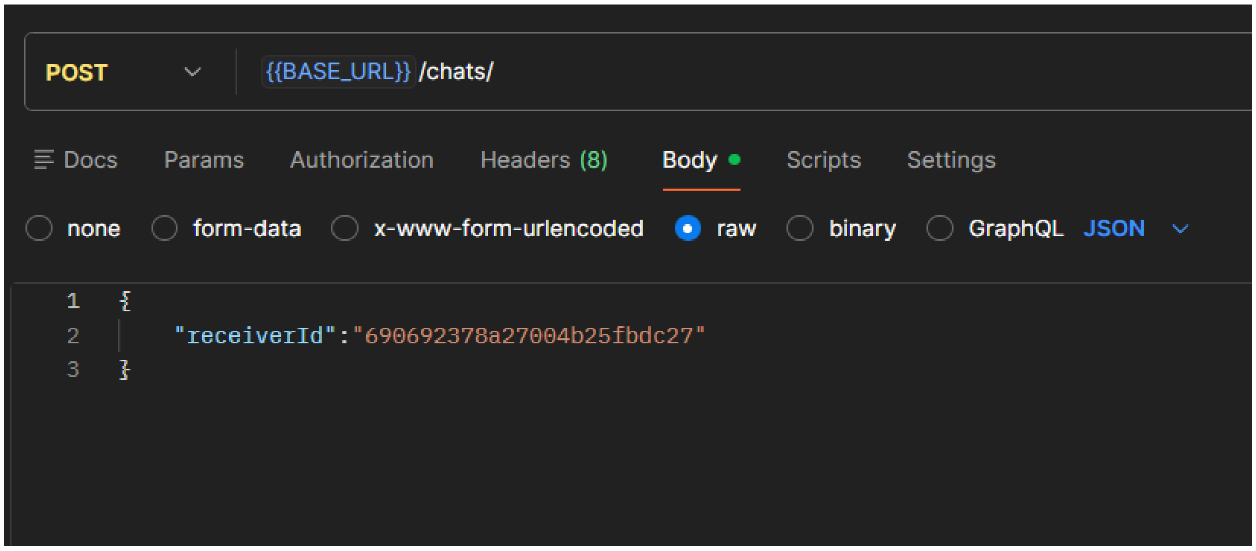
chats

POST add chat

PUT read chat

GET get chat

GET get chats



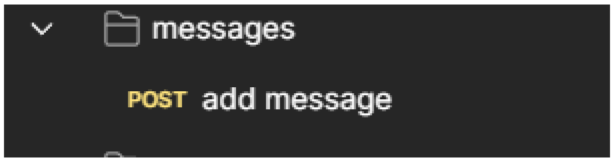
POST {{BASE_URL}} /chats/

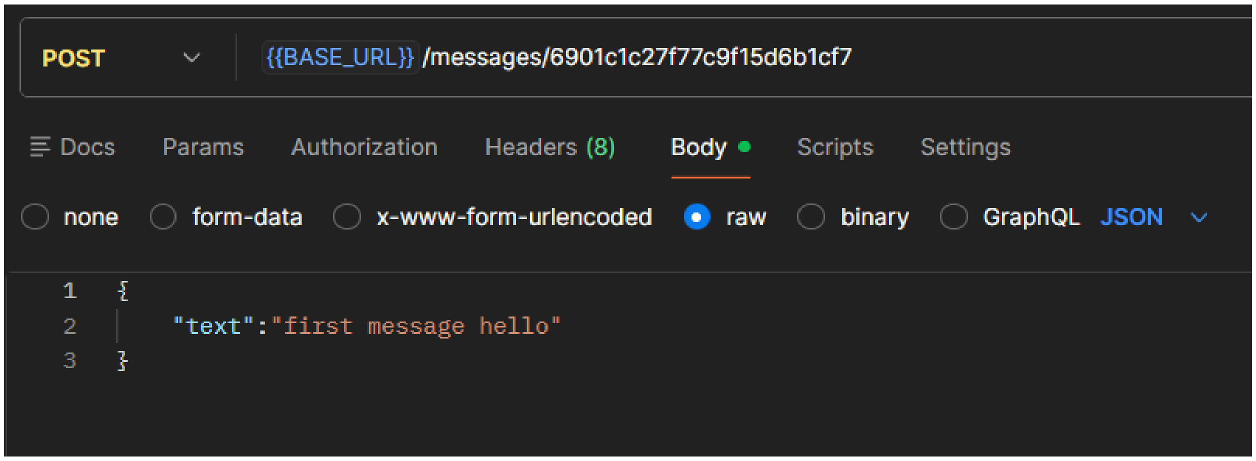
Docs Params Authorization Headers (8) Body Scripts Settings

none form-data x-www-form-urlencoded [✓ raw] binary GraphQL JSON

```
{
    "receiverId":"690692378a27004b25fbdc27"
}
```

8.2.5 messages




POST {{BASE_URL}} /messages/6901c1c27f77c9f15d6b1cf7

```
{
    "text":"first message hello"
}
```

Chapter 9: Testing & Quality Assurance

9.1. Test Strategy

We employed **Manual Black-Box Testing**. Each feature was tested against the user stories defined in Chapter 4.

9.2. Test Cases

Authentication Tests

ID	Test Scenario	Test Steps	Expected Result	Status
TC01	Valid Registration	1. Go to /register 2. Enter valid unique data 3. Submit	Redirect to login page. DB shows new user.	**PASS**

TC02	Duplicate Email	1. Register with existing email	Alert: "User already exists".	**PASS**
TC03	Valid Login	1. Enter correct credentials	Redirect to Home. Cookie set.	**PASS**
TC04	Protected Route	1. Logout 2. Try to access /profile	Redirect to login page.	**PASS**

Functionality Tests

ID	**Test Scenario**	**Test Steps**	**Expected Result**	**Status**
TC05	Search Filter	1. Select City: London 2. Select Type: Buy	List shows only buying options in London.	**PASS**
TC06	Save Post	1. Open Post 2. Click Bookmark icon	Button turns yellow/black. Post added to Profile.	**PASS**
TC07	Chat Real-time	1. User A sends msg 2. Check User B screen	Message appears instantly without refresh.	**PASS**

TC08	Update Profile	1. Change Avatar URL 2. Save	Navbar avatar updates immediately.	**PASS**

Chapter 10: Future enhancements

10.1. Future Enhancements

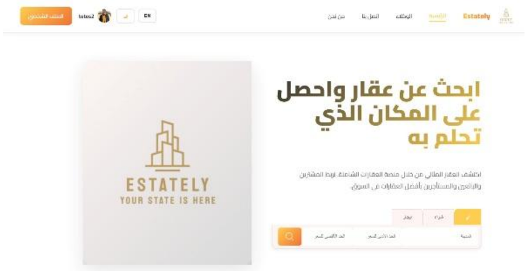
- **AI Recommendations:** Suggest properties based on user browsing history.
- **Mortgage Calculator:** Add a financial tool for buyers.
- **Virtual Tours:** Support 360-degree video uploads.
- **Admin Panel:** A dedicated route for admins to ban users or delete spam posts.

Conclusion

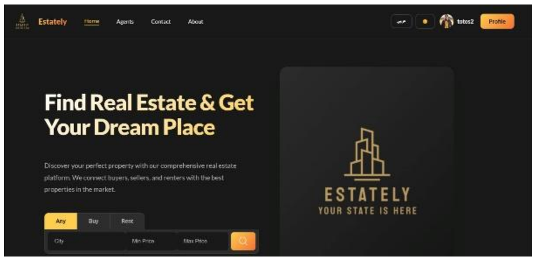
The **Estately** project represents a significant step forward in modernizing real estate transactions. By successfully implementing a full-stack solution using React, Node.js, and Socket.io, the team has delivered a platform that is not only functional but also secure, fast, and user-centric.

The integration of real-time chat and interactive mapping solves the core problems of latency and visualization identified in the market analysis. The rigorous testing phase confirmed the system's stability, and the modular architecture ensures that Estately is ready for future scalability and feature expansion. This project stands as a testament to the power of the MERN stack in building complex, real-world applications.

Screenshots for our website:

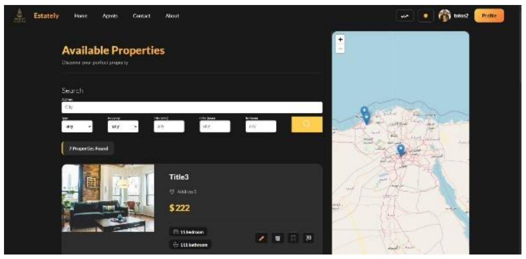


ابحث عن عقار واحصل على المكان الذي تحلم به



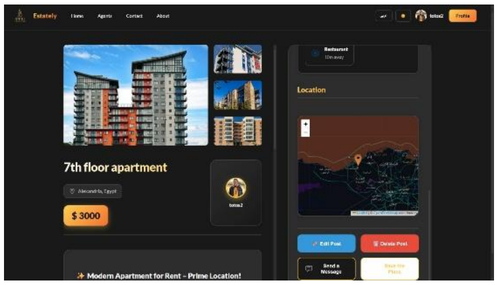
Find Real Estate & Get Your Dream Place

Discover your perfect property with our comprehensive real estate platform. We connect buyers, sellers, and renters with the best properties in the market.



Available Properties

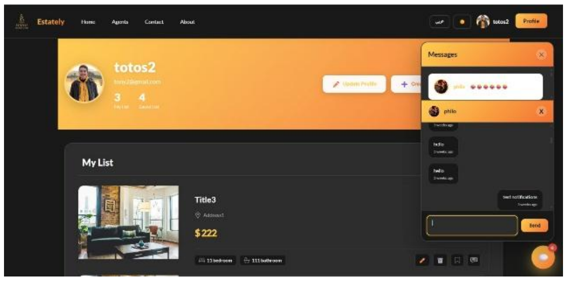
Discover your perfect property



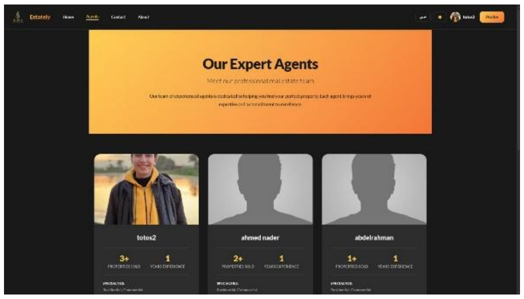
7th floor apartment

$ 3000

Modern Apartment for Rent – Prime Location!



My List



Our Expert Agents

Meet our professional real estate team