

Smart Agriculture Monitoring System

Universidade de Aveiro

Sebastian D. González, Daniel Nascimento
Pedrinho



Smart Agriculture Monitoring System

Dept. de Eletrónica, Telecomunicações e Informática
Universidade de Aveiro

sebastian.duque@ua.pt(103690), dpedrinho01@ua.pt(107378)

4 de novembro de 2024

Conteúdo

1	Introduction	1
2	Application Overview	2
2.1	Login	2
2.2	Real-time Data Visualization	3
2.2.1	Home tab - Data Visualization and PDF Download	3
2.2.2	Map tab	4
2.3	Dark mode	5
3	Technical Implementation	6
3.1	MQTT Sensor Simulation	6
3.1.1	Simulated Sensor Apps	6
3.1.1.1	Data Generation and Transmission	6
3.1.1.2	MQTT Connection	7
3.1.1.3	Interfaces	7
3.1.1.4	Control Options	7
3.2	Data Storage	8
3.3	DB Structure	9
3.4	Dependencies	9
4	Challenges and Solutions	10
4.1	ChangeNotifier	10
4.2	DB Updating	11
5	Future Work	12
6	Conclusion	13
7	Repository and Contributions	14

Lista de Figuras

1.1	App Icon	1
2.1	Login interface screenshot	2
2.2	Home tab	3
2.3	Pdf download	4
2.4	Map with all markers	4
2.5	Sensor 1	4
2.6	Sensor 2	4
2.7	Light Mode	5
2.8	Dark Mode	5
3.1	Sensor 1	7
3.2	Sensor 2	7
3.3	Error Sensor1	8
3.4	Error Sensor2	8

Introduction

The **Smart Agriculture Monitoring System** is a mobile application developed to help users in tracking essential environmental conditions in agricultural terrains. Utilizing the MQTT protocol, this app enables real-time monitoring of various sensor data, including temperature, atmospheric pressure, luminosity and humidity. This app providing instant access to these metrics and it aims to support farmers, agronomists, and agricultural technologists in making informed decisions and monitor plants and harvest.

This monitoring system aims the growing need for precision in agriculture, where data insights and monitoring can significantly impact productivity and sustainability. With our user-friendly interface and robust backend integration, the application ensures reliable data transmission and visualization. This allowing users to respond to environmental changes and unpredictable situations.



Figura 1.1: App Icon

Application Overview

Our application incorporates several essential features in order to meet the demands of modern agriculture. These functionalities are designed to provide users with critical insights and efficient management of the environmental data.

2.1 Login

The app incorporates a secure login system to ensure that data remains personalized and protected for each user, having each user their associated data received and sensor locations. The login requires users to have an account with unique username and password.

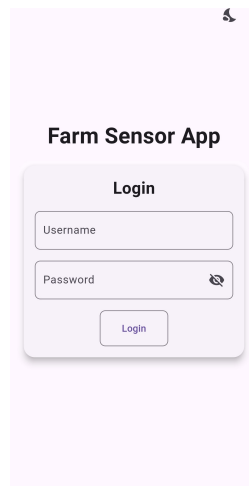



Figura 2.1: Login interface screenshot

When opening the app, users are presented with textboxes to enter their username and password. To enhance usability, the login screen also includes a **Show Password** button , allowing users to toggle the visibility of their password as they type. This is very helpful to avoid input errors by the users.

2.2 Real-time Data Visualization

Once the login is successful the user is presented with the **Real-time Data Visualization** feature that serves as the core of the whole application. This functionality displays incoming data from sensors (temperature, humidity, pressure and luminosity).

To enhance user experience, the application's primary interface is structured with two tabs to enable users to use different features without having to leave the main screen.

2.2.1 Home tab - Data Visualization and PDF Download

This tab provides users with access to all collected data from every sensor in real-time. It displays continuous updates of all data for metrics temperature, humidity, pressure, and luminosity presenting a comprehensive view of environmental conditions across all the monitored areas.

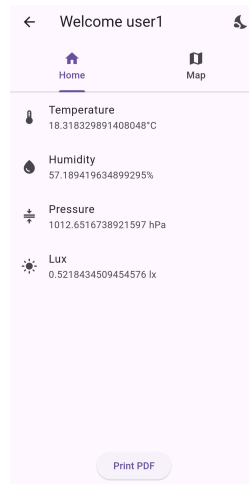


Figura 2.2: Home tab

In addition to live data visualization, this tab includes a **PDF Download** feature, allowing users to generate a pdf report of the collected sensor data at the moment the button is pressed. This feature allows users to easily document and share measures to monitor the plants.

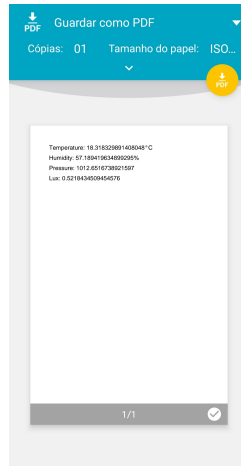


Figura 2.3: Pdf download

2.2.2 Map tab

The Map Tab provides a visual interface that allows users to view the geographical distribution of all sensors deployed in the agricultural fields. Each sensor is represented by a marker on the map. When users press in a marker, a popup appears displaying the latest reading from that sensor and plotting the real-time reading in a graph.



Figura 2.4: Map with all markers

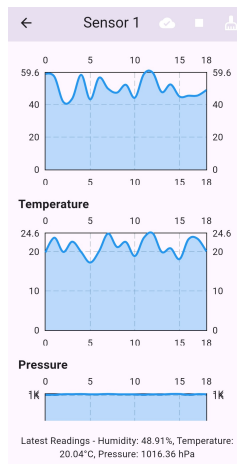


Figura 2.5: Sensor 1

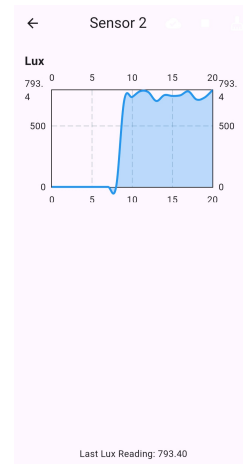


Figura 2.6: Sensor 2

With this geo-visualization, users get to determine the particular areas that require focus and, in cases when the area is extensive, monitor changers in the environment in sections of the field. The map helps in understanding different areas of the field affected by certain environmental conditions at a given time.

2.3 Dark mode

We also added a **Dark Mode button** in all tabs and on the log screen in order to switch to a darker interface theme across the entire application. We added this feature in order to the app to be more adaptable to varying lighting conditions and improving visual comfort across all functionalities.

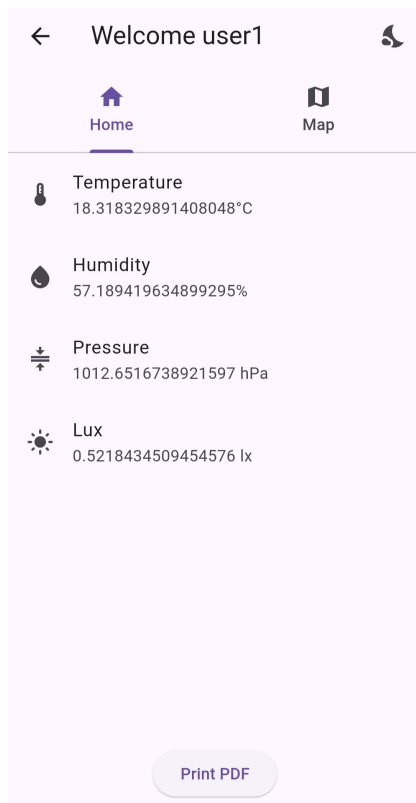


Figura 2.7: Light Mode

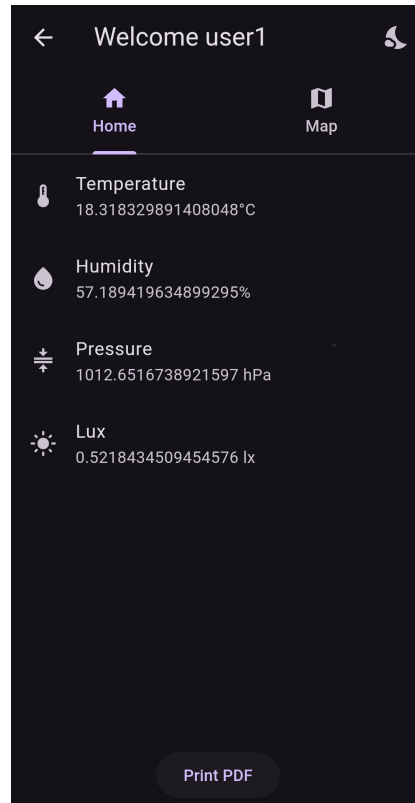


Figura 2.8: Dark Mode

Technical Implementation

3.1 MQTT Sensor Simulation

We decided to simulate the received data through an **MQTT Sensor Simulation system**. This allows us to mimic real-world environmental data in a managed, simulated environment. By using **MQTT** (Message Queuing Telemetry Transport), we simulate live sensor readings for luminosity, temperature, humidity, and pressure through an external app that is sending all the data. The main app is supported by another external Flutter app that generates and sends sensor data periodically, replicating real-time sensor updates through an MQTT broker. We created two **Simulated Sensor Apps** to achieve our goals.

3.1.1 Simulated Sensor Apps

Both **Simulation Sensor app** closely mimics a real sensor network, helping perfect data processing, visualization, and alerting features of the **Smart Agriculture Monitoring System** without needing actual physical sensors. The simulation creates random values within specified ranges for each parameter depending in which application its being used.

3.1.1.1 Data Generation and Transmission

The simulation app creates random data readings within realistic ranges for each environmental parameter. Each reading is then packaged into a **JSON** object and then encoded for the MQTT transmission. For example, a typical message may contain fields such as temperature, humidity, pressure, and a unique **sensor_id**.

```
1 {  
2   "type": "sensor_reading",  
3   "id": 1,  
4   "humidity": 78.67669398834957,  
5   "temperature": -5.245224817695348,  
6   "pressure": 143.6935320125267  
7 }
```

Listing 3.1: JSON message from Sensor1

3.1.1.2 MQTT Connection

The simulated data is published into a specific **MQTT topic** called **test/flutter/topic** and uses the **mqtt_client** dart package to establish a connection to the test.mosquitto.org **MQTT broker**.

3.1.1.3 Interfaces

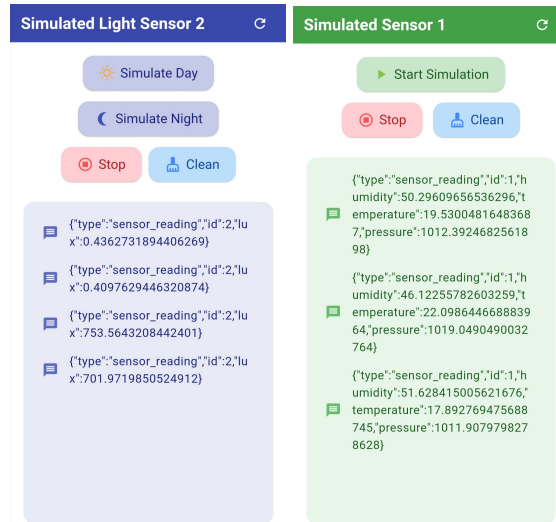


Figura 3.1: Sensor 1

Figura 3.2: Sensor 2

3.1.1.4 Control Options

The simulation app includes several user controls for managing the data flow and for debugging purposes:

- **Start Simulation:** Initiates the periodic data transmission, allowing the app to send new sensor readings.
- **Stop Simulation:** Pauses data transmission.
- **Clear Messages:** Clears the message log displayed on the screen, helping users keep track of only the most recent data.
- **Refresh button:** allows users to quickly re-establish the connection to the MQTT broker, in case of a disconnection, ensuring they receive the latest sensor data without needing to restart the application. When the connection to the MQTT broker fails, a popup appears to inform users of the issue, allowing them to reconnect.

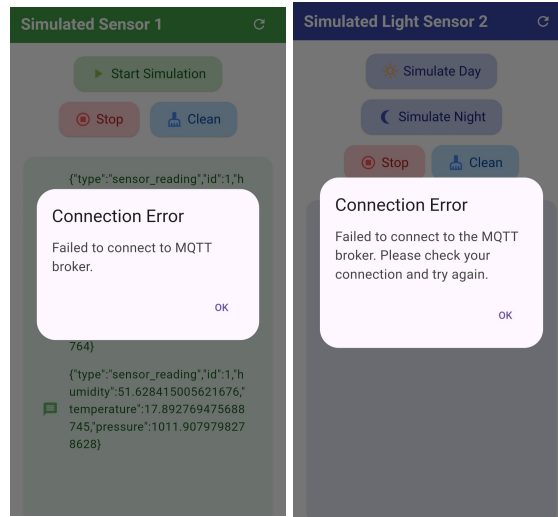


Figura 3.3: Error Sensor1

Figura 3.4: Error Sensor2

3.2 Data Storage

When the data transmission ends by pressing the appropriate button, the latest measures of all the sensors (including the default measures if some data value hasn't received data yet) are all sent to a local database, that stores the measured values in accordance to the current logged in user.

The database in question is a simple hive box database, containing all of the users information, including associated markers for the Map tab, latest measurements and login information.

```

1  if (result != null) {
2      Provider.of<SensorData>(context, listen: false).updateSensorData(
3          result['humidity'],
4          result['temperature'],
5          result['pressure'],
6          result['lux'],
7      );
8      user.temperature = result['temperature'];
9      user.humidity = result['humidity'];
10     user.pressure = result['pressure'];
11     user.lux = result['lux'];
12     boxUsers.putAt(i, user);

```

Listing 3.2: Receiving data and Updating DB

For simplicity sake, no information is encrypted.

3.3 DB Structure

The database itself simply contains the users of our application. Each user contains its log in details, some information relevant to Map Marker creation, as well as the last measured values in the application itself

```
1  return User(  
2      username: fields[0] as String,  
3      password: fields[1] as String,  
4      width: fields[2] as double,  
5      height: fields[3] as double,  
6      latitude: fields[4] as double,  
7      longitude: fields[5] as double,  
8      lastValue: fields[6] as String,  
9  );
```

Listing 3.3: User Fields as present in user.g.dart

3.4 Dependencies

1. **flutter_map**: Its used to display the map in the application, enabling a location-based visualization.
2. **latlong2**: Provides geographical coordinates support for **flutter_map**, allowing easy handling of latitude and longitude data, for example for the markers location.
3. **flutter_map_marker_popup**: Adds popups to map markers to show the plotting of the measures of the sensors.
4. **hive**: It's a lightweight, NoSQL database used for the local data storage, in our project, allowing the app to store user data and sensor records efficiently.
5. **hive_flutter**: Integrates Hive with Flutter, simplifying database initialization and making it compatible with Flutter.
6. **shared_preferences**: Used for storing simple, persistent key-value pairs, like user preferences or theme mode, ensuring data persistence across app sessions.
7. **pdf**: This allows the app to generate the PDF report, enabling users to export data.
8. **printing**: Facilitates the actual printing of PDFs or sharing them with other apps, supporting users in saving or sharing generated reports.
9. **mqtt_client**: Manages the MQTT protocol for real-time communication with the sensors.
10. **fl_chart**: Used to create and plot graphs, providing visual representation of the sensor data.

Challenges and Solutions

4.1 ChangeNotifier

Following the presentation, we identified areas to improve in our program and decided to integrate [ChangeNotifier](#). This update helps our application handle state more effectively, ensuring real-time updates across components without the need for manual widget refreshing. With **ChangeNotifier**, we aim to create a more responsive and user-friendly experience, as data changes will now instantly update throughout the app in the sensor readings in all tabs(Home and Map). This improvement not only boosts performance but also adheres to state management best practices.

```
1 class SensorData with ChangeNotifier {
2   double temperature = 0.0;
3   double humidity = 0.0;
4   double pressure = 0.0;
5   double lux = 0.0;
6
7   void updateSensorData(double newTemperature, double newHumidity,
8     double newPressure, double newLux) {
9     temperature = newTemperature;
9     humidity = newHumidity;
10    pressure = newPressure;
11    lux = newLux;
12    notifyListeners();
13  }
```

Listing 4.1: Defining the ChangeNotifier Class in main.dart

```
1 void main() async {
2   ...
3   runApp(
4     MultiProvider(
5       providers: [
6         ChangeNotifierProvider(create: (_) => SensorData()),
7       ],
8       child: const MainApp(),
9     ),
10  );
```

```
11 }
```

Listing 4.2: ChangeNotifierProvider is used to make SensorData accessible throughout the widget tree in main.dart

```
1 class _HelloWorldPageState extends State<HelloWorldPage> {
2   @override
3   void initState() {
4     super.initState();
5
6     // Subscrição no stream do MQTT para atualizar o SensorData
7     mqttService.sensorStream.listen((data) {
8       final sensorData = Provider.of<SensorData>(context, listen: false
9     );
10    if (data['id'] == 1) {
11      sensorData.updateSensorData(
12        data['temperature'] ?? sensorData.temperature,
13        data['humidity'] ?? sensorData.humidity,
14        data['pressure'] ?? sensorData.pressure,
15        sensorData.lux,
16      );
17    } else if (data['id'] == 2) {
18      sensorData.updateSensorData(
19        sensorData.temperature,
20        sensorData.humidity,
21        sensorData.pressure,
22        data['lux'] ?? sensorData.lux,
23      );
24    }
25  });
26 }
```

Listing 4.3: MQTT updates the ChangeNotifier (SensorData) with new sensor values.

4.2 DB Updating

Initially, no data was stored in the database as part of the application design. However, following a review presentation, it was decided that storing measurement data would significantly enhance both functionality and user experience. By retaining measurement records in the database, the application gains the ability to display historical data, giving users access to past measurements and fostering a sense of continuity. This approach allows for a more comprehensive view of measurement trends over time, making the application feel more cohesive and dynamic, while also aligning with best practices for data persistence and analysis.

Future Work

For future possible work, there is always the option of implementing more sensors. Reaching internet connectivity, with to-be-defined purpose could also favor the application in terms of its scalability. Lastly, the database can be improved to be more robust and less error prone, though no errors could currently be detected.

Conclusion

With this project, we learned, from scratch, how to develop an application in flutter, as well as how to implementing several technologies together, including databases, connectivity, external sensors, and the overall inner workings of an application using Flutter.

Repository and Contributions

The project repository is available at the following [Github](#) link.

The contributions to the project were equally divided (50/50) between the two group members.