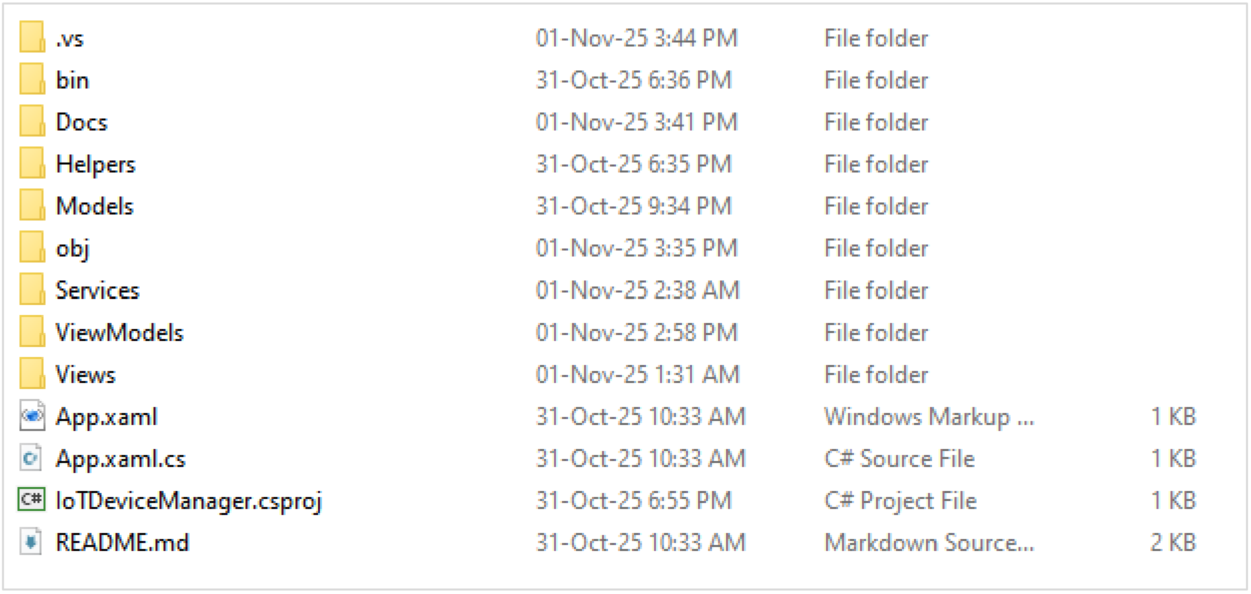
Part 1: Practical coding & System Flow

Q1. Windows-Based IoT Tool

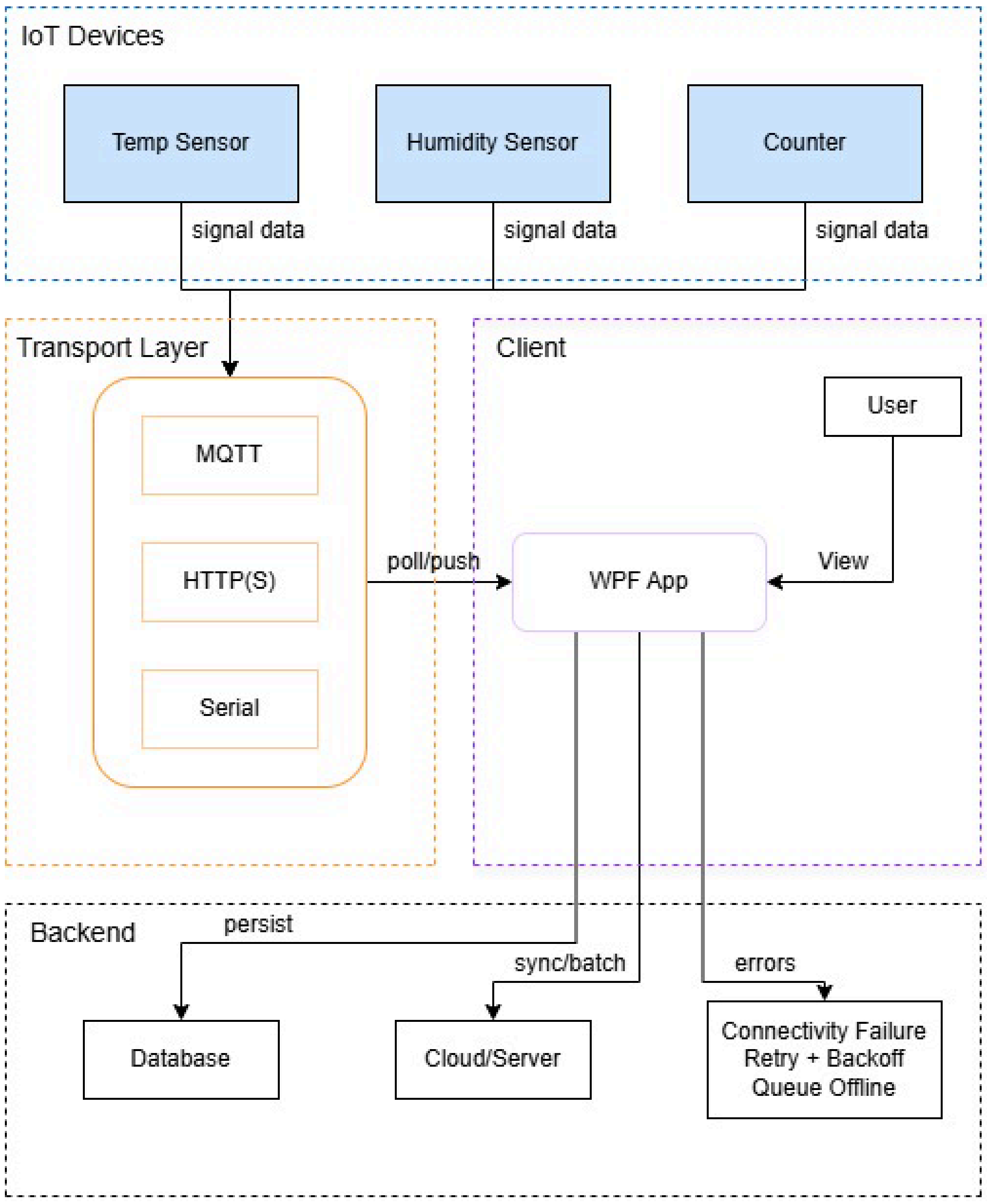
- Main Application in IoTDeviceManager folder.
- Simplified UI and data separation in MVVM format.
- Initial comes with default 5 mock data.
- Allow users to add, update and delete a device.
- Simulating device connectivity.
- Logging for add, update, delete and toggle status.



.vs	01-Nov-25 3:44 PM	File folder	
bin	31-Oct-25 6:36 PM	File folder	
Docs	01-Nov-25 3:41 PM	File folder	
Helpers	31-Oct-25 6:35 PM	File folder	
Models	31-Oct-25 9:34 PM	File folder	
obj	01-Nov-25 3:35 PM	File folder	
Services	01-Nov-25 2:38 AM	File folder	
ViewModels	01-Nov-25 2:58 PM	File folder	
Views	01-Nov-25 1:31 AM	File folder	
App.xaml	31-Oct-25 10:33 AM	Windows Markup ...	1 KB
App.xaml.cs	31-Oct-25 10:33 AM	C# Source File	1 KB
IoTDeviceManager.csproj	31-Oct-25 6:55 PM	C# Project File	1 KB
README.md	31-Oct-25 10:33 AM	Markdown Source...	2 KB

Q2. System architecture & data flow (located: IoTDeviceManager\Docs)

- Every device emits signal to transport layer (MQTT, HTTP(S), Serial)
- The transport layer then pushes the data received to WPF app
- Users view the real-time updated data.
- Any data received in WPF will log into local database and sync through Cloud.
- User modification will also be saved and logged.
- If connectivity failure happens, the system will log retry connecting and backoff queue offline. In my case, the connectivity is only simulated.



Q3. Device Communication Handling (Simulation)

Simulation separated into two layers:

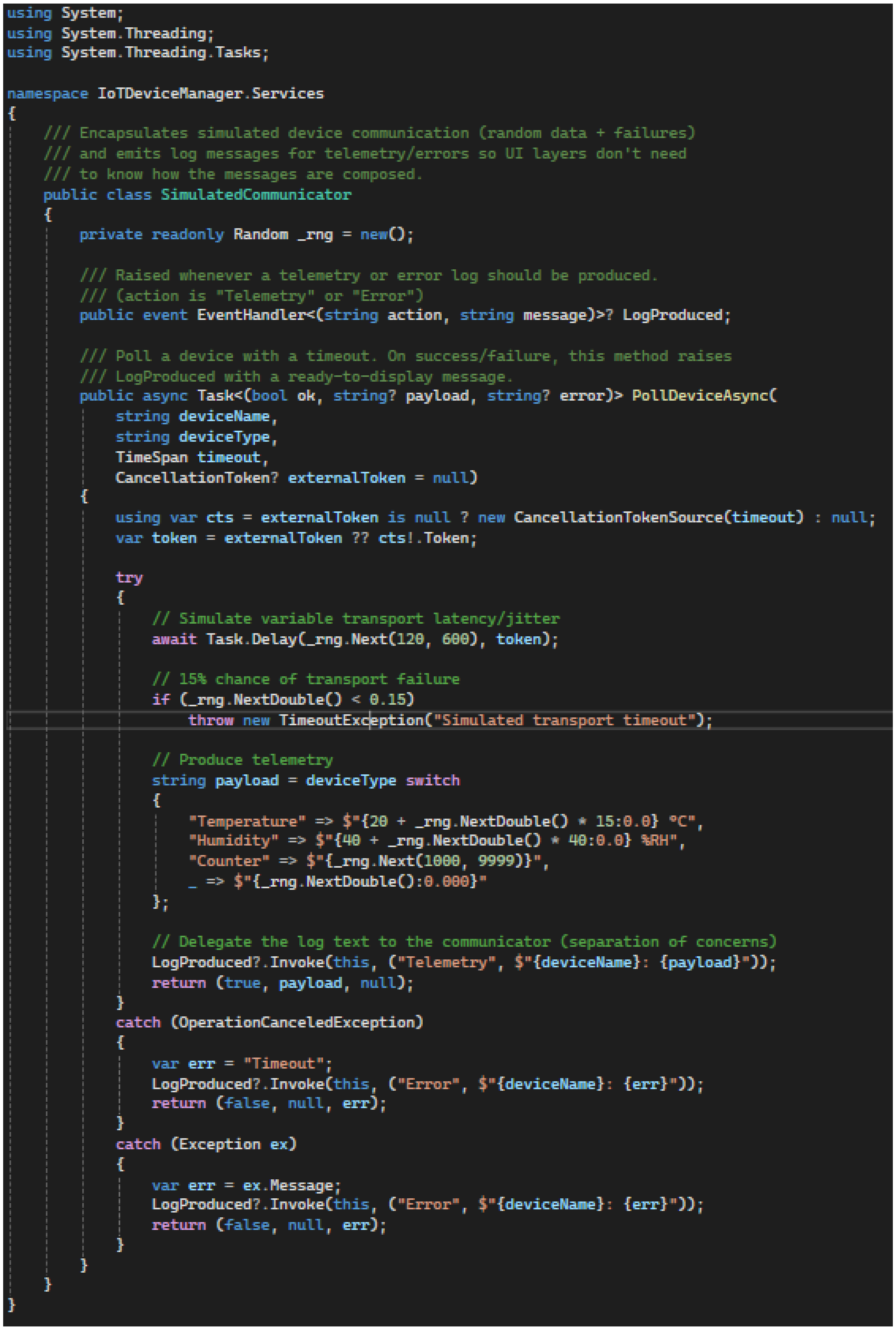
1. **SimulatedCommunicator** (located: IoTDeviceManager\Services)
    - Generates random but acceptable telemetry per device type.
    - Introduces realistic latency/jitter and failure cases (timeouts, unreachable).
    - Emits ready-to-display log lines "Telemetry" and "Error" so the UI doesn't compose messages.
2. **DeviceService** (located: IoTDeviceManager\Services)
    - Uses a WPF DispatcherTimer to poll a random device every 1.5 s.
    - Randomly toggles IsOnline (~10% chance) to simulate chances of disconnect in real environment.
    - Calls SimulatedCommunicator.PollDeviceAsync(...) on success updates device fields and fires TelemetryReceived (UI pulses the row).
    - Forwards communicator logs to the UI.

Real-time display:

- On success, DeviceService updates Device.LastData + LastSeenUtc, the UI binding instantly displays the new value.
- The Logs panel shows either telemetry lines (e.g., Boiler Temp 01: 27.4 °C) or *Error* lines (e.g., Room Humidity: Timeout).

Disconnection handling:

- If a device is offline (IsOnline = false) or a timeout occurs, SimulatedCommunicator raises an error log entry, "Timeout" or "Device unreachable".
- DeviceService surfaces that error via TelemetryReceived so the UI can react.



```csharp
using System;
using System.Threading;
using System.Threading.Tasks;

namespace IoTDeviceManager.Services
{
    /// Encapsulates simulated device communication (random data + failures)
    /// and emits log messages for telemetry/errors so UI layers don't need
    /// to know how the messages are composed.
    public class SimulatedCommunicator
    {
        private readonly Random _rng = new();

        /// Raised whenever a telemetry or error log should be produced.
        /// (action is "Telemetry" or "Error")
        public event EventHandler<(string action, string message)>? LogProduced;

        /// Poll a device with a timeout. On success/failure, this method raises
        /// LogProduced with a ready-to-display message.
        public async Task<(bool ok, string? payload, string? error)> PollDeviceAsync(
            string deviceName,
            string deviceType,
            TimeSpan timeout,
            CancellationToken? externalToken = null)
        {
            using var cts = externalToken is null ? new CancellationTokenSource(timeout) : null;
            var token = externalToken ?? cts!.Token;

            try
            {
                // Simulate variable transport latency/jitter
                await Task.Delay(_rng.Next(120, 600), token);

                // 15% chance of transport failure
                if (_rng.NextDouble() < 0.15)
                    throw new TimeoutException("Simulated transport timeout");

                // Produce telemetry
                string payload = deviceType switch
                {
                    "Temperature" => $"{20 + _rng.NextDouble() * 15:0.0} °C",
                    "Humidity" => $"{40 + _rng.NextDouble() * 40:0.0} %RH",
                    "Counter" => $"{_rng.Next(1000, 9999)}",
                    _ => $"{_rng.NextDouble():0.000}"
                };

                // Delegate the log text to the communicator (separation of concerns)
                LogProduced?.Invoke(this, ("Telemetry", $"{deviceName}: {payload}"));
                return (true, payload, null);
            }
            catch (OperationCanceledException)
            {
                var err = "Timeout";
                LogProduced?.Invoke(this, ("Error", $"{deviceName}: {err}"));
                return (false, null, err);
            }
            catch (Exception ex)
            {
                var err = ex.Message;
                LogProduced?.Invoke(this, ("Error", $"{deviceName}: {err}"));
                return (false, null, err);
            }
        }
    }
}
```

Real implementations differ on MQTT and Serial:

MQTT:

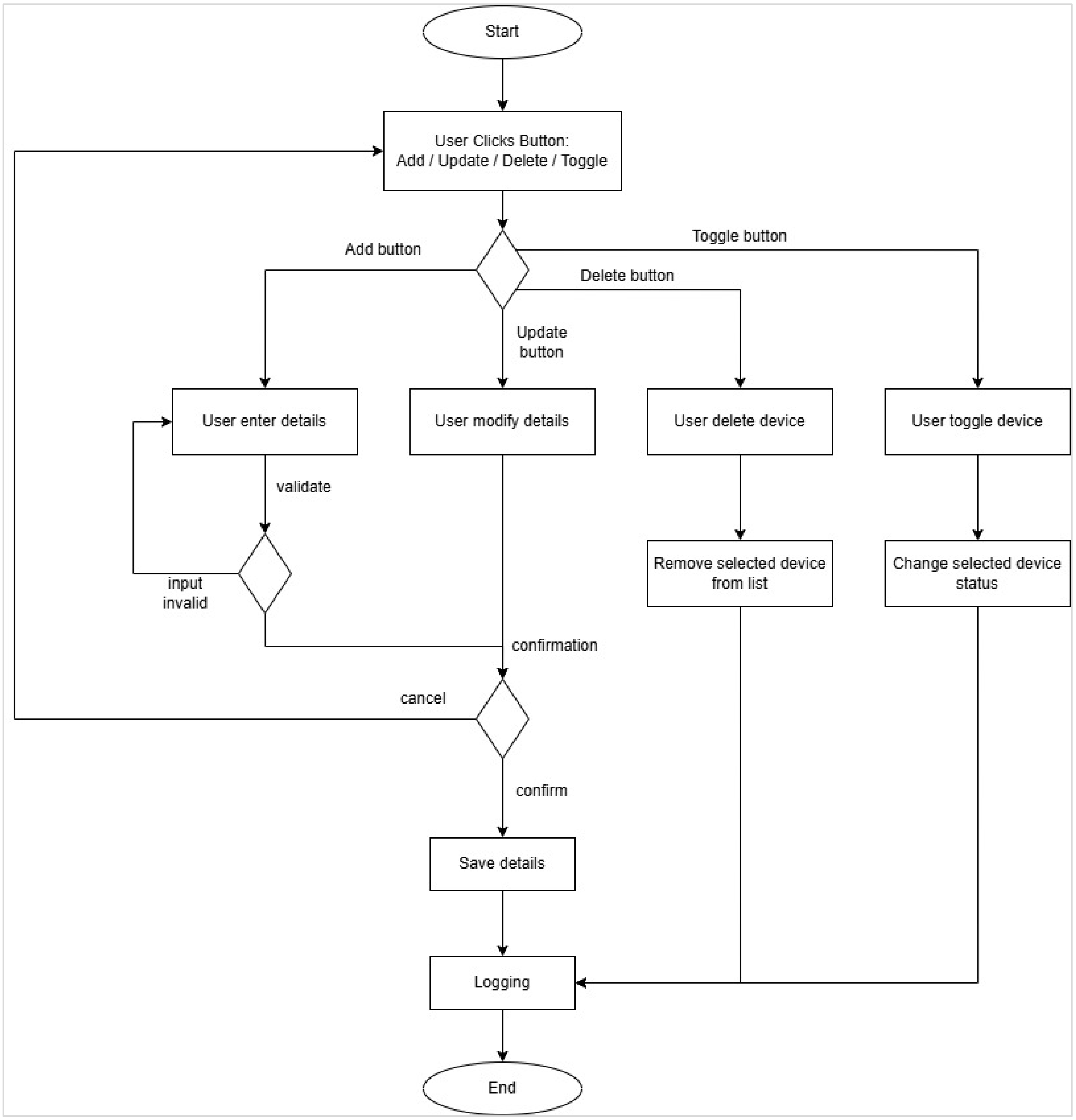
- Uses a publish–subscribe model where devices send publish data to topics, and the app subscribes to receive them in real time.
- Very efficient for low-bandwidth and many device scenarios.
- Handles automatic reconnect, Quality of Service (QoS) and message retention.
- Requires a broker such as Mosquitto, Azure IoT Hub, AWS IoT Core.

Serial:

- Used for direct connection between a PC and hardware sensor.
- Data is sent over a COM port using the SerialPort class.
- Requires defining a clear message format (start/stop bytes, checksums) and handling timeouts or disconnections.
- Suitable for prototyping or lab setups.

Q4. Flowchart documentation (located: IoTDeviceManager/Docs)

A simple flow chart diagram demonstrates the operation flow of the WPF application.



Part 2: Applied knowledge and design thinking

Q1. Troubleshooting Scenario (Debugging Focus)

1. Possible Root Cause

   From coding perspective:

   - **Timer or async race condition**: The polling timer might overlap with UI updates, causing data not to refresh properly.
   - **Dispatcher thread blocking**: Long operations (e.g., random data generation or I/O) might run on the UI thread instead of background threads.
   - **Event not triggered**: TelemetryReceived or PropertyChanged might not fire if the event is unsubscribed or handled incorrectly.

   From UI perspective:

   - **Binding not refreshed:** Missing INotifyPropertyChanged or wrong binding mode (OneTime instead of OneWay).
   - **Threading issue:** Background updates not marshaled back to the UI thread.
   - **Sorting/filtering bug:** The DataGrid may not visually update if filtered or sorted incorrectly.

2. Debugging Plan and Tools

   a. Reproduce issue

      - Run the app again and note when/which device stops updating.
      - Observe whether the Logs still showing new telemetry. This is to isolate the issue could occur in UI or backend.

   b. Inspect logs and timers

      - Add more detailed logging messages to confirm the timer is firing.
      - Use debug message "Debug.WriteLine()" or a simple logger to trace each event.

   c. Visual Debugging

      - Use Visual Studio's Live Visual Tree and Live Property Explorer to confirm binding updates.

Q2. User Manual / Guide

1. **Starting Application**
   a. Launch the IoTDeviceManager.exe which is located at: *IoTDeviceManager\bin\Debug\net8.0-windows*
   b. Or use cmd to run with dotnet build, dotnet run.
   c. Or use visual Studio Code 2022 to run IoTDeviceManager.csproj (F5)
2. Main action
   a. Add Device
   - A small dialog appears allow user to enter the Device Name and Type.
   - Click Confirm to add or Cancel to discard.
   - The new device appears instantly in the list, and an "Added" log entry is shown.

   d. Update Device
   - Select a device in the list.
   - Click "Update Device" or double-click the row.
   - Modify the name/type then click Confirm to update and Cancel to discard.
   - A log entry records the update.

   e. Delete Device
   - Select a device in list.
   - Click "Delete Device"
   - The device is removed from the list and a "Deleted" log entry appears.

   f. Toggle Status
   - Select a device in list.
   - Click "Toggle Status"
   - The device switches between **Online** and **Offline**, with a corresponding log entry.

   g. View Logs
   - User can view logging at bottom part of the main window.
   - Logs show real-time telemetry, connection errors, and user actions.
   - Each log has an action tag (Telemetry, Error, Add, Update, Delete, Toggle).

Q3. Real-World Extension Plan (Design Thinking)

What cloud services would you integrate

Azure IoT Hub for Device Connectivity

- Provides secure communication and message routing between devices and the cloud.
- Acts as a managed message broker that supports millions of connected devices.
- Handles device registration, telemetry ingestion and command delivery automatically.
- Each device in the WPF application can be registered with IoT Hub.
- Devices publish telemetry data to IoT Hub, and the desktop client **s**ubscribe**s** to receive updates in real time.

How would you scale for hundreds of devices?

MQTT Broker Architecture for Scalability

- Enables low-latency, high-volume telemetry streaming.
- Efficiently manages continuous, real-time data exchange between devices and the server.
- Reduces bandwidth usage and connection overhead for large-scale deployments.

Load-Balanced Microservices

- Introduced for data ingestion and processing to ensure modular scalability.
- Allows each service like telemetry processing, logging, device management to scale independently.
- Maintains consistent system performance even during peak load conditions.

Time-Series Database for Telemetry Storage

- Stores all logs and telemetry data efficiently in time-based order.
- Optimized for fast read/write operations and high data compression.
- Supports time-based queries and analytics for trend visualization and reporting.

What features to prioritize?

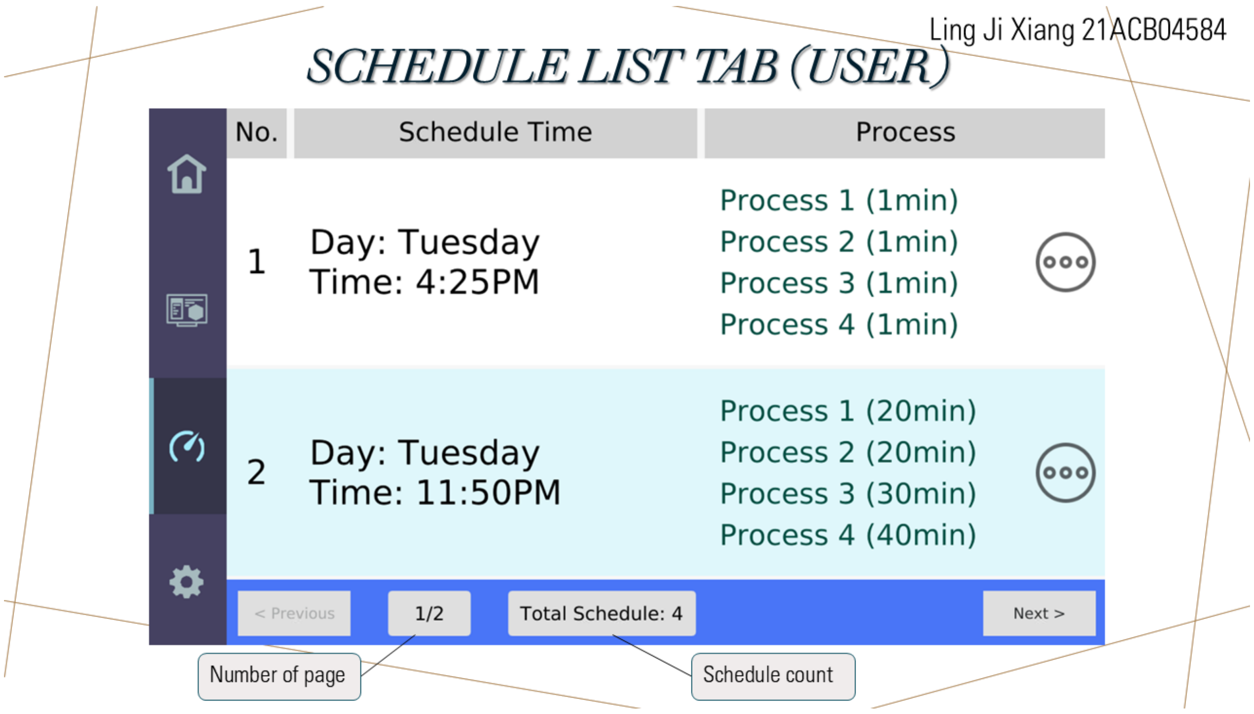
- Device Provisioning & Group Management (by type or location).
- Integration with mobile app for monitoring on-the-go.
- Dashboard customization with charts and alerts.

Q4. Relevant work

Below shows the work I done during my internship period. The project I assigned is pump relay scheduling system. This system allows users to set their desired time period to activate the relay during each process. The system includes auto tab which allows users to set time schedule and manual tab which allows user to manually turn on the relay.

Schedule List Tab

This tab is to show the user the schedule that is already set. The process detail are also shown to user, like how long will the respective process will run.

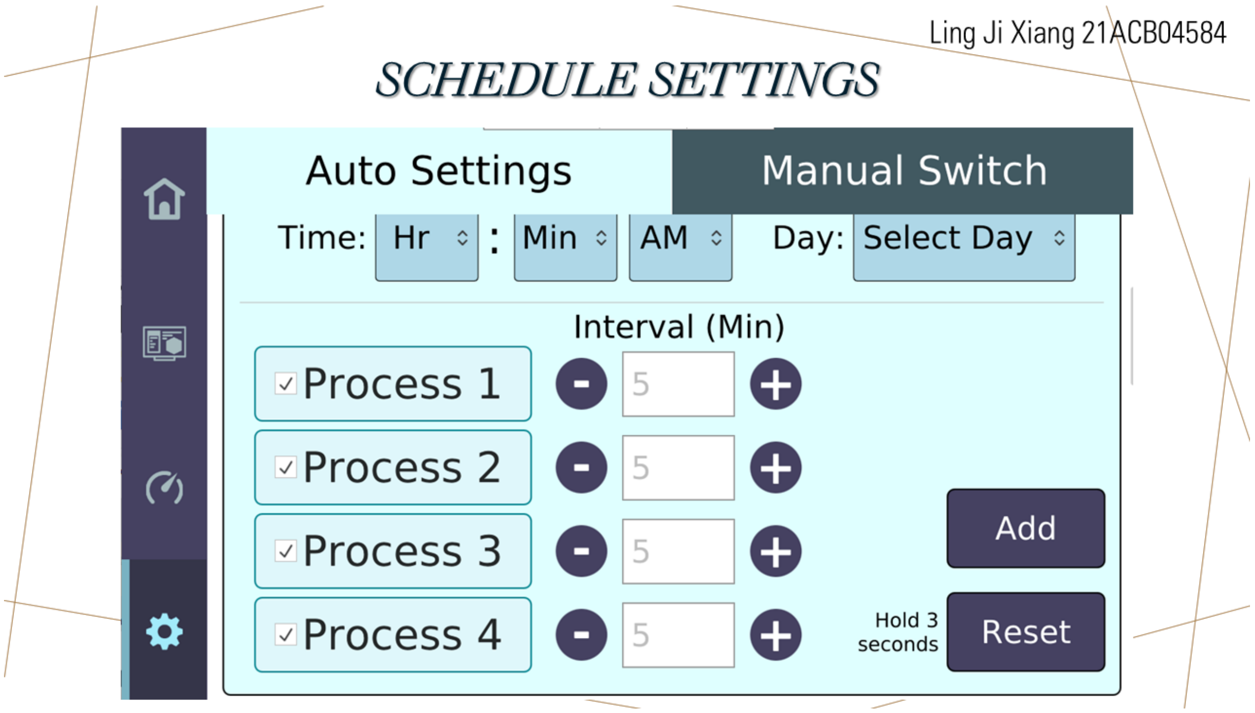

Ling Ji Xiang 21ACB04584

SCHEDULE LIST TAB (USER)

No.	Schedule Time	Process
1	Day: Tuesday Time: 4:25PM	Process 1 (1min) Process 2 (1min) Process 3 (1min) Process 4 (1min)
2	Day: Tuesday Time: 11:50PM	Process 1 (20min) Process 2 (20min) Process 3 (30min) Process 4 (40min)

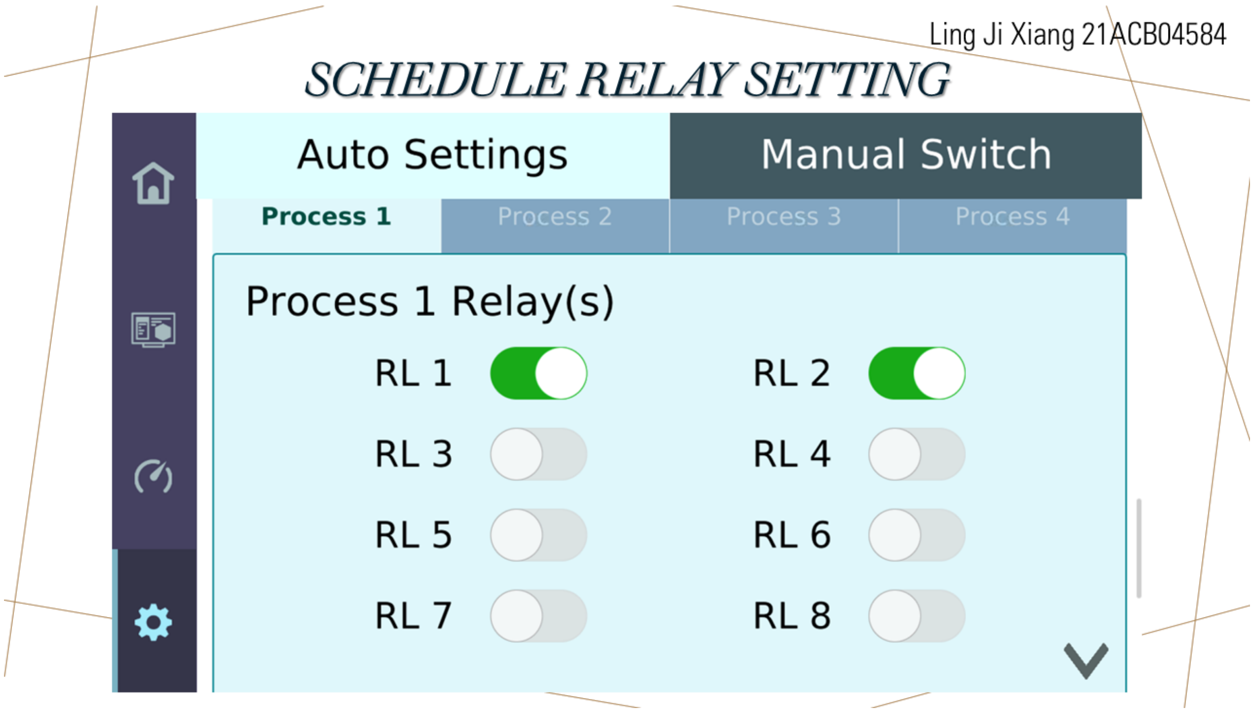
Schedule Settings

This tab is the main function of the scheduling system, which user add the time and interval for each process.



Schedule Relay Setting

This tab let user to choose which relay to turn on during the specific process.

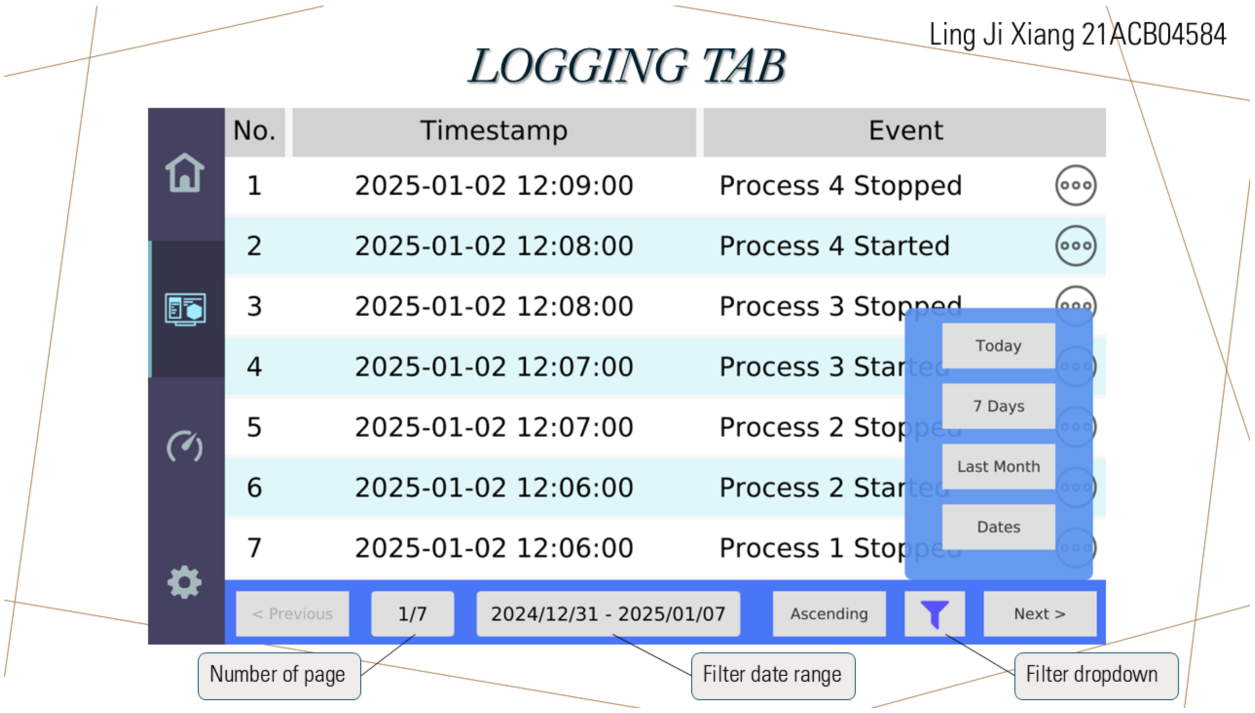


Process 1 Relay(s)

RL 1 [✓]
RL 2 [✓]
RL 3
RL 4
RL 5
RL 6
RL 7
RL 8

Logging Tab

This tab show the logging for every process event. It also allow user to filter the logs by days and ascending/descending.


LOGGING TAB

Ling Ji Xiang 21ACB04584

No.	Timestamp	Event
1	2025-01-02 12:09:00	Process 4 Stopped
2	2025-01-02 12:08:00	Process 4 Started
3	2025-01-02 12:08:00	Process 3 Stopped
4	2025-01-02 12:07:00	Process 3 Started
5	2025-01-02 12:07:00	Process 2 Stopped
6	2025-01-02 12:06:00	Process 2 Started
7	2025-01-02 12:06:00	Process 1 Stopped