

NEST - Architectures and Service Platforms

Version 1.0

Table of Contents

NEST Device	5
• NEST Device Firmware	5
• Testing and Validation	6
NEST Simulation	8
• NEST Simulation	8
• Automated Deployment Workflow	9
ThingsBoard	10
• ThingsBoard Dashboard Overview	10
• NEST Device Profile Configuration	11
• NEST Rule Chain Logic	13
Telegram	14
• NEST Telegram Controller	14
• NEST API Client	15
• System Configuration	23

Welcome to NEST's Documentation

Welcome to the documentation of the NEST project. This document provides a comprehensive guide to understanding, using, and developing a technical solution for the automated management of poultry farms.

In this documentation, you will find detailed explanations of the system's architecture, which focuses on egg quality monitoring and protection. It covers the integration of physical hardware with IoT service platforms and the use of edge computing to ensure low-latency responses.

Whether you are a developer, agricultural engineer, or farm manager, this documentation aims to serve as a valuable resource for understanding and extending the capabilities of the Next-generation Edge System for Tracking (NEST).

What is NEST?

NEST (Next-generation Edge System for Tracking) is an innovative project developed to solve real-world challenges in smart farming. The system automates the monitoring of incubation conditions by periodically measuring environmental variables like temperature and humidity.

The system implements a hybrid architecture where physical and virtual worlds coexist:

- Physical NEST Device: Based on an ESP32 microcontroller, it integrates sensors for temperature, humidity, and weight, alongside actuators such as an intelligent RFID lock and servomotors for automatic doors.
- Simulation Environment: A Python-based software layer that replicates the behavior of multiple nodes, allowing for system scalability testing.
- ThingsBoard Integration: Acts as the primary service orchestrator for data aggregation, rule-based alerts, and digital twin management.

Beyond simple telemetry, NEST features Autonomous Edge Decision-making. For example, if the system detects a weight increase (an egg) but identifying sensors show the hen has left, it automatically triggers actuators to close the nest door, protecting the production from predators.

Code Documentation

NEST Device

NEST Device Firmware

The **NEST (Next-generation Edge System for Tracking)** firmware is an ESP32-based IoT application designed for smart poultry farming. It integrates environmental monitoring, access control, and weight measurement using a multitasking approach powered by **FreeRTOS**.

Main Features

- **Real-time Environmental Monitoring:** Periodic reading of temperature and humidity via DHT22.
- **Access Control:** High-frequency RFID scanning (MFRC522) for hen and farmer identification.
- **Weight Measurement:** High-precision egg detection using a load cell and HX711 amplifier.
- **Remote Commanding:** Bidirectional communication with ThingsBoard for actuator control (Servos and RGB LED).
- **Fault Tolerance:** Concurrent error tracking for sensors and network connectivity.

Firmware Architecture

The firmware utilizes **FreeRTOS** to manage two core concurrent tasks, ensuring that high-speed events (like RFID detection) do not interfere with periodic telemetry.

Task Management

- **taskTelemetry (Periodic):** Executes every 10 seconds (default). It performs data fusion from all sensors and publishes a comprehensive JSON payload to the MQTT broker.
- **taskRFID (Async):** Executes every 250ms. It provides “Instant Publish” capabilities, sending a telemetry update immediately upon detecting a new card to minimize latency for door operations.

Resource Synchronization

To prevent race conditions and hardware conflicts, the system implements the following **Mutexes**:

- `mqttMutex` : Synchronizes access to the Shared MQTT client and WiFi radio.
- `spiMutex` : Manages the SPI bus contention between the MFRC522 RFID reader and other peripherals.

Technical Reference

`String checkRFID()`

Scans for the presence of an RFID tag. It includes a forced Wakeup sequence to ensure tags are detected even if held continuously in front of the reader.

Returns:

The UID of the detected card in Hex format, or an empty string if none.

`void setLedColor(String color)`

Updates the color of the Common Anode RGB LED.

Parameters:

color – The target color name (off, red, green, blue, on).

`void operateDoor(String command)`

Drives two SG90 servomotors to specific angles to simulate a physical door opening or closing.

Parameters:

command – “open” to trigger the entry position, otherwise “closed”.

`void mqttCallback(char *topic, byte *payload, unsigned int length)`

Processes incoming JSON messages from ThingsBoard. It extracts **Shared Attributes** to remotely update the door state, LED feedback, or the telemetry reporting frequency.

Testing and Validation

To ensure the reliability of the NEST system, each hardware peripheral underwent a dedicated **Unitary Testing** phase. This process verified that both the physical wiring and the specific software drivers worked correctly before integration into the main FreeRTOS multitasking firmware.

Hardware Unitary Tests

The following procedures were executed to validate each component of the ecosystem:

- **DHT22 (Temperature and Humidity):**

- **Objective:** Data consistency and single-bus timing.
- **Method:** Measured ambient conditions and verified output accuracy.
- **Reference:** [ESP32 with DHT22 Guide](#)

- **Load Cell (HX711):**

- **Objective:** Accuracy and calibration of weight sensing for egg detection.
- **Method:** Applied known weights and verified results in the Serial Monitor.
- **Reference:** [ESP32 Load Cell and HX711 Tutorial](#)

- **RFID (MFRC522):**

- **Objective:** Successful UID identification and SPI bus stability.
- **Method:** Scanned multiple tags to ensure unique hex strings were captured correctly.
- **Reference:** [RFID MFRC522 Video Guide](#)

- **Servos (SG90):**

- **Objective:** Precision of movement and power stability under load.
- **Method:** Commanded open and closed positions to simulate door operation.
- **Reference:** [Servo Control Video Tutorial](#)

- **RGB LED:**

- **Objective:** Color mixing and PWM duty cycle validation.
- **Method:** Executed a test cycle to display Red, Green, and Blue colors for local feedback.

System Integration Results

After the unitary tests, a **System Integration Test** was conducted to confirm the end-to-end data pipeline. As shown in the local feedback logs, the system successfully transitions through states (e.g., from `WAITING_FOR_HEN` to `HEN_INSIDE`) based on real-time sensor inputs.

All telemetry data—including temperature, humidity, weight, and UID—is successfully synchronized with the **ThingsBoard** dashboard via secure MQTT.

NEST Simulation

NEST Simulation

This module provides a Python-based simulation environment that replicates the behavior of multiple NEST nodes. It allows for scalability testing and validation of the system's logic in parallel with physical hardware.

Operational Logic

The simulation runs a **Finite State Machine (FSM)** that mimics the life cycle of a poultry nest. The system transitions through states based on random persistence and environmental conditions.

- **`WAITING_FOR_HEN`**: Idle state with no weight or UID detected.
- **`HEN_INSIDE`**: Simulates hen entry with random weight (2000g-3500g) and a specific UID.
- **`EGGS_DEPOSITED`**: Represents the hen leaving while the weight of the eggs remains.
- **`PERSON_COLLECTING`**: Authorized user (Farmer) collecting production, identified by a unique UID.

Code Documentation

`class nest_sim.SmartNest(token, name)[source]`

Bases: `object`

Class representing a Smart Nest node (NEST).

Manages internal state, weight simulation logic, and user identification through a Finite State Machine (FSM).

`__init__(token, name)[source]`

Initializes the nest with credentials and default values.

Parameters:

- **token** – ThingsBoard access token for the device.
- **name** – Identifiable name for logs.

`update_logic()[source]`

Updates the cyclic logic. The farmer UID is sent once, while the hen UID persists while she is inside.

`nest_sim.on_message(client, userdata, msg)[source]`

Callback executed when an MQTT message (Shared Attributes) is received.

Enables remote control of telemetry frequency, door status, and LED color from the ThingsBoard dashboard.

`nest_sim.run()[source]`

Main function to start the MQTT client and the simulation loop.

Automated Deployment Workflow

The NEST ecosystem supports a hybrid environment where physical nodes and simulated entities interact seamlessly. To manage this scalability, the project utilizes a batch-driven automated deployment.

Provisioning File (tokens.txt)

The tokens.txt file serves as a local database for device credentials. Each line maps a unique ThingsBoard Access Token to a Device Name.

`tokens.txt`

```
HUyaoCGGK6Pmu8nH3AWr,NEST_2
XAWy4XST70V5WmEHwV7h,NEST_3
kPfpX4EumVHdRR2icNjP,NEST_4
```

These tokens allow the simulation to connect as NEST 2, NEST 3, and NEST 4, which are provisioned under the same NEST Device Profile as the physical hardware.

Execution Script (run_all.bat)

The run_all.bat script automates the parallel execution of multiple virtual nodes. It iterates through the provisioning file and launches a separate process for each device.

run_all.bat

```
@echo off
for /F "tokens=1,2 delims=," %%A in (tokens.txt) do (
    start "%%B" cmd /k python nest_sim.py %%A %%B
)
```

Technical Workflow

- Parsing: The script uses a for loop with a comma delimiter to extract the token (%%A) and the name (%%B).
- Parallelism: The start command opens a new command prompt window for each instance, allowing the nodes to run concurrently.
- Persistence: The /k flag keeps the terminal open after the script execution, enabling the developer to monitor the local feedback and MQTT logs for each node.
- Integration: Once launched, all nodes synchronize their Shared Attributes with ThingsBoard, appearing immediately on the Geographic Map and the Entities Summary Table.

ThingsBoard

ThingsBoard Dashboard Overview

The NEST dashboard serves as the primary interface for the real-time monitoring and management of poultry farm nodes. It integrates telemetry data, geospatial information, and remote control capabilities into a single cohesive view.

Main Components and Functionalities

1. Geospatial Monitoring

The dashboard utilizes an **OpenStreetMap** widget to track the physical location of each nest. By mapping `latitude` and `longitude` attributes, the system provides a clear geographical overview of the distributed nodes across the facility.

2. Environmental & Production Telemetry

Real-time data visualization is provided for critical environmental and production metrics:

- **Climate Monitoring:** Continuous tracking of **Temperature** and **Humidity** levels to ensure optimal conditions for the hens and eggs.
- **Production Detection:** Monitoring of the **Weight** sensor, which is the primary indicator of a new egg being laid.
- **Power Management:** Tracking of **Battery** levels for physical ESP32 nodes to prevent downtime.

3. Edge Control & Actuators

The dashboard enables direct interaction with the hardware through **Shared Attributes** widgets:

- **Access Control:** Real-time status and manual override of the **RFID Lock**.
- **Door Automation:** Control and monitoring of the **Servo** motor that operates the nest entrance.
- **Digital Twin Logic:** Visualization of the autonomous edge logic, showing when a nest is “Locked” or “Unlocked” based on the presence of an egg and the absence of the hen.

4. Data Analysis & History

- **Latest Values:** High-visibility cards showing the most recent telemetry received.
- **Time-Series Charts:** Historical graphs (e.g., weight trends or temperature fluctuations over the last 24 hours) to assist in long-term farm management and analysis.

5. Scalability & Entity Management

Through the use of **Entity Aliases** (such as “Actual NEST”), the dashboard can dynamically switch between different physical devices or simulated Python nodes, allowing the system to scale easily as more nests are added to the farm.

NEST Device Profile Configuration

The **NEST Device Profile** acts as a technical blueprint within ThingsBoard, ensuring that all physical and simulated nodes follow a standardized set of rules and communication protocols. It centralizes the logic for alarm generation and data transport security.

Technical Specifications

- **Transport Protocol:** The profile is configured to use **MQTT** as the primary transport stack.
- **Data Format:** All incoming telemetry and outgoing commands are structured using **JSON**.
- **Provisioning Strategy:** It uses the **Allow Create New Devices** strategy, enabling the automated deployment of simulated nodes via the `run_all.bat` workflow.

Alarm Rules and Operational Logic

The profile contains specialized rules to transform raw telemetry into actionable alerts for the farmer:

1. Environmental Bounds

- **Temperature Alarm:** Automatically triggers a “Warning” or “Critical” alarm if the temperature deviates from the optimal incubation range.
- **Humidity Alarm:** Monitors ambient moisture levels to protect egg quality, triggering alerts if values fall outside predefined thresholds.

2. Security and Productivity

- **Unauthorized Access:** Generates an alarm if a tag is scanned that does not match an authorized Hen or Farmer UID.
- **System Inactivity:** Includes a “Device Inactivity” rule that alerts the farmer if a NEST node stops sending telemetry, indicating a potential hardware or connectivity failure.

3. Edge Status Monitoring

- **Attribute Sync:** Tracks the synchronization of **Shared Attributes** (such as the door state or RGB LED color) to ensure the Digital Twin accurately reflects the physical state of the nest.

Integration with Service Platforms

The profile is linked to the **NEST Rule Chain**. This ensures that as soon as telemetry is received, it is processed to determine if an automated action is required, such as notifying the user via **Telegram** when a critical alarm is created.

NEST Rule Chain Logic

The **NEST Rule Chain** acts as the central intelligence of the system within ThingsBoard. It processes incoming sensor data, manages a **Finite State Machine (FSM)** for the nest's life cycle, and automates physical responses and external notifications.

Core Workflow

1. Message Classification and Validation

- **Message Type Switching:** The system first differentiates between `Post telemetry`, `Post attributes`, and other message types via the **Check Message Type** node.
- **Telemetry Validation:** The **Check Telemetry Message** node ensures incoming data contains necessary keys such as `temperature`, `humidity`, `weight`, `uid`, `error`, or `init`.
- **Hardware Error Handling:** If an `error` key is detected, the **Hardware Error** node creates a **CRITICAL** severity alarm including error details and a timestamp.

2. Digital Twin and FSM Management

The system manages the nest's state autonomously through specialized logic:

- **Finite State Machine (FSM): Using the `state` shared attribute, the Check FSM State and Change FSM State nodes control transitions between:**
 - `Waiting for Hen`: Remains until the UID matches the `henUID`.
 - `Hen Inside`: Active while the hen is detected.
 - `Eggs Inside`: Triggered when the hen leaves and weight is detected; it calculates estimated eggs (`weight / avgWeight`) and sets the door to `closed`.
 - `Recolecting Eggs`: Allows access to the farmer (`validUID`) to reset the cycle.
- **Access Control (UID):** The **Open Door** node validates if the incoming `uid` matches the `validUID`. If it matches, it toggles the `door` attribute and sets the LED `rgb` to **Green**. Unrecognized UIDs trigger the **Reject Door Use** node, setting the LED to **Red**.

3. Environmental Monitoring and Alarms

- **Dynamic Thresholds:** The **TempRange** and **HumRange** nodes fetch client attributes (`maxTemp`, `minTemp`, etc.) to use as bounds.
- **Alarm Life Cycle:**
 - If temperature or humidity exceeds limits, **Temperature/Humidity Out of Bounds** alarms are created.
 - Alarms are automatically cleared when telemetry returns to the safe range.
- **Egg Notification:** The **Eggs Inside** filter triggers a specific alarm when the egg count is greater than zero.

4. External Notifications

- **Telegram Integration:** The rule chain connects to the Telegram API to send updates to chat ID `1624527516`.
- **Real-time Alerts:** Notifications are sent for created/cleared environmental alarms and when eggs are detected or collected.

Technical Nodes

- **TBEL Scripting:** Extensively used for FSM transitions, egg estimation logic, and RGB status management.
- **Attribute Enrichment:** The **Get Attributes** nodes (TempRange, HumRange, UIDs, FSM) load configuration parameters into metadata before evaluation.
- **REST API Call:** Executes POST requests to the Telegram Bot.
- **Data Persistence:** Sensor data is saved via **Save Timeseries**, and device states are pushed to **SHARED_SCOPE** to sync with the physical hardware.

Telegram

NEST Telegram Controller

This module implements a high-level Telegram interface for the NEST IoT system. It acts as a bridge between the end-user and the physical or simulated NEST nodes, providing real-time telemetry, door control, and environmental configuration.

Operational Logic

The bot operates using an **asynchronous event-driven architecture** managed by the *python-telegram-bot* library. It maintains a stateful session for each user via *user_data* to ensure commands are executed against the correct device.

- **Authentication Flow:** Users must authenticate via the `/login` command. The module uses a decorator-based system to restrict access to protected API endpoints.
- **Contextual Nest Selection:** Before interacting with sensors or actuators, a user must select a “Current NEST” from the available registry. This context is persisted across the session.
- **Menu Navigation:** The system uses a nested **Inline Keyboard** structure, allowing users to navigate from a global list of nests to specific device sub-menus (Telemetry, Door, Eggs, etc.).
- **Validation Decorators:**
 - `@require_login`: Ensures an active API token exists before sending PUT/POST requests.
 - `@require_nest_selected`: Ensures a Target Device ID is present in the context.

Command Hierarchy

The bot implements several command categories to manage the IoT ecosystem:

1. **System Commands:** `/start`, `/help`, and `/status` for connectivity health checks.
2. **Actuator Control:** * **Door**: Toggle states between *open* and *closed*. * **RGB**: Visual status feedback control.
3. **Environmental Tuning:** * Set min/max thresholds for **Temperature** and **Humidity** to trigger automated alerts on the physical hardware.
4. **Production Logging:** * Update **Egg Types** (Hen vs. Quail) to adjust weight calculation logic for production metrics.

NEST API Client

This module provides a robust interface for communicating with the IoT Platform backend. It handles all HTTP infrastructure, including authentication persistence, telemetry polling, and bidirectional attribute synchronization.

Operational Logic

The client acts as a high-level wrapper around the `requests` library, implementing specific patterns for IoT device management:

- **Session Management:** Uses a centralized `requests.Session` to maintain persistent headers (JWT Token) and optimize connection pooling across multiple API calls.
- **Two-Phase Attribute Updates:** Implements a “command-and-verify” pattern. When an attribute (like door status or temperature thresholds) is changed, the client performs up to three verification retries to ensure the physical device or the twin has successfully registered the new state.
- **Corpus-Specific Logic:**
 - **Telemetry:** Fetches time-series data using specific keys (humidity, temperature, weight).
 - **Attributes:** Manages both *Client Attributes* (thresholds, GPS) and *Shared Attributes* (door, RGB status).

Verification Protocol

To ensure system reliability, the `_verify_attribute_change` method is invoked after any write operation. It handles:

1. **Numeric Tolerance:** Compares floating-point values (like Temperature) with a small epsilon (0.001) to account for precision differences.
2. **Retry Backoff:** Introduces a short delay between attempts to allow the backend and the device to synchronize.
3. **Result Integrity:** Returns a boolean status and a descriptive message, allowing the Telegram interface to report precise errors to the user.

Code Documentation

`class api_client.APIClient(base_url: str)[source]`

Bases: `object`

Client to interact with the IoT Platform API.

Handles authentication, telemetry retrieval, and device attribute management (door, temperature, humidity, and location).

Parameters:

base_url (*str*) – The root URL of the IoT API.

__init__(base_url: str)[source]

Initializes the API client with a base URL and default headers.

login(username: str, password: str) → str | None[source]

Authenticates the user and saves the JWT token for subsequent requests.

Parameters:

- **username** (*str*) – The API username.
- **password** (*str*) – The API password.

Returns:

The JWT token if successful, None otherwise.

Return type:

Optional[*str*]

get_telemetry(device_id: str) → Dict | None[source]

Retrieves the latest timeseries telemetry data for a specific device.

Parameters:

device_id (*str*) – The unique identifier of the device.

Returns:

Dictionary containing telemetry keys and values, or None if the request fails.

Return type:

Optional[Dict]

get_door_status(*device_access_token: str*) → Dict | None[\[source\]](#)

Retrieves the ‘door’ attribute from the shared attributes of a device.

Parameters:

device_access_token (*str*) – The access token for the specific device.

Returns:

Dictionary with shared attributes, or None if the request fails.

Return type:

Optional[Dict]

set_door_status(*device_access_token: str, status: str*) → Tuple[bool, str][\[source\]](#)

Updates the door status and performs a verification loop to confirm the change.

Parameters:

- **device_access_token** (*str*) – The access token for the specific device.
- **status** (*str*) – The target status (‘open’ or ‘closed’).

Returns:

A tuple containing (success_status, feedback_message).

Return type:

Tuple[bool, str]

get_temperature_attributes(*device_access_token: str*) → Dict | None[\[source\]](#)

Fetches the client-side temperature thresholds (maxTemp, minTemp).

Parameters:

device_access_token (*str*) – Device access token.

Returns:

Dict containing maxTemp and minTemp.

Return type:

Optional[Dict]

set_temperature_attribute(device_access_token: str, attribute: str, value: float) → Tuple[bool, str][source]

Sets a temperature threshold and verifies the update.

Parameters:

- **device_access_token** (str) – Device access token.
- **attribute** (str) – The attribute name ('maxTemp' or 'minTemp').
- **value** (float) – The temperature value to set.

Returns:

Tuple (success, message).

Return type:

Tuple[bool, str]

get_humidity_attributes(device_access_token: str) → Dict | None[source]

Fetches the client-side humidity thresholds (maxHum, minHum).

Parameters:

device_access_token – Device access token.

Returns:

Dict containing maxHum and minHum.

Return type:

Optional[Dict]

set_humidity_attribute(device_access_token: str, attribute: str, value: float) → Tuple[bool, str][source]

Sets a humidity threshold and verifies the update.

Parameters:

- **device_access_token** – Device access token.
- **attribute** – The attribute name ('maxHum' or 'minHum').
- **value** – The percentage value to set.

Returns:

Tuple (success, message).

Return type:

Tuple[bool, str]

get_location_attributes(*device_access_token: str*) → Dict | None[*source*]

Retrieves current GPS coordinates (latitude, longitude) from the device.

Parameters:

device_access_token – Device access token.

Returns:

Dict containing latitude and longitude.

Return type:

Optional[Dict]

set_location_attributes(*device_access_token: str, latitude: float, longitude: float*) →

Tuple[bool, str][*source*]

Updates the device location coordinates and verifies the change.

Parameters:

- **device_access_token** – Device access token.
- **latitude** – Latitude float.
- **longitude** – Longitude float.

Returns:

Tuple (success, message).

Return type:

Tuple[bool, str]

`get_eggs_attribute(device_access_token: str) → str | None`[\[source\]](#)

Retrieves the ‘eggs’ shared attribute count.

Parameters:

device_access_token – Device access token.

Returns:

String value of egg count, or None.

Return type:

Optional[str]

`get_weight_attributes(device_access_token: str) → Dict | None`[\[source\]](#)

Retrieves weight configuration attributes (avgWeight, minWeight).

Parameters:

device_access_token – Device access token.

Returns:

Dict with avgWeight and minWeight.

Return type:

Optional[Dict]

`set_weight_attributes(device_access_token: str, avg_weight: float, min_weight: float) → Tuple[bool, str][source]`

Updates the weight reference for egg type detection.

Parameters:

- **device_access_token** – Device access token.
- **avg_weight** – Average weight in grams.
- **min_weight** – Minimum weight in grams.

Returns:

Tuple (success, message).

Return type:

Tuple[bool, str]

`get_egg_type(device_access_token: str) → str[source]`

Infers the egg type (Hen, Quail) based on the configured average weight.

Parameters:

- **device_access_token** – Device access token.

Returns:

'Hen', 'Quail', or 'Unknown'.

Return type:

str

`get_rgb_attribute(device_access_token: str) → str | None[source]`

Retrieves the shared 'rgb' status from the device.

Parameters:

- **device_access_token** – Device access token.

Returns:

RGB color name/value as string, or None.

Return type:

Optional[str]

is_logged_in() → bool[source]

Checks if the client has an active authentication token.

Returns:

True if a token exists, False otherwise.

Return type:

bool

logout()[source]

Invalidates the local token and clears the authorization headers.

System Configuration

The `config` module serves as the central repository for constants, credentials, and environment-specific settings. It defines the mapping between the physical IoT nodes and the logical representations used by the Telegram bot.

Component Overview

This module is structured into three main segments:

1. **Authentication Secrets:** Contains the Telegram Bot API token. It is the primary credential for the BotFather interface.
2. **Device Registry:** A static dictionary named `NESTS` that defines the hardware available in the system. Each entry links a human-readable display name to its internal UUID (`device_id`) and its private `access_token`.
3. **Connectivity Settings:** Defines the URI structure for the backend platform, distinguishing between the standard Device API (v1) and the Plugins API for telemetry and authentication.

Session Variables

The module also declares placeholders for runtime data such as `USER_TOKEN`, `USERNAME`, and `CURRENT_NEST`. These variables are updated dynamically during the bot's execution to maintain the state of the user session.

Module Documentation

Configuration module for the NEST IoT Telegram Bot.

This module contains the sensitive credentials, API endpoints, and the static registry of NEST devices used by the system.

config.TELEGRAM_TOKEN = '8560879729:AAEHgMY-NWSmXSejZ07uWvmfoxJfSWEWbCw'

The unique authentication token provided by BotFather.

config.API_BASE_URL = 'https://srv-iot.diatel.upm.es'

Root URL for the IoT Platform API.

config.API_V1_BASE_URL = 'https://srv-iot.diatel.upm.es/api/v1'

Base URL for Device API v1 (Attributes and RPC).

config.API_PLUGINS_BASE_URL = 'https://srv-iot.diatel.upm.es/api'

Base URL for Plugin-based APIs (Telemetry and Auth).

config.NESTS = {'NEST1': {'access_token': 'hNxbPHZG1A1Rft0LHAVO', 'device_id': 'b5697430-f455-11f0-b5e6-d92120c3d6c8', 'display_name': '🟡 NEST 1'}, 'NEST2': {'access_token': 'HUyaoCGGK6PMu8nH3AWr', 'device_id': 'e838eeb0-f458-11f0-b5e6-d92120c3d6c8', 'display_name': '🟡 NEST 2'}, 'NEST3': {'access_token': 'XAWy4XST70V5WmEHwV7h', 'device_id': 'ec94f5d0-f458-11f0-b5e6-d92120c3d6c8', 'display_name': '🟡 NEST 3'}, 'NEST4': {'access_token': 'kPfpX4EumVHdRR2icNjP', 'device_id': 'f62effa0-f458-11f0-b5e6-d92120c3d6c8', 'display_name': '🟡 NEST 4'}}

Dictionary mapping NEST identifiers to their technical credentials and display names. Each entry contains a 'device_id' (UUID) and an 'access_token' (Secret).

config.USER_TOKEN = None

Stores the active JWT session token after a successful login.

config.USERNAME = None

Stores the username of the currently authenticated user.

config.CURRENT_NEST = None

Stores the ID of the NEST device currently being controlled by the user.

