**Full Stack Development with MERN**

**Introduction**

**Project Title**

**HouseHunt: Finding Your Perfect Rental Home**

**Team Members**

**M.Reshma**

**Lanka J Siva Sai Sravan Kumar**

**L.Uma**

**Lavanya Konda**

**Team ID:LTVIP2025TMID53139**

**Purpose**

The purpose of a project titled "**House Hunt: Finding Your Perfect Rental Home**" is to streamline and enhance the rental experience for both tenants and landlords through a user-friendly digital platform. This project aims to address common challenges in the rental process by providing tools and features that facilitate efficient communication, property management, and decision-making.

**Key Objectives:**

1. **Simplify Property Search:** Offer an intuitive interface for tenants to search for rental properties based on specific criteria such as location, budget, and amenities.
2. **Enhance Landlord-Tenant Communication:** Provide channels for clear and documented interactions between landlords and tenants, fostering transparency and trust.
3. **Streamline Maintenance Requests:** Enable tenants to report maintenance issues easily, and allow landlords to manage and track these requests efficiently.
4. **Facilitate Lease Management:** Assist in the creation, storage, and management of lease agreements, ensuring all parties have access to important documents.
5. **Incorporate Future Enhancements:** Plan for the integration of features like GPS-based property listings, live chat support, and expansion to various platforms to meet evolving user needs.

By focusing on these objectives, the project seeks to create a comprehensive solution that benefits both renters and property owners, making the rental process more efficient and user-centric.

**Features**

Here are the key features that “**House Hunt: Finding Your Perfect Rental Home**” include to help users efficiently find and secure their ideal rental:

**🏡 1. Location-Based Search & Neighbourhood Insights**

* Filter properties by proximity to work, schools, transit, groceries, parks, safety, etc.
* Provide neighborhood profiles—public transport options, crime stats, vibe, nearby amenities.

notes and follow-up reminders. **2. Customizable Checklists: Must-haves, Nice-to-haves & Deal-breakers**

* Let users rank features (e.g., 2 BHK, balcony, pet-friendly, parking)
* Clearly flag deal-breakers to quickly filter out unsuitable listings.

**3. Amenity Filters & Personalization**

* Include in-unit laundry, parking, dishwasher, AC, storage space.
* Highlight renter-preferred perks like pet zones, air filtration, coworking/social areas.

**4. Virtual Tours & In-Person Inspection Tools**

* 3D or VR tour integration to pre-screen listings.
* Inspection checklist (appliances, outlets, plumbing, structural issues).

**5. Safety & Legal Compliance**

* Check secure locks, cameras, well-lit grounds.
* Guidance or reminders for lease registration, tenant rights, deposits based on local rules.

**6. Budgeting & Cost Breakdown**

* Track rent, utilities (included vs. separate), parking fees, insurance.
* Provide tools to estimate total monthly cost vs. budget.

**7. Lease & Landlord Coordination**

* Compare lease lengths (6‑month, 12‑month, flexibility).
* Built-in messaging for maintenance requests and document management

**8. Community Insights & Social Features**

* Show on-site amenities (gyms, lounges, pet areas, events like happy hours).
* Reviews or feedback from past/current tenants

**9. Negotiation Support & Market Data**

* Provide average rent comps for similar properties to support negotiation.
* Suggested negotiation strategies for lease renewal and timing based on vacancy trends.

**10. Organization & Alerts**

**.** Automated alerts when new listings match your criteria

* Save, compare, tag favorites; maintain viewing

**Description**

Here’s a smooth, narrative-style description of HouseHunt:

HouseHunt is a full-stack application, crafted to simplify how tenants and landlords connect over rental properties. Built on the MERN stack—React for the frontend, Node.js and Express for server-side logic, and MongoDB for data storage—it allows owners to upload house details including images, videos, rent, amenities, and geographic boundaries known as geofences. These geofences are drawn by landlords using Radar.io, while the frontend integrates Google Maps to present listing locations visually.

Renters can explore active listings using filters like budget, amenities, location, and indicates area risk levels . Favoriting listings, submitting booking requests, and real-time chatting with landlords are all supported, fostering a responsive and user-friendly experience.

Development challenges include integrating geofencing with address geocoding and handling the mapping system. Future plans include enabling tenant reviews, adding user verification, and expanding neighborhood insights.

In essence, HouseHunt is a cloud-enabled, map-integrated rental marketplace where creating and searching listings, applying filters, messaging in real time, and handling secure authentication come together in one seamless full-stack platform.

**Scenario**

Here are three user scenarios that illustrate how HouseHunt works end-to-end—showcasing its core flows for tenants, landlords, and admins:

**🧑‍💼 Scenario 1: Tenant Searching and Booking**

Priya, a software engineer, wants a 2 BHK near her office in Secunderabad under ₹25,000/month. She logs in, applies filters (location radius, budget, pet-friendly), and views listings on a Google Map. She clicks on a suitable property, checks images, details, and geofence info drawn by the landlord. She favorites it and sends a booking request. The landlord receives the alert, reviews her profile, and approves. Priya then initiates a real-time chat to coordinate a visit and digital lease signing.

**🏠 Scenario 2: Landlord Listing with Geofence**

Ravi, a landlord, registers and uploads a listing—adding photos, rent amount, 3 BHK details, amenities, and sets a geofence around the property using Radar.io. This geofence shows up as a highlighted zone on the map for renters. Once he publishes the listing, it passes admin review and goes live. Ravi then receives booking inquiries, reviews tenant profiles, and chats with interested parties through the app.

**👨‍💻 Scenario 3: Admin Management**

Meena, an admin, accesses the dashboard to review new listings. She sees Ravi’s recent submission, checks photos and location details, and approves it. Later, she monitors bookingand can deactivate listings or block users if policy violations occur.

These scenarios capture the real-world workflows enabled by HouseHunt’s full-stack design: from user authentication and listing creation to geolocation, booking approval, and live chat communication—all under an admin-governed environment.

**Technical Architecture**

Solution architecture for a "house-hunt" application will vary depending on the specific features and scale you envision, but I can provide a comprehensive overview covering common components and considerations.

Here's a breakdown of a potential solution architecture, using a modern, scalable approach:

Solution Architecture for a House-Hunt Application

1. Overview & Goals:

The goal of this architecture is to provide a robust, scalable, and user-friendly platform for individuals to search, discover, and potentially manage properties for rent or sale. Key considerations include:

* Scalability: Handling a growing number of users and property listings.
* Performance: Fast search results and responsive user interface.
* Reliability: High availability and data integrity.
* Security: Protecting user data and property information.
* Extensibility: Easy to add new features (e.g., virtual tours, AI recommendations).
* Maintainability: Simple to manage and update.

2. Key Components & Technologies:

a. Client Applications:

* Web Application:
  + Technologies: React, Angular, Vue.js (for Single Page Application - SPA)
  + Deployment: Static site hosting (e.g., AWS S3 + CloudFront, Netlify, Vercel).
* Mobile Applications:
  + Technologies: React Native, Flutter (cross-platform) or native (Swift/Kotlin for iOS/Android).

b. Content Delivery Network (CDN):

* Purpose: To cache static assets (images, CSS, JavaScript) closer to the users, reducing latency and improving load times.
* Technologies: AWS CloudFront, Cloudflare, Akamai.

c. API Gateway:

* Purpose: Acts as a single entry point for all client requests. Handles request routing, authentication, rate limiting, and API versioning.
* Technologies: AWS API Gateway, Kong, Apigee, NGINX Plus.

d. Authentication and Authorization:

* Purpose: Manages user registration, login, session management, and access control.
* Technologies:
  + Managed Services: AWS Cognito, Auth0, Firebase Authentication.
  + Self-hosted: OAuth 2.0, OpenID Connect with services like Keycloak.

e. Backend Microservices:

This is the core business logic, broken down into smaller, independent services for better scalability, maintainability, and fault isolation.

* Property Service:
  + Functionality: CRUD operations for property listings (add, edit, delete, view), property details (photos, descriptions, amenities, price, location).
  + Technologies: Node.js (Express), Python (Django/Flask), Java (Spring Boot), Go.
* User Service:
  + Functionality: User profiles, saved searches, favorite properties, contact information.
  + Technologies: Same as Property Service.
* Search/Indexing Service:
  + Functionality: Advanced search capabilities (filters by price, location, type, amenities), full-text search, geo-spatial search. This is crucial for a house-hunt application.
  + Technologies: Elasticsearch, Apache Solr.
* Notification Service:
  + Functionality: Sending email alerts (new listings, price changes), SMS notifications, push notifications to mobile apps.
  + Technologies: AWS SNS, SendGrid, Twilio, Firebase Cloud Messaging.
* Image/File Storage Service:
  + Functionality: Storing and serving property images, documents (e.g., floor plans).
  + Technologies: AWS S3, Google Cloud Storage, Azure Blob Storage.
* Messaging/Queueing Service:
  + Purpose: Decoupling services, handling asynchronous tasks (e.g., image processing, sending notifications).
  + Technologies: Apache Kafka, AWS SQS, RabbitMQ.
* Payments Service (Optional, if transactions are involved):
  + Functionality: Handling payments for premium listings, agent subscriptions, etc.
  + Technologies: Stripe, PayPal, Braintree APIs.
* Analytics/Logging Service:
  + Purpose: Collecting application logs, user behavior data, and performance metrics for monitoring, debugging, and insights.
  + Technologies: ELK Stack (Elasticsearch, Logstash, Kibana), Grafana, Prometheus, AWS CloudWatch, Google Cloud Logging.

f. Database Layer:

* Property Database:
  + Type: Relational Database (SQL) for structured data and strong consistency.
  + Technologies: PostgreSQL, MySQL, Amazon RDS, Google Cloud SQL.
* User Database:
  + Type: Can be Relational (for user profiles, preferences) or NoSQL (for more flexible schema).
  + Technologies: PostgreSQL, MongoDB (for flexible profiles), DynamoDB.
* Search Index Database:
  + Type: Specialized for fast full-text and complex queries.
  + Technologies: Elasticsearch (typically used for the Search/Indexing Service).
* Geolocation Database (for advanced geo-spatial queries):
  + Technologies: PostGIS (extension for PostgreSQL).

g. Caching Layer:

* Purpose: Stores frequently accessed data in-memory to reduce database load and improve response times.
* Technologies: Redis, Memcached, Amazon ElastiCache.

h. Monitoring & Logging:

* Purpose: To observe the health and performance of the application, detect issues, and troubleshoot.
* Technologies: Prometheus & Grafana, ELK Stack (Elasticsearch, Logstash, Kibana), Datadog, New Relic, AWS CloudWatch.

i. CI/CD (Continuous Integration/Continuous Deployment):

* Purpose: Automates the build, test, and deployment process, enabling faster and more reliable releases.
* Technologies: Jenkins, GitLab CI/CD, GitHub Actions, AWS CodePipeline, CircleCI.

j. Infrastructure as Code (IaC):

* Purpose: Manages and provisions infrastructure through code, ensuring consistency and repeatability.
* Technologies: Terraform, AWS CloudFormation, Ansible.

3. Deployment Strategy:

* Cloud-Native: Leverage cloud providers like AWS, Google Cloud Platform (GCP), or Microsoft Azure for their managed services, scalability, and global reach.
* Containerization: Docker for packaging applications and their dependencies.
* Orchestration: Kubernetes (EKS, GKE, AKS) for deploying, scaling, and managing containerized applications. This provides self-healing, load balancing, and automated rollouts.

4. Scalability Considerations:

* Horizontal Scaling: Add more instances of stateless services as demand grows.
* Database Sharding/Read Replicas: Distribute database load across multiple servers.
* Caching: Reduce database hits.
* Message Queues: Handle spikes in requests and decouple services.
* CDN: Offload static content delivery.

5. Security Considerations:

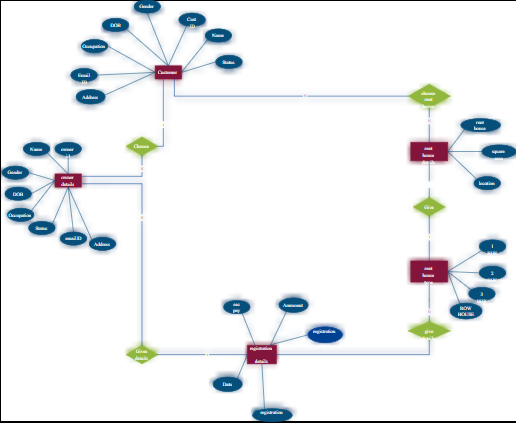
* HTTPS/SSL: Encrypt all communication.
* Authentication & Authorization: Strong user authentication and granular access control.
* Data Encryption: Encrypt data at rest and in transit.
* Input Validation: Prevent injection attacks (SQL, XSS).
* API Security: Rate limiting, API keys, OAuth.
* Regular Security Audits: Penetration testing, vulnerability scanning.
* Principle of Least Privilege: Grant only necessary permissions to services and users.

6. Future Enhancements:

* AI/ML for Recommendations: Suggesting properties based on user preferences and behavior.
* Virtual Tours/AR Integration: Immersive property viewing experiences.
* Chatbots/AI Assistants: For quick queries and support.
* Agent/Broker Portal: Dedicated features for real estate professionals.
* CRM Integration: For lead management and follow-ups.

This comprehensive solution architecture provides a solid foundation for building a robust and scalable house-hunt application.

**ER Diagram**

****

* User: Represents both renters and landlords  
  *Attributes*: UserID, Name, Email, Phone, Role (Renter/Landlord), PasswordHash
* Property: Rental listings created by landlords  
  *Attributes*: PropertyID, Title, Description, Address, Latitude, Longitude, Rent, Amenities  
  – One Landlord (User) can own multiple Properties (1:M)
* Geofence: Spatial boundaries used for map filtering  
  *Attributes*: GeofenceID, PropertyID (FK), PolygonData  
  – Each Property can have one geofence (1:1)
* Lease/Booking: Tenant's request or booking record  
  *Attributes*: LeaseID, PropertyID (FK), TenantID (FK from User), StartDate, EndDate, Status  
  – One Property can have multiple Leases, one Tenant can have many leases (M:1)
* ChatMessage: Real-time communication  
  *Attributes*: MessageID, LeaseID (FK), SenderID (FK), MessageText, Timestamp  
  – One Lease can have multiple ChatMessages (1:M)
* Review (Future feature): Ratings for properties or landlords  
  *Attributes*: ReviewID, PropertyID (FK), TenantID (FK), Rating, Comment, Date  
  – Many Reviews per Property (1:M)
* MaintenanceRequest: Tenant-submitted issues  
  *Attributes*: RequestID, PropertyID (FK), TenantID (FK), Description, Status, CreatedAt  
  – One Property can have multiple MaintenanceRequests (1:M)

This schema covers essential HouseHunt flows:

* Landlords create Properties and associate Geofences for map filtering.
* Tenants search based on location/amenities, then initiate a Lease request.
* ChatMessages are tied to each Lease for real-time coordination.
* Optional Reviews and MaintenanceRequests add user feedback and support.

**Setup Phase**

**Pre-Requisites**

🧩 Core Skills

* HTML & CSS: Strong understanding of markup, responsive design, Flexbox/Grid layout, media queries and forms
* JavaScript (ES6+): Mastery of variables, functions, arrow functions, promises, async/await, modular code
* JavaScript fundamentals: Solid grasp of data types, arrays, objects, closures, DOM manipulation

🔄 Full-Stack Development

* React.js: Component-based UI, hooks (useState, useEffect), routing, state management with Context or Redux
* Node.js and Express: Building RESTful APIs, middleware, routing, error handling, async programming
* MongoDB + Mongoose: Schema modeling, CRUD operations, indexing, aggregation pipelines

🧰 Tools & Fundamental Practices

* Git & GitHub: Version control, branching, collaborating
* Command Line: Unix shell commands (ls, cd, etc.) for navigating and automation
* Environment & Config: NPM/Yarn, .env management, structured project setup

🧠 Problem-Solving & Data Structures

* Practice algorithms and data structures (arrays, maps, recursion) to write efficient.

✅ Summary Checklist

By mastering:

* Web fundamentals (HTML, CSS, JS)
* Frontend (React + state management)
* Backend (Node/Express, RESTful design)
* Database (MongoDB with schema modeling)
* Tools (Git, CLI, deployment platforms)
* Optional: Authentication, real-time chat, file storage

.

Here are the key prerequisites for developing a full-stack application using Node.js, Express.js, MongoDB, React.js:

**Node.js and npm:**

Node.js is a powerful JavaScript runtime environment that allows you to run JavaScript code on the server-side. It provides a scalable and efficient platform for building network applications.

Install Node.js and npm on your development machine, as they are required to run JavaScript on the server-side.

Download[: https://nodejs.org/en/download/](https://d.docs.live.net/ee4bfc48f225ef35/Documents/:%20https:/nodejs.org/en/download/ )

Installation instructions: [https://nodejs.org/en/download/package-manager/](https://nodejs.org/en/download/package-manager/ )

**Express.js:**

Express.js is a fast and minimalist web application framework for Node.js. It simplifies the process of creating robust APIs and web applications, offering features like routing, middleware support, and modular architecture.

Install Express.js, a web application framework for Node.js, which handles server-side routing, middleware, and API development.

Installation: Open your command prompt or terminal and run the following command:

**npm install express**

**MongoDB:**

MongoDB is a flexible and scalable NoSQL database that stores data in a JSON-like format. It provides high performance, horizontal scalability, and seamless integration with Node.js, making it ideal for handling large amounts of structured and unstructured data.

Set up a MongoDB database to store your application's data.

Download: [https://www.mongodb.com/try/download/community](https://www.mongodb.com/try/download/community )

Installation instructions: [https://docs.mongodb.com/manual/installation/](https://docs.mongodb.com/manual/installation/ )

**React.js:**

React.js is a popular JavaScript library for building user interfaces. It enables developers to create interactive and reusable UI components, making it easier to build dynamic and responsive web applications.

Install React.js, a JavaScript library for building user interfaces.

Follow the installation guide: <https://reactjs.org/docs/create-a-new-react-app.html>

**HTML, CSS, and JavaScript**: Basic knowledge of HTML for creating the structure of your app, CSS for styling, and JavaScript for client-side interactivity is essential.

**Database Connectivity**: Use a MongoDB driver or an Object-Document Mapping (ODM) library like Mongoose to connect your Node.js server with the MongoDB database and perform CRUD (Create, Read, Update, Delete) operations. To Connect the Database with Node JS go through the below provided link:

[https://www.section.io/engineering-education/nodejs- mongoosejs-mongodb/](https://www.section.io/engineering-education/nodejs-%20mongoosejs-mongodb/ )

**Front-end Framework**: Utilize Reactjs to build the user-facing part of the application, including entering complaints, status of the complaints, and user interfaces for the admin dashboard.

For making better UI we have also used some libraries like material UI and boostrap.

**Version Control**: Use Git for version control, enabling collaboration and tracking changes throughout the development process. Platforms like GitHub or Bitbucket can host your repository.

Git: Download and installation instructions can be found at:

[https://git-scm.com/downloads](https://git-scm.com/downloads )

**Development Environment**: Choose a code editor or Integrated Development Environment (IDE) that suits your preferences, such as Visual Studio Code, Sublime Text, or WebStorm.

• Visual Studio Code: Download from <https://code.visualstudio.com/download>

To run the existing Video Conference App project downloaded from GitHub:

Follow below steps:

Clone the Repository:

* Open your terminal or command prompt.
* Navigate to the directory where you want to store the e-commerce app.
* Execute the following command to clone the repository:

**Git clone :<https://github.com/Mallampallireshma/House-Hunt.git>**

Install Dependencies:

• Navigate into the cloned repository directory:

cd complaint-registery

• Install the required dependencies by running the following commands:

cd client

npm install

cd server

npm install

Start the Development Server:

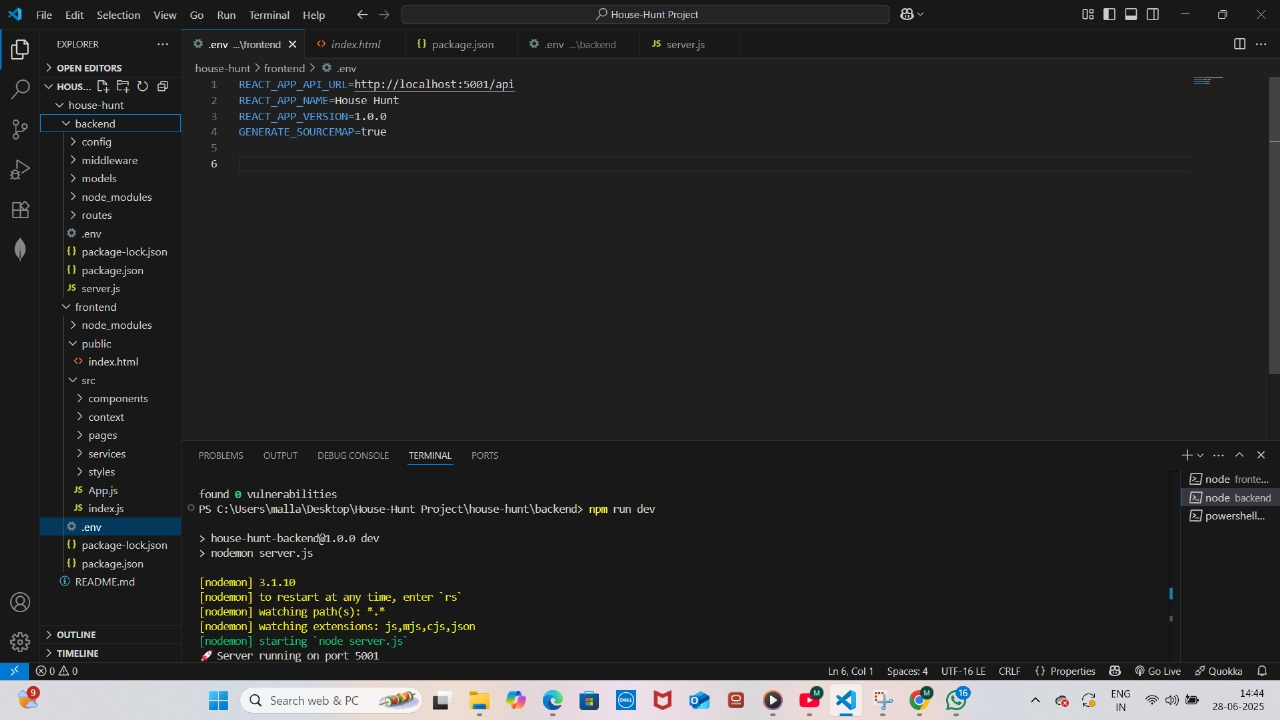
• To start the development server, execute the following command:

npm start

• **House-Hunt**: finding your perfect rental home will be accessible at

<http://localhost:3000>

**project structure**

****

**Application Flow**

**🧭 1.User Journey Map**

Based on typical renter behavior

Discovery Phase: User lands on homepage → sees clear CTAs like “Start Your Search” → enters basic criteria (location, budget).

* Exploration Phase: User browses property listings with map overlay, filters, and hover details → narrows down favorites.
* Engagement Phase: User views a listing → checks images, amenities, geofence, and booking options → sends inquiry or booking request.
* Interaction Phase: Tenant and landlord connect via real-time chat → schedule a visit and sign lease.
* Closing Phase: User confirms booking; admin approves listing → workflow completes.

**🔄 2. Task Flow for Booking a Property**

[Homepage] → [Search Form] → [Results List/Map] → [Select Property]

→ [Property Detail] → [Click “Request Booking”] → [Filled Booking Form]

→ [Submit] → [Booking Confirmation]

**‍API Documentation**

**Base URL**: <http://localhost:3000>

General

Get /-check if server is working.

Response:{“message”: ”working”}

**Project Flow**

Use the code in:[**https://github.com/Mallampallireshma/House-Hunt.git**](https://github.com/Mallampallireshma/House-Hunt.git)

Or follow the videos below for better understanding.

**Milestone 1:**

**Project Setup and Configuration:**

1. **Create project folders and files:**

Now, firstly create the folders for frontend and backend to write the respective code and install the essential libraries.

* + Client folders.
  + Server folders

1. **Install required tools and software:**

For the backend to function well, we use the libraries mentioned in the prerequisites. Those libraries includes

* + Node.js.
  + MongoDB.
  + Bcrypt
  + Body-parser

Also, for the frontend we use the libraries such as

* + React Js.
  + Material UI
  + Bootstrap
  + Axios

**Milestone 2**:

**Backend Development:**

* **Set Up Project Structure:**
* Create a new directory for your project and set up a package.json file using npm init command.
* Install necessary dependencies such as Express.js, Mongoose, and other required packages.
* **Set Up Project Structure:**
  + Create a new directory for your project and set up a package.json file using npm init command.
  + Install necessary dependencies such as Express.js, Mongoose, and other required packages.
* **Create Express.js Server:**
  + Set up an Express.js server to handle HTTP requests and serve API endpoints.
  + Configure middleware such as body-parser for parsing request bodies and cors for handling cross-origin requests.
* **Define API Routes:**
  + Create separate route files for different API functionalities such as authentication, stock actions, and transactions.
  + Implement route handlers using Express.js to handle requests and interact with the database.
* **Implement Data Models:**
  + Define Mongoose schemas for the different data entities like Bank, users, transactions, deposits and loans.
  + Create corresponding Mongoose models to interact with the MongoDB database.
  + Implement CRUD operations (Create, Read, Update, Delete) for each model to perform database operations.
* **User Authentication:**
  + Implement user authentication using strategies like JSON Web Tokens (JWT) or session-based authentication.
  + Create routes and middleware for user registration, login, and logout.
  + Set up authentication middleware to protect routes that require user authentication.

**Milestone 3:**

**Database development:**

**1. Conceptual Modeling & Entities**

Start with identifying your core entities and relationships — the fundamental pieces of your system:

* **User** (landlords, tenants, admins)
* **Property** (listings with address, rent, amenities)
* **Geofence** (spatial boundary tied to a property)
* **Booking/Lease** (tenant's request, dates, status)
* **Payment** (amount, date, method)
* **ChatMessage** (real-time messages tied to a booking)
* **MaintenanceRequest** (tenant-initiated issue tracking)
* **Review** (tenant feedback on property/landlord)

**2. Logical Design & Relationships**

Translate the ERD into tables or collections:

* **Properties** reference **Users** (landlords)
* **Bookings** reference both **Property** and **User** (tenant)
* **Payments** belong to a **Booking**
* **ChatMessages** link to **Booking** (or a thread with participant IDs)
* **MaintenanceRequests** connect Property and Tenant
* **Geofence**: one-to-one with a Property
* **Reviews** are linked to Property and Tenant

**3. NoSQL Schema (e.g., MongoDB)**

Align with MERN stack:

* Collections: users, properties, bookings, etc.
* Embed simple sub-documents (like amenities, images) within properties.
* Use references for relationships (propertyID, tenantID).
* Employ hybrid modeling: embed small arrays (e.g., recent messages) and normalize bulk data (like bookings).

**5. Normalization, Constraints & Validation**

* Store data normalized to prevent redundancy (1NF–3NF)
* Use constraints like NOT NULL, UNIQUE, FK, and enum/enum-like fields.
* In MongoDB, enforce schema rules via Mongoose and validate on insert/update.
* Consider soft deletion with isActive flags rather than dropping records

**6. Data Integrity & Transactions**

* Use ACID transactions for multi-step operations (e.g., booking + payment).
* In MongoDB, wrap related document updates in transactions; in SQL, utilize database transactions.

**7. Indexing & Performance**

* Add indices on foreign keys (propertyID, tenantID), sorting fields (startDate), and geospatial fields (location, geofence) for fast queries.
* Employ caching (Redis) for frequent reads like active listings or popular properties.

**8. Evolutionary Design & Regression Safety**

* Adopt evolutionary schema design: plan small refactors, migrations via tools like migrate-mongo or SQL migrations.
* Add regression tests to ensure that updates do not break existing functionality

**Milestone 4:**

🚀 **1. Initialize Your Project**

Use Vite – a fast, modern alternative to CRA:

npm create vite@latest house-hunt-frontend -- --template react-ts

cd house-hunt-frontend

npm install

This sets up a React + TypeScript starter with lightning‑fast hot-reload behavior.

🗂 **2. Organize Your Folder Structure**

Create a scalable, feature-oriented structure:

src/

├── assets/ # images, icons, fonts

├── components/ # generic UI (Button, Modal, Input)

├── features/ # domain features (auth/, listings/, chat/)

├── hooks/ # reusable logic (useAuth, useMap)

├── pages/ # route-level views (Home, Login, ListingDetail)

├── services/ # API calls (axios wrappers)

├── store/ # global state (Redux or Context)

├── utils/ # helpers (formatDate, validators)

├── App.tsx

└── main.tsx

This aligns with recommended patterns for maintainable React apps.

**🧠 3. Implement Best Practices**

* Functional Components & Hooks: Use function components with hooks for cleaner, testable code .
* Reusable Components: Keep components small, focused, and reusable—for example an InputField or ListingCard
* Absolute Imports: Configure jsconfig.json to avoid complex relative paths:

{ "compilerOptions": { "baseUrl": "src" }, "include": ["src"] }

* Styling: Use CSS-in-JS (e.g., Styled Components/Emotion) for modular theming [reddit.com+12reddit.com+12reddit.com+12](https://www.reddit.com/r/react/comments/ngvrii?utm_source=chatgpt.com).
* Linting & Formatting: Set up ESLint + Prettier to maintain consistent code style

🔒 4. State & API Management

* State Management: Use Context for auth/global state, and toolkits like Redux or React Query for data fetching.
* API Layer: Create a services/apiClient.ts to wrap axios calls and centralize endpoints.

🧪 5. Testing Setup

Use Jest + React Testing Library for component and logic tests:

npm install -D jest @testing-library/react @testing-library/jest-dom

⚙️ 6. Environment & Deployment

* Create .env files to manage variables (e.g., VITE\_API\_BASE\_URL) and add them
* Deploy on Vercel, Netlify, or Firebase Hosting—they offer seamless React support

🔄 7. Developer Workflow

* Commit Strategy: Use small, clear commits and PRs via GitHub.
* CI/CD: Automatically lint, type-check, test, and deploy on pushes using GitHub Actions.
* Pull Requests: Enforce code reviews and tests before merging to main.

✅ Quick Start Checklist

1. Set up with Vite and TypeScript.
2. Scaffold folders: components, features, pages, hooks, services, store, etc.
3. Write small, focused components using hooks.
4. Use absolute imports and meaningful naming.
5. Add ESLint, Prettier, and testing tools.
6. Create global state and API layers.
7. Add environment configurations and deploy.

**Milestone 5:**

**Project implementation**

