<goal> You’re an industry-veteran software engineer responsible for crafting high-touch features for large-scale desktop applications. You excel at creating detailed technical specifications for features, and understanding how different features connect and nest within each other.

You must review the <context> below and use it to output a thorough, no-stone-unturned feature specification document.

DO NOT WRITE CODE IN THIS OUTPUT UNLESS IT’S PSEUDOCODE FOR A TECHNICAL SITUATION.  
</goal>

<format> Structure your output as follows:

**File System**

* Folder and file structure for both front-end (UI) and back-end repositories

**Feature Specifications (do this for all features, including post MVP, below is an example of how to do so)**

**Feature 1: Item Inventory Management**

* **Feature goal**: Provide a comprehensive inventory tracking system where users can add, modify, view, and delete inventory items (name, quantity, expiration date, category).
* **API relationships**:
  + Interaction with the local database to store item information.
  + If cloud syncing is introduced later, interaction with cloud storage for backup and cross-device access.
* **Detailed feature requirements**:
  + CRUD operations for inventory items
  + Search and filter functionality based on item attributes (name, category, quantity).
  + Category tagging for items (e.g., dairy, produce).
  + Auto-generated reorder suggestions based on low inventory levels.
  + Alert for expiring items
* **Detailed implementation guide**:
  + Use **SQLite** or **Realm** for local database storage.
  + Each inventory item is an entity with fields: item\_name (text), quantity (int), expiration\_date (date), category (text).
  + CRUD operations should have validation:
    - Create: Ensure item\_name is not empty. Ensure quantity is a positive integer.
    - Read: Search by name, filter by category, and sort by expiration date.
    - Update: Allow partial updates (e.g., changing quantity or expiration date without modifying other fields).
    - Delete: Soft delete with a flag (e.g., is\_deleted boolean).
  + For reordering: Check inventory every 24 hours, flag items with quantity below reorder threshold (set by user or AI).
  + Use **MVVM (Model-View-ViewModel)** pattern for UI.
  + For macOS (SwiftUI) and Windows (C# WPF), ensure responsiveness and accessibility (keyboard navigation, large fonts).

**Feature 2: User Roles & Access Management**

* **Feature goal**: Allow user role management to control who has access to add, modify, or delete inventory items.
* **API relationships**:
  + Link user role data with inventory actions to restrict access based on role (Admin, Staff).
* **Detailed feature requirements**:
  + Role-based permissions (Admin: full access, Staff: view only or limited edit access).
  + Admin can assign roles to users.
  + Basic user login screen, authenticated via role-based authentication.
* **Detailed implementation guide**:
  + Create a **User** table with fields: user\_id (int), username (text), password\_hash (text), role (text).
  + Admin should have the ability to modify user roles.
  + Use **bcrypt** or another secure hashing function for password storage.
  + Restrict views or edits of certain inventory items based on user roles.
  + Use session or local authentication for login and role verification.
  + User role and permissions should be stored locally for faster checks.

**Feature 3: Reorder Suggestions & AI Integration**

* **Feature goal**: Provide reorder suggestions based on historical inventory data and user-set thresholds.
* **API relationships**:
  + If AI is involved, use external services (OpenAI, or another API) to generate reorder recommendations based on past consumption data.
* **Detailed feature requirements**:
  + Track inventory usage history over time.
  + Allow users to set reorder thresholds (e.g., reorder if quantity falls below 5).
  + AI-driven reorder suggestions, possibly integrated with a model trained on past usage.
* **Detailed implementation guide**:
  + Store usage history in a **UsageHistory** table: usage\_id (int), item\_id (int), usage\_date (date), quantity\_used (int).
  + Reorder suggestions are calculated by checking usage over the past month and comparing it to current inventory levels.
  + If AI is used: Generate reorder suggestion based on trends identified by a trained model.
  + API integration: If using OpenAI or another service, ensure proper error handling for API limits and performance concerns.
  + Allow manual override by users for reorder suggestions.

**System Diagram**

* Provide a detailed architecture diagram for the MVP:
  + **Frontend**: C# WPF or SwiftUI for inventory views, role management, and interaction with local database.
  + **Backend**: Local database (SQLite or Realm) storing items, usage history, and user roles.
  + **AI Service** (optional in the future): Integrated via API for reorder suggestions.
  + Ensure that all components interact smoothly with local storage and user roles.

**Questions & Clarifications**

* What happens if a user deletes an item that has already been flagged for reorder?: Answer – suggestion an intelligent/intuitive solution.
* Are there any specific integrations with external hardware (e.g., barcode scanners) that should be considered? Answer- this can be a post MVP feature.

**List of Architecture Consideration Questions**

* How will we handle data migration if we later decide to add cloud syncing? Answer – you may suggest a solution.
* If multiple roles interact with the same inventory list at once, should there be locking mechanisms to prevent data conflicts? Answer – yes.
* What will happen when a user moves between different devices (e.g., macOS and Windows)? Will local data sync between them, or is it purely local? Answer – primarily local but you may recommend a more efficient solution.

</format> <warnings-and-guidelines> <warning-1>Do not leave out steps. This absolutely must be a step-by-step output that, when passed to a human, accurately describes in exact detail what needs built</warning-1> <warning-2>This is not a code writing step. Only pseudocode if needed to guide the user. This is a stage of detailed feature specifications</warning-2> <guideline-1> For each FEATURE, make sure you also consider each of these items:

1. **System Architecture Overview**
   * High-level architecture diagram/description
   * Technology stack selection with justification
   * Deployment architecture
   * Integration points with external systems
2. **Database Schema Design**
   * Entity-relationship diagrams
   * Table definitions with all fields, types, and constraints
   * Indexing strategy
   * Foreign key relationships
   * Database migration/versioning approach
3. **Comprehensive API Design**
   * RESTful/GraphQL endpoints with full specifications
   * Request/response formats with examples
   * Authentication and authorization mechanisms
   * Error handling strategies and status codes
   * Rate limiting and caching strategies
4. **Frontend Architecture**
   * Component hierarchy with parent-child relationships
   * Reusable component library specifications
   * State management strategy
   * Routing and navigation flow
   * Responsive design specifications
5. **Detailed CRUD Operations**
   * For each entity:
     + **Create operation**: validation rules, required fields
     + **Read operation**: filtering, pagination, sorting
     + **Update operation**: partial updates vs. full replacement
     + **Delete operation**: soft delete vs. hard delete, cascading
6. **User Experience Flow**
   * User journey maps
   * Wireframes for key screens
   * State transitions and loading states
   * Error handling from user perspective
7. **Security Considerations**
   * Authentication flow details
   * Authorization matrix (roles and permissions)
   * Data validation and sanitization rules
   * Protection against common vulnerabilities (CSRF, XSS, etc.)
8. **Testing Strategy**
   * Unit test requirements
   * Integration test scenarios
   * End-to-end test flows
   * Performance testing thresholds
9. **Data Management**
   * Data lifecycle policies
   * Caching strategies
   * Pagination and infinite scrolling implementation
   * Real-time data requirements
10. **Error Handling & Logging**

* Structured logging format
* Error classification and prioritization
* Monitoring and alerting thresholds
* Recovery mechanisms

</guideline-1> </warnings-and-guidelines> <context> I’d like to build a \*\*local/desktop inventory management app\*\* for a \*\*school kitchen\*\*.

You should take inspiration from apps like **Trello** (for task management) and **Inventory Tracker**, but this app will be significantly different in the following ways:

* It’s focused on **simple, easy-to-use inventory management** for non-tech-savvy users (i.e., kitchen staff).
* The app will primarily process data **locally**, not requiring a constant internet connection.
* **AI will be used during development** (via tools like Cursor or Trae AI) to assist with tasks like generating boilerplate code, database structure, and UI components.

Here is the full extent of how the app should function as an MVP:

* **Item Inventory List**: Display all inventory items with details (name, quantity, expiration date, category).
* **Item Search/Filter**: Allow users to search or filter by item name or category.
* **Item Tracking**: Track quantities and notify when inventory is low or needs reordering.
* **Reorder Suggestions**: Automatically generate reorder suggestions based on historical usage (AI-driven).
* **Item Addition/Modification**: Allow users to add new items, edit existing ones, and update quantities.
* **User Management**: Basic role management (e.g., Admin, Staff), ensuring different users have appropriate access levels.

<other-critical-notes> \*\*WHAT\*\*: I’m building a \*\*local/desktop inventory management app\*\* for \*\*school kitchens\*\*. \*\*WHO\*\*: This app is for \*\*non-technical kitchen staff\*\* who need to keep track of inventory. \*\*WHY\*\*: This app solves the problem of \*\*manual inventory tracking\*\* and \*\*wasted resources\*\* due to stockouts or overstocking. \*\*HOW\*\*: This app is different because it is \*\*AI-assisted during development\*\* and \*\*runs entirely locally\*\*, with an emphasis on simplicity for non-tech users. </other-critical-notes> <current-tech-choices> - \*\*Frontend/UI\*\*: The frontend will be built using \*\*C# with WPF\*\* for a Windows-based desktop app and \*\*SwiftUI\*\* for macOS. - \*\*Database\*\*: Local storage will be managed using \*\*SQLite\*\* or \*\*Realm\*\* for simplicity and efficiency in local data handling. - \*\*AI Tools\*\*: The development process will use \*\*Cursor/Trae AI\*\* to assist with code generation, database design, and feature automation. - \*\*No cloud connectivity\*\* is required for the MVP, but there may be a future option for \*\*syncing data\*\* across multiple machines. - \*\*Analytics\*\*: If needed, analytics for app usage can be collected using \*\*Posthog\*\* or another suitable tool. </current-tech-choices> </context>