ChatGPT

Cannabis Traceability MVP Platform Scaffold

Overview and Objectives

A budding cannabis seedling (early growth stage). This MVP traceability platform tracks each cannabis plant from the seed stage onward. The goal is to simulate a seed-to-sale system: recording events like planting and harvesting, and ensuring data integrity at every step. By capturing key lifecycle events (seed planting, growth milestones, harvest) in a minimalistic web app, regulators, operators, and auditors can visualize the supply chain. The system uses a modern TypeScript stack (Node/NestJS and React) with a PostgreSQL database, all containerized via Docker for easy development and deployment.

This scaffold provides a complete project structure compatible with GitHub Copilot suggestions. It emphasizes **modularity**, separation of concerns, and simplicity. All interfaces are in English and follow an **Apple-style simplicity** with cannabis-themed accents – meaning plenty of white space, clean sans-serif typography, and intuitive layouts [1]. Despite the cannabis domain, the UI avoids overwhelming visuals, opting for a professional look with subtle green accents to reflect the industry. Every component is kept minimal and clear, focusing on core functionality rather than ornate design.

Project Structure and Tech Stack

The project is organized into distinct frontend and backend directories, each with its own tooling and Docker configuration. Below is an overview of the repository structure:

```
cannabis-traceability-mvp/
├── README.md                   # Documentation for setup and usage
├── docker-compose.yml          # Orchestration of services (frontend,
backend, db)
├── backend/                    # NestJS backend service (Node.js + Express)
│   ├── Dockerfile              # Docker image for backend
│   ├── package.json            # Node dependencies (NestJS, etc.)
│   └── src/
│       ├── main.ts             # Bootstraps the NestJS application
│       ├── app.module.ts       # Root module importing all feature modules
│       └── modules/            # Feature modules grouping controllers/
services
│           ├── user/
│           │   ├── user.module.ts
│           │   ├── user.service.ts
│           │   └── user.controller.ts
│           ├── plant/
│           │   ├── plant.module.ts
│           │   ├── plant.service.ts
│           │   └── plant.controller.ts
│           └── harvest/
│               ├── harvest.module.ts
```

```
│               ├── harvest.service.ts
│               └── harvest.controller.ts
├── frontend/                   # React frontend (TypeScript + Tailwind CSS)
│   ├── Dockerfile              # Docker image for frontend
│   ├── package.json            # Frontend dependencies (React, etc.)
│   ├── tailwind.config.js      # Tailwind CSS configuration
│   └── src/
│       ├── index.tsx           # App entry point
│       ├── App.tsx             # Application root component (defines routes/
layout)
│       ├── assets/             # (Optional) images, icons, etc.
│       ├── components/         # Reusable UI components (e.g., NavBar,
FormWizard)
│       ├── pages/              # Page-level components for each view
│       │   ├── Login.tsx
│       │   ├── Dashboard.tsx
│       │   ├── LifecycleExplorer.tsx
│       │   ├── BlockchainView.tsx
│       │   └── Wizard.tsx      # Wizard for planting/harvesting (Operator
only)
│       └── context/            # Contexts for global state (e.g.,
AuthContext)
│           └── AuthContext.tsx
└── postgres-data/
# Directory (or Docker volume) for PostgreSQL data persistence
```

Tech Stack:

- **Frontend:** React (with **TypeScript**) for robust type-safe components, styled using **Tailwind CSS** for rapid UI development. React Router handles page navigation, and Context API (or a lightweight state management) handles global user session state (like current role).
- **Backend:** NestJS (Node.js framework) with TypeScript, chosen for its **modular architecture** and built-in support for structured code organization [2]. NestJS uses Express under the hood and encourages splitting features into modules (with controllers and providers) for maintainability. This modular approach means we can encapsulate traceability features (planting, harvesting, user auth) into separate units [3], making the codebase developer-friendly and easy to extend.
- **Database:** PostgreSQL for persistence (e.g. storing plant records, though in this MVP most data is mocked). A Dockerized Postgres instance is included, and the backend is set up to connect to it (via environment variables for host, user, password, etc.). Initial data can be seeded via scripts or inserted on first run; however, the MVP primarily uses **hardcoded or in-memory data** to simulate functionality without requiring complex migrations.
- **Containerization:** Docker is used for both the backend and frontend, and **Docker Compose** orchestrates them along with the Postgres service. This allows one-command startup of the entire stack in a local environment [4], ensuring consistency across development machines. The containers expose standard ports (React dev server on **3000**, NestJS API on **3001**, Postgres on **5432**).

The use of **TypeScript** across the stack ensures type consistency and developer confidence, leveraging GitHub Copilot's strengths to autocomplete and suggest code. With the scaffold in place, Copilot can

help flesh out the business logic (for example, generating service methods or React component internals) based on the clear interfaces and types we define.

Docker & Development Environment

To streamline development, a **Docker Compose** configuration is provided that defines all required services: the React frontend, NestJS backend, and Postgres database. Developers can launch the entire environment with a single command (`docker-compose up`) and have everything running together. Key aspects of the Docker setup include:

- **Docker Compose File (`docker-compose.yml`):** This file defines three services:
- `backend` (NestJS app),
- `frontend` (React app),
- `db` (PostgreSQL database).

Each service uses an image or build context and necessary configuration. For example, the NestJS service is built from the `./backend` directory (using its Dockerfile) and maps port **3001**. The React service builds from `./frontend` and maps port **3000**. The Postgres service uses the official **postgres:14-alpine** image (lightweight) and maps port **5432** for local access [5]. An excerpt from the compose file might look like:

```
version: '3.8'
services:
  backend:
    build:
      context: ./backend
      dockerfile: Dockerfile
    env_file: ./backend/.env               # contains DB connection strings,
etc.
    ports:
      - "3001:3001"
    depends_on:
      - db
    volumes:
      - ./backend/src:/app/src            # optional: mount code for hot-
reload in dev
  frontend:
    build:
      context: ./frontend
      dockerfile: Dockerfile
    env_file: ./frontend/.env
    ports:
      - "3000:3000"
    volumes:
      - ./frontend/src:/app/src           # optional: mount code for HMR in
dev
  db:
    image: postgres:14-alpine
    ports:
      - "5432:5432"
```

```
    environment:
      POSTGRES_USER: trace_user
      POSTGRES_PASSWORD: trace_password
      POSTGRES_DB: traceability
    volumes:
      - postgres-data:/var/lib/postgresql/data
volumes:
  postgres-data:
```

In the above: - We use environment variables (or an `.env` file) to configure the database (e.g. database name, username, password) so that the NestJS app can connect. - We mount volumes for code directories in dev mode so that changes to source files on the host trigger live reload inside the containers (for React's dev server and NestJS with hot reload). This enables a smooth developer experience where edits reflect immediately without rebuilding the containers [6]. - The Postgres data is stored in a named volume (`postgres-data`) to persist the database between runs.

- **Backend Dockerfile:** A simple Dockerfile based on Node.js (Alpine) is used for the backend. It sets the working directory, installs dependencies, copies the source, and runs the NestJS application. For example:

```
FROM node:18-alpine
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
# Use NestJS in watch mode for development:
CMD ["npm", "run", "start:dev"]
```

This ensures the NestJS server (on port 3001) starts in development mode (with hot reload via `nodemon` or Nest CLI's watch mode). In production, the Dockerfile could be multi-stage to build the app and run a trimmed production server, but for MVP we prioritize ease of local iteration.

- **Frontend Dockerfile:** Similarly, the frontend uses a Node 18 image to run the React development server. For example:

```
FROM node:18-alpine
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
CMD ["npm", "start"]   # assuming create-react-app or similar starts on
port 3000
```

In a real deployment, we'd build static files and serve via Nginx or a cloud host, but here we use the dev server for simplicity. The environment variables `REACT_APP_BACKEND_URL` (or host/port) can be passed via the `.env` or docker-compose so the frontend knows how to reach the backend API (e.g., `REACT_APP_BACKEND_URL=http://localhost:3001` for local dev) [7].

- **Running the stack:** The README provides instructions to run `docker-compose up --build` to build images and start containers. Once up, the app is accessible at **http://localhost:3000** (frontend) and the API at **http://localhost:3001** (backend) [4]. Developers can log in with the test credentials to explore different role views. Stopping and removing containers is just as easy with `docker-compose down`. This containerized setup ensures that onboarding a new developer is trivial (no need to manually install Node, Postgres, etc., just Docker).

Backend – NestJS API (Node.js + PostgreSQL)

The backend is a **NestJS** application divided into logical modules corresponding to our domain entities and features. NestJS was chosen for its robust structure: it forces a separation of concerns (controllers for routing, providers for business logic, modules for organization) and is built in TypeScript for reliability [2].

Project Modules: We define the following NestJS modules (each in `src/modules/...`): - **User Module:** Handles user login and role management (though in this MVP, it's a mock). Contains a `UserService` (with hardcoded users and a simple method to validate credentials) and a `UserController` for endpoints like `POST /auth/login`. This module also could handle the **2FA** simulation – e.g., an endpoint to verify a 2FA code (which might just check against a static code like "123456" for demo purposes). - **Plant Module:** Manages plantings (the "seed" stage). `PlantService` holds a list of planted seeds (in-memory or retrieved from DB). It has methods like `getAllPlants()` and `createPlant(dto)` which either fetch mock data or insert into the database. `PlantController` exposes endpoints such as: - `GET /plants` – return a list of all planted seed entries (each entry could have an ID, strain name, location, and timestamp). - `POST /plants` – accept a new planting event (with details like strain, location) and return the created record (for MVP, it can just echo the input with a generated ID and timestamp). - **Harvest Module:** Manages harvest events. `HarvestService` might maintain a list of harvest records. Key methods might be `getAllHarvests()` and `recordHarvest(dto)`. The corresponding `HarvestController` provides: - `GET /harvests` – list all harvest events (each might include an ID, reference to a plant, yield amount, drying/curing status, and timestamp). - `POST /harvests` – create a new harvest record (in practice linking to a plant ID that was previously planted). In the MVP, this will accept a plant identifier (or select from existing plants) plus yield and status, and return a mock record.

All modules are imported into the root `AppModule` so Nest can assemble the complete application. The NestJS CLI or generator could be used to scaffold these modules, controllers, and services quickly [8]. For example, `nest g module plant`, `nest g service plant`, `nest g controller plant` would set up the boilerplate files automatically. By the end, our `src` folder will have a clear structure with separate folders for each feature, reflecting Nest's best practices [3].

Mock Data and Services: Since this is a prototype, we avoid complex business logic or database operations. Services return **hardcoded data structures** or use in-memory storage: - We might define a simple interface or class for Plant (with fields like `id, strain, location, plantedAt, harvested=false`) and similarly for Harvest (with `id, plantId, yield, harvestedAt, status`). The `PlantService` can initialize with a couple of dummy plants (for example, one or two entries to show on the dashboard or explorer), and the `HarvestService` can be initially empty (to be populated when an operator records a harvest via the wizard). - The `UserService` can contain a static list of users, e.g.:

```
const USERS = [
  { username: "regulator1", password: "pass123", role: "Regulator" },
  { username: "operator1",  password: "pass123", role: "Operator" },
  { username: "auditor1",   password: "pass123", role: "Auditor" }
];
```

A method `validateUser(username, password)` checks this list and returns the user if credentials match. No actual hashing or security – this is just to simulate login. If a user is found, we might simulate issuing a JWT or session; however, since the frontend is mock-authenticated, we can simply return a dummy token or user object. The 2FA step can be simulated by requiring an extra call or client-side prompt (without actual SMS/email). For instance, after a successful login response, the frontend navigates to a "Enter 2FA code" screen – any 6-digit code input could be considered valid for the demo.

Sample Controller Implementation: As an example, the **PlantController** might look like:

```
@Controller('plants')
export class PlantController {
  constructor(private readonly plantService: PlantService) {}

  @Get()
  getAllPlants() {
    return this.plantService.getAllPlants();  // returns an array of plant
records
  }

  @Post()
  createPlant(@Body() dto: CreatePlantDto) {
    return this.plantService.createPlant(dto);  // creates a new plant
(mocked)
  }
}
```

And the **PlantService** could be:

```
@Injectable()
export class PlantService {
  private plants: PlantRecord[] = [];  // in-memory storage for demo

  getAllPlants(): PlantRecord[] {
    return this.plants;
  }

  createPlant(dto: CreatePlantDto): PlantRecord {
    const newPlant: PlantRecord = {
      id: Date.now(),  // unique id (timestamp as proxy)
      strain: dto.strain,
      location: dto.location,
      plantedAt: new Date(),
```

```
      harvested: false
    };
    this.plants.push(newPlant);
    return newPlant;
  }
}
```

This shows how minimal the logic is – just enough to simulate adding and retrieving records. Other modules would follow a similar pattern. The **HarvestService** might reference the PlantService to check if a plant exists or to mark it harvested, but in an MVP we might skip deep validations.

Database Integration: The NestJS app includes configuration (e.g., using `TypeORM` or `Prisma` if desired) to connect to PostgreSQL. For demonstration, we might not implement full DB logic, but the scaffold has the pieces in place: - A database connection configuration (in `app.module.ts`, importing a TypeORM module or using NestJS ConfigModule with `DATABASE_URL`). For example, using TypeORM:

```
TypeOrmModule.forRoot({
  type: 'postgres',
  host: process.env.POSTGRES_HOST || 'db',  // 'db' is the Docker service
name for Postgres
  port: parseInt(process.env.POSTGRES_PORT) || 5432,
  username: process.env.POSTGRES_USER || 'trace_user',
  password: process.env.POSTGRES_PASSWORD || 'trace_password',
  database: process.env.POSTGRES_DB || 'traceability',
  autoLoadEntities: true,
  synchronize: true, // auto-create tables from entities (OK for dev/MVP)
}),
```

Then each module could define an Entity (e.g., PlantEntity, HarvestEntity) and use a repository. **However, to keep things simple, the MVP can use just the services with in-memory data** and skip actual DB calls – the presence of Postgres is mostly to show the intended stack and allow future implementation. The initial data seeding could simply be done in the service constructors or a dedicated seeding script (run on container startup) that inserts a couple of rows in the database for demonstration.

API Endpoints Summary: The REST API surface for the MVP might include: - `POST /auth/login` – Verify credentials (against the hardcoded list) and start a session (mock). Returns a dummy JWT or session token and the user role. - `POST /auth/verify-2fa` – (If implementing 2FA step) Validate a code. In the mock, accept any code or a fixed code, and return success/failure. - `GET /plants` – Get all plant records. - `POST /plants` – Create a new plant record (seed planting). - `GET /harvests` – Get all harvest records. - `POST /harvests` – Create a new harvest event (with a plant reference). - `GET /lifecycle` – (Optional) Return a combined list of events (plants and harvests) for the **Lifecycle Explorer** page. This could aggregate data from PlantService and HarvestService, or query the database join if using real DB. For simplicity, this can return an array of events with a type field (e.g. `{ type: 'planted', plantId: 123, strain: 'ABC', time: ... }` and `{ type: 'harvested', plantId: 123, yield: 500, time: ... }`). - `GET /integrity` – (Optional) Return hash values or integrity check info for the **Blockchain Integrity** view. For example, could return a list of recent events with their computed SHA-256 hashes.

These endpoints all return **mocked JSON data** – suitable for the frontend to render but not performing real business operations or security checks. NestJS automatically serializes our returned objects to JSON (for simple types) [9], so we can just return plain objects or arrays from the controller methods and Nest will handle the HTTP response.

Frontend – React Application (with Tailwind CSS)

The frontend is a single-page React application (bootstrapped with a tool like Create React App or Vite, using the TypeScript template). We use **Tailwind CSS** for styling to achieve a clean, modern interface quickly. Tailwind is configured via `tailwind.config.js` to scan our `src/**/*.{tsx,html}` files for class names, and the base Tailwind directives (`@tailwind base; @tailwind components; @tailwind utilities;`) are included in the main CSS (e.g., `index.css`). This gives us utility classes to craft the minimalist design – lots of whitespace, rounded buttons, and a simple color scheme (perhaps neutral grays and a green accent to hint at the cannabis theme).

Responsive Design: All pages and components are built to be **responsive**. We utilize Tailwind's utility classes (like `flex`, `grid`, and responsive prefixes such as `md:w-1/2`, `sm:hidden` etc.) to ensure the layout works on mobile and desktop. The navigation might collapse into a hamburger menu on small screens or stack vertically. Testing on a phone-sized viewport and a desktop viewport verifies that the design remains intuitive and usable.

Navigation & Routing: The app likely uses **React Router** for client-side routing between pages: - After login, users are routed to their role-specific dashboard. - The navigation menu (perhaps a sidebar or topbar) shows links to pages the user can access (Dashboard, Lifecycle Explorer, Blockchain Integrity, etc., with some conditional links for certain roles). - We implement conditional rendering for menu items and routes based on the current user's role. For example, an Operator will see a "Operations Wizard" link, while a Regulator might see a "Compliance Dashboard" link. An Auditor might see an "Audit Logs" or lifecycle explorer link. This role-aware menu ensures users only see relevant sections, improving clarity and security [10]. In practice, we define a roles constant and a mapping of which menu items each role should see:

```
// Example role definition and menu mapping
enum Role { REGULATOR = 'Regulator', OPERATOR = 'Operator', AUDITOR =
'Auditor' }

const menuByRole: Record<Role, Array<{path: string, label: string}>> = {
  [Role.REGULATOR]: [
    { path: '/dashboard', label: 'Dashboard' },
    { path: '/explorer', label: 'Lifecycle Explorer' },
    { path: '/integrity', label: 'Blockchain Integrity' },
  ],
  [Role.OPERATOR]: [
    { path: '/dashboard', label: 'Dashboard' },
    { path: '/wizard', label: 'Start Process' },       // wizard for plant/
harvest
    { path: '/explorer', label: 'Lifecycle Explorer' },
    { path: '/integrity', label: 'Blockchain Integrity' },
  ],
  [Role.AUDITOR]: [
    { path: '/dashboard', label: 'Dashboard' },
```

```
    { path: '/explorer', label: 'Lifecycle Explorer' },
    { path: '/integrity', label: 'Blockchain Integrity' },
  ],
};
```

The navigation component uses the current user's role (from context or props) to determine which links to display, as illustrated in similar React RBAC examples [11]. This ensures **role-based UI** differences: each role experiences a tailored interface (only Operator sees the wizard, etc.) – *"show exactly what each user should see, no more, no less"* [10].

State Management and Auth Context: We implement a simple global state to track the logged-in user and role. React's Context API is sufficient here: - `AuthContext` provides `{ user, role, login, logout }` to the component tree. When a user "logs in" through the mock form, we set the `user` and `role` in context and possibly store them in `localStorage` for persistence across page refresh (since we aren't doing a real auth flow with cookies or JWT). - The context is used by components to decide what to render. For example, the navigation bar component accesses `AuthContext` to get `role` and then maps the menu items accordingly [12]. We can also protect certain routes (e.g., if the user tries to navigate to the wizard without being an Operator, we redirect or show "Unauthorized"). This can be done via a simple wrapper component or route guards, but given the small scope, conditional rendering in the components might suffice.

UI Pages/Views:

1. **Login Screen:** This is the entry point of the app. It includes fields for username and password, and possibly a "Login" button. Since we simulate 2FA, the login might be a two-step form:
2. Step 1: Username & Password – On submit, call the backend `POST /auth/login`. If the credentials match a user, the backend responds with success (and perhaps a flag that 2FA is required).
3. Step 2: 2FA Code – If required, the UI then shows a second input for the 6-digit code. In the MVP, we do not send a real code; we can display a message like "Enter the code sent to your device" and accept any input (or a fixed code like "000000"). This step is purely to mimic the flow. After submitting, the user is considered logged in.

The Login page is styled simply: a centered card with the app logo/title, input fields, and a submit button. Tailwind helps with form styling (`input` classes for borders, padding, focus states, etc.). We also ensure basic validation feedback (e.g., if wrong credentials, show an error message – which for MVP can be triggered by the backend returning an error status or by checking the response).

> **Note:** No actual authentication tokens are needed for the prototype; we trust the front to manage state. However, we could store a dummy JWT returned by backend in localStorage for completeness, and include it in headers on API calls – simulating how it *would* work in a real app.

1. **Dashboard (Role-Specific):** After login (and 2FA), the user lands on a Dashboard page. This page is customized per role:
2. **Regulator's Dashboard:** could show high-level compliance KPIs or alerts. For instance, number of active plants, number of harvests this week, any alerts (like discrepancies) – all dummy values. Perhaps a list of recent activities across the industry (like "Harvest completed at Farm A", "Lab test failed for Batch X") to mimic oversight.

3. **Operator's Dashboard:** shows operational info, e.g., tasks that need attention. Maybe "X plants need harvest", "Y packages awaiting label", or simply a welcome message and a shortcut to start the *Plant/Harvest Wizard*.
4. **Auditor's Dashboard:** might display auditing metrics or a summary of data integrity status. For MVP, it could be as simple as showing a count of events recorded and a prompt to use the explorer to deep dive.

The Dashboard will include the navigation menu and a header that indicates the user's role (e.g., "Welcome, Operator [Name]"). We use conditional components to render different content for each role; this can be done by checking the `role` context or by making separate Dashboard components and choosing one based on role.

1. **Operations Wizard (Operator Only):** This is a multi-step form that guides an **Operator** through recording a full cultivation cycle event:
2. **Step 1: Plant Seed.** The operator enters details of a new planting: strain name (text), grow location (text or dropdown if we had predefined sites), and timestamp (which can default to now). On submission, an API call `POST /plants` is made to record this. For the MVP, this will just add to the in-memory list and return the created plant record (with an ID).
3. **(Optional) Step 1.5:** Show a confirmation or details of the recorded planting (e.g., "Plant ID 123: Strain ABC planted at Greenhouse 7 on 2025-08-27 17:26").
4. **Step 2: Harvest Crop.** This step could either follow immediately (if we are simulating planting and harvesting in one flow), or be separate. In reality, harvesting happens much later than planting, but for demonstration, we can allow the wizard to continue to harvest for the newly planted seed. The operator selects which plant to harvest (a dropdown of plant IDs or names from the PlantService data), enters the yield amount (e.g., in grams), and a status (e.g., "drying" or "dried"). On submission, a `POST /harvests` call records the harvest.
5. **Completion:** After harvest is recorded, the wizard might display a summary "Plant X harvested successfully with yield Y", and possibly mention a hash was generated (tie-in with blockchain view) or prompt the operator to label the batch (out of scope for this MVP).

The Wizard is implemented as a component with internal state to track the current step. It can use a simple step counter or a state machine. Tailwind can style the wizard form with progress indicators at the top (like "Step 1 of 2") and nice form inputs. We make sure the wizard view is **only accessible to Operators** – if somehow a regulator or auditor route to `/wizard`, we either hide the link and/or redirect them away (could show a "404" or "Unauthorized" page).

1. **Cannabis Lifecycle Explorer:** This page provides a view of the **full lifecycle of cannabis items** in the system, primarily for Regulators and Auditors (Operators can see it too, but it's especially useful for oversight). The explorer could be a timeline or table view of events:
2. It might list each plant and the events associated with it (Planting -> Harvest -> (if extended, processing -> etc.). In MVP, since we have just planting and harvest, the explorer can show a list of all plants, and under each, the associated harvest event (if any).
3. Alternatively, it could list all events in chronological order (mixing plantings and harvests) with identifiers linking them. For example:

```
Plant #123 "Blue Dream" planted at Greenhouse 7 — Aug 1, 2025, 10:00
AM
→ Harvested (Plant #123) – Yield: 500g dried — Sep 15, 2025, 4:00 PM
Plant #124 "Purple Kush" planted at Outdoor Field — Aug 5, 2025, 09:30
```

```
AM
(not yet harvested)
```

4. The data for this view comes from the backend (perhaps via `GET /lifecycle` combining plants and harvests). Since it's mocked, the backend can fabricate a few entries on first load. We can use a simple React component to render the timeline or a table. Tailwind's utility classes help to style the timeline (borders, spacing, maybe an icon for plant vs harvest). We might even use a third-party lightweight component or just flexbox to simulate a timeline.

This explorer gives auditors and regulators a high-level *traceability* view – they can see if every plant has a harvest record, and if the timestamps make sense (in a real system, they'd watch for gaps or irregularities). For now, it's mainly a read-only page demonstrating that data is being tracked across steps.

1. **Blockchain Integrity View:** This is a special page meant to simulate how data would be secured on a blockchain or immutable ledger for integrity. In the MVP UI, this could be a simple panel that shows the hashes of events:
2. We can list recent events (or all events) along with a computed **SHA-256 hash** of their key data. For instance, if a plant was planted with ID 123 and strain "Blue Dream" at time T, we can take a string like `"PLANT:123:Blue Dream:T"` and hash it to produce a 64-character hexadecimal string. Similarly for harvest events. The frontend can either compute these (using a JS crypto library) or, easier, the backend can provide the hash in the data it returns (since Node can compute SHA-256 easily).
3. The page might display something like:

   ```
   Event                                  | SHA-256 Hash
   -------------------------------------- |
   ----------------------------------------
   Plant #123 "Blue Dream" planted @ 2025-08-01 10:00 | a3c4f... (64 chars)
   Harvest #987 of Plant #123 @ 2025-09-15 16:00 (500g) | 9baf1... (64
   chars)
   ```

   Each hash is unique to the event data; if data were tampered, the hash would change, demonstrating tamper-evidence.
4. We also show maybe a button "Recompute Hash" to illustrate the concept. But in general, this view educates the user that every recorded action can be hashed and later verified. In a real implementation, these hashes would be written to a blockchain ledger (like in a system using QLDB or Hyperledger, as envisioned) [13] [14], providing an immutable audit trail. For MVP, we just mimic the hashing part and display the results.

This feature is inspired by the idea that *blockchain provides an immutable, transparent ledger of events* for cannabis supply chain [15]. By recording a cryptographic fingerprint of each event, we ensure **non-repudiation and tamper-evidence** in the system [13]. The UI doesn't actually connect to a blockchain network; it's a **mocked integrity panel** to make the concept concrete for users. We can mention (in a tooltip or note) that "In a full implementation, these hashes would be submitted to a blockchain or immutable ledger for audit purposes."

Visual Design and UX Notes: All frontend pages follow a consistent minimalist design. Borrowing from Apple's design ethos, we use **crisp typography and ample whitespace** [1]. The color scheme could include white backgrounds, black/gray text, and a primary accent color (perhaps a green that evokes

cannabis but in a subdued, modern tone). Icons (from a library like Heroicons or FontAwesome) are used sparingly to label actions (for example, a seedling icon for planting, a scissors or basket icon for harvest, a shield or lock for the blockchain integrity section). These add visual cues without clutter. We ensure that even a user unfamiliar with the system can navigate easily: the menu labels are clear, and each page has a heading or title explaining what it is.

Accessibility is considered by using proper HTML elements (forms, labels, buttons) and sufficient color contrast. Tailwind makes it easy to adjust classes to meet contrast guidelines if needed.

Users and Role-Based Access

The platform defines **three user roles** with distinct perspectives: **Regulator**, **Operator**, and **Auditor**. Each role has a mocked user account for testing: - **Regulator:** Represents government or authority personnel overseeing compliance. In our scaffold, a regulator user (e.g., username "regulator1") will see a dashboard with compliance stats and have access to the lifecycle explorer and integrity views to verify that operators are recording everything properly. Regulators do **not** use the wizard (they don't directly plant or harvest; they supervise). - **Operator:** Represents a licensed cannabis business operator (e.g., a cultivator or manufacturer). The operator user (e.g., "operator1") sees a dashboard geared towards operations (perhaps tasks or recent activity) and crucially has access to the **Plant/Harvest Wizard** to record cultivation events. They also can view the lifecycle explorer (to see their own data) and the blockchain integrity view (to ensure their records are logged). - **Auditor:** Represents an external or internal auditor who reviews the records for accuracy and compliance. The auditor user ("auditor1") has a dashboard possibly showing audit flags (if any) and can access the lifecycle explorer and blockchain hash view to cross-check records. Like regulators, auditors do not perform operations, so they won't have the wizard available.

Mock Login & 2FA: Authentication is simulated. The login page has hardcoded acceptable credentials for each role as mentioned. When a user logs in, the frontend will set their role in the global state. We implement a **conditional rendering** such that if no user is logged in, the app shows only the Login screen (or routes unauthenticated users to login). After login, the role dictates the UI.

To simulate **Two-Factor Authentication**, after entering the correct username/password, we show a second input for a one-time code. This is purely a UX simulation (no SMS or email actually sent). We can hardcode an expected code (like "111111") or simply accept any input as valid for now. The point is to illustrate a 2FA step in the flow. The UI can show a fake countdown or a message "Enter the 6-digit code from your authenticator app." Once submitted, we mark the user as fully authenticated.

The role information is stored in the `AuthContext` and also perhaps in a cookie/localStorage for persistence. No secure token exchange is needed for this prototype; all API calls are open or use a dummy header if needed. This approach is **developer-friendly** because it bypasses the complexity of OAuth/JWT, focusing on the core role-based behavior.

Role-Based UI Control: Throughout the app, we use the user's role to control access: - Navigation menu generation uses the `menuByRole` mapping as described, so only relevant links appear [16] [17]. - Individual components or routes check role as needed. For example, the Wizard page component might do:

```
const { role } = useContext(AuthContext);
if (role !== 'Operator') {
```

```
  return <p>Unauthorized</p>; // or redirect to home
}
// ...otherwise render wizard steps
```

- We could also use React Router's capability to protect routes. For instance, define routes and wrap them in a component that checks the role before rendering (similar to the concept of a ProtectedRoute in RBAC tutorials [18]). Given the small scope, a simple inline check like above is sufficient.

This ensures that even if a user without proper role somehow finds out a URL, they cannot use functionality not intended for them (in a real app, the backend would also enforce this on the API side, but since our backend is mock, we enforce at least on UI).

By structuring role management in one place (the AuthContext and menu logic), we make it easy to modify or extend. If a new role is introduced, we update the roles enum and menu mapping, and the rest of the app can adapt. This approach is aligned with recommended practices for scalable role-based frontends [10] [12].

Blockchain Integrity Feature (Simulated)

One unique aspect of this platform is the integration (albeit simulated) of **blockchain-based integrity checks**. In a production system, each critical event would be hashed and stored on a blockchain or immutable ledger to guarantee data integrity and auditability [15]. For the MVP, we implement the core of this idea in a simplified form: - When the backend records an event (planting or harvesting), it immediately computes a **SHA-256 hash** of that event's data [13]. For example, the data might be a JSON or string like `{"type":"plant","id":123,"strain":"ABC","user":"operator1","timestamp":"2025-08-27T17:26:00Z"}`. The SHA-256 hash of this (essentially a 256-bit fingerprint) is generated. We don't actually send this to any external ledger, but we retain it in memory or as part of the response. - The backend can include the hash in the response for the event creation API. For instance, the response to `POST /plants` could be:

```
{
  "id": 123,
  "strain": "Blue Dream",
  "location": "Greenhouse 7",
  "plantedAt": "2025-08-27T17:26:00Z",
  "hash": "a3c4f5...d1"
}
```

where `hash` is the SHA-256 of the concatenated relevant fields. Similarly, the harvest response would include a hash. These hashes are computed using a library like Node's built-in `crypto` (`crypto.createHash('sha256').update(data).digest('hex')`). - The **Blockchain Integrity UI** then displays these hashes to the user as proof-of-record. We emphasize that if someone altered the data, the hash would change, but since the hash is presumably also stored on a tamper-proof ledger, any discrepancy would be caught. (In a real system, as described in the project context, these hashes might be sent to Amazon QLDB or Hyperledger Fabric for immutability [14], and a transaction ID or proof would be returned. The MVP stops at just showing the hash generation.)

This feature helps illustrate the concept of an **immutable audit trail**: every event is "chained" by its hash. For demonstration, we may allow the user (regulator/auditor) to pick an event and recompute its hash on the client side to verify it matches the stored hash. This proves that the data wasn't tampered with between recording and viewing.

By integrating this in the MVP, stakeholders can see how blockchain enhances traceability: it provides *transparency and accountability at every stage* of the supply chain [15]. The implementation is intentionally lightweight (no actual blockchain nodes or network calls), focusing instead on the *idea* – the SHA-256 hashes as stand-ins for blockchain transactions.

Additional Features and Nice-to-Haves

Beyond the core pages, we include a few extra touches to make the prototype feel more complete and aligned with the requirements:

- **Offline vs Online Indicator:** The app can detect network status and show a small indicator (e.g., a colored dot or text in the header) to inform the user if the app is offline. This could use the `navigator.onLine` property and a window event listener for `'online'/'offline'` events, or simply simulate it (since the environment is local). For demonstration, we might include a toggle in the UI to force "Offline Mode" to see how the app reacts – perhaps by showing a banner "You are offline. Some features may be unavailable." The logic for offline can be as simple as:

```
const [online, setOnline] = useState(true);
useEffect(() => {
  const updateStatus = () => setOnline(navigator.onLine);
  window.addEventListener('online', updateStatus);
  window.addEventListener('offline', updateStatus);
  return () => {
    window.removeEventListener('online', updateStatus);
    window.removeEventListener('offline', updateStatus);
  };
}, []);
```

  And then use `online` state to display a green dot (online) or red dot (offline) in the corner of the nav bar. This provides visual feedback of network status, which is a nod to real-world usage where the app might be used in the field on a tablet (and connectivity could be intermittent).

- **Tailwind UI components:** We leverage Tailwind to ensure the UI is not only clean but also interactive. For example, we use Tailwind classes for hover effects on buttons (`hover:bg-green-600` to darken on hover), disabled states (`disabled:opacity-50` for future module links), and responsive adjustments (`md:flex-row` vs `flex-col` layouts). We might use a pre-built component for things like modals or tooltips if needed (Tailwind UI library or similar), but given the minimal design, likely not many fancy components are needed.

- **Future Module Placeholders:** On the sidebar or menu, we reserve space for future expansion beyond cannabis. For instance, below the cannabis-specific links, we might show grayed-out menu items like "Alcohol (Coming Soon)" and "Mushrooms (Coming Soon)". These would be non-clickable (`<span className="text-gray-400 cursor-not-allowed">Alcohol</span>`),

just to indicate that the platform is envisaged to extend to other regulated substances. This is purely a UI element – no functionality, but it demonstrates a forward-looking design. It can be implemented by including those items in the `menuByRole` for roles that should eventually see them, but marking them as disabled in the component. For example:

```
{ menuItems.map(item => (
    <li key={item.label} className={item.disabled ? 'opacity-50 cursor-
not-allowed' : ''}>
      {item.disabled
        ? <span>{item.label}</span>
        : <NavLink to={item.path}>{item.label}</NavLink>
      }
    </li>
  ))
}
```

Where `item.disabled` is true for "Alcohol" and "Mushrooms" entries loaded from a config.

- **README and Documentation:** The scaffold includes a `README.md` that explains how to set up and run the project. It would outline steps such as:

- Prerequisites: Docker and Docker Compose installed.
- How to start: `docker-compose up --build` (which builds images for frontend/backend and starts Postgres).
- Credentials for login (listing the mock usernames/passwords for each role).
- Description of available URLs (http://localhost:3000 for app, and possibly http://localhost:3001 for API if one wants to hit the endpoints directly).
- Basic troubleshooting (e.g., if port conflicts occur, or reminding to use the right .env files).

Having this in the scaffold ensures a new developer or tester can quickly get the MVP running without guesswork.

- **Dev-Friendly Setup:** Because this is aimed at quick iteration, we have included features like hot-reload via volume mounts, simple mock data (so no complex environment needed), and the ability to run either in Docker or directly on host (developer can still do `npm start` in frontend or `npm run start:dev` in backend if they prefer that over Docker). This flexibility can be mentioned in docs. The idea is to lower the barrier to experiment with the code, including using GitHub Copilot – for example, a developer can open this project in VSCode with Copilot enabled and easily ask for implementations of the stubbed methods or generate additional components.

Conclusion

This project scaffold outlines a complete **seed-to-sale traceability MVP** for the cannabis industry, fulfilling the requirements with a modular, containerized TypeScript codebase. It separates concerns between a NestJS backend (providing RESTful APIs with mock data and structured modules) and a React frontend (offering a role-specific user experience with minimalistic design). By using Docker Compose, we ensure that both the development and future deployment environments are easy to manage and reproducible.

The UI is intentionally kept simple and intuitive – *"Apple-style simplicity"* in action – to allow users to focus on functionality. Each user role sees a tailored interface: regulators and auditors can explore the entire lifecycle of cannabis products and verify data integrity, while operators can efficiently record their cultivation activities. The inclusion of a blockchain integrity simulation, even at MVP stage, demonstrates forward-thinking design: highlighting how cryptographic hashes can secure the data [15] and providing a foundation for real blockchain integration later.

With this scaffold in place, developers can leverage GitHub Copilot to rapidly fill in any remaining pieces: for instance, writing the actual code for services, refining the UI components, or even extending the platform. The project is ready for **fast iteration** – new features like adding lab tests or inventory tracking can be added by generating new modules or components following the established patterns. In summary, the provided structure is comprehensive yet approachable, aiming to **simulate a full traceability platform** without heavy dependencies, while keeping the door open for future enhancements in authentication, real data handling, and expanded regulatory coverage.

Sources:

1. Agba, D. *Initiating a NestJs app with PostgreSQL using Docker* – DEV Community [19] (example Docker Compose for Postgres service and env setup).
2. Cortisse, A. *Setting up dev environment for React/NestJS applications* – arnaudcortisse.com (2024) [5] [4] (Docker Compose with NestJS, React, Postgres services; dev workflow).
3. Purani, A. *Role-Based Menu Rendering in React* – SevenSquare Tech (July 2025) [10] [11] (concept of conditional UI by role and code example using context for menu items).
4. Vercel Design Guidelines – *Apple-style simplicity* in UI (clean, uncluttered interface with ample whitespace and clear typography) [1] .
5. High Peaks, *Blockchain and Cannabis: Seed-to-sale Tracking* (2023) – on using blockchain for transparent seed-to-sale traceability [15] .
6. RenewEdge Solutions, *Blockchain Commit Lifecycle* (2025) – describes hashing events (SHA-256) and ledger submission for traceability integrity [13] .
7. Mandal, D. *NestJS Controllers* – Medium (Apr 2024) [2] (NestJS key features: TypeScript, modular architecture).

---

1 Frontend design guidelines - v0 by Vercel
https://v0.app/chat/frontend-design-guidelines-o666EUJHd3z

2 Controllers (NestJS Core Series 01) | by Deepak Mandal | Medium
https://danimai.medium.com/controllers-in-nestjs-fb0ce27935a2

3 8 19 Initiating a NestJs app with PostgreSQL using Docker - DEV Community
https://dev.to/blankgodd/initiating-a-nestjs-app-with-postgresql-using-docker-4bag

4 5 6 7 Trying out NestJS part 1: Setting up dev environment for your React / NestJS applications that rocks
https://arnaudcortisse.com/blog/trying-out-nestjs-part-1/

9 Controllers | NestJS - A progressive Node.js framework
https://docs.nestjs.com/controllers

10 11 12 16 17 18 Role-Based Menu Rendering in ReactJS (With GitHub Code)
https://www.sevensquaretech.com/role-based-menu-reactjs-with-github-code/

13 14 Blockchain Commit Lifecycle.docx
file://file-2uaCYDkoqSeSsifXuWEYL3

15 The Intersection of Blockchain and Cannabis
https://highpeaks.com/the-intersection-of-blockchain-and-cannabis-tracking-seed-to-sale/