Final Project
MAUI Notes Application
YT0797 C#.NET Development, Thomas More

Gilles Van Pellicom

December 15, 2025

Contents

Notes on the Process **2**

1 Project Overview **3**
1.1 Description . . . 3
1.2 High-Level Architecture . . . 3

2 Requirements and Fulfillment **4**
2.1 Functional Requirements . . . 4
2.2 Architectural Rationale . . . 4

3 Database Design **5**
3.1 Schema . . . 5
3.2 Example Data . . . 5

4 Docker Setup **6**
4.1 Rationale . . . 6
4.2 Docker Compose . . . 6
4.3 API Dockerfile . . . 6

5 ASP.NET REST API **7**
5.1 Overview . . . 7
5.2 Endpoint Overview . . . 7
5.3 Endpoint Usage . . . 7
5.3.1 Retrieval . . . 7
5.3.2 Creation and Modification . . . 8
5.3.3 Deletion and Search . . . 8

6 Application Wiring and MVVM Architecture **9**
6.1 High-Level Architecture . . . 9
6.2 MVVM Responsibilities . . . 9
6.2.1 Views . . . 9
6.2.2 ViewModels . . . 10
6.2.3 Service Layer . . . 10
6.3 Dependency Injection and Composition . . . 10
6.4 Backend Wiring Overview . . . 10
6.5 End-to-End Interaction Flow . . . 11
6.6 Architectural Rationale . . . 11

7 Platform-Specific Configuration **12**
7.1 Android . . . 12
7.2 macOS (Mac Catalyst) . . . 12

8 Outcome **13**

Notes on the Process

Reproducability

- **Development and Testing Environment:**
  - **Platform:** macOS Tahoe 26.2
  - **Hardware:** M3 Pro, 36 GB LPDDR5
  - **IDE:** Rider 2025.3.0.4
  - **Framework:** .NET 9.0.11@arm64, .NET MAUI 9.0.120/9.0.100
  - **Backend Runtime:** ASP.NET 9.0.11
  - **Container Platform:** Docker v29.1.2

All backend components are containerized and can be reproduced using Docker without requiring any local database or API installation. This ensures consistent behavior across development machines and eliminates configuration drift.

Chapter 1

Project Overview

1.1 Description

This project is a cross-platform notes application built using .NET MAUI. The application targets Android and desktop macOS through Mac Catalyst.

Users can create, edit, delete, and search textual notes. All note data is persisted in a MySQL database, accessed exclusively through a REST API written in ASP.NET. The MAUI client functions purely as a consumer of this API and does not contain persistence logic.

1.2 High-Level Architecture

This project is intentionally split into three layers:

- **MAUI Client** — Presentation and interaction.
- **ASP.NET REST API** — Application logic and validation.
- **MySQL Database** — Persistent storage.

Each layer has a clearly defined responsibility. This separation avoids leaking infrastructure concerns into the UI and keeps platform-specific code isolated from business logic.

Chapter 2

Requirements and Fulfillment

2.1 Functional Requirements

- **MVVM architecture:** Implemented using MAUI data binding and view models.
- **CRUD functionality:** Notes can be created, edited, and deleted.
- **Search:** Notes can be queried by matching title and content.
- **Remote persistence:** Notes are stored in a MySQL database.
- **REST-based communication:** All data access occurs via HTTP.
- **Cross-platform support:** Android and macOS (Mac Catalyst).

2.2 Architectural Rationale

The decision to introduce an explicit API layer, rather than connecting the MAUI application directly to the database enforces a strict separation of concerns and ensures that persistence logic, validation, and query behavior remain centralized.

Further, this architecture improves testability. The API can be exercised independently using HTTP tooling, while the MAUI client can be developed and debugged without a direct dependency on database connectivity details. Using MVVM within the client further separates presentation from state management, reducing UI complexity and improving maintainability.

Chapter 3

Database Design

3.1 Schema

The database consists of a single table representing notes.

Table 3.1: Notes Table Schema

Column	**Type**	**Attributes**	**Description**
id	INT	PRIMARY KEY, AUTO_INCREMENT	Unique identifier
title	LONGTEXT	NOT NULL	Note title
content	LONGTEXT	NOT NULL	Note content
date	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP	Creation timestamp

3.2 Example Data

Table 3.2: Example Notes

ID	*TITLE*	*CONTENT*	*DATE*
1	MAUI Architecture	Discussion of MVVM bindings, observable properties, command usage, and view model lifecycle behavior in the MAUI client.	2025-01-10 09:14
2	API Design Decisions	Rationale behind REST endpoint structure, stateless request handling, and consistency of JSON payloads.	2025-01-17 16:02
3	Docker Infrastructure	Notes covering Docker Compose orchestration, health checks, volume persistence, and service startup ordering.	2025-01-24 11:48

Chapter 4

Docker Setup

4.1 Rationale

Docker is used to guarantee consistent runtime behavior across development and testing environments. It removes the need for local database setup and ensures that API and database versions remain aligned.

4.2 Docker Compose

Docker Compose orchestrates the API and database containers and defines networking, startup order, and persistence behavior.

Key characteristics of the setup include:

- **Health checks:** The API only starts once MySQL is fully available.
- **Named volumes:** Database data persists independently of container lifecycle.
- **Service discovery:** Containers communicate using service names rather than hard-coded addresses.

4.3 API Dockerfile

The API is built using a multi-stage Dockerfile.

The build stage compiles and publishes the application using the .NET SDK image. The final stage contains only the published output and the ASP.NET runtime, resulting in a smaller and more secure image. Configuration is supplied through environment variables rather than hard-coded values.

Chapter 5

ASP.NET REST API

5.1 Overview

The API is implemented using ASP.NET and exposes a RESTful interface over HTTP. It is stateless and communicates exclusively via JSON.

All logic related to validation, ordering, and search behavior resides in the API. The client remains agnostic of storage details and interacts solely through defined endpoints.

5.2 Endpoint Overview

- `GET /notes` — Retrieve all notes (newest first)
- `GET /notes/{id}` — Retrieve a single note
- `POST /notes` — Create a new note
- `PUT /notes/{id}` — Update an existing note
- `DELETE /notes/{id}` — Delete a note
- `GET /notes/search?query=...` — Search title and content

5.3 Endpoint Usage

5.3.1 Retrieval

Get all notes

```
curl -X GET "http://localhost:8080/notes"
```

Get note by ID

```
curl -X GET "http://localhost:8080/notes/1"
```

5.3.2 Creation and Modification

Create a new note

```
curl -X POST "http://localhost:8080/notes" \
-H "Content-Type: application/json" \
-d '{
  "title": "My Test Note",
  "content": "This is a note created via curl."
}'
```

Update an existing note

```
curl -X PUT "http://localhost:8080/notes/1" \
-H "Content-Type: application/json" \
-d '{
  "title": "Updated title",
  "content": "Updated content",
  "date": "2025-02-01T12:00:00Z"
}'
```

5.3.3 Deletion and Search

Delete a note

```
curl -X DELETE "http://localhost:8080/notes/1"
```

Search notes

```
curl -X GET "http://localhost:8080/notes/search?query=project"
```

Chapter 6

Application Wiring and MVVM Architecture

This chapter describes how this project is wired end-to-end, focusing on the separation of concerns between UI, application logic, transport, and persistence. The MAUI client follows the MVVM pattern strictly, while the backend exposes a narrow REST surface consumed through a dedicated service layer.

6.1 High-Level Architecture

This project is composed of two clearly separated applications:

- A **.NET MAUI client**, responsible for presentation, user interaction, and navigation.
- An **ASP.NET Core REST API**, responsible for persistence, querying, and data integrity.

The MAUI application never accesses the database directly. All data access is mediated through HTTP calls to the API. This separation keeps the client platform-agnostic and allows the backend to evolve independently.

6.2 MVVM Responsibilities

The MAUI application adheres to MVVM without shortcuts or hybrid patterns.

6.2.1 Views

Views are implemented as XAML pages and contain no application logic. Their responsibilities are limited to layout and binding.

- UI elements bind to ViewModel properties using MAUI data binding.
- User actions (button clicks, item selection) are routed to `ICommand` instances exposed by the ViewModel.
- No services or HTTP clients are referenced from Views.

6.2.2 ViewModels

ViewModels act as the sole owner of UI state and interaction logic.

- Expose observable state using bindable properties and collections.
- Translate user intent into service calls.
- Perform navigation through MAUI Shell routing.

ViewModels are unaware of transport details such as URLs, JSON, or HTTP verbs. Their dependency is expressed only in terms of interfaces.

6.2.3 Service Layer

The service layer provides a thin abstraction over the REST API.

- `INotesApiService` defines the contract for note operations.
- `NotesApiService` implements this contract using `HttpClient`.
- Serialization and endpoint mapping are contained entirely within this layer.

This indirection prevents HTTP concerns from leaking into ViewModels and keeps the application testable.

6.3 Dependency Injection and Composition

All wiring is performed in `MauiProgram.cs`, which acts as the composition root.

- ViewModels and Views are registered with appropriate lifetimes.
- A platform-aware `HttpClient` is configured once and reused.
- Services are injected via constructors, not resolved manually.

This ensures a single, explicit location where object lifetimes and dependencies are defined.

6.4 Backend Wiring Overview

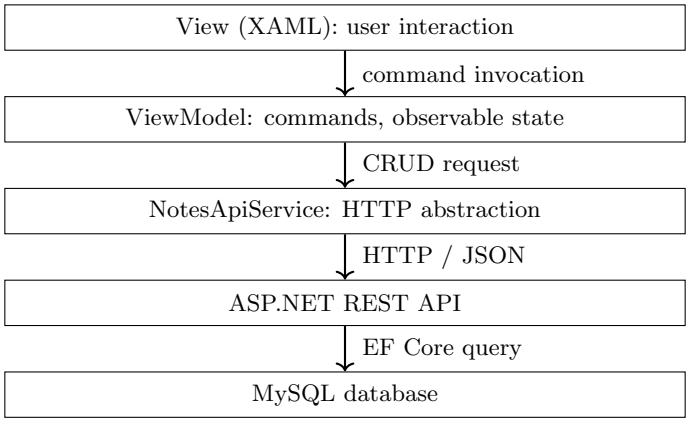
On the backend side, the ASP.NET API follows a similarly layered approach.

- Controllers expose REST endpoints only.
- Entity Framework Core handles database access via `NotesDbContext`.
- The controller depends on the DbContext through dependency injection.

No business logic is embedded in the controller beyond basic validation and request handling. Persistence logic remains confined to the data layer.

6.5 End-to-End Interaction Flow

The following diagram illustrates the complete interaction path, from user action to persistence and back. Each layer communicates only with its immediate neighbor.



Each layer communicates only with its immediate neighbor. The MAUI client has no direct knowledge of persistence, and the database is never accessed from the UI or ViewModel layers.

Figure 6.1: End-to-end user interaction and data flow across the MAUI client and backend API.

6.6 Architectural Rationale

This wiring enforces clear boundaries:

- UI changes do not affect persistence or transport logic.
- API changes are localized to the service layer.
- Platform-specific concerns remain outside application logic.

The result is a codebase that is predictable to reason about, straightforward to extend, and resistant to architectural drift as the project grows.

Chapter 7

Platform-Specific Configuration

7.1 Android

The Android manifest defines permissions and application-level configuration.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
          package="com.finalproject.maui_app">

    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
    <uses-permission android:name="android.permission.INTERNET"/>

    <application android:networkSecurityConfig="@xml/network_security_config"
                 android:enableOnBackInvokedCallback="true"></application>
</manifest>
```

One detail which might feel unfamiliar to native Android developers is that the `application` tag is intentionally empty. In .NET MAUI, application wiring and lifecycle integration are generated at build time unless explicitly overridden. The manifest therefore acts as a declarative configuration layer rather than a full application definition.

7.2 macOS (Mac Catalyst)

macOS applications require explicit entry points and lifecycle integration.

- **Info.plist:** Declares application capabilities, including network access and background behavior.
- **AppDelegate.cs:** Integrates with the native macOS application lifecycle and handles OS-level events.
- **Program.cs:** Defines the application entry point and initializes the MAUI host for Mac Catalyst.

These files provide the necessary bridge between MAUI's cross-platform abstractions and macOS' native application model, which remains more explicit than its mobile counterparts.

Chapter 8

Outcome

This project resulted in a functional cross-platform notes application with a clear separation between presentation, application logic, and persistence.

Both Android and macOS builds were tested successfully. The backend infrastructure is fully reproducible using Docker, and the architecture allows future extensions without structural changes to existing components. A video recording of a demonstration of this project will be provided seperately.