**MessFinder System Design**

**1. System Architecture**

* **Overview**: MessFinder employs a **client-server architecture** with a React front-end for user interaction, a Node.js/Express back-end for API logic, and Firebase for authentication, data storage, and file uploads. The design prioritizes simplicity to meet the 1-month deadline, leveraging Firebase’s managed services to minimize setup and maintenance.
* **Components**:
  + **Client**: React web app providing interfaces for owners (post/edit listings) and users (search, view, contact).
  + **Server**: Node.js with Express handles APIs for search and messaging, complementing Firebase’s direct CRUD operations.
  + **Database**: Firebase Firestore stores listings, users, and inquiries in a NoSQL structure.
  + **Storage**: Firebase Storage manages room photos.
  + **Authentication**: Firebase Authentication secures owner and user logins.
* **Architecture Diagram** (text-based):

text

Copy

[Owner/User] --> [React Front-end]

|

v

[Node.js/Express APIs] --> [Firebase: Firestore, Storage, Auth]

* **Data Flow**:
  + Owners/users interact with the React UI (e.g., post listing, search rooms).
  + React sends requests to Express APIs or directly to Firebase (via SDK).
  + Express processes complex queries (e.g., search filters) and interacts with Firestore.
  + Firebase handles authentication, stores data, and serves photos.
  + Responses are rendered in the UI (e.g., search results, listing details).

**2. Tech Stack**

* **Front-end**: React.js
  + Reason: Component-based, dynamic UI, beginner-friendly, integrates well with Firebase.
  + Libraries: Axios (API calls), Tailwind CSS (responsive styling), React Router (navigation).
* **Back-end**: Node.js + Express
  + Reason: Lightweight, JavaScript-based, handles custom API logic (e.g., search filters).
  + Libraries: firebase (SDK), cors, dotenv.
* **Database**: Firebase Firestore
  + Reason: NoSQL, real-time updates, managed service, simplifies CRUD operations.
* **Storage**: Firebase Storage
  + Reason: Easy photo uploads, free tier sufficient for demo (1–5 images per listing).
* **Authentication**: Firebase Authentication
  + Reason: Secure, email/password login, quick setup.
* **Hosting**: Vercel or Firebase Hosting
  + Reason: Free, easy deployment for web app.
* **Tools**:
  + Git/GitHub: Version control.
  + Figma: UI mockups (by non-developers).
  + Postman: API testing.
  + VS Code: Development IDE.

**3. Database Schema (Firebase Firestore)**

Firestore uses a NoSQL structure with collections and documents. The schema is designed for efficient storage of user data, room listings, and inquiries.

* **Users Collection** (owners and users):

json

Copy

{

"uid": "user123", *// Firebase Auth UID*

"email": "user@example.com",

"role": "owner" | "user", *// Differentiates owners and users*

"name": "John Doe",

"phone": "1234567890"

}

* **Listings Collection** (room details):

json

Copy

{

"id": "listing456", *// Auto-generated Firestore ID*

"owner\_id": "user123", *// Links to Users collection*

"location": {

"state": "Karnataka",

"district": "Bangalore Urban",

"city\_area": "Koramangala",

"pin": "560034"

},

"room\_details": {

"beds": 3,

"desks": 3,

"capacity": "3-person"

},

"amenities": ["Wi-Fi", "Water", "Electricity"],

"food": {

"available": true,

"type": "Home-cooked",

"price": "Included" | 2000

},

"price": 5000, *// Monthly rent per person (INR)*

"photos": ["storage\_url1", "storage\_url2"], *// Firebase Storage URLs*

"created\_at": "2025-05-01" *// Timestamp*

}

* **Inquiries Collection** (user-owner communication):

json

Copy

{

"id": "inquiry789", *// Auto-generated ID*

"listing\_id": "listing456",

"user\_id": "user789",

"message": "Is the room available next month?",

"created\_at": "2025-05-01"

}

* **Notes**:
  + Listings are auto-published (no admin panel).
  + Location fields are populated from a JSON file (sourced by non-developers) for dropdowns.
  + Photos are stored in Firebase Storage, with URLs saved in Firestore.

**4. UI Structure**

The UI is built with React components, designed to be intuitive and mobile-responsive using Tailwind CSS. The structure is streamlined for the 1-month timeline.

* **Pages**:
  + **Home**: Landing page with search bar (dropdowns) and “Login/Register” links.
  + **Owner Dashboard**:
    - Form to post/edit listings (dropdowns for location, inputs for room details, file upload for photos).
    - Table listing owner’s postings (edit/delete buttons).
    - Inquiry view (messages from users).
  + **User Search**:
    - Search form with dropdowns (State, District, City/Area, PIN) and filters (price, room type, food).
    - Results grid/list (thumbnail, price, location).
  + **Listing Details**:
    - Full details (photos, amenities, food, price).
    - Contact button (sends inquiry or shows owner’s phone/email).
  + **Auth Pages**: Login/register forms (Firebase Authentication).
* **Components**:
  + Navbar: Navigation (Home, Owner Dashboard, Search, Login/Logout).
  + ListingForm: Owner’s form for posting/editing.
  + SearchForm: Dropdowns and filters for users.
  + ListingCard: Displays listing summary in search results.
  + ListingDetails: Full listing view with contact button.
* **Responsive Design**:
  + Tailwind CSS ensures mobile-friendly layouts (e.g., grid adjusts to single-column on phones).
  + Dropdowns are touch-friendly for usability.

**5. Workflow**

* **Owner Workflow**:
  1. Register/login via Firebase Authentication.
  2. Access dashboard, fill out listing form (location dropdowns, room details, photo upload).
  3. Submit listing (saved to Firestore, photos to Storage).
  4. View/edit/delete listings in a table.
  5. Receive and view user inquiries (messages or email).
* **User Workflow**:
  1. Browse as guest or login/register.
  2. Use search page to select location (State, District, etc.) and filters (price, food).
  3. View results in a grid, click for details.
  4. Contact owner via inquiry form (saved to Firestore) or displayed contact info.
* **System Workflow**:
  1. React UI sends requests (e.g., post listing, search) to Express APIs or Firebase SDK.
  2. Express handles complex queries (e.g., multi-field search) using Firestore queries.
  3. Firebase Authentication secures access; Firestore stores data; Storage handles photos.
  4. Responses are rendered in the UI (e.g., search results, listing details).

**6. API Endpoints (Node.js/Express)**

Express APIs complement Firebase’s direct CRUD operations for custom logic. Key endpoints include:

* **POST /api/listings**: Create a listing (owner).
  + Input: Listing data (location, room details, etc.).
  + Action: Saves to Firestore, returns listing ID.
* **GET /api/listings/:id**: Fetch a listing by ID (user details page).
  + Output: Listing data with photos.
* **PUT /api/listings/:id**: Update a listing (owner).
  + Input: Updated fields.
  + Action: Updates Firestore document.
* **DELETE /api/listings/:id**: Delete a listing (owner).
  + Action: Removes from Firestore.
* **GET /api/search**: Search listings by location and filters.
  + Query params: state, district, city\_area, pin, price\_min, price\_max, room\_type.
  + Output: Array of matching listings.
* **POST /api/inquiries**: Send an inquiry (user).
  + Input: Listing ID, user ID, message.
  + Action: Saves to Firestore.
* **GET /api/inquiries/:owner\_id**: Fetch inquiries for an owner.
  + Output: Array of inquiries.

**7. Security and Performance**

* **Security**:
  + Firebase Authentication secures logins (email/password).
  + Firebase Security Rules restrict access:
    - Owners can only post/edit their listings.
    - Users can read listings but only write inquiries.
  + Input validation in Express prevents invalid data (e.g., XSS).
* **Performance**:
  + Firestore queries optimized for small datasets (100–200 listings for demo).
  + Lazy loading for photos (React).
  + Tailwind CSS minimizes CSS bundle size.
* **Scalability**:
  + Firebase’s free tier supports 500 users and 200 listings.
  + Express APIs can scale with load balancers if needed (beyond demo scope).

**8. External Dependencies**

* **Location Data**: JSON file with India’s State/District/City/PIN data (sourced by non-developers from GitHub or government portals).
  + Example structure:

json

Copy

[

{

"state": "Karnataka",

"districts": [

{

"name": "Bangalore Urban",

"cities": ["Koramangala", "Indiranagar"],

"pins": ["560034", "560038"]

}

]

}

]

* + Stored client-side or in Firestore for dropdown population.
* **Firebase Free Tier**:
  + Sufficient for demo (1GB Storage, 10GB Firestore reads/writes).
  + Requires Firebase project setup with API keys.

**Notes for Implementation**

* **Simplicity**: Firebase handles auth, CRUD, and storage, reducing back-end complexity. Express is used only for search and inquiries.
* **Team Alignment**: Non-developers provide location data and mockups, integrating directly into dropdowns and UI.
* **1-Month Feasibility**: No admin panel reduces effort by ~20%, focusing on owner/user flows. React and Firebase enable rapid development.
* **Extensibility**: Future features (e.g., admin panel, geolocation) can be added post-demo.