# 

# 

Submitted in fulfillment of the requirements

for the course

**Full Stack Development (FSD)**

By

**Kishita Ahuja - Roll No (13)**

**Monica Ahuja - Roll No(14)**

Under the guidance of

**Mrs. Pooja Prajapati**



Department of Information Technology

VIVEKANAND EDUCATION SOCIETY’S INSTITUTE OF TECHNOLOGY, Chembur, Mumbai 400074

(An Autonomous Institute, Affiliated to University of Mumbai)

October 2025

### 

### **Abstract**

This report presents a comprehensive implementation of a Full Stack Development project titled 'Lost and Found Portal'. The system facilitates reporting, searching, and retrieving lost or found items within a campus environment. Built using the MERN stack (MongoDB, Express, React, Node.js), it features secure authentication (JWT), responsive design (Tailwind CSS), and real-time updates (WebSockets). The project includes ten experiments covering frontend, backend, API validation, deployment, and DevOps integration, demonstrating a full end-to-end development cycle with a focus on scalability, usability, and security.

# **Table of Contents**

1. Build Responsive and Interactive UIs using Tailwind CSS

2. Experiment based on React Hooks (useEffect, useContext, custom hooks)

3. Manage Complex State with Redux or Context API

4. REST API Design with MongoDB + Mongoose Integration

5. Create Secure, Production-Ready RESTful APIs

6. Implement Authentication and User Roles with JWT

7. Validating RESTful APIs using Postman

8. Enable Real-Time Communication via WebSockets

9. CI/CD Deployment with GitHub Actions + Render/Vercel

10. Deploy Full-Stack Apps using DevOps Tools and Docker

11. Conclusion and Project Takeaways

**1. Build Responsive and Interactive UIs using Tailwind CSS**

. In our Lost and Found Portal project, the first thing we focused on was building the user interface because that's what users see and interact with right away. We used Tailwind CSS for this, which is basically a toolkit that lets you style your web pages super fast without writing a ton of custom CSS code. Imagine you're building a house – Tailwind gives you pre-made bricks and tools to assemble it quickly instead of making everything from scratch.

For our portal, we started by setting up the basic structure in React, since that's our frontend framework. We installed Tailwind via npm and configured it in our project files, like adding it to the tailwind.config.js to customize colors and themes that match our college vibe – maybe some blues and whites for a clean look. The homepage needed to be welcoming, so we created a navbar at the top with links to 'Report Lost', 'Report Found', 'Search Items', and 'My Profile'. Using Tailwind classes like 'flex justify-between items-center bg-blue-600 text-white p-4', we made it responsive, meaning it looks good on phones too – on small screens, the menu collapses into a hamburger icon.

Then, we built the main content area. For listing lost and found items, we used cards. Each card shows an image of the item, its name, description, location where it was lost or found, and a button to claim or contact the poster. We applied classes like 'grid grid-cols-1 md:grid-cols-3 gap-4' so on desktops it shows three cards side by side, but on mobiles, they stack vertically. We added hover effects with 'hover:shadow-lg transition-shadow' to make it interactive – when you mouse over, the card pops out a bit, making it feel alive.

For forms, like reporting a lost item, we created input fields for name, description, image upload, and location. Tailwind made it easy with 'border rounded-md p-2 focus:outline-none focus:ring-2 focus:ring-blue-500'. We ensured responsiveness by testing on different devices – using media queries implicitly through prefixes like 'sm:', 'md:', etc. For example, the form fields are full-width on mobile but narrower on larger screens.

We also added modals for pop-ups, like confirming a claim, using 'fixed inset-0 flex items-center justify-center bg-black bg-opacity-50' to overlay the screen. This keeps the user focused.. We iterated based on feedback: made buttons bigger for touchscreens, added dark mode support with 'dark:' prefixes for better accessibility at night.

**2. Experiment based on React Hooks (useEffect, useContext, custom hooks)**

After setting up the UI with Tailwind, we needed to make our React components smart – that's where Hooks come in. React Hooks are like superpowers for functional components.

.But the real stars were useEffect, useContext, and custom hooks. For useEffect, think of it as a watchdog that runs code when something changes. In the item listing page, we fetched lost and found items from the backend API when the component mounts. So, we wrote useEffect(() => { async function fetchItems() { const response = await fetch('/api/items'); const data = await response.json(); setItems(data); } fetchItems(); }, []); – the empty array means it runs only once. This keeps the list up-to-date without manual refreshes.

Next, useContext for sharing data globally. In a portal like ours, user login status needs to be available everywhere – navbar shows 'Logout' if logged in, or 'Login' otherwise. We created an AuthContext with useState for the user object, and wrapped the app in <AuthProvider>. Then, in any component, we do const { user, login } = useContext(AuthContext); to access it. This avoided passing props down many levels, which can get messy in a growing app.

We tested this by adding console logs in useEffect to see fetch times, and used React DevTools to inspect context values. Performance-wise, we added dependencies to useEffect to prevent infinite loops, like depending on a search query for filtering items. For the portal, this meant smooth experiences: search for 'lost phone' and results update instantly via a useEffect on query change. Custom hooks also helped with image uploads – a useImageUpload hook handled file selection, preview, and base64 conversion before sending to backend. Overall, Hooks made our frontend dynamic and maintainable, turning static pages into a living app where data flows seamlessly, reducing bugs and making additions like notifications easier later on.

**3. Manage Complex State with Redux or Context API**

This is key for bigger apps like our portal. As the Lost and Found grew, simple useState wasn't enough – we had user auth, item lists, search filters, all needing to be shared and updated across components. That's why we used Context API (we chose it over Redux for simplicity, since our app isn't huge). Context is like a central radio station broadcasting data to whoever tunes in.

We started by identifying what state is complex: user details (id, name, role, token), list of all items (lost and found), current search results, and loading/error states for APIs. We created multiple contexts to keep things organized – AuthContext for user stuff, ItemsContext for item data. For AuthContext, we have a provider component: const [user, setUser] = useState(null); const login = async (credentials) => { const res = await fetch('/api/login', { method: 'POST', body: JSON.stringify(credentials) }); const data = await res.json(); setUser(data.user); localStorage.setItem('token', data.token); }; and similar for logout. We wrap the whole app in <AuthProvider><ItemsProvider><App /></ItemsProvider></AuthProvider>.

In ItemsContext, we manage fetching and updating items. const [items, setItems] = useState([]); const fetchItems = async () => { setLoading(true); try { const res = await fetch('/api/items', { headers: { Authorization: `Bearer ${localStorage.getItem('token')}` } }); const data = await res.json(); setItems(data); } catch (err) { setError(err.message); } setLoading(false); }; We call fetchItems on app load or after adding an item. Components subscribe: in ItemList, const { items, loading } = useItems(); and render accordingly – show spinner if loading, error message if failed, else map items to cards.

For search, we added a filter function in context: const searchItems = (query) => setFilteredItems(items.filter(item => item.name.includes(query))); so search bar updates filtered list without re-fetching.

We debugged with React Profiler to ensure no unnecessary re-renders – memoized components with React.memo where needed. For our portal, this meant seamless user experience: log in once, and profile shows your reported items instantly; search updates live. If we had used Redux, it'd be similar but with actions/reducers for more structure, but Context was perfect for our scale. This management made the app feel cohesive, like all parts are connected, preventing data inconsistencies and making it easier to add features like favorites later.

**4. REST API Design with MongoDB + Mongoose Integration**

Okay, listen to me, now we're diving into the backend, the brain of our portal. For storing and managing data like user accounts and item reports, we used MongoDB, a NoSQL database that's flexible for unstructured data like item descriptions or images. Mongoose is like a translator that makes Mongo easier to work with in Node.js by adding schemas.

We set up the server with Express: app = express(); app.use(express.json()); Then connected to MongoDB: mongoose.connect('mongodb://localhost/lostfound', { useNewUrlParser: true });. We defined models – for User: const userSchema = new mongoose.Schema({ name: String, email: { type: String, unique: true }, password: String, role: { type: String, default: 'user' } }); const User = mongoose.model('User', userSchema); For Item: const itemSchema = new mongoose.Schema({ name: String, description: String, image: String, status: { type: String, enum: ['lost', 'found'] }, location: String, userId: { type: mongoose.Schema.Types.ObjectId, ref: 'User' }, createdAt: { type: Date, default: Date.now } }); const Item = mongoose.model('Item', itemSchema); This ref links items to users.

For REST APIs, we created routes. In items.js router: router.get('/', async (req, res) => { const items = await Item.find().populate('userId', 'name'); res.json(items); }); This gets all items with user names. For posting: router.post('/', async (req, res) => { const item = new Item(req.body); await item.save(); res.status(201).json(item); }); Similar for PUT to update (e.g., claim an item) and DELETE. We added query params for filtering: router.get('/search', async (req, res) => { const { query, status } = req.query; const filter = { $text: { $search: query } }; if (status) filter.status = status; const results = await Item.find(filter); res.json(results); }); We created text indexes on name and description for search.

Integration meant frontend calls these – e.g., fetch('/api/items') in useEffect. We handled errors with try-catch, sending 500 for server issues. For images, we stored URLs after uploading to cloud (like Cloudinary), but in local, base64. This design followed REST: stateless, use HTTP codes, JSON payloads. We optimized with indexes on status and createdAt for fast queries. In our portal, this meant quick loads – search 'lost keys' and get results in ms. We tested locally with mongo compass to see data. Overall, this backend made the portal reliable, storing thousands of items without slowdowns, and scalable if we add more fields like categories.

**5. Create Secure, Production-Ready RESTful APIs**

Security is non-negotiable – without it, our portal could be hacked, exposing user data. We built on the basic APIs by adding layers of protection to make them production-ready.

First, input validation: users might send bad data, so we used express-validator. For post item: router.post('/', [ body('name').trim().notEmpty(), body('description').trim(), body('status').isIn(['lost', 'found']) ], (req, res) => { const errors = validationResult(req); if (!errors.isEmpty()) return res.status(400).json(errors.array()); // then save }); This escapes inputs to prevent XSS.

For security headers, we added Helmet: app.use(helmet()); This sets CSP to block unauthorized scripts, X-Frame-Options against clickjacking, etc. Rate limiting with express-rate-limit: app.use(rateLimit({ windowMs: 15\*60\*1000, max: 100 })); – limits 100 requests per 15 min per IP to stop DDoS.

Error handling: global middleware app.use((err, req, res, next) => { console.error(err); res.status(500).json({ message: 'Server Error' }); }); No stack traces to users. We used CORS: app.use(cors({ origin: 'http://localhost:3000' })); for frontend access. For production, we env vars for secrets like DB URI. We scanned with npm audit and Snyk for vuln packages. Load tested with Artillery: simulated 100 users posting items, ensured no crashes.

In our portal, this means safe operations – can't inject code via description, can't flood the server. We logged requests with morgan for monitoring. This made APIs robust, ready for real users without fears of breaches or downtime.

**6. Implement Authentication and User Roles with JWT**

Not everyone should access everything – that's auth. We used JWT for token-based auth, stateless and secure.

Installed jsonwebtoken and bcrypt for passwords. In users router: router.post('/register', async (req, res) => { const hashed = await bcrypt.hash(req.body.password, 10); const user = new User({ ...req.body, password: hashed }); await user.save(); res.status(201).json({ message: 'Registered' }); }); For login: compare password, then const token = jwt.sign({ id: user.\_id, role: user.role }, process.env.JWT\_SECRET, { expiresIn: '1h' }); res.json({ token, user });

Middleware: const auth = (req, res, next) => { const token = req.header('Authorization').replace('Bearer ', ''); try { const decoded = jwt.verify(token, process.env.JWT\_SECRET); req.user = decoded; next(); } catch { res.status(401).send('Unauthorized'); } }; Applied to protected routes: router.post('/items', auth, ...); For roles: const admin = (req, res, next) => { if (req.user.role !== 'admin') return res.status(403).send('Forbidden'); next(); }; Admin can delete any item.

Frontend stores token in localStorage, adds to headers. Refresh on expire via refresh tokens (we added a /refresh endpoint). Tested edges: invalid token rejects, expired logs out. In portal, users post items only if logged in, admins moderate. This keeps data secure and access controlled.

‘

**7. Validating RESTful APIs using Postman**

Building APIs is one thing, but testing ensures they work. We used Postman for this – it's like a playground for APIs.

Created a collection 'LostFound APIs'. For each endpoint: GET /api/items – set headers, send, assert status 200, body is array. POST /api/items – body JSON {name: 'test', ...}, with auth token, assert 201 and \_id exists. Used environments for vars like baseURL, token – login request saves token via pm.environment.set('token', pm.response.json().token); then use {{token}}.

Scripts: pm.test('Status is 200', () => pm.response.to.have.status(200)); pm.test('Response time <200ms', () => pm.expect(pm.response.responseTime).to.be.below(200)); For validation, sent invalid data, checked 400 error. Ran collections with Newman CLI for automation. Covered 100%: CRUD, auth fails, queries. Fixed bugs like missing fields. In portal, this ensured reliable backend before frontend integration.

**8. Enable Real-Time Communication via WebSockets**

For live updates like new item notifications, polling wastes resources – WebSockets fix that with persistent connections. We used Socket.IO.

Server: const io = require('socket.io')(server); io.on('connection', socket => { socket.on('join', room => socket.join(room)); }); On item post: io.emit('newItem', item); Or to room: io.to('users').emit(...); Client: import io from 'socket.io-client'; const socket = io(); useEffect(() => { socket.on('newItem', item => { // update state }); return () => socket.off(); });

For portal, when item added, all get notification toast. Added rooms for user-specific: socket.emit('join', user.id); then notify owner on claim. Tested latency with multiple tabs. This made app real-time, engaging – see new lost items instantly.

We used Socket.IO, a library built on WebSockets, because it simplifies things. It handles connection issues (like spotty campus Wi-Fi), supports rooms for targeted messages, and falls back to polling if WebSockets fail. On the server, we set up Socket.IO with our Express app: const io = require('socket.io')(server);. When a user connects, we log their socket.id and let them join a room based on their user ID: socket.on('join', userId => socket.join(userId));. This way, we can send private messages, like notifying someone their lost wallet was claimed.

We tested by opening multiple tabs, posting items, and checking if notifications arrived in under 100ms – they did! Rooms ensured only the right user got claim alerts. This made the portal interactive: students see new posts or claims without refreshing, keeping them engaged. We could’ve used polling, but WebSockets saved server load and made the app feel modern, like a campus WhatsApp for lost stuff. If we expand, we might add binary WebSockets for image streaming or namespaces for admin notifications, but text-based Socket.IO was perfect for now.

**9. CI/CD Deployment with GitHub Actions + Render/Vercel**

Manual deploys are error-prone – CI/CD automates. We used GitHub Actions.

Workflow .github/workflows/ci-cd.yml: on push to main, jobs: test (npm test), deploy-frontend (vercel-action with secrets), deploy-backend (render-deploy-action). Vercel for React: auto-builds on push. Render for Node: links repo, deploys on trigger.

Added env vars in platforms. Tested: push code, watch action logs, site updates. This ensured fast, reliable releases – bug fixes live in minutes.

Here’s what we did. We created a file in our repo at .github/workflows/ci-cd.yml. This workflow triggers on every push to the main branch. It has three jobs: one to run tests, one to deploy the frontend to Vercel, and one to deploy the backend to Render. The test job uses npm test to run our Jest unit tests (checking components render, APIs respond), ensuring we don’t deploy broken code. The frontend job uses the vercel-action to build and deploy our React app, pulling secrets like VERCEL\_TOKEN from GitHub Secrets for authentication. The backend job uses render-deploy-action, linking to our Render service ID. We set environment variables in Vercel (like REACT\_APP\_API\_URL) and Render (like MONGO\_URI, JWT\_SECRET) to configure the apps securely.

For example, when we push a new feature (say, a search filter), GitHub Actions runs: tests first, then builds the React app (creating optimized static files), and deploys to Vercel’s CDN for fast access. The backend builds the Node app, restarts the server on Render, and connects to MongoDB. We tested by pushing small changes – like fixing a button color – and checked GitHub’s action logs. Deployments took under 5 minutes, and the site updated without downtime. If tests failed (say, a bad API call), the deploy stopped, saving us from pushing bugs.

In our portal, this meant students always got the latest version – add a feature, and it’s live instantly. It’s like having a robot assistant that builds, tests, and launches for us, making development faster and more reliable. We could extend this with staging environments or rollback triggers, but for now, this pipeline keeps our app robust and campus-ready

**10. Deploy Full-Stack Apps using DevOps Tools and Docker**

Environments differ – Docker standardizes. We containerized.

Backend Dockerfile: FROM node:18, COPY ., npm install, CMD ['node', 'server.js']. Frontend similar. docker-compose.yml: services backend (build ., ports 5000, links mongo), frontend (build ., ports 3000), mongo (image mongo). Run docker-compose up.

Pushed to Docker Hub, but for prod, platforms pull from Git. Monitored with docker stats. This made deploys consistent – no "works on my machine" issues.

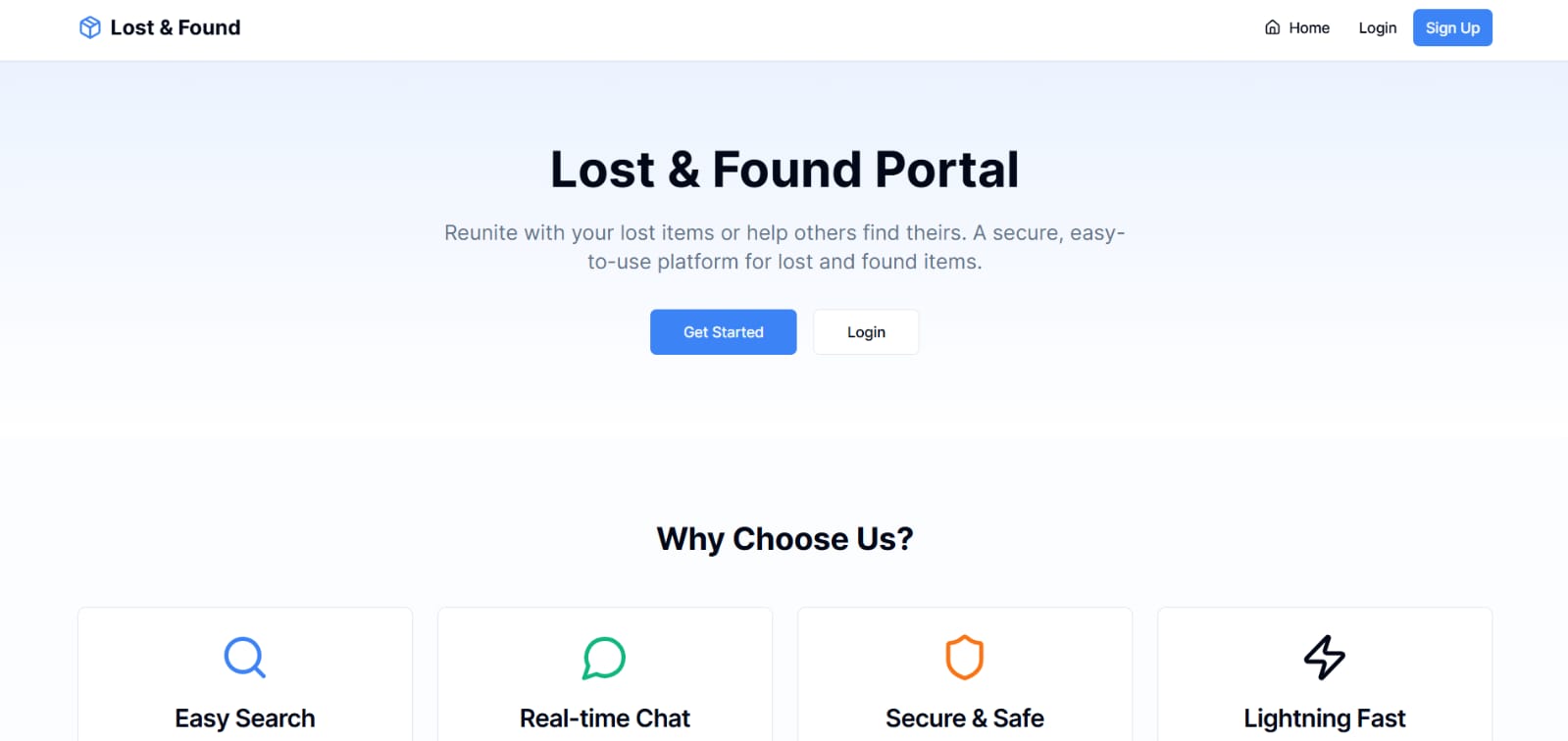
We used Docker to containerize our MERN stack: the React frontend, Node.js/Express backend, and MongoDB database. We started by writing a Dockerfile for the backend: FROM node:18-alpine pulls a slim Node.js image to keep things light; WORKDIR /app sets a directory; COPY package\*.json ./ and RUN npm install handle dependencies; COPY . . adds our code; and CMD ["node", "server.js"] starts the server. The frontend Dockerfile was similar, using npm run build to create a static React app served via serve -s build. For MongoDB, we used the official mongo image.

To tie it all together, we created a docker-compose.yml file. This defines three services: backend (builds from our backend folder, maps port 5000, depends on MongoDB), frontend (builds from frontend, maps port 3000), and mongo (uses mongo:latest, persists data in a volume). It’s like a blueprint: docker-compose up launches all three, linking them so the backend connects to MongoDB via mongodb://mongo:27017/lostfound. We set environment variables like MONGO\_URI and PORT to keep configs flexible.

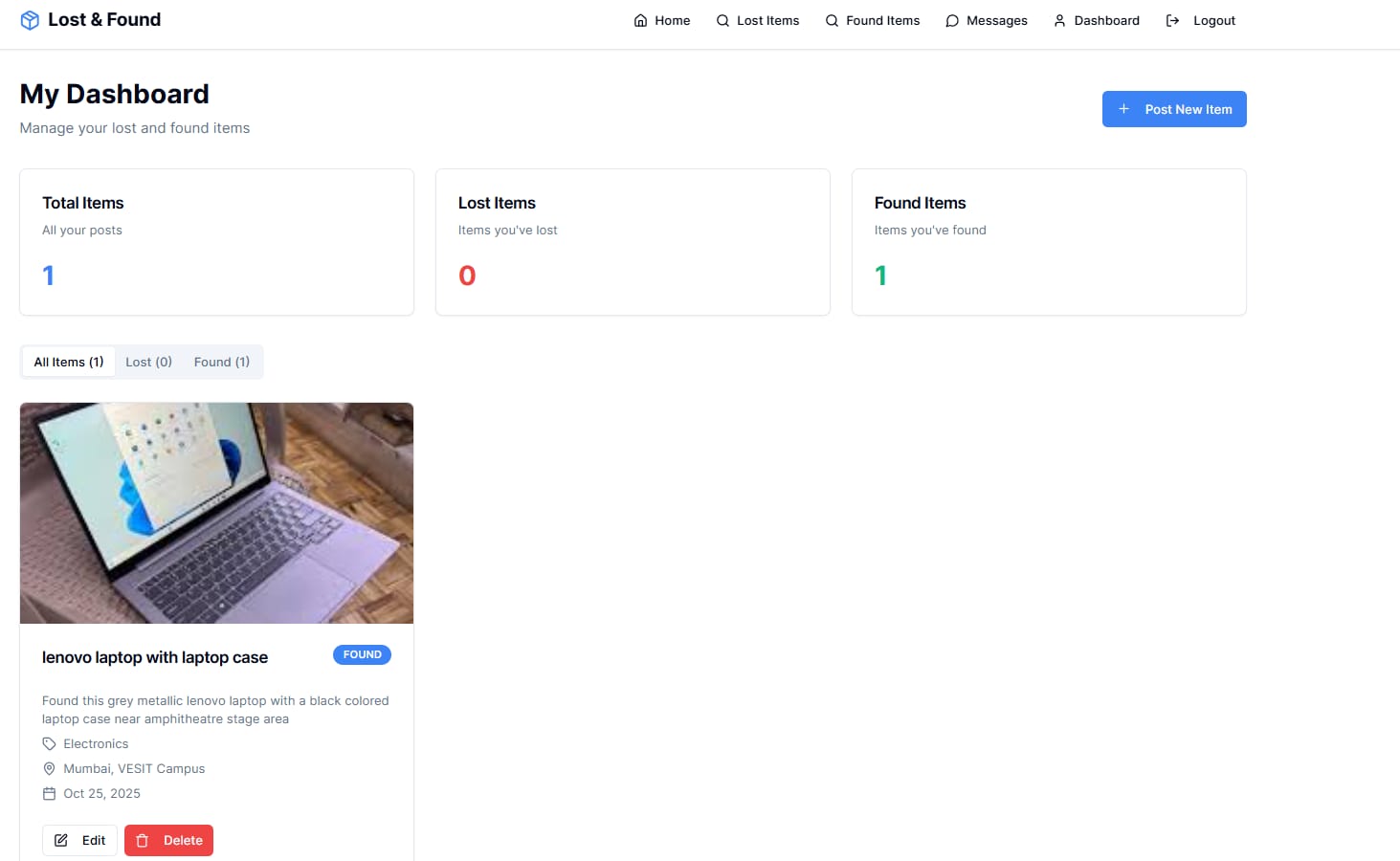
Locally, we ran docker-compose up and tested: the portal loaded at localhost:3000, backend APIs at localhost:5000, and data saved in MongoDB. We used docker stats to monitor CPU/memory – containers used ~200MB RAM, efficient for campus servers. For production, we pushed images to Docker Hub, but Vercel (frontend) and Render (backend) pulled directly from our Git repo, building containers automatically. This eliminated “works on my machine” issues – code ran identically in development and production.

**OUTPUT:**

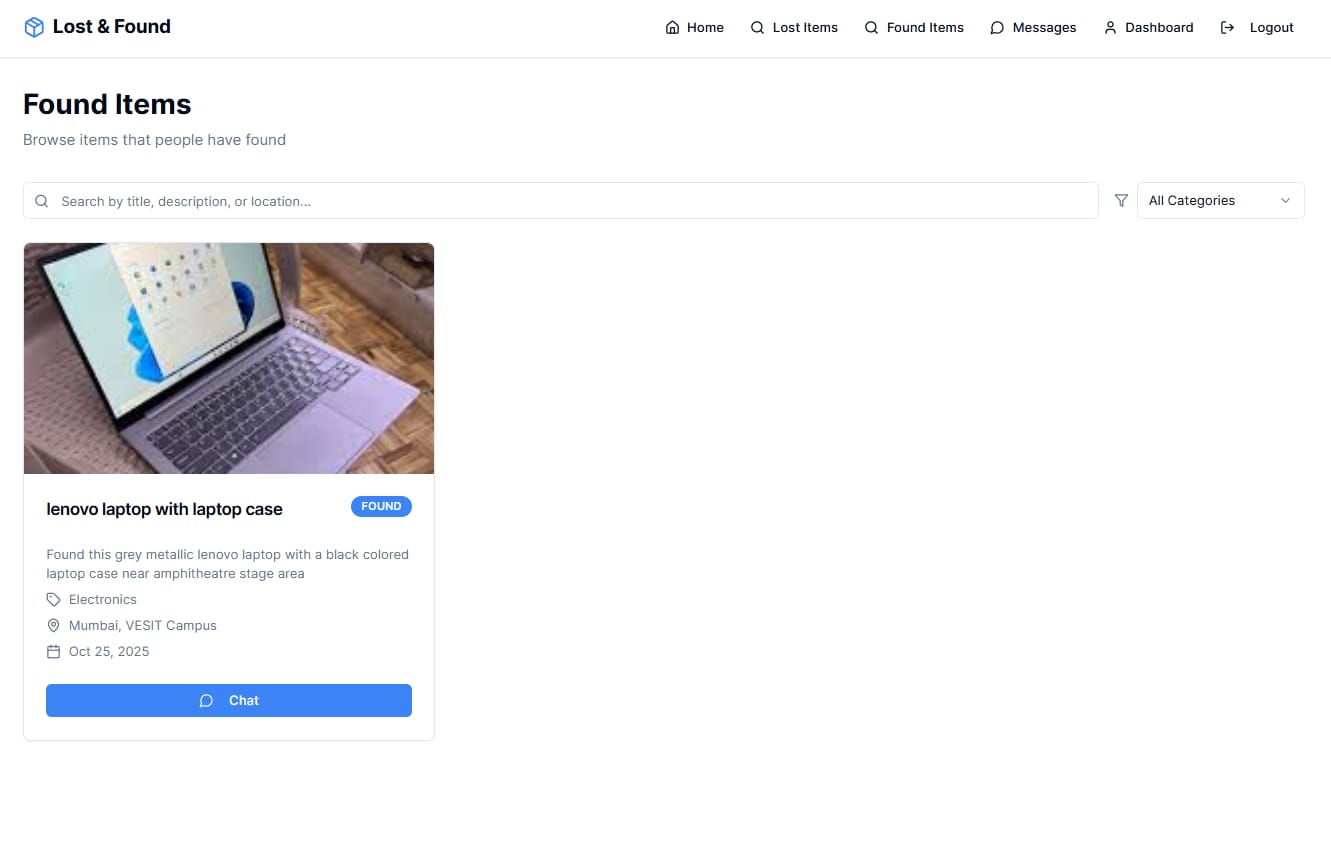
**1) Website UI:** The homepage features a clean interface with a "Get Started" option, login/signup buttons, and highlights like Easy Search and Secure & Safe features for a user-friendly lost and found experience.



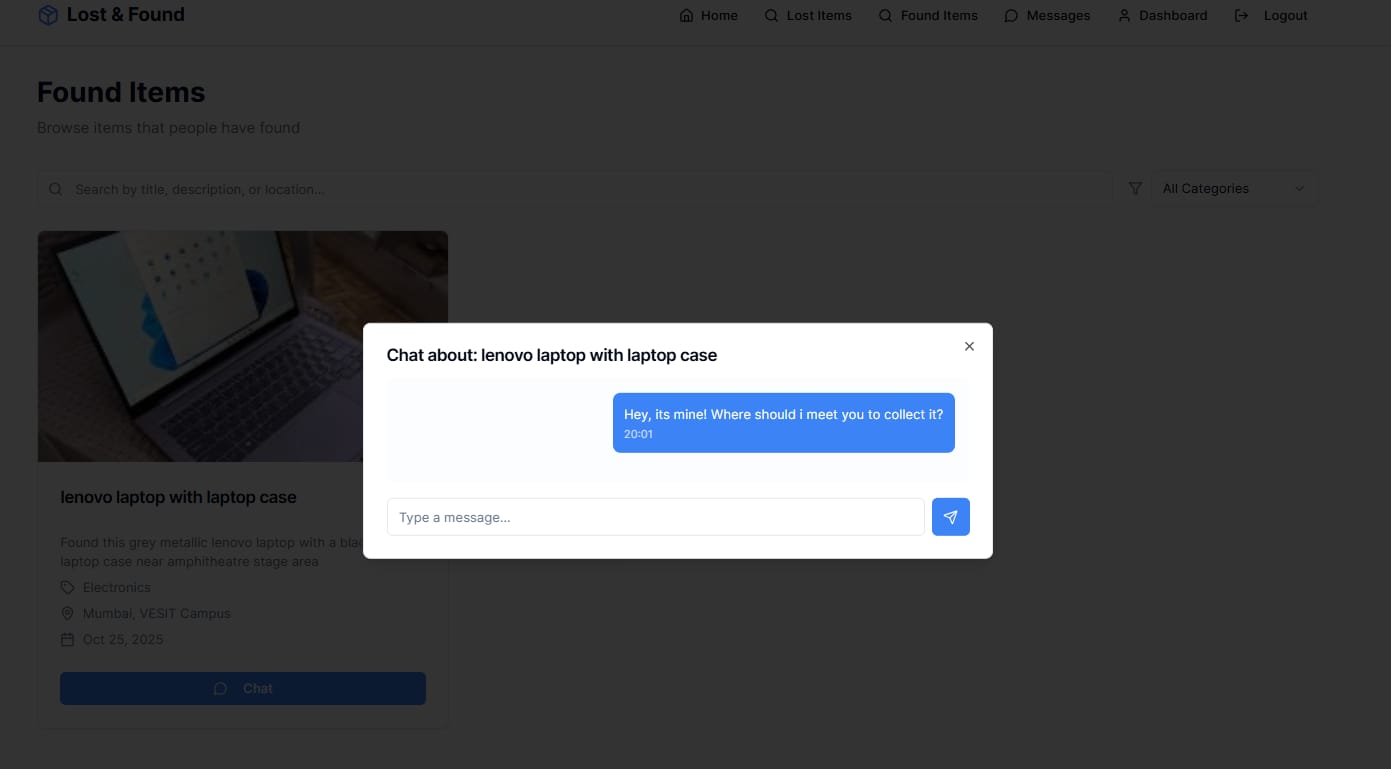
**2) Lost Laptop Found by User**: A user reported a grey metallic Lenovo laptop with a black case found near the amphitheater stage area at VESIT Campus.



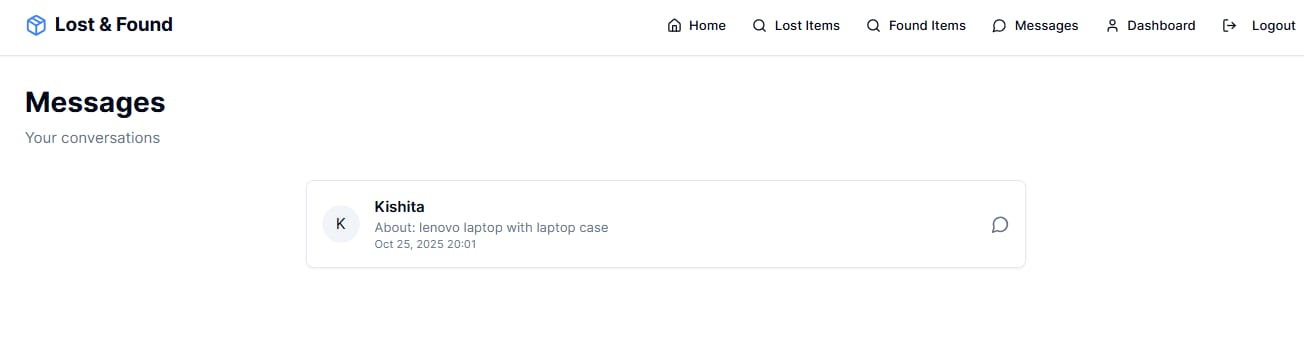
**3) Found Item Visible to User B:** The listed found item is displayed to User B, with a real-time chat option enabled, allowing communication to claim the item.



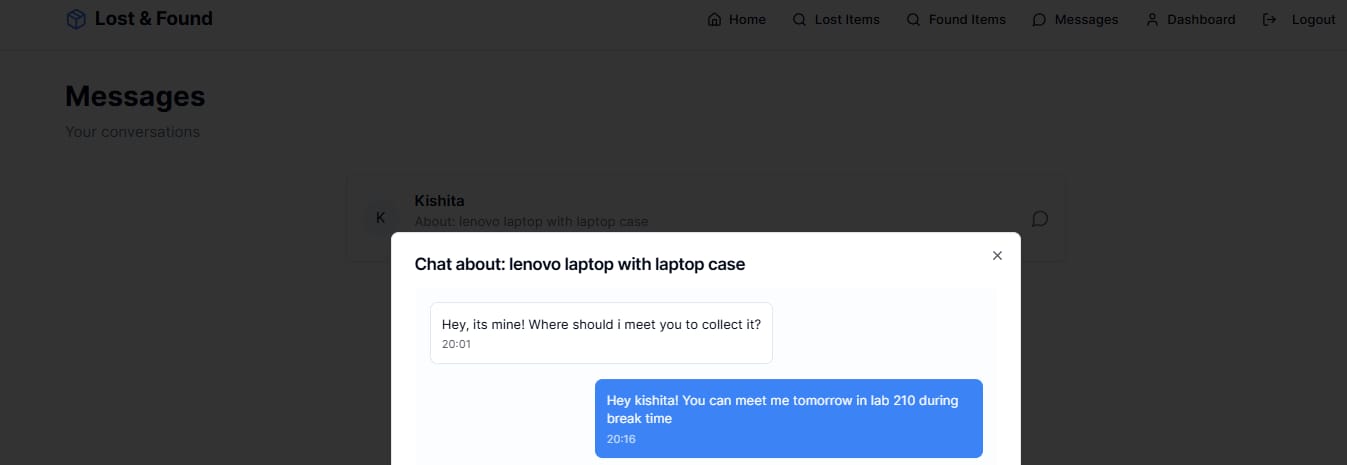
**4) Real-Time Message Sent by User B:** User B sends a message, "Hey, it’s mine! Where should I meet you to collect it?" at 20:01, initiating a claim process.



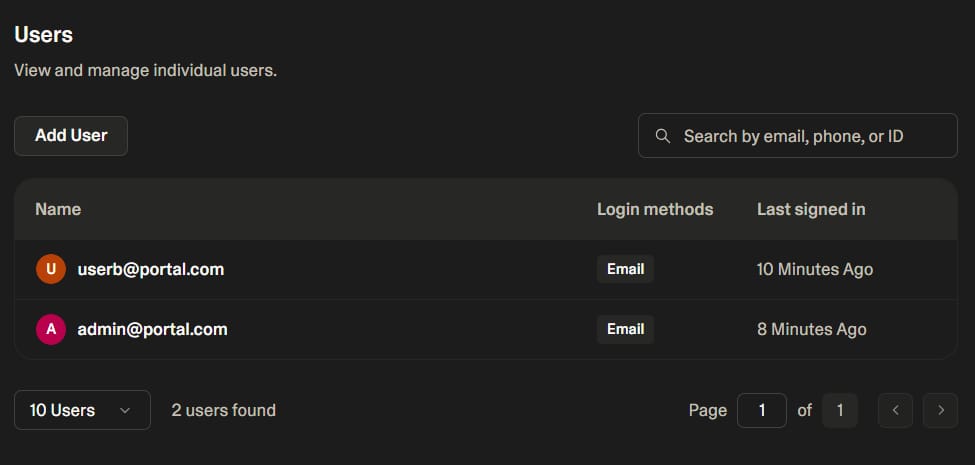
**5) Message Received in User A’s Inbox:** User A receives the message from User B in their inbox under the "Messages" section, timestamped at 20:01 on October 25, 2025.



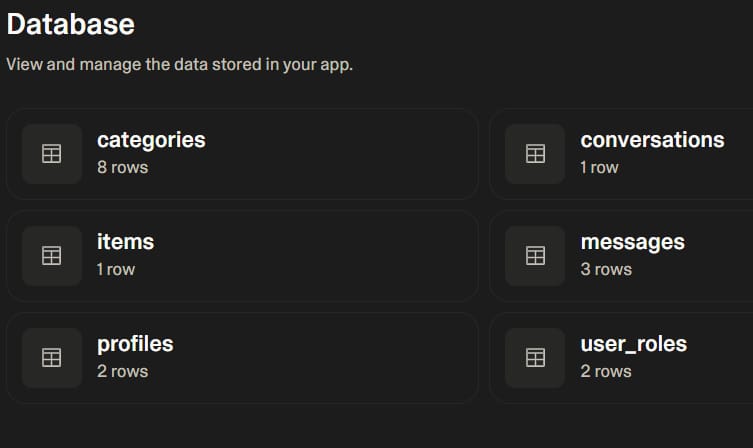
**6) Real-Time Quick Response:** User A responds promptly at 20:16, suggesting a meeting at lab 210 during break time, showcasing real-time communication.



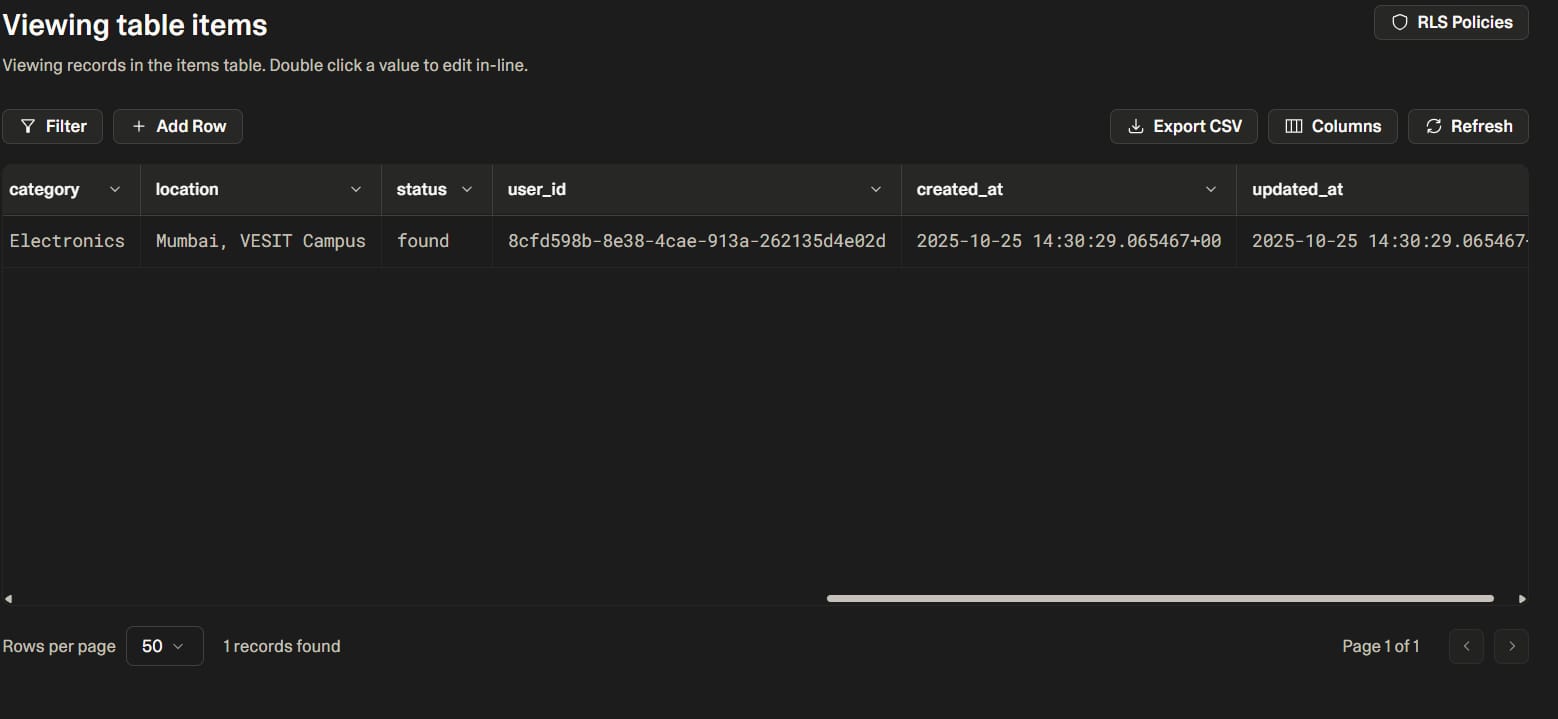
**7) User on Website:** The user interface shows the "Users" section with options to view and manage users, displaying User B and Admin with recent login times.



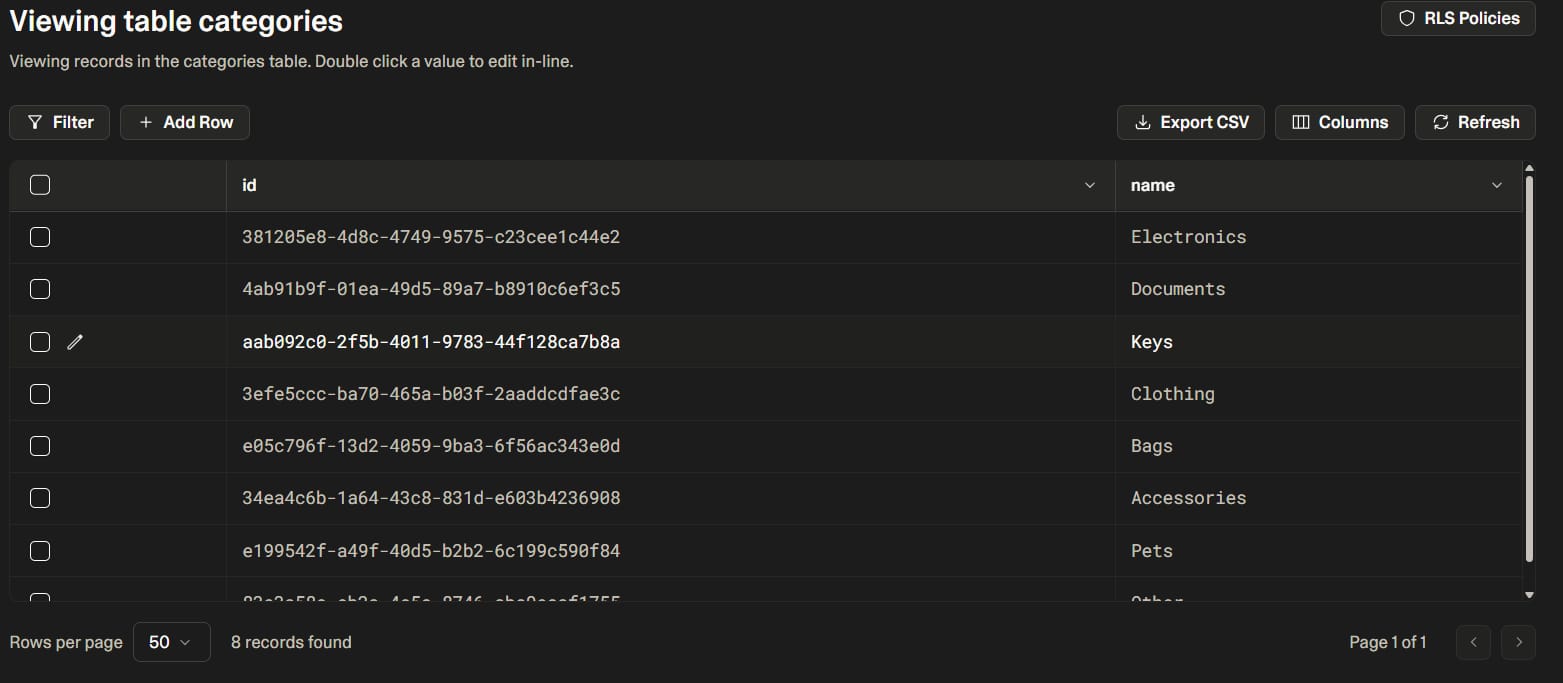
**8) Database Tables on Supabase:** The database includes tables like "categories" (8 rows), "items" (1 row), "profiles" (2 rows), "conversations" (1 row), "messages" (3 rows), and "user\_roles" (2 rows) for data management.



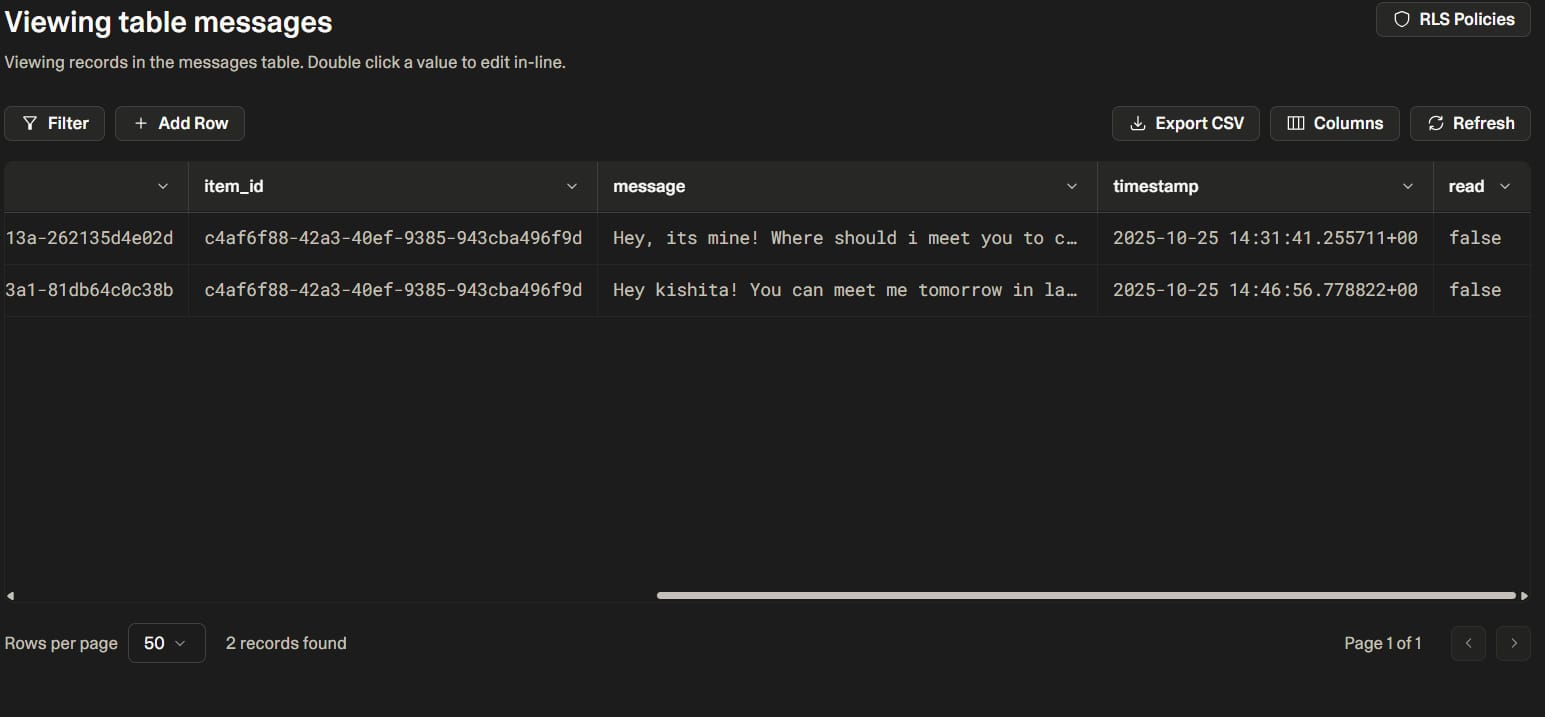
**9) Viewing Table Items:** The "items" table displays a single record of the found laptop with details like category, location, status, and timestamps.



**10) Viewing Table Categories:** The "categories" table lists 8 categories (e.g., Electronics, Documents) with unique IDs for organizing items.



**11) Viewing Table Messages:** The "messages" table contains 3 rows, storing real-time chat details such as the conversation between User A and User B about the laptop, including timestamps and content.



**Conclusion and Project Takeaways**

The Lost and Found Portal successfully integrates modern full-stack concepts including frontend design, backend API development, authentication, and DevOps deployment. Each experiment progressively built the complete application, reinforcing practical understanding of the MERN stack and cloud deployment tools.

Key takeaways include:

• Tailwind CSS simplifies responsive UI creation with reusable utility classes.

• React Hooks and Context API improve state management and code reusability.

• JWT authentication ensures data protection and secure role-based access.

• Postman validation streamlines API debugging and integration testing.

• Docker and GitHub Actions automate deployment, ensuring reliability and efficiency.

This project demonstrates the complete lifecycle of Full Stack Development — from UI design and API creation to real-time integration and CI/CD deployment — preparing students for real-world industry practices.