**Frontend:**

* **React**: A modern JavaScript library for building interactive user interfaces.
* **Progressive Web App (PWA)**: To enable offline functionality and provide a mobile-friendly experience.

**Backend:**

* **Node.js**: A JavaScript runtime for building scalable backend services.
* **Express.js**: A lightweight framework for creating RESTful API endpoints.

**Database:**

* **MySQL**: A robust relational database to store and manage inventory data.

**Additional Tools:**

* **IndexedDB (Frontend)**: For offline storage when the app is in PWA mode.
* **Local MySQL Setup (Development)**: Hosting the database locally during development.

**Why This Stack Works:**

1. **Scalable:** React + Node.js can handle both small and large projects.
2. **Flexibility:** MySQL allows you to structure data in a way that supports future growth.
3. **Offline Functionality:** PWA and IndexedDB provide seamless offline support.
4. **Widely Supported:** All components have large communities, excellent documentation, and plenty of tutorials.

**Backend Documentation for Inventory Management System**

**Overview**

This backend application is built using **Node.js** with **Express.js** for handling API requests and **MySQL** for the database. It manages inventory categories, subcategories, and items, and provides endpoints for interacting with the data.

**Setup**

**Prerequisites**

* **Node.js** (v20.11.1)
* **npm** (v10.2.4)
* **MySQL Server**

**Installation Steps**

1. Clone or create the project directory.
2. Initialize the project:
3. npm init -y
4. Install dependencies:
5. npm install express mysql2 cors body-parser dotenv
6. Create a .env file to store environment variables securely:
7. DB\_HOST=localhost
8. DB\_USER=root
9. DB\_PASSWORD=YourPassword
10. DB\_NAME=InventorySystem
11. PORT=5000
12. Ensure .env is added to .gitignore to prevent it from being uploaded to version control.

**Project Structure**

project-folder/

|-- index.js # Entry point of the application

|-- package.json # Project metadata and dependencies

|-- .env # Environment variables

**Features Implemented**

**Middleware**

* **CORS**: Enables cross-origin resource sharing for frontend-backend communication.
* **Body-Parser**: Parses incoming JSON request bodies.

**MySQL Connection**

The backend connects to the MySQL database using the mysql2 library. Connection settings are stored in the .env file for security.

const db = mysql.createConnection({

host: process.env.DB\_HOST,

user: process.env.DB\_USER,

password: process.env.DB\_PASSWORD,

database: process.env.DB\_NAME,

});

db.connect((err) => {

if (err) {

console.error('Error connecting to MySQL:', err);

} else {

console.log('Connected to the MySQL database!');

}

});

**API Endpoints**

**Root Endpoint**

* **GET /**
  + Response: Confirms the server is running.
* API is running!

**Categories**

* **GET /categories**
  + Fetches all categories from the Categories table.
  + Example Response:
  + [
  + { "idCategory": 1, "Name": "General Surgical Instrumentation" },
  + { "idCategory": 2, "Name": "Internal Fixation" }
  + ]

**Items by Subcategory**

* **GET /items/:subcategoryId**
  + Fetches all items belonging to a specific subcategory based on SubcategoryId.
  + Parameters:
    - subcategoryId: ID of the subcategory.
  + Example Request:
  + GET /items/1
  + Example Response:
  + [
  + { "idItem": 1, "Name": "Halstead Mosquito", "Details": "Used for clamping blood vessels", "Quantity": 50 },
  + { "idItem": 2, "Name": "Kelly Artery Forceps", "Details": "Curved and straight styles available", "Quantity": 40 }
  + ]

**Next Steps**

1. **Extend API Functionality**:
   * Add endpoints for creating, updating, and deleting categories, subcategories, and items.
   * Implement search and filtering functionality.
2. **Implement Pagination**:
   * Useful for handling large datasets in categories or items.
3. **Connect Backend to Frontend**:
   * Use React to build the user interface.
4. **Error Handling**:
   * Standardize error responses and log errors for debugging.
5. **Offline Support**:
   * Integrate offline functionality using IndexedDB for the frontend.

**Notes**

* Always use parameterized queries to prevent SQL injection.
* Test the API endpoints thoroughly using tools like Postman.

**References**

* **Node.js**: <https://nodejs.org/>
* **Express.js**: <https://expressjs.com/>
* **MySQL**: <https://www.mysql.com/>
* **Postman**: <https://www.postman.com/>

**Vet Inventory System Progress Summary**

**Folder Structure Restructuring**

* Restructured the project directory for better organization:
* VetInventorySystem/
* ├── backend/
* │ ├── index.js
* │ ├── routes/
* │ │ ├── categories.js
* │ │ └── items.js
* │ │ └── subcategories.js
* ├── frontend/
* │ ├── src/
* │ ├── public/
* │ └── package.json
* Separated frontend and backend into their respective directories for better modularity and scalability.
* Renamed inventory-backend to backend and inventory-frontend to frontend for simplicity.

**Backend Updates**

1. **Route Modularization**:
   * Moved backend routes to a new routes/ directory:
     + categories.js: Handles CRUD operations for categories.
     + items.js: Manages item-related operations.
     + subcategories.js: Handles subcategory deletion.
   * Updated index.js to use the modular routes.
   * const categoriesRoutes = require('./routes/categories');
   * const itemsRoutes = require('./routes/items');
   * const subcategoriesRoutes = require('./routes/subcategories');
   * app.use('/categories', categoriesRoutes);
   * app.use('/items', itemsRoutes);
   * app.use('/subcategories', subcategoriesRoutes);
2. **Database Connection Injection**:
   * Added middleware to inject the MySQL database connection (req.db) into all route handlers for cleaner code:
   * app.use((req, res, next) => {
   * req.db = db;
   * next();
   * });
3. **Removed Duplicate Routes**:
   * Cleaned up duplicate implementations of POST /categories and PUT /items/:id for better maintainability.
4. **Added Pagination and Search**:
   * Pagination for /categories to handle large datasets:
   * router.get('/', (req, res) => {
   * const { page = 1, limit = 10 } = req.query;
   * const offset = (page - 1) \* limit;
   * db.query('SELECT \* FROM Categories LIMIT ? OFFSET ?', [parseInt(limit), offset], (err, results) => {
   * if (err) {
   * res.status(500).send(err);
   * } else {
   * res.json(results);
   * }
   * });
   * });
   * Search functionality for categories by name:
   * router.get('/search', (req, res) => {
   * const { q } = req.query;
   * db.query('SELECT \* FROM Categories WHERE Name LIKE ?', [`%${q}%`], (err, results) => {
   * if (err) {
   * res.status(500).send(err);
   * } else {
   * res.json(results);
   * }
   * });
   * });

**Frontend Updates**

1. **Routing Setup**:
   * Added react-router-dom for frontend routing.
   * Separated frontend pages into src/pages with components for Categories and Items.
   * src/pages/
   * ├── Categories.js
   * └── Items.js
2. **Navigation and Dynamic Data Loading**:
   * Categories.js lists categories and uses Link to navigate to Items.js for specific category items.
   * Items.js dynamically fetches and displays items based on the category ID from the URL.
3. **Axios Integration**:
   * Set up an apiService.js file in src/services for managing API calls to the backend.
   * import axios from 'axios';
   * const API = axios.create({
   * baseURL: 'http://localhost:5000',
   * });
   * export const getCategories = async () => API.get('/categories');
   * export const getItemsBySubcategory = async (subcategoryId) => API.get(`/items/${subcategoryId}`);
4. **Tested API Connectivity**:
   * Confirmed frontend components fetch data correctly from backend routes.

**Next Steps**

1. **Backend**:
   * Add role-based access control (e.g., admin vs. regular users).
   * Implement authentication using JWT.
   * Write unit tests for API endpoints.
2. **Frontend**:
   * Add forms to create, update, and delete categories and items.
   * Improve UI with a CSS framework (e.g., Bootstrap or TailwindCSS).
   * Add search and pagination to category and item lists.
3. **General**:
   * Test the full workflow (create, read, update, delete) end-to-end.
   * Plan deployment for testing (e.g., use ngrok for backend exposure and deploy frontend to Netlify or Vercel).

Let me know if you’d like to tackle any of these steps now!

**Vet Inventory System: Recent Updates Summary**

**Backend Updates**

1. **Item Management**:
   * Added functionality to handle CRUD operations for items.
     + **Create**: Endpoint to add new items.
     + **Update**: Endpoint to modify existing items.
     + **Delete**: Endpoint to remove items.
2. **Category Management**:
   * Implemented backend routes for categories:
     + **Fetch All Categories**: Endpoint to retrieve all categories.
     + **Search Categories**: Added search capability.
     + **CRUD Operations**: Create, update, and delete functionalities.
3. **Backend Refactoring**:
   * Structured routes for better organization (categories.js and items.js).
   * Added error handling and validation for all endpoints.

**Frontend Updates**

1. **Add Item Functionality**:
   * Created AddItem.js for adding items to categories.
   * Fixed navigation to redirect back to the **Manager Dashboard** view after adding an item.
2. **Manager Dashboard**:
   * Refined Manager.js to:
     + Display categories and their items dynamically.
     + Add buttons for editing and deleting categories and items.
     + Include navigation for adding new items linked to their respective categories.
   * Enhanced error handling for user interactions (e.g., validation for empty fields).
3. **Navigation Improvements**:
   * Adjusted routes to ensure managers navigate seamlessly between categories and items.
   * Redirects now point to manager-specific views where applicable (e.g., /manager?category={id}).
4. **User Interface Adjustments**:
   * Added input fields and buttons for adding and editing categories directly from the manager dashboard.
   * Designed category and item lists with clear actions (edit/delete).

**Notable Fixes**

1. **Add Item Page Issue**:
   * Resolved issue where adding an item redirected to the public user view instead of the manager dashboard.
   * Updated AddItem.js to navigate back to /manager?category={id}.
2. **Category Navigation Issue**:
   * Ensured that selecting a category on the manager dashboard dynamically loads its items.
   * Fixed URL query handling to persist the selected category context.

**Next Steps**

1. **Permission Management**:
   * Implement role-based access to restrict add/edit/delete functionalities to managers only.
2. **Design Enhancements**:
   * Improve UI for manager-specific actions (e.g., dedicated modals or pages for editing categories/items).
3. **Search Optimization**:
   * Add a search bar to filter categories and items dynamically.
4. **Testing**:
   * Conduct end-to-end testing for all manager dashboard functionalities.

This summary reflects the work completed since the last update, focusing on the functionality and user experience of the **Manager Dashboard** and its integration with backend endpoints.

**Summary of Recent Additions and Updates**

**1. Authentication Enhancements**

* **Token-Based Role Redirection**:
  + Updated the App.js file to include RedirectBasedOnRole, a mechanism that decodes the token on login and redirects users based on their roles (Manager or User).
  + Enhanced login functionality to handle role-based navigation dynamically.
* **Persistent Role Validation**:
  + Integrated role validation using jwtDecode to persist user roles across page refreshes.
  + Fixed issues with invalid tokens by removing them from local storage.

**2. Improved Security**

* **Backend Role Validation**:
  + Added role validation middleware (authorizeRole) to critical backend routes like adding categories and items.
  + Ensured secure actions are validated both on the frontend and backend.

**3. User Management**

* **Create New Accounts**:
  + Managers can now create new user accounts through the CreateAccount page.
  + Added a form to handle new account creation with Username, Password, and Role inputs.
  + Ensured the backend validates inputs and hashes passwords securely using bcrypt.
* **Protected User Creation**:
  + Restricted account creation to Manager role via backend middleware.

**4. UI/UX Enhancements**

* **Delete Confirmation**:
  + Implemented a confirmation modal for category and item deletions to prevent accidental deletions.
* **Error Handling**:
  + Added descriptive error messages for failed actions (e.g., unauthorized actions, validation errors).
  + Used frontend error states to display issues to users clearly.

**5. Debugging Fixes**

* Resolved backend issues caused by leftover debugging statements.
* Replaced improper usage of Debugging with proper logging mechanisms.

**6. Refactoring**

* **Manager Dashboard**:
  + Cleaned up redundant code for managing categories and items.
  + Centralized authorization headers for API calls in apiService.js.
* **Reusable Components**:
  + Created a ProtectedRoute component to handle route protection and a RedirectBasedOnRole component for role-based navigation.

**Pending Improvements**

* **Persistent Role State**:
  + Optionally, integrate Context API or Redux to manage user authentication and roles globally.
* **Role-Based Dashboard**:
  + Consolidate User and Manager dashboards for a cleaner, conditional UI.
* **Enhanced Notifications**:
  + Add toast notifications or modals for feedback on success/failure of actions.

Let me know if you’d like to focus on any of these pending items next!