alarm-thingy: An IoT Smart Alarm to Help You Wake Up More Gracefully

Angelo Galavotti
Master's in Artificial Intelligence
University of Bologna
Bologna, Italy
angelo.galavotti@studio.unibo.it

I. Introduction

Traditional alarm clocks often rely on loud, jarring sounds that can disrupt not only the user but also others nearby. This conventional approach to waking can lead to stress, irritation, and an unpleasant start to the day. To address these challenges, this project presents an innovative IoT alarm system designed to offer a more personalized and non-intrusive waking experience.

By incorporating bed presence detection technology using a pressure mat sensor, the systems adapts to the user behaviour, and activates the alarm only when necessary. In addition, the alarm incorporates smart features, including sleep time tracking and weather-based alarm tones.

The user can interact with the system either through a browser-based frontend or via a Telegram bot. The browser-based interface provides a user-friendly UI where alarms can be created, deleted, modified, or toggled, and includes a settings menu that allows customization of preferences. Additionally, the Web App features a modal showing various statistics, including a Grafana-based dashboard and information on average delay of the data transmission.

In this document, the system's architecture, implementation, and evaluation will be presented, providing an overview of the whole design process.

II. Project's Architecture

With regard to technical details, the system leverages an ESP32 board, to which a set of sensors and modules are attached. In particular:

- *Sound Manager Module*: a DFPlayer Mini module which allows to load music from a MicroSD and receives commands from the ESP32 unit. In addition, it serves as the DAC for the speakers.
- *Speaker*: a simple sound system which outputs the alarm ringtone.
- *Pressure sensor*: it outputs a binary value to indicate the presence of pressure.
- *Blue LED*: it signals network status during its setup process and the rate at which the information is sent to a node.

Together, these components make up the edge device.

To allow for the complete implementation of the IoT pipeline, the project uses a set of services, which can be run on different nodes or on a single machine:

- *Data Proxy*: a Python application which implements both the backend used by each UI and the alarm logic. It also publishes the data to the InfluxDB and Grafana instances.
- *InfluxDB*: a time-series database that collects and stores data from the alarm.
- *MQTT broker*: a server which acts effectively as a mediator between the ESP32 and the data proxy server, using the MQTT protocol.
- *Grafana*: a service that displays a real-time graph of the received data, along with the current daily average sleep time.
- *Data analysis module*: it gives insights into the amount of hours slept by the user and other information. It also includes a separate script which computes the accuracy of the sensor.

Each service is associated with a Docker container, which is connected via a network defined through Docker compose. The only exception is the Data Proxy, for reasons that will be later explained.

The user can interact with these services through either of these options:

- *Frontend web application*: a front end application made in React, which leverages the Data Proxy as backend server. It allows to add, remove, update and toggle alarms. In addition, it features a data visualization menu which displays some useful information, such as the time slept in the last 24 hours and the current weather condition. Finally, the application includes a settings menu, allowing users to customize various options, including the sampling rate.
- *Telegram Bot*: a bot which allows the user to interact with various functionalities of the alarm through a series of commands. Many of the functionalities of this module are shared with the React-based frontend.

The data is sent to the Data Proxy from the alarm through either the HTTP or MQTT protocols. The user can choose which one of the protocols to use, and can switch to either of them in real time through the press of a button. The settings

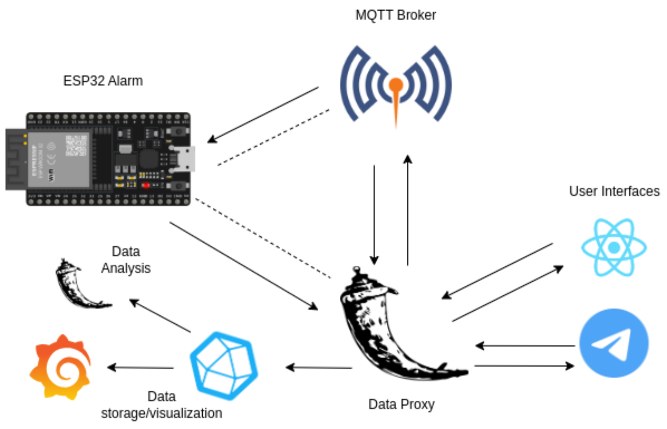


Fig. 1. A descriptive diagram of the system

(including the sampling rate) are communicated from the Data Proxy to the ESP32 using the MQTT Broker.

In order to receive and send the information to the MQTT Broker, the ESP32 must somehow know its IP. However, when dealing with a router or access point that uses a DHCP with Dynamic IP assignment, inserting manually the addresses each time the alarm is restarted may be unintuitive. As such, the system is designed so that the address of the ESP32 is retrieved automatically.

A diagram of the connection is shown in Figure 1. The dashed lines indicate a non-mandatory connection between nodes. In particular, the dashed line between the ESP32 and the MQTT Broker represents the situation in which the user switches to a MQTT connection for the sending the sensors data, while the dashed line between the Data Proxy and the ESP32 indicates the TCP connection that is established at startup for sending the Broker IP to the ESP32.

III. Project's Implementation

This section provides a detailed overview of the implementation and functionalities of each node and service outlined in the previous section.

A. Alarm

As previously mentioned, the alarm system is built around an ESP32 board as its core component. Its firmware is coded in MicroPython, a choice which sped up the deployment of new features. Conversely, MicroPython lacks the extensive community support that C++ enjoys, resulting in a significantly smaller selection of available libraries. Specifically, a custom library was needed to enable the ESP32 to interface with the DFPlayer Mini for music playback[1]. The library was also modified to increase the maximum volume threshold.

The firmware of the alarm operates as follows:

- At startup, the ESP32 attempts to connect to the Wi-Fi router. Afterwards, it establishes a TCP connection with the Data Proxy, which sends the MQTT Broker IP to it. Such address is then used to connect to the Broker.
- The ESP32 probes the sensor data and sends it to the Data Proxy using either HTTP, or to the MQTT Broker through MQTT. In the latter case, the ESP32 publishes the data through the `iot_alarm/data` topic.
- At the same time, the ESP32 is subscribed to the topic `iot_alarm/command` to receive commands such as starting the alarm or stopping it. In addition, the ESP32 uses this topic to receive setting changes, such as the sampling rate.
- The alarm starts only if the user lies on the pressure mat. In order to filter the data, it uses a simple moving average system, which involves computing the average of latest K values of the sensor. The default value of K is 10, however the user can set the window size through the settings menu of the Web App. Based on the weather data received through the `iot_alarm/weather` topic, the alarm will play one of three different chimes.
- After the alarm is activated by the Data Proxy through the `start_alarm` command, it can be stopped if the user gets up or if a `stop_alarm` signal is received via MQTT.

[1]The library can be found at https://github.com/lavron/micropython-dfplayermini

Additionally, *alarm-thingy* features an "angry mode", which can be enabled by the user through the Web App. This feature consists in the alarm playing a metal or energetic song if 30 seconds pass without the user getting out of bed or deactivating the alarm via the Web App or Telegram Bot.

To notify the user about the connections status, the alarm plays a sound effect in the following events:

- when the alarm has connected to the Wi-Fi router (accompanied by the Blue LED fading).
- when the alarm is waiting for the connection to the MQTT Broker.
- when the alarm has connected to the MQTT Broker.

In addition, the Blue LED will blink each time the data was sent successfully.

The alarm supports the mDNS protocol, meaning that the ESP32 can be accessed on the network through the hostname `esp32_alarm`, without needing to specify the IPs manually. In the event that the server or MQTT Broker go offline, the firmware includes a fallback mechanism that automatically attempts to reconnect to both components.

The sampling rate functionality of the alarm is implemented by starting a timer in milliseconds and periodically checking whether the timer has reached the value corresponding to the configured sampling rate.

B. Data Proxy

The Data Proxy is implemented as a Python program using the Flask library, along with its extensions Flask-MQTT and Flask-CORS. These extensions allow for communication with the MQTT Broker via MQTT and enable Cross-Origin Resource Sharing (CORS), respectively. It features API routes to manage the alarms, which function as the backend for both the Web App and the Telegram Bot.

At startup, the Data Proxy operates as follows:

- It loads the alarms from an `alarm.json` file.
- In another thread, it starts the alarm logic.
- Finally, it starts the Flask server and connects to the MQTT Broker.

To provide a functioning alarm system, the Data Proxy checks each instance in the `alarm.json` file, and compares its properties with the current weekday and the current time. In case an alarm has been triggered, it sends a 'GET' request to the weather API[2] and publishes the weather data to the `iot_alarm/weather` MQTT topic.

Since the Data Proxy needs to directly communicate with the ESP32 and resolve its hostname using mDNS, it cannot be configured as a Docker container. Instead, it must be executed as a standalone script.

C. MQTT Broker

The MQTT Broker is implemented through a Docker container using the official Eclipse Mosquitto[3] image. The MQTT Broker manages to the following topics:

- `iot_alarm/command`, handles commands to start an alarm and stop the alarm currently running on the ESP32. It also manages changes in the alarm settings.
- `iot_alarm/sensor_data`, supports the transmission of sensor data when the alarm is set to MQTT mode.
- `iot_alarm/weather`, handles the transmission of weather data.
- `iot_alarm/delay`, transmits the delay data to all clients subscribed to this topic.

D. Web App Frontend

The Web App frontend represents the main interface through which the user can interact with alarm-thingy. It is implemented in React, using Tailwind and daisyUI components to design the UI. It enables the user to create, modify, delete and toggle alarms. The frontend includes a Settings menu that allows to setup some of the properties of alarm-thingy, including:

- the sampling rate, which corresponds to the rate at which the data is sent to the Data Proxy.
- the volume of the alarm (i.e. the sound intensity).
- the size of the window for the moving average.

The Settings menu also allows to stop the current active alarm, to enable 'angry mode' and to use MQTT instead of HTTP for data transmission, among other settings. At startup, the list of alarms is fetched from the backend. Whenever the alarm properties or the settings have been changed, a post request is made to the backend.

Finally, the frontend includes an "Information" menu. This menu displays a Grafana dashboard showing the sensor state and its moving average over the last 30 minutes, along with details about the mean HTTP request delay[4] and the total sleep duration in the past 24 hours. This data is retrieved via HTTP requests from the Data Proxy and Data Analysis modules, respectively.

The Information menu also displays the current weather conditions for the location specified by the user, along with the likelihood of the user being in bed at the current time. This likelihood is calculated using a Prophet model via the Data Analysis module.

Some screenshots image of the Web App, its Settings menu and Information menu are shown in Figure 3, 4 and 5.

E. Grafana/InfluxDB

The Grafana and InfluxDB instances are set up as Docker containers, using the official Grafana and InfluxDB images. Before using their respective features, the user must first login to each service using the default credentials[5]. The InfluxDB container is already set up with a bucket and an organization. Similarly, the Grafana instance is set up beforehand to retrieve data from InfluxDB and provide the real-time data visualization.

The Grafana module also includes two dashboards that displays the cumulative average of the user's sleep duration, and the sensor state over the last 10 hours.

F. Data Analysis Module

The Data Analysis module is comprised of two parts:

- A server implemented with Python and Flask, to which the modules can make queries and receive information on the amount of hours the user has slept. To compute this value, it retrieves the necessary data from the InfluxDB instance.
- A simple Python script, which computes the accuracy of the of the sensor based on the data collected and a ground truth provided as input to the script.

The figures shown in Table I of Section IV were computed by this module.

The Data Analysis module includes a Python program that trains a Prophet model using synthetic data to enable the Web App to display the likelihood of the user being in bed at any given time. The synthetic data is designed to "force" the model to predict the user being in bed between 9 PM and 5 AM. However, the module also includes scripts that allow training an additional model on real user data retrieved from InfluxDB. Due to the generally low quality of the results produced by the model trained on real data, the use of synthetic data was adopted instead.

Additionally, the Data Analysis server calculates the cumulative average of daily sleep duration in a separate thread. This value is then stored in InfluxDB, enabling Grafana to display it on its dashboard. To compute it, the script first analyzes the database for existing records and calculates the overall average. Subsequently, it updates the running average whenever it detects that a day has passed.

[2]The Weather API is provided by https://open-meteo.com/

[3]More info at https://mosquitto.org/

[4]The request delay information is unavailable when using the MQTT protocol for sensor data transmission.

[5]The default credentials are *username*: admin, *password*: admin123

G. *Telegram Bot*

The Telegram Bot was implemented using the 'python-telegram-bot'[6] Python library. It provides an alternative way for users to interact with the ESP32 and manage alarms. The commands that the bot supports are the following:

- `/start`: starts the bot. On startup, it prints a welcoming message showing the
- `/add_alarm HH:MM weekdays`: adds a new alarm for time HH:MM, and which is repeat on the weekdays list specified by the user. The weekdays must be inputted as a list of integers.
- `/toggle_alarm <alarm_id>`: enables or disables an alarm without deleting it.
- `/delete_alarm <alarm_id>`: deletes an alarm from `alarms.json`.
- `/update_alarm <alarm_id> HH:MM weekdays`: updates the properties of alarm.
- `/list_alarms`: lists all alarm entries from the `alarms.json` file.
- `/stop_alarm`: stops the alarm that is currently running.

The Bot is public and can be accessed by any user with a Telegram account, by accessing the link t.me/alarm_thingy_bot. An example of interaction with the Telegram Bot can bee seen in Figure 6.

IV. Results

A. *User detection accuracy*

To evaluate the user detection capabilities of alarm-thingy, the sensor was tested in four trials, each lasting approximately one hour. Specifically, the accuracy was determined by comparing the total sleeping time calculated by the sensor with a ground truth. This comparison was conducted using a Python script that employs the same function as the one in the Data Analysis module to compute the total sleeping time. The results are presented in Table I, along with the mean of the recorded values.

Trial	**Accuracy**
1	94.06%
2	88.21%
3	96.72%
4	92.11%
Mean	92.78%

TABLE I. Measures obtained in the four trials.

B. *Mean Latency*

The mean latency is calculated using the cumulative average method. The ESP32 computes the total latency of the HTTP request, and sends the data via MQTT to the Data Proxy server, which in turn provides an API endpoint to fetch the moving average of the latency.

The results, displayed in Figure 2, were obtained by running the script for 45 minutes. To ensure more consistent and less noisy results, a small number of outliers were excluded from the analysis. The graph shows how the mean latency hovers around 430 ms, and increases over time. The mean value of the averages presented in this graph is 397.61 ms.

[6]Details can be found at https://pypi.org/project/python-telegram-bot/

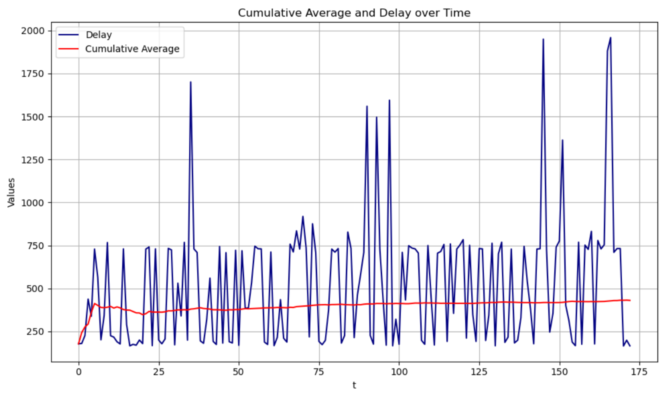
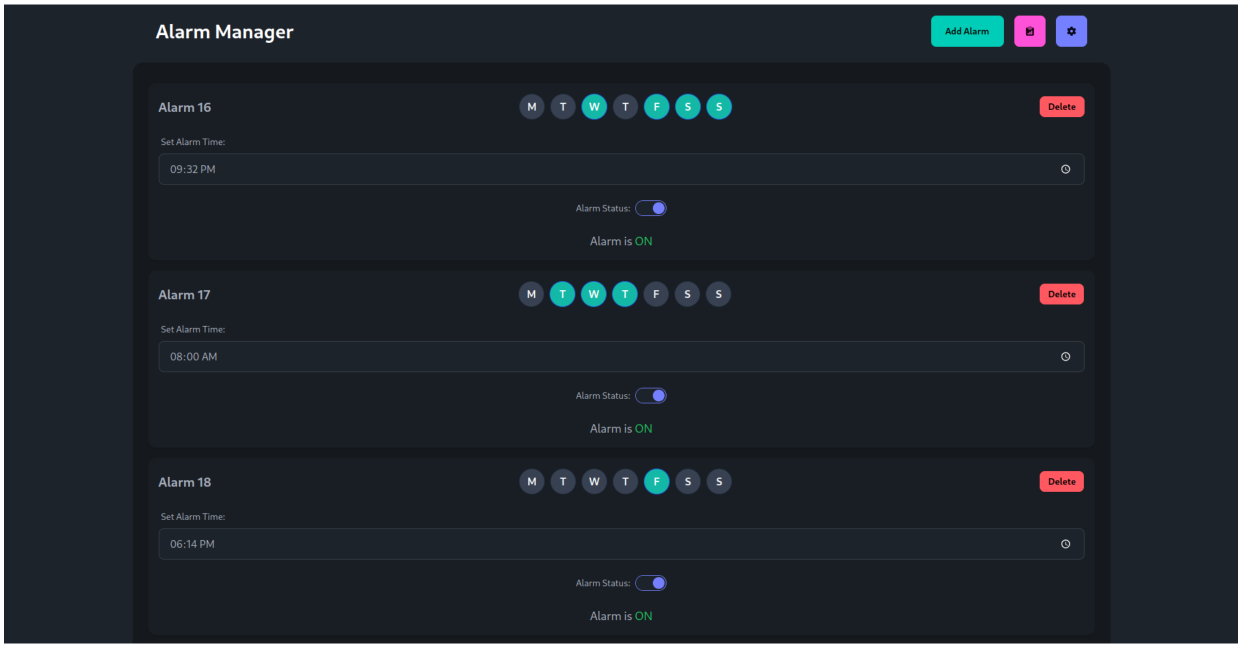


Fig. 2. A graph showing the cumulative average of the delays.


Alarm Manager

Add Alarm

Alarm 16 — M T W T F S S — Delete

Set Alarm Time: 09:32 PM

Alarm Status:

Alarm is ON

Alarm 17 — M T W T F S S — Delete

Set Alarm Time: 08:00 AM

Alarm Status:

Alarm is ON

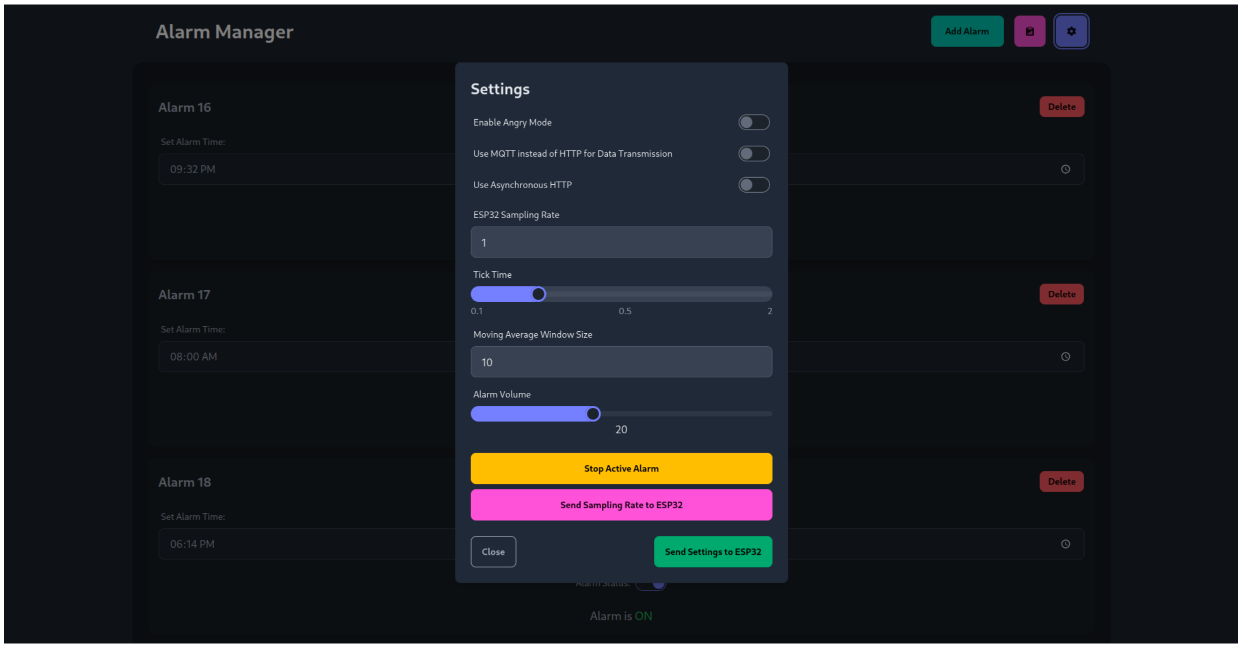
Alarm 18 — M T W T F S S — Delete

Set Alarm Time: 06:14 PM

Alarm Status:

Alarm is ON

Fig. 3. The Web App landing page.


Settings

Enable Angry Mode

Use MQTT instead of HTTP for Data Transmission

Use Asynchronous HTTP

ESP32 Sampling Rate: 1

Tick Time: 0.1 — 0.5 — 2

Moving Average Window Size: 10

Alarm Volume: 20

Stop Active Alarm

Send Sampling Rate to ESP32

Close

Send Settings to ESP32

Fig. 4. The Settings Modal Menu.

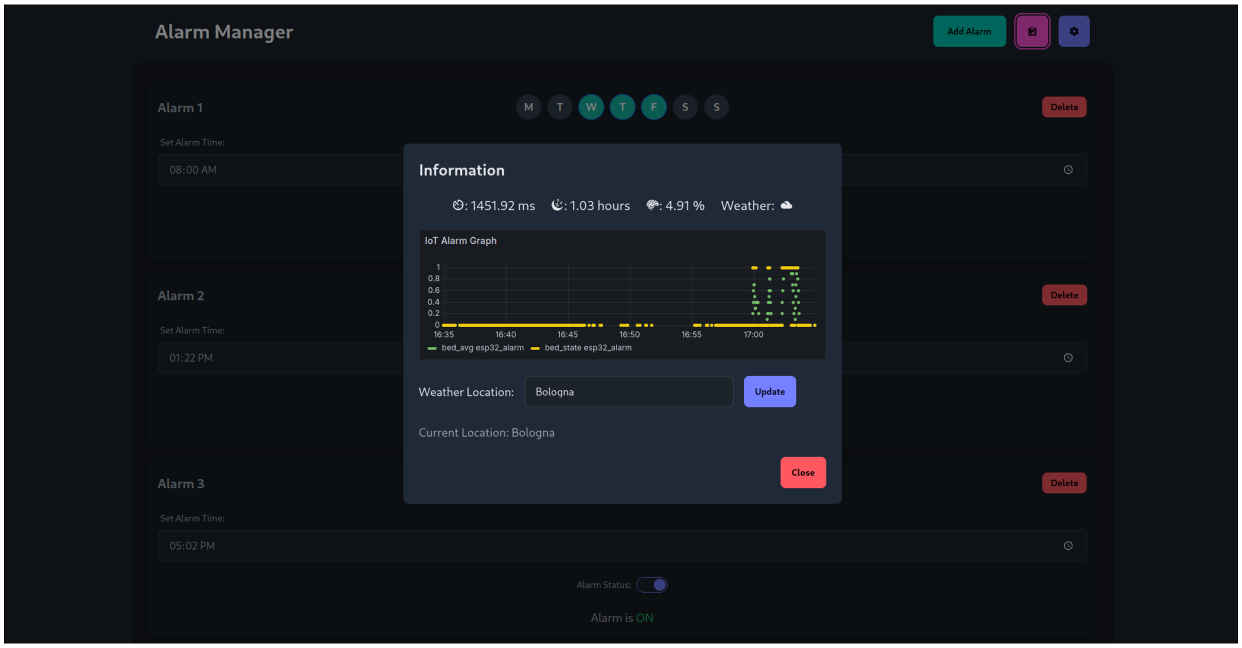
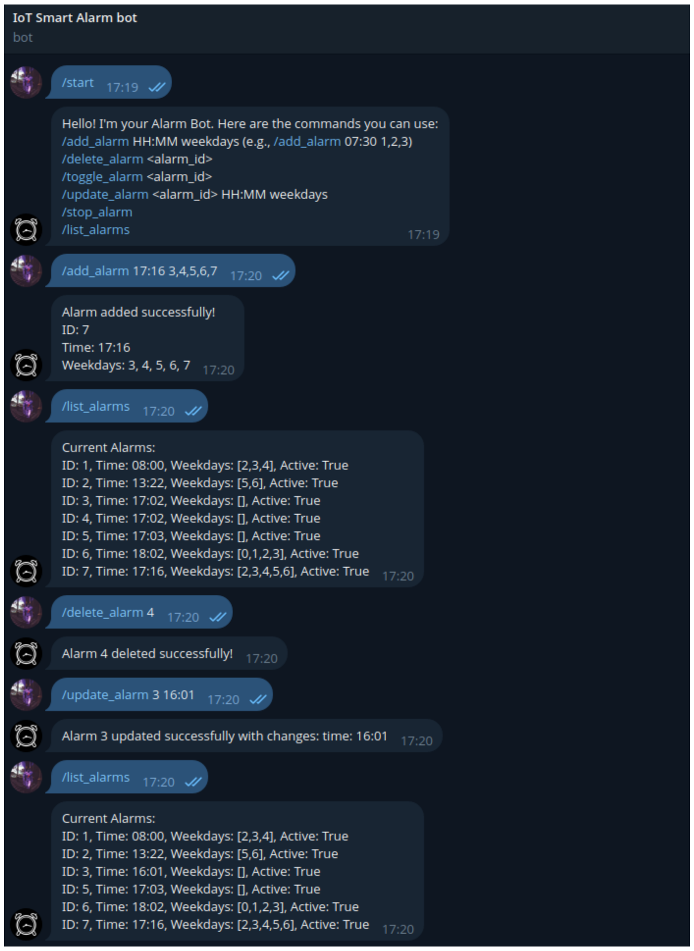


Fig. 5. The Information Modal Menu.



IoT Smart Alarm bot
bot

/start 17:19

Hello! I'm your Alarm Bot. Here are the commands you can use:
/add_alarm HH:MM weekdays (e.g., /add_alarm 07:30 1,2,3)
/delete_alarm <alarm_id>
/toggle_alarm <alarm_id>
/update_alarm <alarm_id> HH:MM weekdays
/stop_alarm
/list_alarms 17:19

/add_alarm 17:16 3,4,5,6,7 17:20

Alarm added successfully!
ID: 7
Time: 17:16
Weekdays: 3, 4, 5, 6, 7 17:20

/list_alarms 17:20

```
Current Alarms:
ID: 1, Time: 08:00, Weekdays: [2,3,4], Active: True
ID: 2, Time: 13:22, Weekdays: [5,6], Active: True
ID: 3, Time: 17:02, Weekdays: [], Active: True
ID: 4, Time: 17:02, Weekdays: [], Active: True
ID: 5, Time: 17:03, Weekdays: [], Active: True
ID: 6, Time: 18:02, Weekdays: [0,1,2,3], Active: True
ID: 7, Time: 17:16, Weekdays: [2,3,4,5,6], Active: True 17:20
```

/delete_alarm 4 17:20

Alarm 4 deleted successfully! 17:20

/update_alarm 3 16:01 17:20

Alarm 3 updated successfully with changes: time: 16:01 17:20

/list_alarms 17:20

```
Current Alarms:
ID: 1, Time: 08:00, Weekdays: [2,3,4], Active: True
ID: 2, Time: 13:22, Weekdays: [5,6], Active: True
ID: 3, Time: 16:01, Weekdays: [], Active: True
ID: 5, Time: 17:03, Weekdays: [], Active: True
ID: 6, Time: 18:02, Weekdays: [0,1,2,3], Active: True
ID: 7, Time: 17:16, Weekdays: [2,3,4,5,6], Active: True 17:20
```

Fig. 6. An example of interaction with the Telegram Bot.