**Launch Features (MVP)**

**Item Inventory List & Search/Filter**

A core feature that provides a comprehensive, real-time view of all food and supply items in the kitchen. Users can quickly see item names, current quantities, expiration dates, and categories at a glance. The interface will include intuitive search and filtering options, allowing staff to instantly find items by name or category, streamlining the process of checking stock levels.

* **List of Core Requirements or Functions**
  + Display items in a clear, sortable table or grid format.
  + Columns for Item Name, Quantity, Expiration Date, and Category.
  + Real-time search bar that filters the list as the user types.
  + Dropdown or selectable filters for one or more categories.
  + Visual cues for item status (e.g., color-coding for items nearing expiration).
* **Tech Involved**
  + **C# with WPF (Windows) / SwiftUI (macOS):** To build the native UI components, including the data grid/table and search/filter controls.
  + **SQLite:** To query the local database for all inventory items and apply search/filter criteria efficiently.
* **Main Requirements**
  + The UI must be performant on older hardware (Windows 10, 4+ years old).
  + The data display must be clean and easy to read for non-technical users.

**Item Tracking & Low Stock Notifications**

This feature automates the monitoring of inventory levels. The system will track the quantity of each item and automatically flag items that fall below a predefined threshold. It will generate clear, non-intrusive notifications to alert staff when an item is running low or nearing its expiration date, helping to prevent stockouts and reduce food waste.

* **List of Core Requirements or Functions**
  + Allow an Admin to set a "low stock" threshold for each item.
  + Track item quantity changes (additions/deductions).
  + Monitor expiration dates daily.
  + Generate a persistent but unobtrusive notification within the app for low stock or expiring items.
  + Create a dedicated "Needs Attention" view or section summarizing all flagged items.
* **Tech Involved**
  + **C# with WPF / SwiftUI:** For creating the notification UI elements and the "Needs Attention" dashboard view.
  + **SQLite:** To store and query quantity levels, thresholds, and expiration dates.
  + **Background Service/Thread:** A simple background task that runs on app launch to check for low stock and expiring items.
* **Main Requirements**
  + Notifications must be clear and actionable.
  + The background check should be lightweight and not impact application performance.

**Item & Category Management**

Enables kitchen staff to maintain an accurate inventory by easily adding new items, editing existing item details, or performing manual daily updates of quantities. Admins will also have the ability to manage the list of available categories (CRUD operations), ensuring the inventory structure meets the kitchen's specific needs.

* **List of Core Requirements or Functions**
  + A simple form for adding a new item with fields for Name, Initial Quantity, Expiration Date, and Category.
  + Ability to select an existing item and edit its details.
  + A straightforward interface for daily stock updates (e.g., "End of Day Count").
  + An Admin-only interface for Creating, Reading, Updating, and Deleting inventory categories.
  + Pre-populate with a few static, common categories (e.g., Dairy, Produce, Dry Goods).
* **Tech Involved**
  + **C# with WPF / SwiftUI:** To create the data entry forms and modal dialogs.
  + **SQLite:** To handle the Create, Read, Update, Delete (CRUD) operations for both inventory items and categories.
* **Main Requirements**
  + Forms must be simple and require minimal typing.
  + Input validation to prevent incorrect data entry.

**User & Role Management**

Provides a system for managing user access with three distinct roles. This ensures that staff members have the appropriate level of access for their responsibilities, protecting sensitive operations and data.

* **List of Core Requirements or Functions**
  + A simple login screen.
  + **Admin Role:** Full control. Can manage inventory, users, categories, and view reports.
  + **Manager Role:** Can view all inventory data and access reports, but cannot edit users, categories, or system settings. Can update item quantities.
  + **Staff Role:** Basic access. Can view inventory and update item quantities only. Cannot see reports or access settings.
  + An Admin-only area for creating and deleting user accounts.
* **Tech Involved**
  + **C# with WPF / SwiftUI:** To build the login screen and the user management interface.
  + **SQLite:** A dedicated table to store user credentials (with hashed passwords) and their assigned roles.
  + **Password Hashing Library:** A standard library (e.g., System.Security.Cryptography in .NET) to securely store passwords.
* **Main Requirements**
  + Password storage must be secure (hashed and salted).
  + The application must clearly enforce the three-tiered role-based permissions.

**Future Features (Post-MVP)**

**Rule-Based Reorder Suggestions**

This feature will analyze historical consumption to provide simple, rule-based reorder suggestions. Instead of just flagging low stock, it will recommend *how much* to reorder based on past usage, helping to optimize stock levels.

* **List of Core Requirements or Functions**
  + Log all quantity changes with a timestamp and user ID to build a history of usage.
  + Develop a simple algorithm (e.g., calculate average daily usage over the last 30 days).
  + Generate a "Suggested Reorder" list with recommended quantities based on the rule.
  + Allow Admins to review and approve suggestions.
* **Tech Involved**
  + **SQLite:** To store historical transaction data for every item.
  + **Core Business Logic (.NET Standard Library):** The calculation logic will reside here.
* **Main Requirements**
  + The analysis must run locally and quickly.
  + The suggestion logic should be transparent and easy for an Admin to understand.

**Reporting Dashboard**

A dedicated section for Managers and Admins to view key inventory metrics. This provides insights into consumption trends, waste, and purchasing patterns, enabling better decision-making.

* **List of Core Requirements or Functions**
  + Summary of current inventory value.
  + Report on items nearing expiration to prioritize usage.
  + Historical usage charts for specific items or categories.
  + Ability to export reports to CSV or PDF.
* **Tech Involved**
  + **C# with WPF / SwiftUI:** To build the dashboard UI.
  + **Data Visualization Library (e.g., LiveCharts for WPF):** To create simple graphs and charts.
  + **SQLite:** To query the necessary data for the reports.
* **Main Requirements**
  + Reports should be accessible only to Manager and Admin roles.
  + Data visualizations must be simple and easy to interpret.

**System Diagram**

Here is a clean system diagram for the MVP, illustrating the recommended architecture with a shared core library.



**Architecture Recommendations**

**1. Core Logic Abstraction (Future-Proofing for Web/Mobile)**

To maximize code reuse and prepare for future platforms (like a web dashboard), the application's architecture should be split into two main parts:

* **UI Layer (WPF/SwiftUI):** This layer is platform-specific and only handles what the user sees and interacts with. It contains no business logic.
* **Core Logic Layer (.NET Standard Library):** This is a shared, platform-agnostic library that contains all the critical logic:
  + **Business Rules:** How to track inventory, check for low stock, manage users.
  + **Database Operations:** All the code that reads from and writes to the SQLite database.
  + **Data Models:** The definitions for Item, User, Category, etc.

**Benefit:** When you decide to build a web dashboard, you can reuse this entire Core Logic library. The new web project will only need to build a new UI layer that calls into the same, already-tested core functions. This saves significant time and reduces bugs.

**2. Database Choice and Schema (SQLite)**

For a local-first, free, and stable database, **SQLite** is the ideal choice. It's a serverless, file-based database that is extremely reliable and well-supported on every platform, including Windows, macOS, and web servers.

**Future-Proofing the Schema:** To ensure an easy migration to a web-based system, we should design the schema with web concepts in mind from the start:

* **Use Universally Unique Identifiers (UUIDs):** Instead of using simple auto-incrementing integers (1, 2, 3) as primary keys for your tables, use UUIDs (e.g., f81d4fae-7dec-11d0-a765-00a0c91e6bf6).
  + **Why?** When you eventually sync data from multiple kitchens to a central database, integer IDs will clash (e.g., two different kitchens will both have an "Item #5"). UUIDs are globally unique, preventing these conflicts entirely and making synchronization trivial.
* **Add Timestamp Fields:** Every table (Items, Categories, etc.) should have CreatedAt and UpdatedAt timestamp columns. This is crucial for synchronization, as it allows a central server to know which version of the data is the newest.