Assignment — Inventory & Sales Orders App (C#/.NET)

# What to build (scope is only these two modules)

• Items (Inventory): list/search/filter/sort/paginate; create/update item; inventory adjustments (+/–) with reason codes; optimistic concurrency (ETag/rowversion).  
• Sales Orders: create draft header; add/update/remove lines; soft reservation against available stock; Confirm to decrement stock; duplicate-line confirmation.  
  
No other modules. Focus on app quality and UX. Do not integrate with any external system; use simulated data (see “Data Simulation”).

# Stack & Target (chosen for App Stores)

• Framework: .NET 8 + .NET MAUI (Single Project)  
• Targets: iOS, Android, Windows, Mac  
• UI: MAUI XAML Pages (MVVM pattern)  
• Local database: SQLite via EF Core  
• DI: Microsoft.Extensions.DependencyInjection  
• Logging: Microsoft.Extensions.Logging  
• Resilience: Polly policies  
  
If fallback needed: WinUI 3 (Windows App SDK).

# Data Simulation (how to work without ERP)

Seed a local SQLite DB from CSV files.  
CSV files: items.csv, customers.csv, salesorders.csv, salesorderlines.csv, inventoryadjustments.csv.  
On first run: import seed CSVs. Settings page allows re-import/export.  
  
Tables: Items, Customers, SalesOrders, SalesOrderLines, InventoryAdjustments, Reservations.

# App Architecture

Project structure:  
• App (MAUI UI)  
• Application (Use cases)  
• Domain (Entities, services)  
• Infrastructure (Persistence, Integration stubs)  
• Shared (abstractions)  
• Tests (unit + integration)  
  
Key Interfaces: IItemRepository, ISalesOrderRepository, IExternalErpClient (stub).

# Functional Requirements — Items

• List columns: Number, Name, Inventory, Unit Price, Active, Last Modified  
• Filter: name, number, inventory range, active toggle  
• Sort: name/price asc/desc  
• Paging: selectable (5–50)  
• Create/Update with validation  
• Inventory adjustments with reason codes  
• Concurrency safety (RowVersion/ETag).

# Functional Requirements — Sales Orders

• Draft header: customer, date, currency  
• Lines: add/update/remove items with qty/price/discount  
• Reservations: reduce available stock until confirm  
• Confirm: validates stock, decrements inventory  
• Duplicate lines: allowed only with confirmation.

# Non-Functional Requirements

• Validation: DataAnnotations/FluentValidation  
• Logging with correlation id  
• Resilience prepared for HTTP (Polly)  
• Config via appsettings & secrets  
• Performance: avoid N+1, use indices  
• Localization: en-US resource file

# Testing

Unit Tests: reservations, pagination/sorting, validation, concurrency.  
Integration Tests: SQLite happy path (import CSV → create draft → confirm).

# Deliverables

1. GitHub repo with code, migrations, tests, sample CSVs, README.  
2. One-page architecture note.

# Packaging Notes

• MAUI single-project assets (icons/splash)  
• Package outputs: Android .aab, Windows MSIX, iOS/macOS description steps  
• Keep logs clean, request minimal permissions.

# Step-by-Step Scaffold for Candidates

1. dotnet new maui -n InventoryOrdersApp  
2. Add EF Core + SQLite  
3. Create DbContext + migrations  
4. CSV import service  
5. Repositories (EF)  
6. Application services (queries, commands)  
7. MVVM pages for Items & Orders  
8. Tests (xUnit)  
9. README with emulator setup & screenshots.

# Additional Information for Candidates

## Purpose of the Assessment

This project is not a final product, but a learning and evaluation exercise. We want to see how you structure a C#/.NET application that is app-store ready, simulate real data workflows, handle validation, concurrency, and error cases, and how you document your problem-solving approach.

## Skills You Will Demonstrate

• .NET MAUI (UI): App-store–ready, cross-platform UI (iOS/Android/Windows/Mac).  
• C# Basics: Classes, async/await, interfaces, dependency injection.  
• Database: Using SQLite with EF Core.  
• Data Simulation: Import/export CSVs and maintain a working local database.  
• Architecture: Organize code into layers (UI, Application, Domain, Infrastructure).  
• Testing: Write at least a few unit tests (e.g., validation, pagination).

## How to Work with Simulated Data

• Use the provided CSV files (items.csv, customers.csv, etc.) as your mock ERP data.  
• On first run, import the CSVs into SQLite (using EF Core migrations).  
• Provide a 'Re-import data' button in the app’s settings page.  
• You may add extra seed rows for testing (e.g., 20+ items, a few customers, and orders).  
  
Think of this as your local ERP sandbox. Later, we’ll plug in a real API.

## Recommended Workflow

1. Set up project: dotnet new maui -n InventoryOrdersApp.  
2. Data Layer: build DbContext and EF Core migrations for Items, Orders, etc.  
3. Import Service: parse CSV into SQLite.  
4. Repositories: implement IItemRepository and ISalesOrderRepository.  
5. Application Layer: add use cases (QueryItems, CreateOrder, ConfirmOrder).  
6. UI: implement ItemsPage and OrdersPage with MVVM.  
7. Validation: ensure rules (e.g., qty > 0).  
8. Testing: write a few unit tests.  
9. README: explain how to run, with screenshots.

## Tips for Success

• Break tasks into small steps.  
• Commit often to GitHub with clear messages.  
• If you don’t know something, **ask me directly** for clarification.  
• Keep code clean and simple.  
• Document assumptions in the README.

## Break tasks into small steps (example roadmap):

1. Create the new MAUI project (dotnet new maui).
2. Add EF Core + SQLite packages.
3. Define entities (Items, Customers, SalesOrders, etc.).
4. Build DbContext and run first migration.
5. Write CSV import service to populate SQLite.
6. Implement repositories (IItemRepository, ISalesOrderRepository).
7. Add Application layer use cases (QueryItems, CreateOrder, ConfirmOrder).
8. Build ItemsPage UI (list + filters + pagination).
9. Build ItemDetailsPage (edit + inventory adjustment).
10. Build SalesOrdersPage (create draft, add lines).
11. Build SalesOrderDetailsPage (confirm order flow).
12. Add unit tests for validation & pagination.
13. Add one integration test with SQLite.
14. Write README with screenshots and run instructions.