**PROJECT DOCUMENTATION**

**Store Manager: Keep track of inventory**

**Project Title: Store Manager: Keep track of inventory**

**Team ID: NM2025TMID36522**

**Team Leader:**

* **Keerthana. B-**  [**24cskeerthanab041@gmail.com**](mailto:%2024cskeerthanab041@gmail.com)

**Team Members:**

* **Jothi sree. H-** [**24csjothisree038@gmail.com**](mailto:24csjothisree038@gmail.com)
* **Karthika. R-** [**24cskarthika039@gmail.com**](mailto:24cskarthika039@gmail.com)
* **KavyaDarsiny.B-** [**24cskavyadarsiny040@gmail.com**](mailto:24cskavyadarsiny040@gmail.com)

**Project Overview**

**Purpose:**

The Store Inventory Management System helps businesses track products, manage stock levels, and monitor sales. It ensures real-time inventory updates, reduces manual errors. The main objective of this project is to simplify stock tracking. It ensures that the right products are available at the right time, improving both business efficiency and customer service.

The benefits of the system include saving time, reducing mistakes, providing real-time data, and giving store owners a clear overview of their products. With this system, managing a store becomes more professional, transparent, and convenient.

### The system will be accessible through any browser.

### Supports multiple roles: Admin, Store Manager, Cashier, and Employee.

### Secure login with role-based access.

### Integration with MongoDB for scalable data storage.

### RESTful APIs to handle backend operations.

**Features:**

* Product management (add, update, delete items).
* Stock monitoring with alerts for low inventory.
* Sales and purchase tracking.
* Secure user authentication (admin, staff).
* Dashboard with analytics and reports.

### Existing solutions like Tally ERP, Zoho Inventory, and Odoo provide powerful inventory and billing features. However, these solutions are often too costly, require technical expertise, and include unnecessary modules for small businesses.

* [**Inventory Management Methods**: Learn various methods that can enhance accuracy and efficiency in retail and e-commerce operations.](https://www.bing.com/ck/a?!&&p=1ceb52df6de065b8b868b6ccf49dfa5e78aeba13e80545191c306e8ba6fa2ce6JmltdHM9MTc1NzgwODAwMA&ptn=3&ver=2&hsh=4&fclid=0ba849ec-67db-6839-222d-5fb366a46956&psq=store+inventory&u=a1aHR0cHM6Ly93d3cubmV0c3VpdGUuY29tL3BvcnRhbC9yZXNvdXJjZS9hcnRpY2xlcy9pbnZlbnRvcnktbWFuYWdlbWVudC9yZXRhaWwtaW52ZW50b3J5LW1hbmFnZW1lbnQuc2h0bWw&ntb=1)
* [**Inventory Examples**: Access downloadable examples of store inventory lists to better understand inventory management.](https://www.bing.com/ck/a?!&&p=123af1865cc469ecbb7eeca0c6998a5faff89972efb5ce13e7db0022419987e9JmltdHM9MTc1NzgwODAwMA&ptn=3&ver=2&hsh=4&fclid=0ba849ec-67db-6839-222d-5fb366a46956&psq=store+inventory&u=a1aHR0cHM6Ly93d3cuZXhhbXBsZXMuY29tL2J1c2luZXNzL3N0b3JlLWludmVudG9yeS5odG1s&ntb=1)
* [**Importance of Inventory Management**: Understand the fundamental role of inventory management in ensuring the right stock at the right time.](https://www.bing.com/ck/a?!&&p=fe80d627d997ce824bcf42b8f6fefdca901747f05b2eab27db6f40be79ce1aaaJmltdHM9MTc1NzgwODAwMA&ptn=3&ver=2&hsh=4&fclid=0ba849ec-67db-6839-222d-5fb366a46956&psq=store+inventory&u=a1aHR0cHM6Ly93d3cuaGFzaG1pY3JvLmNvbS9waC9ibG9nL3JldGFpbC1pbnZlbnRvcnktbWFuYWdlbWVudC8&ntb=1)

**Architecture:**

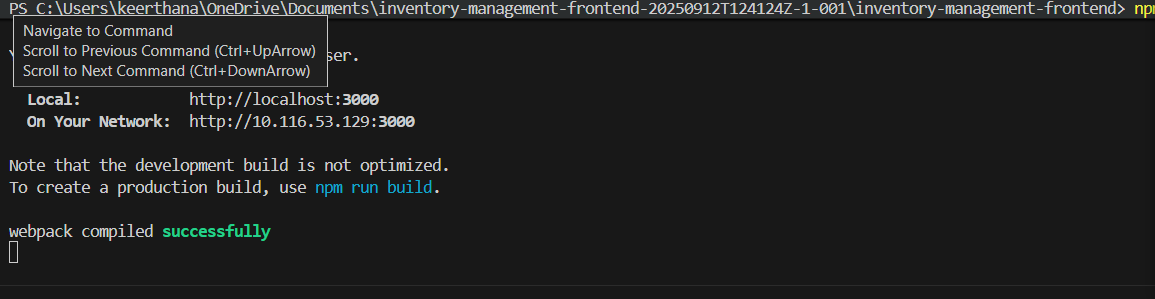
### 

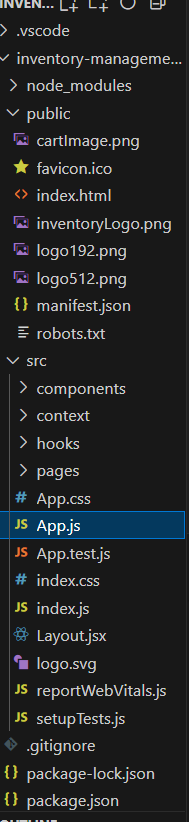
### The Store Manager follows a 3-tier architecture:

### 1. Presentation Layer – React.js frontend for user interaction.

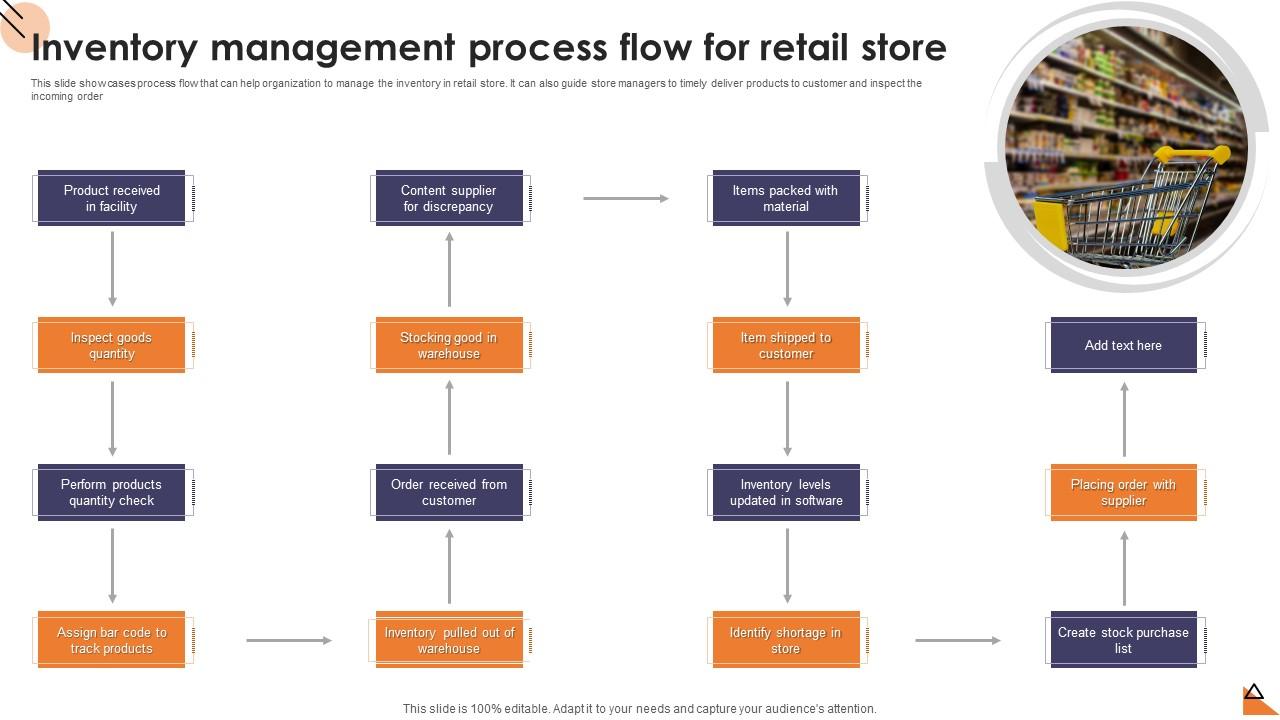
### 2. Application Layer – Node.js & Express.js for API and business logic.

### 3. Data Layer – MongoDB for storing products, employees, and transactions.

****

****

**Set-up Instructions:**



To begin a web development project that uses Node.js, MongoDB, Git, React.js, Express.js, Mongoose, and Visual Studio Code, you must have a solid understanding of several key technologies. The following prerequisites are essential:

### Core Programming Languages

* **JavaScript (ES6+)**: This is the most critical prerequisite. You must have a strong grasp of modern JavaScript, including asynchronous programming concepts like async/await and Promises, as it is the foundation for Node.js, Express.js, React.js, and much of the overall ecosystem.
* **HTML & CSS**: A basic understanding of HTML for structuring web pages and CSS for styling them is necessary, especially for the frontend portion using React.js.

### Development Tools & Concepts

* **Command Line Interface (CLI)**: You need to be comfortable navigating your computer's file system, running commands, and managing dependencies using a terminal.
* **Git & GitHub**: You must know how to use Git for version control. This includes basic commands like clone, add, commit, push, and pull. A GitHub account is essential for hosting and sharing your code.
* **Node Package Manager (npm)**: You must know how to use npm to install and manage project dependencies and to run scripts.

### Backend-Specific Prerequisites

* **Node.js & Express.js**: Understand how a backend server works. You should know how to create routes, handle HTTP requests, and manage middleware with Express.js, which is built on Node.js.
* **MongoDB & Mongoose**: While you'll learn the specifics of these two technologies, a basic understanding of database concepts is helpful. Mongoose is an Object Data Modeling (ODM) library that helps you interact with MongoDB from your Node.js application, so knowing how a schema works and what CRUD (Create, Read, Update, Delete) operations are is beneficial.

### Frontend-Specific Prerequisites

* **React.js**: To work with React, you should be familiar with its core concepts, such as components, state, props, and hooks. Knowing how to manage the user interface and handle user events is fundamental.

### Integrated Development Environment (IDE)

* **Visual Studio Code (VS Code)**: While not a prerequisite for the technologies themselves, VS Code is the recommended and most widely used code editor for this stack. Familiarity with its interface, extensions, and built-in features will significantly boost your productivity.

**Folder Structure:**

**store-inventory-app/**

**│── src/**

**│ ├── App.js → Main application logic (state + functions + layout)**

**│ ├── components/ → Reusable components folder**

**│ │ ├── InventoryForm.js → Form to add new items (name + quantity)**

**│ │ ├── InventoryList.js → Shows list of all inventory items**

**│ │ └── InventoryItem.js → Handles single item (update, delete actions)**

**│ ├── index.js → Entry point of React app (renders <App />)**

**│ └── App.css → Styling for your app**

**Running the Application:**

**FRONTEND:**

* **cd client**
  + cd stands for **change directory**. This command is used to navigate between folders in your file system.
  + client is the name of the folder where the frontend code is located. Many full-stack projects separate the frontend from the backend into folders named client and server (or frontend and backend).
  + This command changes your current location in the terminal from the project's root directory to the client folder.
* **npm start**
  + npm stands for **Node Package Manager**. It's a command-line tool that comes with Node.js and is used to manage project dependencies and run scripts.
  + start is a **pre-defined script** in the package.json file of a project. When you run npm start, it executes a command (often react-scripts start or a similar command) that launches the development server for your frontend application.
  + This server runs locally on your machine, allowing you to view and test your application in a web browser. It also enables features like hot reloading, where changes to your code are automatically reflected in the browser without a manual refresh.

**BACKEND:**

**cd server**

* cd stands for **change directory**. This command is used to move between folders in your file system.
* server is the name of the folder where the backend code is located. In many full-stack projects, the backend and frontend are separated into distinct directories, often named server and client respectively.
* This command changes your current location in the terminal from the project's root directory to the server folder.
* Access: Visit <http://localhost:3000>

**User Interface:**

Let's break down the different user interfaces (UIs) you've mentioned, applying them to the context of a "store inventory" application. Each UI serves a distinct purpose and is designed for a specific type of user.

### 1. Landing Page

This is the first page a user sees when they visit your application's URL. Its primary goal is to provide a brief overview of the application and encourage new users to sign up or existing users to log in.

**Key Elements:**

* **Hero Section**: A prominent section with a compelling headline, a brief description of the app's purpose ("Effortless Inventory Management"), and a clear call-to-action (CTA) button like "Sign Up" or "Get Started."
* **Feature Highlights**: Sections that showcase the main benefits of the app, such as "Real-Time Tracking," "Simplified Stock Management," or "Barcode Scanning Integration."
* **Testimonials/Social Proof**: Quotes or logos from satisfied users or companies to build trust.
* **Navigation**: Links to other pages like "About Us," "Pricing," and "Login."
* **Footer**: Contains contact information, legal links, and social media icons.

For a store inventory app, the landing page would be the public-facing side, designed to attract potential business users.

### 2. Freelancer Dashboard

This UI is designed for a specific user type: a freelancer who uses the inventory app to manage their own small business or a client's inventory. It's a personalized space that provides an at-a-glance summary of their inventory data.

**Key Elements:**

* **Summary Widgets**: Small cards or boxes that show key metrics like "Items in Stock," "Low Stock Alerts," "Recent Sales," and "Total Value of Inventory."
* **Quick Actions**: Buttons or links to perform common tasks quickly, such as "Add New Item," "View Low Stock," or "Generate Report."
* **Recent Activity Feed**: A list of the most recent inventory transactions, such as items added or removed.
* **Navigation**: A sidebar or header menu with links to detailed views like "All Items," "Sales History," "Reports," and "Settings."

The freelancer dashboard provides a streamlined experience for a user who needs to manage a single or small set of inventories efficiently.

### 3. Admin Panel

This UI is for the administrator of the entire application. It's a powerful tool designed to manage all aspects of the system, including users, permissions, and system-wide data. This panel would not be accessible to regular users.

**Key Elements:**

* **User Management**: The ability to view, add, edit, or delete user accounts (e.g., freelancers, store managers).
* **System-Wide Analytics**: Charts and graphs showing overall app usage, such as the total number of users, active inventories, and most popular features.
* **Content Management**: Tools to manage system-wide content, such as a blog or help center articles.
* **Settings**: The ability to configure global application settings.
* **Security & Logs**: Access to a log of system events and the ability to manage security settings.

The admin panel is a backend-focused UI that gives the developer or owner of the application full control over the platform.

### About the Store Inventory

The core of all these UIs is the concept of a **Store Inventory**. This refers to the data model itself, which includes:

* **Items**: The products or goods being tracked. Each item would have attributes like name, quantity, price, SKU (Stock Keeping Unit), and possibly an image.
* **Tracking**: The ability to log changes to the inventory. This could involve recording when items are added, sold, or removed.
* **Status**: The current state of an item (e.g., "In Stock," "Low Stock," or "Out of Stock").

**Testing:**

Testing is a crucial part of the software development lifecycle, and for a store inventory application, it ensures accuracy, reliability, and a good user experience. Here's how you would approach manual testing for your inventory app, along with the specific tools you mentioned.

### Manual Testing during Milestones

Manual testing is the process of a human tester physically interacting with the software to find bugs and ensure it meets the requirements. Performing it during milestones is a strategic approach to catch issues early.

**Milestone 1: Backend API Development**

* **Focus**: The core functions of your application are the **API endpoints** (routes) that handle the inventory data. At this stage, you don't have a frontend yet, so you need to test the backend directly.
* **Testing Scenarios**:
  + **Add an item**: Test the API endpoint for creating a new item. Send a request with a valid item name and quantity, then verify that the item is successfully added to the database.
  + **Retrieve items**: Test the API endpoint that retrieves all inventory items. Make sure the response is a list of all the items you've added.
  + **Update an item**: Test the endpoint for updating an item's quantity or name.
  + **Delete an item**: Test the endpoint that removes an item from the inventory.
* **Tools**: **Postman** is the perfect tool for this. It allows you to create and send HTTP requests (GET, POST, PUT, DELETE) to your backend API endpoints and inspect the responses. You can save your test cases as a "collection" for easy re-execution.

**Milestone 2: Frontend-Backend Integration**

* **Focus**: The goal here is to ensure the frontend (React.js) can successfully communicate with the backend API.
* **Testing Scenarios**:
  + **Create Item**: Use the InventoryForm component in your browser to add a new item. Manually verify that the item appears in the InventoryList and that the data is also reflected in your MongoDB database.
  + **Update/Delete Item**: Interact with the InventoryItem component to update or delete an item. Confirm that the changes are shown on the frontend and are also persisted in the database.
  + **Edge Cases**: Test for scenarios where a user might enter invalid data (e.g., a negative quantity, an empty item name) to see how the frontend and backend handle the error.
* **Tools**: **Postman** (for checking the backend directly) and **Chrome DevTools** (for checking the frontend).

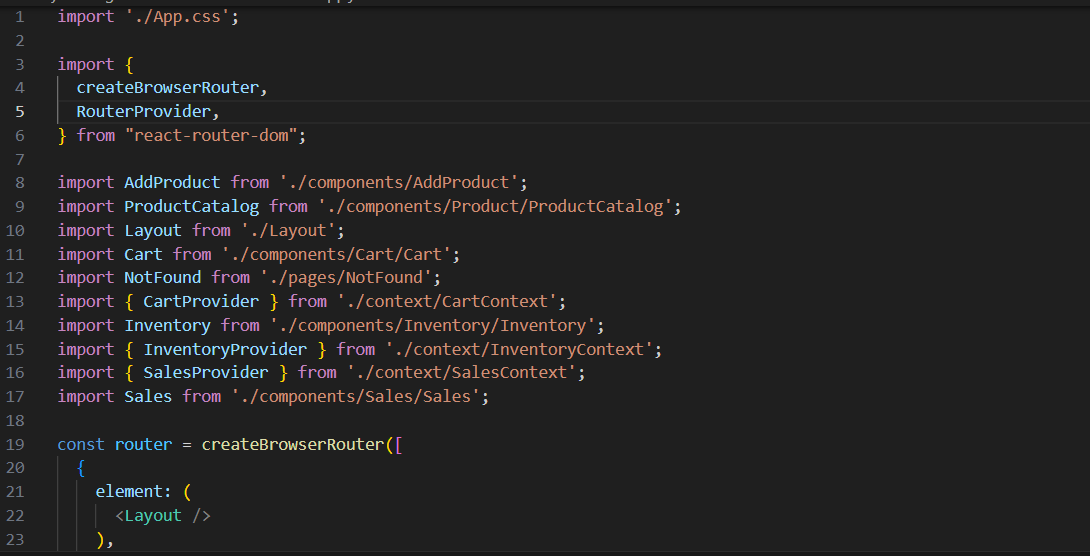
**Milestone 3: Complete User Interface (UI) Functionality**

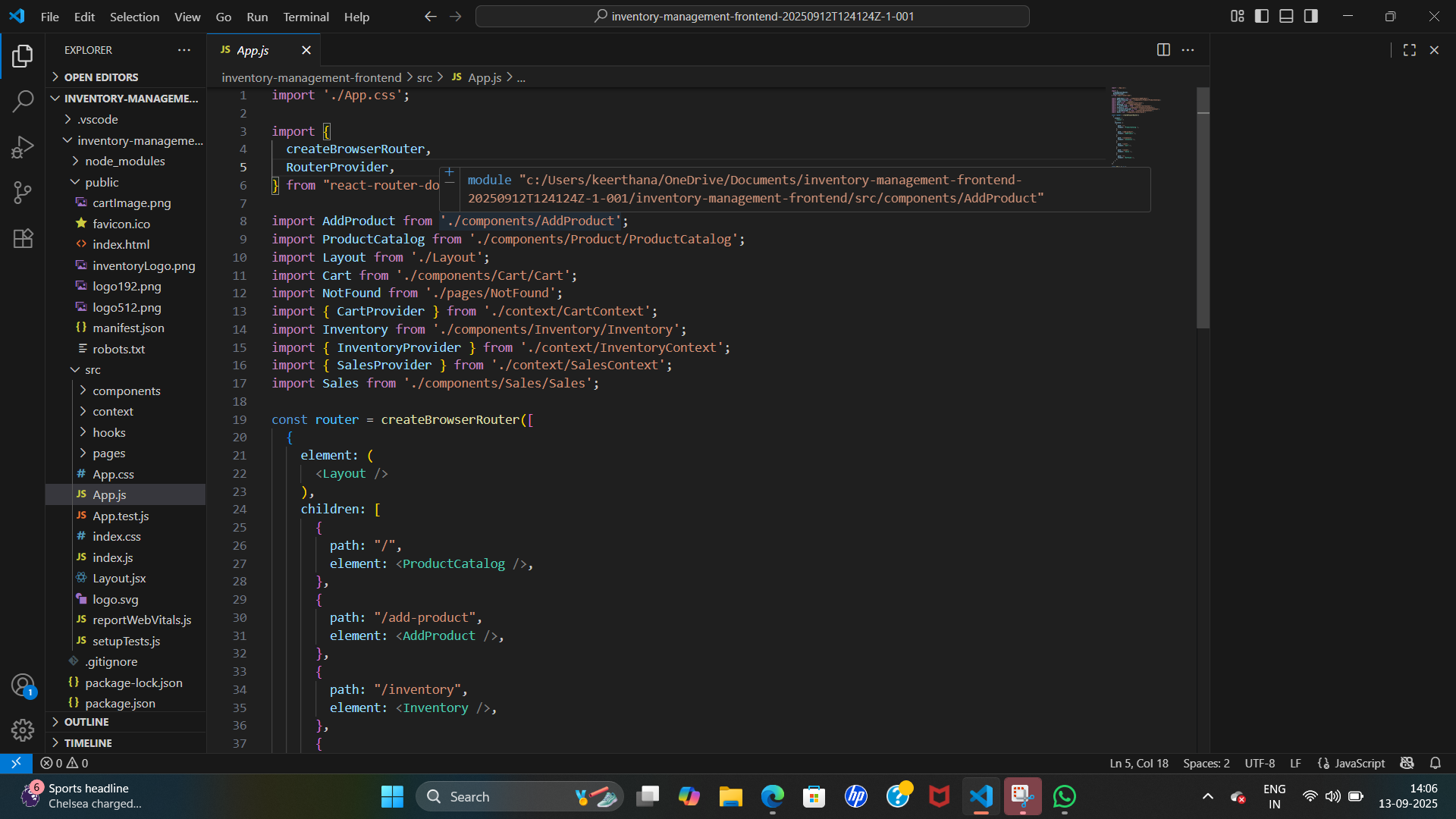
* **Focus**: This is a comprehensive test of the entire application, including the look and feel.
* **Testing Scenarios**:
  + **Full User Workflow**: Go through the entire process from adding an item to updating its quantity, deleting it, and refreshing the page to ensure data persistence.
  + **UI/UX Testing**: Check for visual consistency, responsiveness on different screen sizes, and overall usability.
  + **Data Integrity**: Verify that the numbers in the UI (e.g., total items) are always accurate and reflect the data in the database.

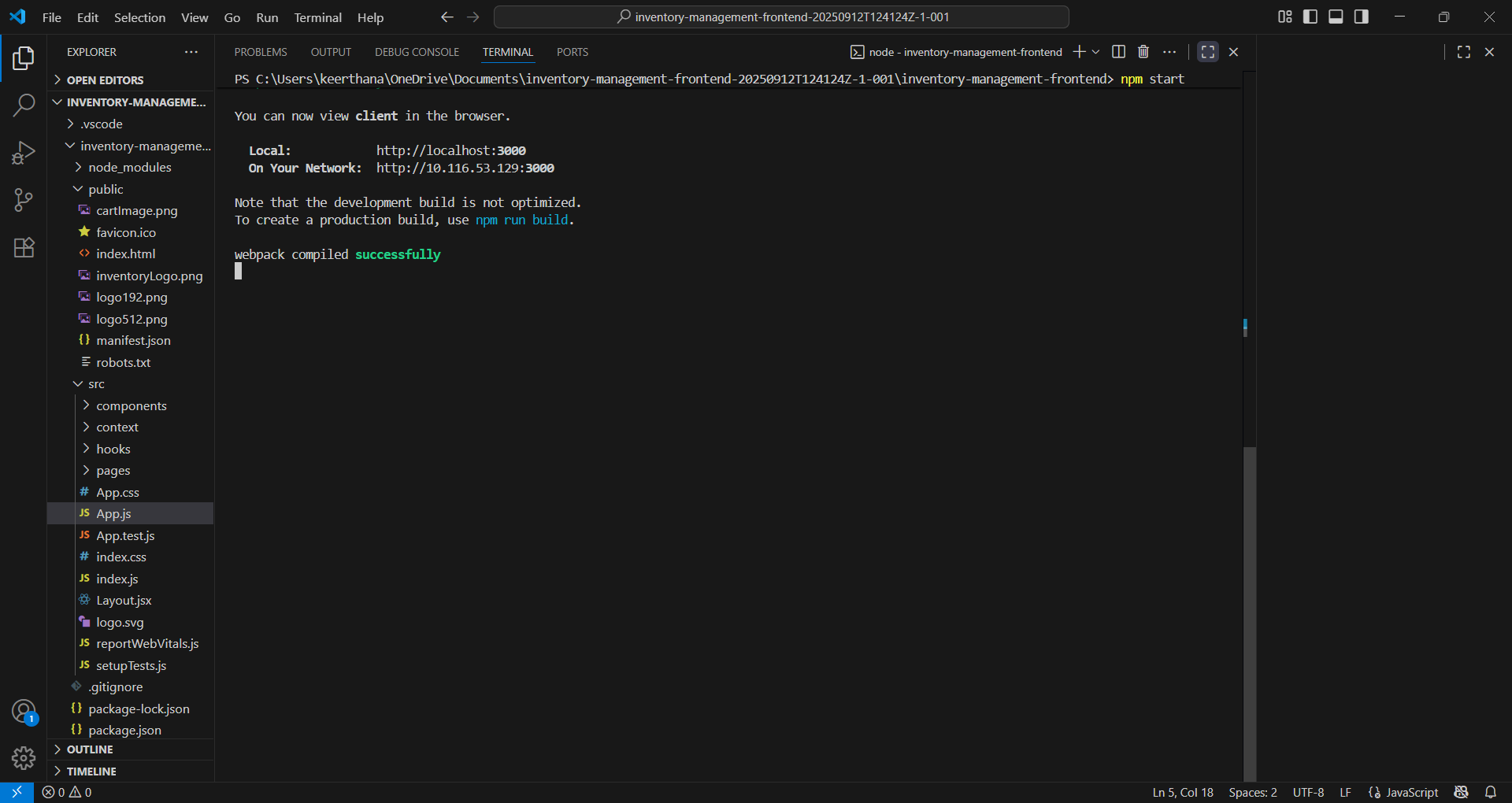
### Testing Tools in Detail

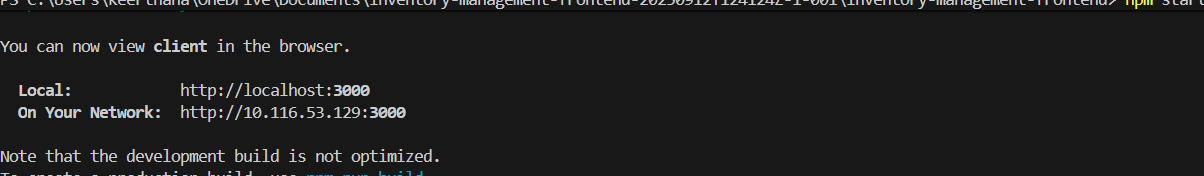
* **Postman**: As a standalone application, Postman is essential for **API testing**. You can use it to:
  + **Validate API Endpoints**: Confirm that each endpoint works as expected by sending requests and checking the status codes (e.g., 200 OK for success, 404 Not Found, 400 Bad Request) and response bodies.
  + **Automate Tests**: You can write simple JavaScript tests within Postman to automatically validate that the response data matches your expectations. This is especially useful for regression testing.
  + **Share Collections**: You can save and share your collection of API tests with other team members, ensuring everyone is testing the same scenarios.
* **Chrome DevTools**: This is your primary tool for **frontend testing**.
  + **Elements Panel**: Inspect and modify the HTML and CSS of your InventoryList and InventoryForm in real time to test how different styles and layouts would look.
  + **Console Panel**: This is critical for debugging. You can check for JavaScript errors that occur during user interactions, view console logs you've added to your code, and interact with your application's state directly.
  + **Network Panel**: A must-have for debugging the frontend-backend connection. You can see every API request your frontend makes, inspect the request headers and body, and analyze the response from the server. This helps you identify if the frontend is sending the wrong data or if the backend is returning an unexpected response.
  + **Application Panel**: This panel lets you inspect your local storage and session storage, which are often used for storing user data like a token. For a simple inventory app, you can use it to check if a user is logged in or out.

**SCREENSHOTS :**

****

****

****

****

**DEMO LINK:**

**https://drive.google.com/file/d/1EiC4EEIEfL21nqLXDQYMt2uqZoYp5L6J/view?usp=drivesdk**

**KNOWN ISSUES:**

Based on the technologies and components you've mentioned for a store inventory app, here are some common issues that are likely to arise, categorized by the area of the application.

### 1. Frontend (React.js)

* **State Management Complexity**: As your app grows, managing the inventory list's state in the main App.js component can become difficult. It's a "known issue" that simple useState can lead to props drilling (passing props down through many nested components).
* **UI/UX Issues**: The layout might not be responsive across all devices (mobile, tablet, desktop). Buttons or forms may not be intuitive, leading to a poor user experience.
* **Component Re-rendering**: If not optimized, the entire InventoryList might re-render every time a single InventoryItem is updated, which can slow down the application, especially with a large number of items.
* **Input Validation**: The InventoryForm might not have robust client-side validation, allowing a user to submit a negative quantity or an empty item name, leading to an API error.

### 2. Backend (Node.js, Express.js, Mongoose)

* **API Inconsistency**: The API endpoints might not follow RESTful conventions, making them difficult to use. For example, using a GET request to delete an item, or having inconsistent URL paths.
* **Error Handling**: The API might not return meaningful error messages. If a database operation fails, the server might crash or return a generic "Internal Server Error" (500), making it hard for the frontend to handle the issue gracefully.
* **Data Validation**: The backend might not validate incoming data from the frontend. This could lead to saving invalid data to the database (e.g., a non-numeric value for quantity). This is a security risk.
* **Connection Management**: The backend might not properly handle the database connection. A common issue is not closing a connection after a request, which can lead to a large number of open connections that degrade performance.

### 3. Database (MongoDB)

* **Schema Design Issues**: The MongoDB schema might be poorly designed, leading to data redundancy or making it difficult to query data efficiently. For example, not having the right indexes on fields that are frequently queried can lead to slow search times.
* **Concurrency Problems**: In a multi-user environment, two users might try to update the same item at the same time. Without proper concurrency controls, this could lead to data corruption.
* **Data Migration**: When you need to change the schema (e.g., add a new field to an item), updating all existing documents in the database can be a complex and risky process.

### 4. Git and Version Control

* **Merge Conflicts**: Multiple developers working on the same files (like App.js or package.json) can lead to merge conflicts when they try to push their code to the same branch.
* **Branching Strategy**: Without a clear branching strategy (e.g., Git Flow), a repository can become cluttered and disorganized, making it hard to track changes.

### 5. Deployment and Operations

* **Environment Variables**: Mismanaging environment variables (e.g., database connection strings, API keys) can lead to security vulnerabilities or make the application fail to start in production.
* **Performance**: The application might perform well locally but struggle under heavy load in a production environment due to unoptimized queries or a lack of server resources.
* **Security**: Not sanitizing user input can lead to **injection attacks**. Storing sensitive data in plain text is another critical security issue.
* **Monitoring**: A lack of monitoring tools means that you won't know when the application has an issue until users report it.

**FUTURE ENHANCEMENTS:**

A well-designed store inventory application is a great starting point, but to make it truly powerful and competitive, you can add a range of features. Here are some key future enhancements for a store inventory app, moving from basic functionality to a more advanced, intelligent system.

### 1. Advanced Inventory Tracking & Identification

* **Barcode and QR Code Scanning**: Instead of manual data entry, implement the ability to use a phone's camera (via the frontend) to scan barcodes or QR codes. The backend would then retrieve or update the item's information automatically.
* **RFID Integration**: For even more advanced tracking, incorporate RFID (Radio-Frequency Identification) technology. This would allow for quick, bulk scanning of items without a direct line of sight, which is invaluable for large warehouses.
* **Batch and Serial Number Tracking**: For products that need strict traceability (e.g., food, electronics), add the ability to track inventory by batch number or unique serial number.

### 2. Enhanced Analytics and Reporting

* **Predictive Analytics & Demand Forecasting**: Use the historical sales data in your MongoDB database to build a predictive model. The app could then suggest optimal reorder points and quantities, preventing stockouts and overstocking.
* **Customizable Reports**: Allow users to generate and export detailed reports on inventory valuation, sales trends, and turnover rates.
* **AI-Powered Insights**: Implement AI to identify slow-moving items ("dead stock") or seasonal trends, providing proactive recommendations to the user.

### 3. Supply Chain and Vendor Management

* **Automated Reordering**: The system could automatically generate and send purchase orders to suppliers when stock levels fall below a predefined threshold, streamlining the replenishment process.
* **Supplier Management**: Create a dedicated section to manage supplier information, track delivery lead times, and monitor supplier performance.
* **Multi-Location Management**: Allow users to manage inventory across multiple physical stores or warehouses from a single dashboard. This would include the ability to track transfers between locations.

### 4. User and Security Enhancements

* **User Roles and Permissions**: Differentiate between user types (e.g., administrator, store manager, employee). An employee might only be able to view and count inventory, while a manager can update prices and place orders.
* **User Activity Logs**: Implement a system to log all user actions (e.g., "John Doe updated the quantity of 'Item X'"). This is crucial for accountability and auditing.
* **Two-Factor Authentication (2FA)**: Add an extra layer of security to the login process to protect user accounts.

### 5. Mobile and Integration Features

* **Dedicated Mobile App**: Beyond a responsive web design, a native mobile app for iOS and Android would provide a better user experience, especially for on-the-go inventory management tasks like receiving shipments.
* **E-commerce Platform Integration**: Seamlessly integrate with popular e-commerce platforms like Shopify or WooCommerce. This would allow the app to automatically sync inventory data across both the physical store and the online store.
* **Accounting Software Integration**: Connect with accounting tools like QuickBooks or Xero to automatically sync inventory costs, sales data, and other financial metrics.

By implementing these enhancements, the application would evolve from a simple list-based tool into a comprehensive, intelligent, and interconnected business solution.

The development of a store inventory application, from concept to completion, is a comprehensive journey that integrates multiple technologies and disciplines. The project is fundamentally a full-stack web application, strategically divided into a frontend and a backend to ensure a clear separation of concerns and promote scalability.

The **frontend**, built with **React.js**, serves as the user-facing layer. Its well-structured folder layout, defined by the core App.js and modular components, facilitates a clean, reusable, and maintainable codebase. This separation is crucial for providing distinct user interfaces—a public-facing landing page, a user-centric freelancer dashboard, and a powerful admin panel—each tailored to a specific user's needs.

The **backend**, powered by **Node.js** and **Express.js**, is the operational backbone. It handles the business logic, manages communication with the database, and exposes RESTful API endpoints. **MongoDB**, as the NoSQL database, offers flexibility in storing item data, while **Mongoose** provides a structured way for the backend to interact with the database. This backend ensures that all inventory data is persistent, secure, and accessible.

Throughout the development process, **version control with Git** is indispensable for collaborative work. **Testing** at every milestone is paramount for quality assurance, with **Postman** for API validation and **Chrome DevTools** for frontend debugging. Addressing common issues such as state management complexity, poor error handling, and security vulnerabilities is a continuous effort that solidifies the application's reliability.

Ultimately, the app's potential extends far beyond basic functionality. By integrating **advanced features** like barcode scanning, predictive analytics, and automated reordering, the application can transform from a simple tracking tool into an intelligent business solution. This evolution from a foundational structure to a feature-rich, integrated system is what defines the successful development of a robust and enduring software product.

THANK YOU