



**Proactive Producer and Processor Networks for  
Troodos Mountains Agriculture  
3PRO-TROODOS  
Prot. No: INTEGRATED/0609/061**

<b>Deliverable type</b>	<i>Report, Software</i>
<b>Work package number and name</b>	<i>WP6: Adapting agricultural and natural resource management to quality certification and climate change</i>
<b>Date</b>	<i>10 Dec 2020, Updated 23 Feb 2023.</i>
<b>Responsible Beneficiary</b>	<i>PA8</i>
<b>Authors</b>	<i>Georgios Michalis, Loizos Kanaris</i>
<b>Classification of Dissemination</b>	<i>Public</i>
<b>Short description</b>	<i>D6.1: Sensor Observation Service, database and irrigation decision support system software platform with report</i>

This document contains information, which is proprietary to the 3PRO-TROODOS consortium. Neither this document nor the information contained herein shall be used, duplicated, or communicated by any means to any third party, in whole or in parts, except with prior written consent of the 3PRO-TROODOS Coordinator.

The information in this document is provided as is and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.



**Ευρωπαϊκή Ένωση**  
Ευρωπαϊκό Ταμείο  
Περιφερειακής Ανάπτυξης



Κυπριακή Δημοκρατία



Διαρθρωτικά Ταμεία  
της Ευρωπαϊκής Ένωσης στην Κύπρο





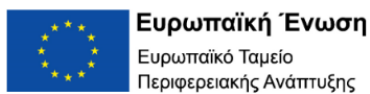
Version	Date	Changes	Partner
V1.0	07-12-2020	Initial Document	PA8, Sigint solutions Ltd
V2.0	23-02-2023	Complete Rework	PA8, Sigint solutions Ltd

*Disclaimer:* The information in this document is subject to change without notice. Company or product names mentioned in this document may be trademarks or registered trademarks of their respective companies.

## All rights reserved

The document is proprietary of the 3PRO-TROODOS Consortium Members. No copying or distributing in any form or by any means is allowed without the prior written agreement of the owner of the property rights.

This document reflects only the authors' view. The Research & Innovation Foundation is not liable for any use that may be made for the information contained herein.





## Contents

Introduction .....	5
Design of a Wireless Sensor Network (WSN) .....	5
Design of a Sensor Observation Service .....	5
Deployment of a WSN.....	5
Software & equipment.....	6
Eagle- PCB design software .....	6
Autodesk Fusion 360 – modelling software .....	7
Visual studio Code .....	7
ESP32 – Microcontroller .....	8
Modem - Waveshare 7600E .....	8
SDI-12 interface .....	8
Sensors .....	9
Sensor Group .....	9
Power System.....	11
System modules and measurement methodology .....	12
System modules .....	12
Logger General description .....	12
Sensor’s Module .....	13
Sensor Observation Service Version 1 .....	15
Sensor Observation Service Version 2 .....	18
Introduction .....	18
Terminology .....	19
Terminology related to the user management.....	19
Installing the platform .....	20
Setting up the 3pro platform .....	23
Initializing devices .....	37





The 3Pro dashboard.....	41
Creating farmer entities and assigning devices .....	54
Downloading Data.....	60
Description of the other utility applications.....	61
Algorithms and implementation.....	75
Introduction on post processing .....	75
Terminology .....	75
Device Properties .....	78
Device Attributes .....	80
3pro device requirements.....	81
Telemetry Transmission formats .....	89
Mathematical calculations / Theory .....	90
Description of implementation.....	94
Implementation .....	97
Conclusion.....	152





## Introduction

Specification and technical report of the topology of the sensor observation service and the wireless sensor network.

To support the 3Pro-Troodos project, SIGINT provides within this technical report the deliverable D.6.1 included into the proposal. More specifically the following activities have been performed:

### Design of a Wireless Sensor Network (WSN)

1. Select a group of sensors to collect environmental data
2. Cherry pick the microcontroller to provide interconnection and data collection between the sensors
3. Design a prototype circuit board (PCB) to support a turnkey solution of connectivity among the sensors and microcontroller
4. Investigate a solid and robust Network solution based on the area morphology
5. Interconnect the WSN system through software development
6. Design a stand-alone power solution for the system
7. Design a solid and weatherproof structure to provide equipment mounting
8. Deploy the software to interconnect each component and gather the data

### Design of a Sensor Observation Service

1. Software architecture of the system
2. Selection of communication architecture including the database scheme and the message format.
3. Design of the software security levels.
4. Implementation of the Beta version of the Software platform.
5. Creation of a pilot live website to test the integration with the sensor network.

**The software – as part of this deliverable – is available online, to access click [here](#)**

### Deployment of a WSN

1. Manufacture & assembly the PCBs
2. Construct and assembly the mounting system

The irrigation scheduling decision support code was developed in cooperation with Adriana Bruggeman and colleagues from The Cyprus Institute (see D6.3).

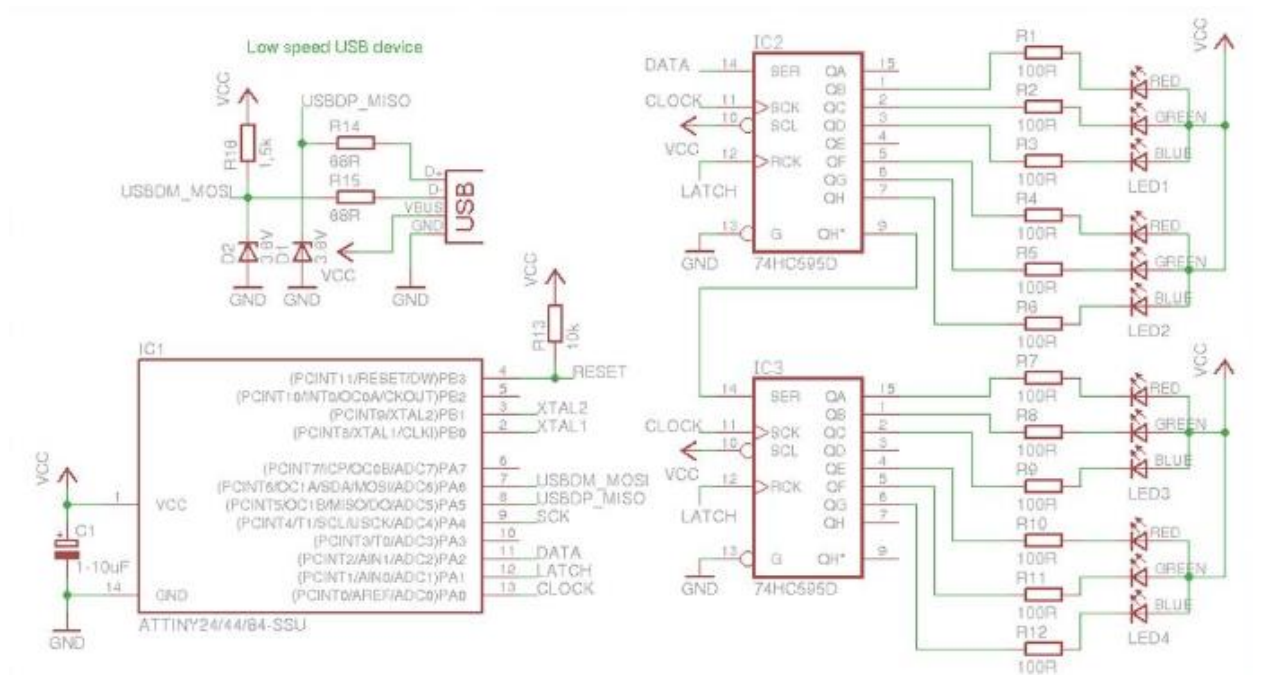
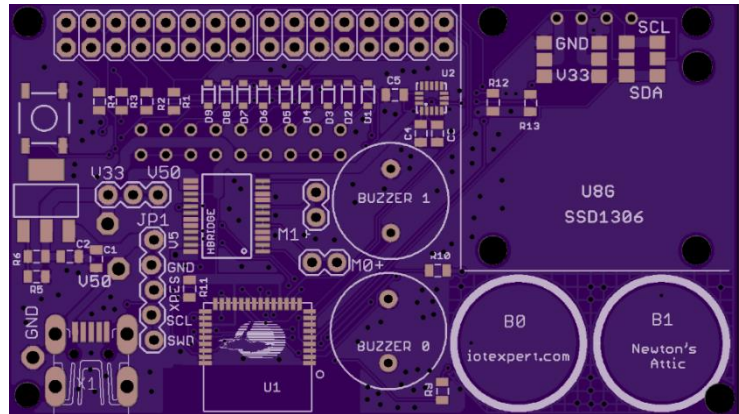


## Software & equipment

For the execution of the scope of work, SIGINT utilized the following equipment and software tools:

### Eagle- PCB design software

**Autodesk EAGLE** is an electronic design automation (EDA) software. Enabling printed circuit board (PCB) designers to seamlessly connect schematic diagrams, component placement, PCB routing, and comprehensive library content.

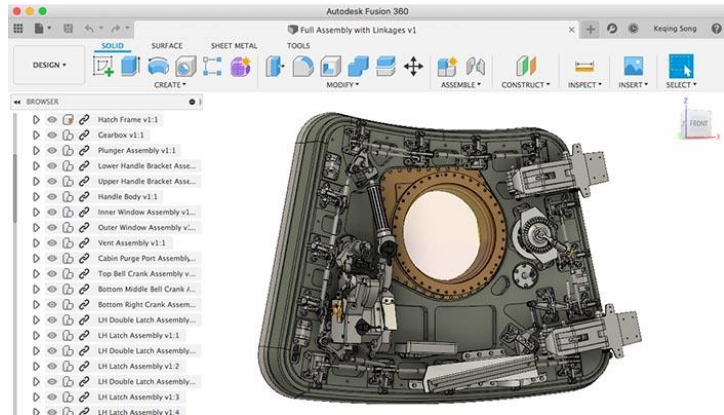




3PRO-TROODOS

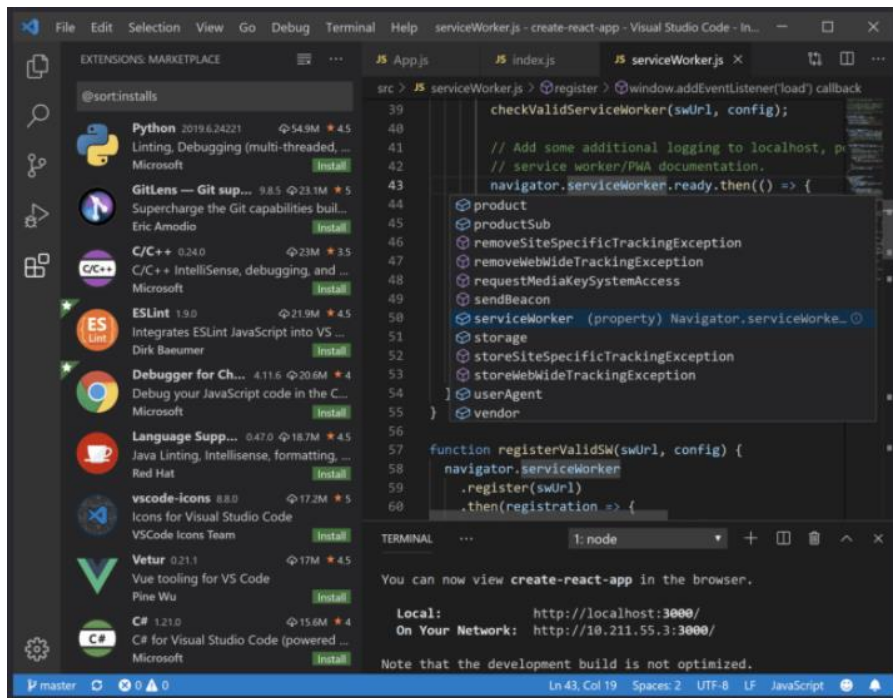
## Autodesk Fusion 360 – modelling software

**Fusion 360** is solid modelling software that allows you to design models and assemblies in 3 dimensions. Provides the breadth of tools to tackle the most complex problems, and the depth to finish critical detail work



## Visual studio Code

**Visual studio code** is a Free integrated development environment build on open source that allows you to code in various programming languages. With the help of various plugins, it can be extended indefinitely. With the help with the platformio plugin it allows to create and upload firmware in a variety of microprocessor family, including ESP32 & ESP8266 families.



## ESP32 – Microcontroller

ESP32 is a series of low-power system on a chip microcontroller with integrated Wi-Fi 802.11 b/g/n and Bluetooth. It is equipped with 12bit ADC, I2C bus, SPI and a couple of GPIOs. The microcontroller is the heart of the system as it gathers all the information from the sensors and dispatch it to databases for further evaluation. Due to its low power consumption it is suitable for stand-alone operation using batteries as feed in source.



## Modem - Waveshare 7600E

Sim7600E, is a multi-band LTE-FDD/LTE-DD/HSPA+ GSM/GPRS/EDGE module. The Sim7600 series integrates multiple satellite high accuracy positioning systems with multiple build-in network protocols, and software functions including abundant interfaces such as UART, USB, I2C, GPIO which is suitable for main IoT applications such as telematics, surveillance devices, remote sensor monitoring etc. Due to the UART interface, it can interface and communicate directly with any microcontroller that supports Serial communication.



## SDI-12 interface.

The TBS01A SDI-12 to UART module is a bi-directional interface for the conversion of commands and data into SDI-12 format and vice versa. The module is Plug and Play, targeting cost sensitive data logging applications. It offers low current consumption, small footprint and easy integration into products which require a SDI-12 interface.





## Sensors

A multitude of sensors have been carefully chosen for the projects.

### Sensor Group

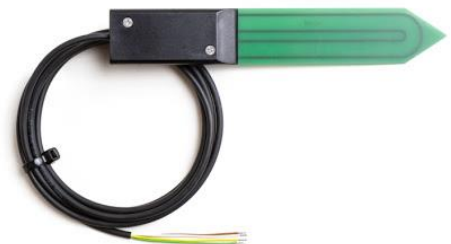
#### VEML7700

16bit dynamic sensor range for ambient light detection from 0 lux to 120000 lux.



#### SMT100

High accuracy soil moisture and soil temperature sensor. Combines the advantages of the low-cost FDR sensor systems with the accuracy of a TDR system. Like a TDR, it measures the travel time of a signal to determine the dielectric constant of the soil. And like a FDR, it converts this dielectric constant into an easy to measure frequency. But unlike an FDR it is not based on a capacitor but utilizes a ring oscillator to transform the signal's travel time into the measure frequency. The resulting frequency (>100 MHz) is high enough to operate well in clayey soils



#### Atmos 14.

Meter group atmos 14. A 4-in 1 temperature/ Relative humidity / Barometric pressure, / Vapor pressure sensor.



**Atmos 22**

Ultrasonic wind sensor. Measures air temperature, air speed, air direction. Its greatest advantage is that it has no moving parts. A lower speed threshold makes it well suited for measuring wind within plant canopies (if needed).



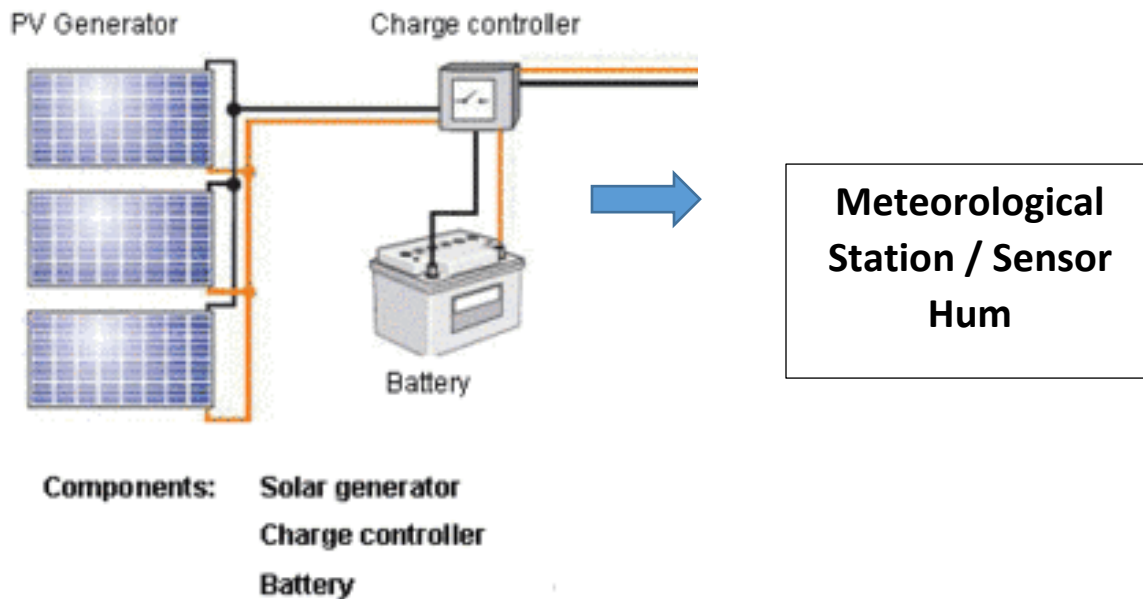
**Davis precipitation Sensor**

Measures rainfall (water level)



## Power System.

Since the system will be installed on remote fields where power is not available, a clean energy production system was installed to provide sufficient power for the system. The systems consist of a solar panel to convert the solar energy to electrical, batteries to store the energy, and a charger regulator to control the power in and out of the batteries. Suitable batteries and photovoltaic cells have been selected for the 2 sensor hubs, so they can have suitable autonomy even on pitch black conditions. Depending on the measurement frequency a minimum of 1-week standby time is available.



## System modules and measurement methodology

### System modules

#### Logger General description

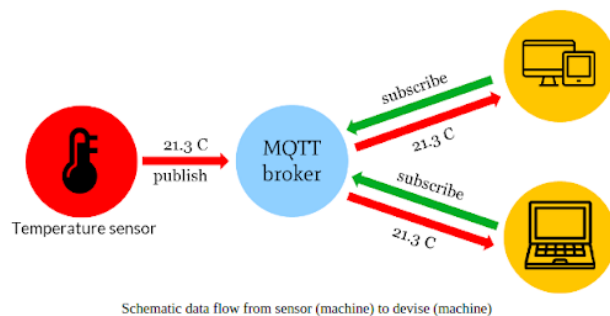
The system consists of a waterproof (IP65) enclosure with integrated environmental sensors , integrated batteries, and photovoltaic cell for easy charging.

It has enough ports to support 1 to 8 soil moisture and temperature sensors, as well the capability to support precipitation and water flow meter sensors. (reducing the active soil sensors to 6), as well 2 SDI protocol sensors. (more can be connected using an expansion board). It can be connected with an internal wifi antenna (in case wifi is available near the installation point), and/or an external antenna for 3G/4G cellular purposes.



Internally all the sensors are connected on the microcontroller ESP32 from Espressif.

An intelligent application has been written in order to connect the microprocessors to Sensor observation Service as well to read sensor's values using various protocols and techniques such as I2C bus, ADC (Analogue to Digital Conversion) and UART communication.



For the purpose of data acquisition and visualization, the MQTT publish-subscribe based protocol was used. Several topics were created publishing messages regarding the environmental measurements as well as system health monitoring info, such as battery voltage, date & time and RSSI. To catch up accurate date and time, the Network Time Protocol was used (NTP). The accurate data and time are critical for the proper synchronization of the sensor's

transmitted parameters. With that being said, only one synchronization event is strictly necessary. The system will then keep the time in its internal clock, with the NTP synchronization acting only to correct any small time drifts that make occur occasionally.

The link between the publish/subscribe clients is achieved by the MQTT broker. The broker is at the heart of any publish/subscribe protocol. Depending on the implementation, a broker can handle up to



thousands of concurrently connected MQTT clients. The broker is responsible for receiving all messages, filtering the messages, determining who is subscribed to each message, and sending the message to these subscribed clients. The broker also holds the sessions of all persisted clients, including subscriptions and missed messages.

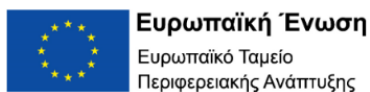
The system wakes up every 10 minutes and takes environmental data. Then it formats the data creating a mqtt payload. Depending the selected power settings, it will then store the data for later transmission, or will transmit the data directly by waking up the cellular interface. It will then go again to low power mode by disabling the modem and putting the microcontroller in low power mode. Even on low power mode the pulse channers are still monitored continuously using a separate ultra low power (ULP) processing unit.

### Sensor's Module

The sensor's module is using a PCB board designed, and assembled by Sigint Solutions Ltd. This board is powered up with Li-On researchable batteries and for the reason of non-energy redundancy the system is set up into deep sleep mode waking up at predefined time intervals (controlled by the NTP). The

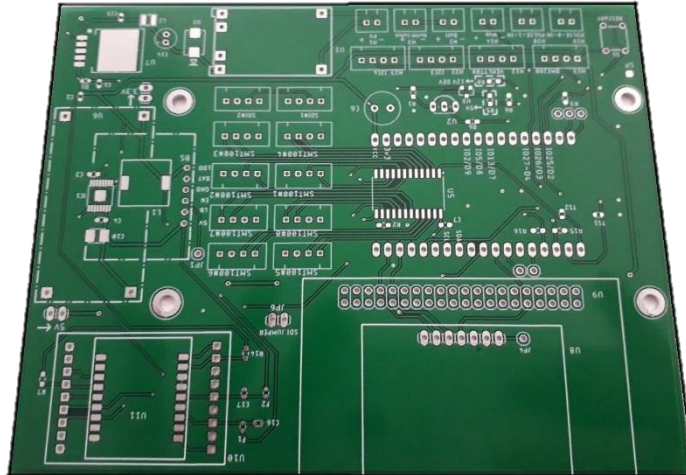
The need of an integrated and solid system led into designing, manufacturing and assembly the PCB that supports the ESP32 and provides connection for a variety of sensors. The PCB is presented below.

The design of the logger is not included in the deliverables. It has been design to simplify the data acquisition, but the sensor observation platform can use 3<sup>rd</sup> party loggers. For example most of the project has been completed using the Truebner loggers.

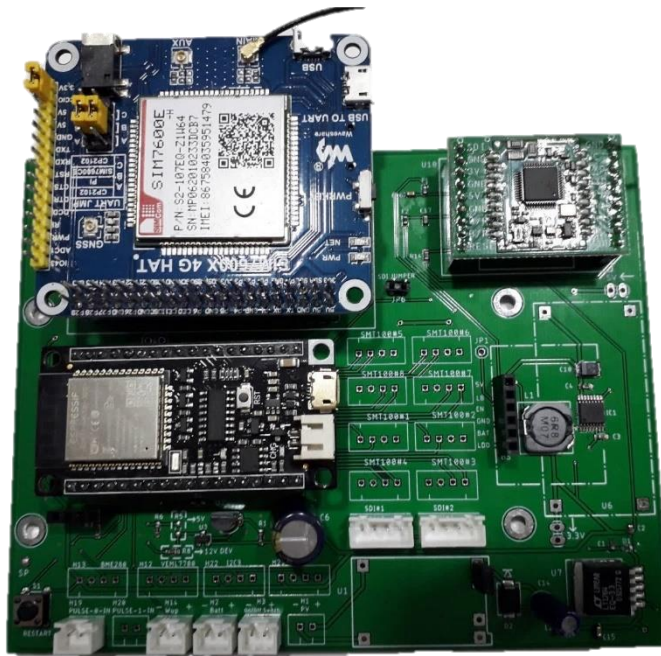




3PRO-TROODOS



Bare PCB - Front Side



PCB – Assembled



## Sensor Observation Service Version 1

The sensor observation service consists of a various software components.

The software platform resides in a dedicated webhosting server. The server hosts the services of the software components:

- The website.
- The Mqtt interface which stores the data in a temporary database.
- A cron job script that preprocesses the data from the temporary database and places them on the permanent database and on the corresponding user. A cron job is just a server task that runs periodically.
- Another cron job that periodically examines the latest stored data for threshold violations and sends notification to the user.

On the interface itself in high level there are multiple user levels. On the top level is the administrator that can see all the data

UTC	Hub	Sensor Id	Value
2020-11-04 01:04:58	BC:DD:C2:CF:A4:EC	WiFi Connected SSID	2.00
2020-11-04 01:04:58	BC:DD:C2:CF:A4:EC	ESP FW version	1.02
2020-11-04 01:04:58	BC:DD:C2:CF:A4:EC	ESP	8.00
2020-11-04 01:04:58	BC:DD:C2:CF:A4:EC	Battery	4.29
2020-11-04 01:04:58	BC:DD:C2:CF:A4:EC	WiFi	-77.00
2020-11-04 01:04:58	BC:DD:C2:CF:A4:EC	BME280A	90.85
2020-11-04 01:04:58	BC:DD:C2:CF:A4:EC	BME280B	1079.45
2020-11-04 01:04:58	BC:DD:C2:CF:A4:EC	BME280C	890.14
2020-11-04 01:04:58	BC:DD:C2:CF:A4:EC	BME280D	9.34
2020-11-04 01:04:58	BC:DD:C2:CF:A4:EC	VEML7700	0.00
2020-11-04 01:04:58	BC:DD:C2:CF:A4:EC	SMT100A1-2	21.33
2020-11-04 01:04:58	BC:DD:C2:CF:A4:EC	SMT100A1-1	4.74
2020-11-04 01:04:58	BC:DD:C2:CF:A4:EC	SMT100B2-2	20.46
2020-11-04 01:04:58	BC:DD:C2:CF:A4:EC	SMT100B2-1	2.95
2020-11-04 01:04:58	BC:DD:C2:CF:A4:EC	SMT100B3-2	-21.23
2020-11-04 01:04:58	BC:DD:C2:CF:A4:EC	SMT100B3-1	0.00
2020-11-04 04:52:57	CB:28:96:9A:5B:AB	WiFi Connected SSID	0.00

One level down there is the “User”, which can have a Group of the sensor hub that he owns (for example a user that wants to monitor the condition in 2 separate locations can have a group with the sensor hubs on each location).



3PRO-TROODOS

[← Back To Agent](#)

Testing Group

Sim card / Ref:Id : .

Longitude : .

Latitude : .

Hub/s : 2      Sensor/s : 18

[👁](#) [✎](#) [🗑](#) [↺](#) [👤](#)

Agros General

Sim card / Ref:Id : .

Longitude : .

Latitude : .

Hub/s : 3      Sensor/s : 37

[👁](#) [✎](#) [🗑](#) [↺](#) [👤](#)

[← Back To Gateway Group](#)

2C:F4:32:50:0D:34

Sensor Hub ID : SH0018

MAC ID : 2cf43250d34

Number of Sensor : 9 [👁](#) [✎](#) [🗑](#) [🔋](#)

BC:DD:C2:CF:A4:EC

Sensor Hub ID : SH0017

MAC ID : BCDDC2CFA4EC

Number of Sensor : 13 [👁](#) [✎](#) [🗑](#) [🔋](#)

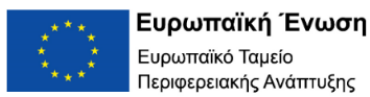
C8:2B:96:9A:50:A8

Sensor Hub ID : SH0016

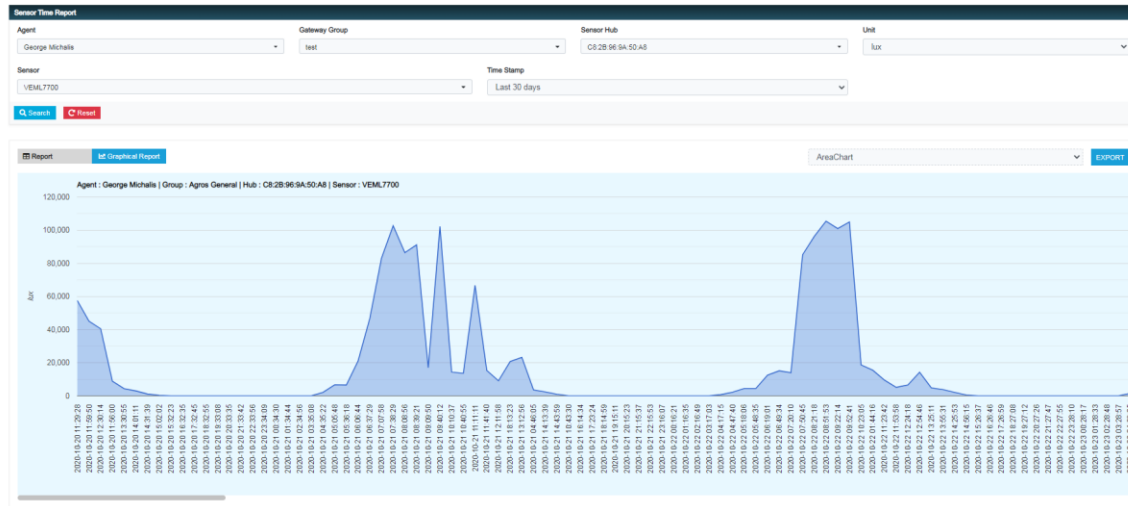
MAC ID : C82B969A50A8

Number of Sensor : 15 [👁](#) [✎](#) [🗑](#) [🔋](#)

The user can plot at any time any of the data collected from the any of the sensors of the hubs, at the desired time range.

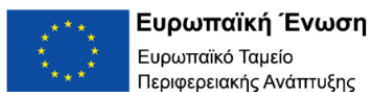






There is also a way to set thresholds that monitor critical values and provide notifications to the user when these criteria have been fulfilled.

Internally the Mqtt payloads from the various sensor hubs are being collected and placed on temporary location in a database. Every minute or so, a routine scans the newly acquired data and processes them by matching them with the user registered devices, checking the criteria of set thresholds set for that user (as mentioned before), etc.





## Sensor Observation Service Version 2

During the initial stages of the project, we made the decision to develop the IoT platform in-house, believing that we had the necessary expertise and resources to build a customized solution that would meet our specific needs. However, as the project progressed, it became clear that developing a fully-featured platform with all the features that we wanted from scratch would be more time-consuming and resource-intensive than we had anticipated. The initial platform was perfectly fine, but it had some performance issues and it wasn't very scalable, and while it would be ok with a few sensors that were going to be used for the project, it wouldn't probably be ok for a wider use.

After careful consideration, we decided to explore other options, including using an open source platform that could be customized to fit our needs. This approach offered several advantages, including:

1. **Reduced development time and cost:** By leveraging an existing open source platform, we were able to save time and resources on development, as we did not have to build the platform from the ground up.
2. **Access to a wider range of features:** Open source platforms are typically developed and maintained by a community of developers who contribute to the platform's features and functionality. This meant that we had access to a wider range of features and capabilities than we would have been able to develop on our own.
3. **Ongoing support and maintenance:** By using an open source platform, we had access to ongoing support and maintenance from the community of developers who were contributing to the platform. This ensured that the platform would remain up-to-date and functional over time.

So, this version of the platform uses an open source version of a platform (thingsboard) as a base, which offers great levels of configurability, scalability and security. The rest of the 3pro functionality has been built on top of that platform using extensive rules, custom dashboards, and widgets, as well as a few modifications in the source code.

### Introduction

To perform the necessary research a number of wireless sensors had to be used to take various measurements of the state of the soil as well as the environmental conditions in an area. To keep all these data easily accessible, a sensor observation IoT platform had to be developed.

On the Sensor Platform, device entities of type "3Pro Device" can be added. This device entity will record the sensor data that is transmitted from the device, and will post process the data accordingly to give some predictions. But for the data to be post processed successfully some specific parameters about a device must be set correctly.





For more information about the platform and well as installation instructions refer to the relevant repository of the platform.

The following document contains instructions about how to setup the 3d pro platform, as well as explanations, terminology, etc.

There are also several repositories with utility tools that can be used to simplify the device configuration and other stuff.

## Terminology

The platform follows a very complicated paradigm for user and device management so a lot of explanation is needed.

### Terminology related to the user management

- Users
  - System Administrator
  - Tenant administrator
  - Assets
  - Devices
  - Telemetry
  - Entities
  - Dashboards
  - Customers
  - Customer Users and Customer Administrators.
1. A user is a user of the platform. It can be a System administrator, tenant administrator, A customer User Or A customer administrator.
  2. System Administrator is a user which is the owner of the thingsboard installation. He has access to all system resources, and the tenant administrators.
  3. The Tenant administrator

The Tenant administrator is a user that has access to tenants (hereby called customers), and can view manage and create dashboards, assets, customers, sub users, add devices etc.

To simplify, assuming that there is not another sub customer at play, the Tenant administrator on his own, can be thought as an owner of devices. The device Assets. Assets is just a way to group devices that belong to specific locations. There is no limit on how many assets and devices



can be nested. For example, a building as a whole can have X apartments with Y devices each, and the full building can have additional Devices. How the relations are designed will be described later in this document.

4. Devices. Devices are devices entities that contain sensors, which send telemetry data. For example Argus hub is a device that contains X amount of sensors and transmits Y amount of telemetry. In thingsboard sensors don't matter. They don't have any representation and it's only the telemetry that matters.

The telemetry entries are called attributes. There 2 type of attributes. The Server attributes, and the Telemetry attributes. More information about them will be described later in this document.

5. Telemetry is the measurements that are transmitted from the devices.
6. Dashboards

A dashboard is an interface that displays information about devices.

A dashboard can be static or dynamic, get entries from groups etc. List all devices belonging in a user, etc. At the same time a dashboard can be shared with multiple users, assuming that they have been configured correctly to pull the information about the devices of individual users. Dashboards can have also separate states. A dashboard state is a dashboard configuration which can be configured to display different things depending the selected device or something similar. more information about the use of dashboard states will be referenced on the tutorial about how to set a dashboard for multiple device types.

7. Customers. A customer (or tenant) is an entity that (like the tenant administrator), can own devices, assets, dashboards etc.

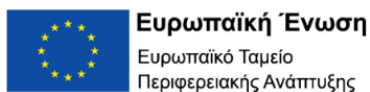
By default a customer is just a business entity. This means that it can own devices assets etc, but by default it doesn't have any users. The Customer entity need to have a "customer user", a "customer administrator" or both.

The customer can belong To the Tenant Administrator, and the customer in turn can have extra subcustomers, etc

## Installing the platform

### Prerequisites

This guide describes how to install ThingsBoard on Ubuntu 18.04 LTS / Ubuntu 20.04 LTS. Hardware requirements depend on chosen database and amount of devices connected to the system. To run ThingsBoard and PostgreSQL on a single machine you will need at least 1Gb of RAM.





### *Step 1. Install Java 11 (OpenJDK)*

ThingsBoard service is running on Java 11. Follow this instructions to install OpenJDK 11:

```
sudo apt update
sudo apt install openjdk-11-jdk
```

Please don't forget to configure your operating system to use OpenJDK 11 by default. You can configure which version is the default using the following command:

```
sudo update-alternatives --config java
```

You can check the installation using the following command:

```
java -version
```

Expected command output is:

```
openjdk version "11.0.xx"
OpenJDK Runtime Environment (...)
OpenJDK 64-Bit Server VM (build ...)
```

### *Step 2. ThingsBoard service installation*

Download installation package.

```
wget https://github.com/thingsboard/thingsboard/releases/download/v3.4.1/thin
gsboard-3.4.1.deb
```

Install ThingsBoard as a service

```
sudo dpkg -i thingsboard-3.4.1.deb
```

### *PostgreSQL Installation*

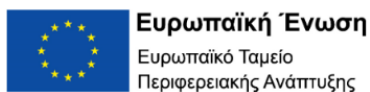
Instructions listed below will help you to install PostgreSQL.

```
# install **wget** if not already installed:
sudo apt install -y wget
```

```
# import the repository signing key:
```

```
wget --quiet -O - https://www.postgresql.org/media/keys/ACCC4CF8.asc | sudo a
pt-key add -
```

```
# add repository contents to your system:
```





```
RELEASE=$(lsb_release -cs)
echo "deb http://apt.postgresql.org/pub/repos/apt/ ${RELEASE}"-pgdg main | su
do tee /etc/apt/sources.list.d/pgdg.list
```

```
# install and launch the postgresql service:
sudo apt update
sudo apt -y install postgresql-12
sudo service postgresql start
```

Once PostgreSQL is installed you may want to create a new user or set the password for the the main user. The instructions below will help to set the password for main postgresql user

```
sudo su - postgres
psql
\password
\q
```

Then, press “Ctrl+D” to return to main user console and connect to the database to create thingsboard DB:

```
psql -U postgres -d postgres -h 127.0.0.1 -W
CREATE DATABASE thingsboard;
\q
```

ThingsBoard Configuration Edit ThingsBoard configuration file

```
sudo nano /etc/thingsboard/conf/thingsboard.conf
```

Add the following lines to the configuration file. Don’t forget to replace **PUT\_YOUR\_POSTGRESQL\_PASSWORD\_HERE** with your real postgres user password:

```
# DB Configuration
export DATABASE_TS_TYPE=sql
export SPRING_DATASOURCE_URL=jdbc:postgresql://localhost:5432/thingsboard
export SPRING_DATASOURCE_USERNAME=postgres
export SPRING_DATASOURCE_PASSWORD=PUT_YOUR_POSTGRESQL_PASSWORD_HERE
# Specify partitioning size for timestamp key-value storage. Allowed values:
DAYS, MONTHS, YEARS, INDEFINITE.
export SQL_POSTGRES_TS_KV_PARTITIONING=MONTHS
```

*Run installation script*

Once ThingsBoard service is installed and DB configuration is updated, you can execute the following script:





```
sudo /usr/share/thingsboard/bin/install/install.sh
```

*Start ThingsBoard service*

Execute the following command to start ThingsBoard:

```
sudo service thingsboard start
```

You then access the platform on the following url:

```
http://localhost:8080
```

The following default credentials are available if you have specified `-loadDemo` during execution of the installation script:

**System Administrator:** `sysadmin@thingsboard.org /sysadmin`

**Tenant Administrator:** `tenant@thingsboard.org / tenant`

**Customer User:** `customer@thingsboard.org / customer`

You can always change passwords for each account in account profile page.

## Setting up the 3pro platform

For the platform to work correctly, various dashboards and settings must be imported.

### *Importing Rule Chains*

Press **Rule Chains** and press the **+** symbol. Then press on the **Import rule chain**



Ευρωπαϊκή Ένωση  
Ευρωπαϊκό Ταμείο  
Περιφερειακής Ανάπτυξης

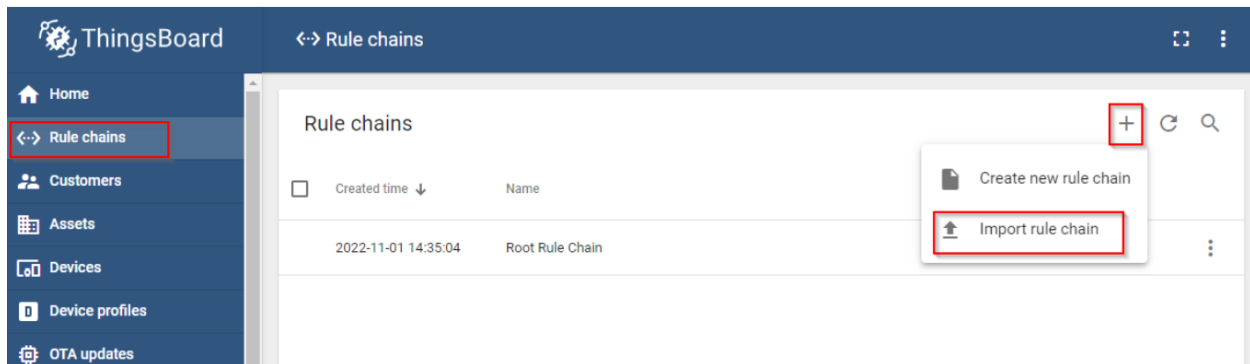


Κυπριακή Δημοκρατία

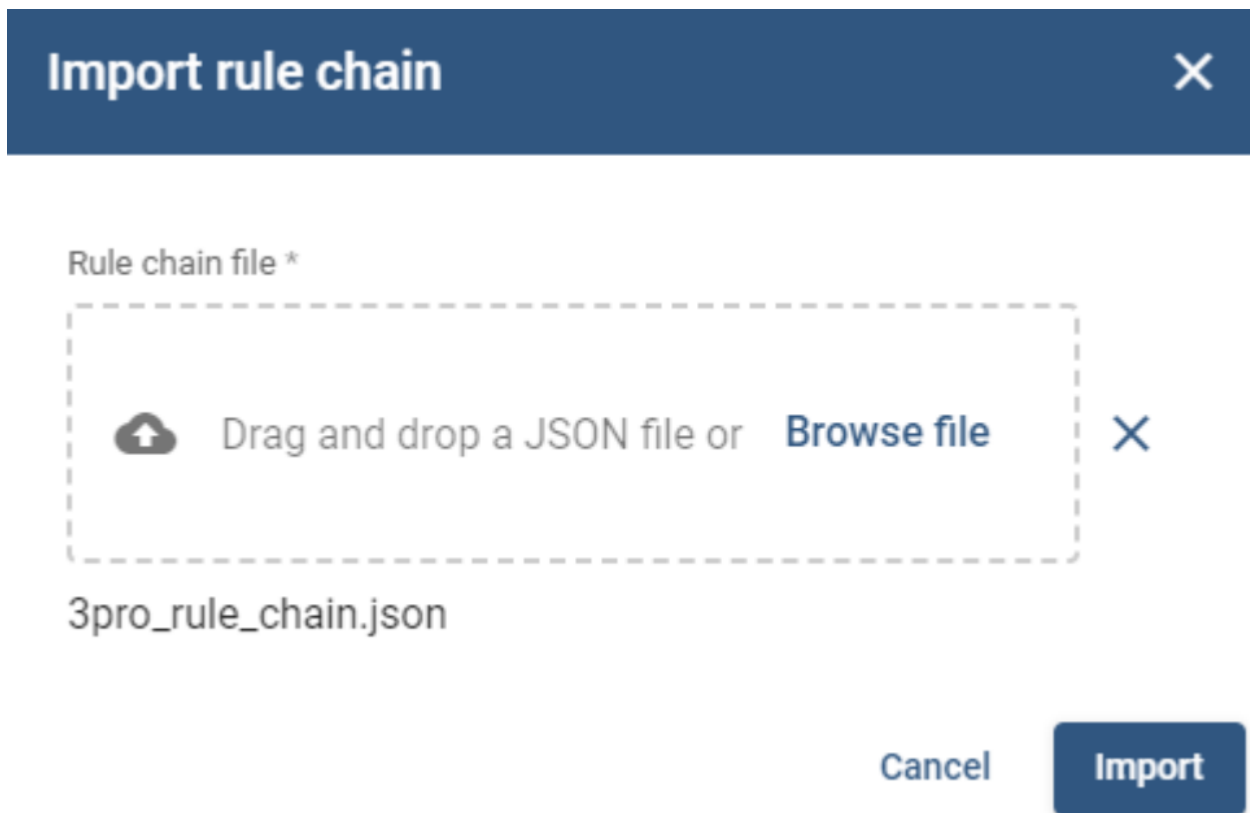


Διαρθρωτικά Ταμεία  
της Ευρωπαϊκής Ένωσης στην Κύπρο





import the file `3pro\_rule\_chain.json` which can be found under `Rule Engine` folder on the repository.



The rule chain `3pro` should appear on the list





<input type="checkbox"/>	Created time ↓	Name	Root	
<input type="checkbox"/>	2022-11-01 15:03:38	3Pro Argus Hub Meteo	<input type="checkbox"/>	⋮
	2022-11-01 14:35:04	Root Rule Chain	<input checked="" type="checkbox"/>	⋮

Press on top of the rule chain name and on the resulting page press the button `copy rule chain id`

Rule chains

### 3Pro Argus Hub Meteo

Rule chain details

Details Attributes Latest telemetry Alarms Events Relations

Open rule chain Export rule chain Make rule chain root Delete rule chain

**Copy rule chain id**

Name  
3Pro Argus Hub Meteo

Debug mode

Description

This will copy the rule chain unique id which will look something like that

`99f7cc50-59e5-11ed-a89c-13427b0e0a64`

Additional information about how the rule chain works can be found at the end of the document on the section about algorithms and implementations.

### [Importing device profiles](#)

Navigate on the `Device profiles` folder, and edit the `3pro\_Device.json` via a text editor.

Replace the `id` with the rule chain id that was copied on the previous step.

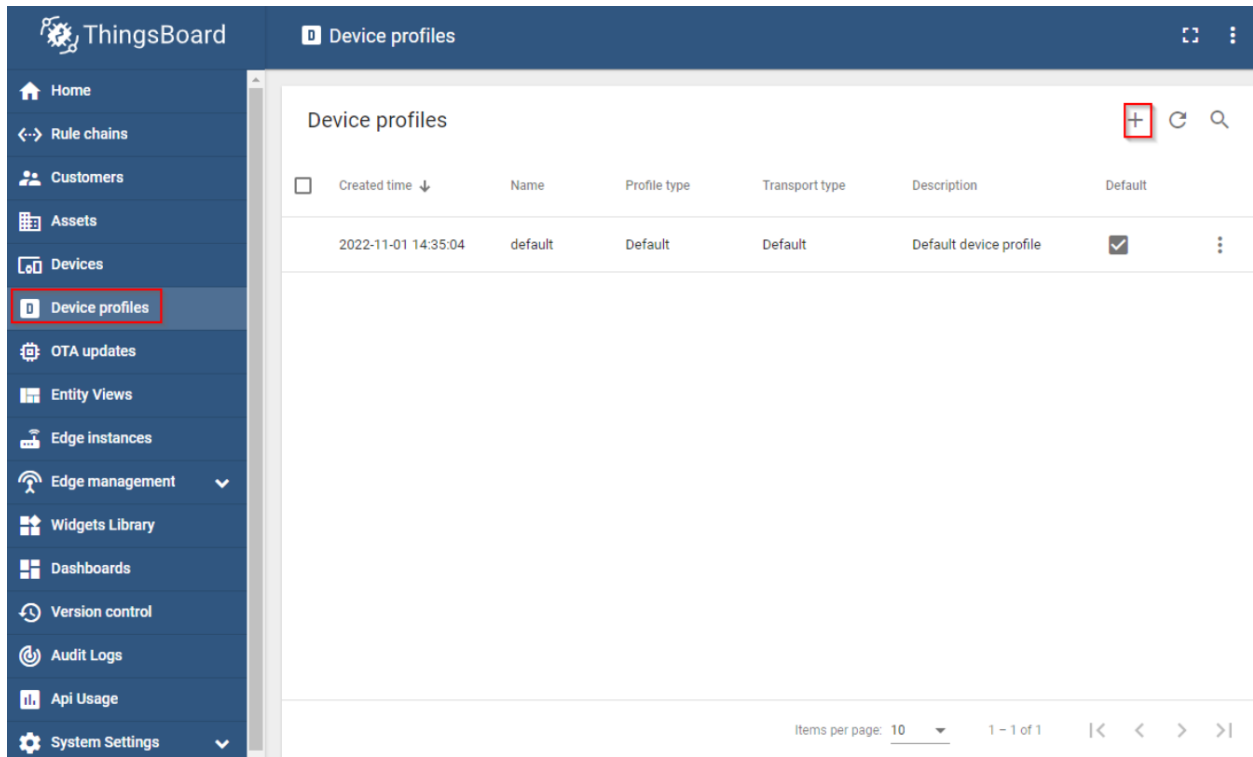




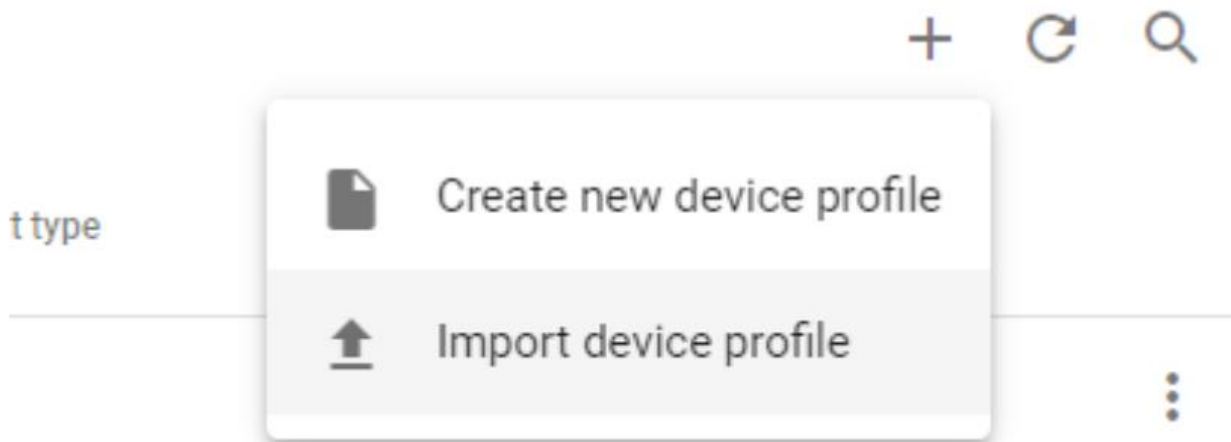
```
{
  "name": "3Pro Device",
  "description": "",
  "image": null,
  "type": "DEFAULT",
  "transportType": "DEFAULT",
  "provisionType": "DISABLED",
  "defaultRuleChainId": {
    "entityType": "RULE_CHAIN",
    "id": "99f7cc50-59e5-11ed-a89c-13427b0e0a64"
  },
  "defaultDashboardId": null,
  "defaultQueueName": "",
  "profileData": {
    "configuration": {
      "type": "DEFAULT"
    }
  }
}
```

Save and close the document.

On the platform, press on the `Device Profiles` and press the `+` symbol.



Press the button `Import Device Profile`



on the new tab that will open drop the `3pro\_Device.json`.





and press import .

## Import device profile ✕

Device profile file \*

 Drag and drop a JSON file or [Browse file](#) ✕

3pro\_argus\_hub\_meteo.json

[Cancel](#) [Import](#)

The imported device profile should appear in the device profile list.

Device profiles + ↻ 🔍

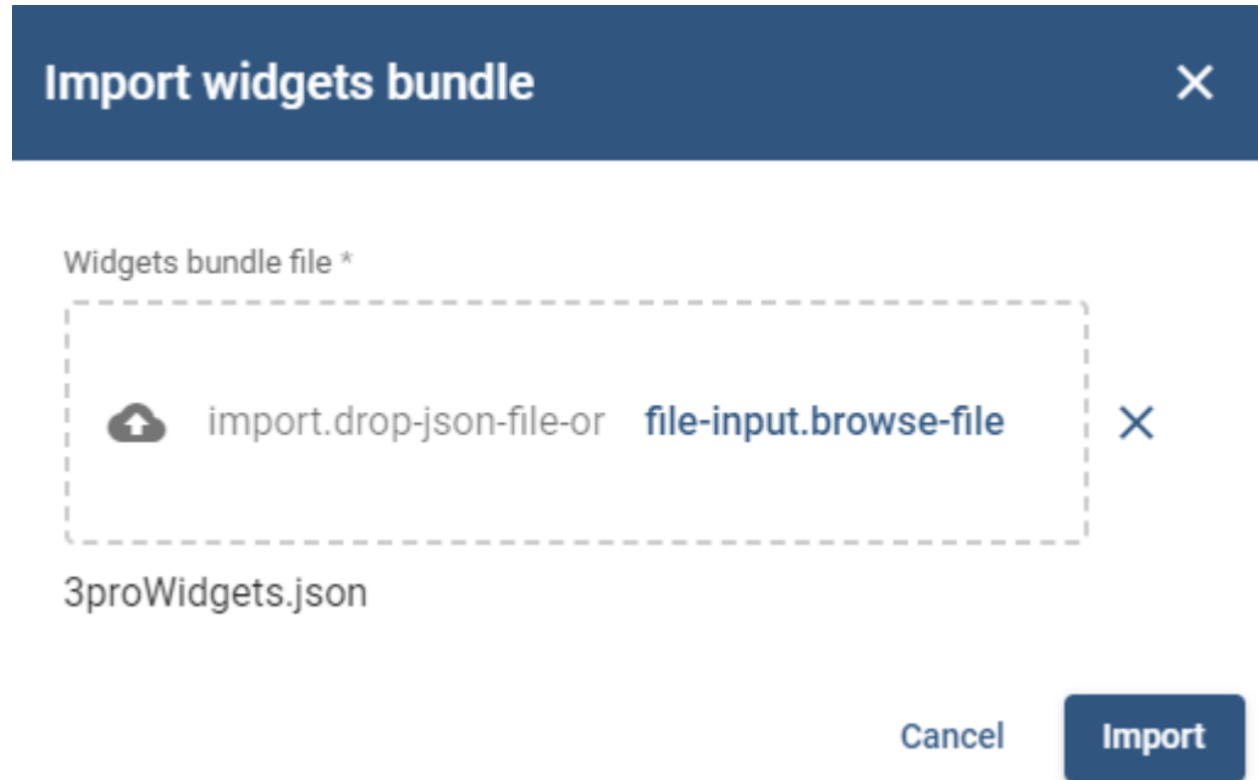
<input type="checkbox"/>	Created time ↓	Name	Profile type	Transport type	Description	Default	
<input type="checkbox"/>	2022-11-01 15:12:30	<b>3Pro Device</b>	Default	Default		<input type="checkbox"/>	⋮
	2022-11-01 14:35:04	default	Default	Default	Default device profile	<input checked="" type="checkbox"/>	⋮





## Importing Widgets

Press on the `Widgets library` button, then on the `+` symbol, and finally on the `Import widgets bundle`  
On the new page select the file `3proWidgets.json` which can be found in the `widgets` folder in this repository.



and press `import`.

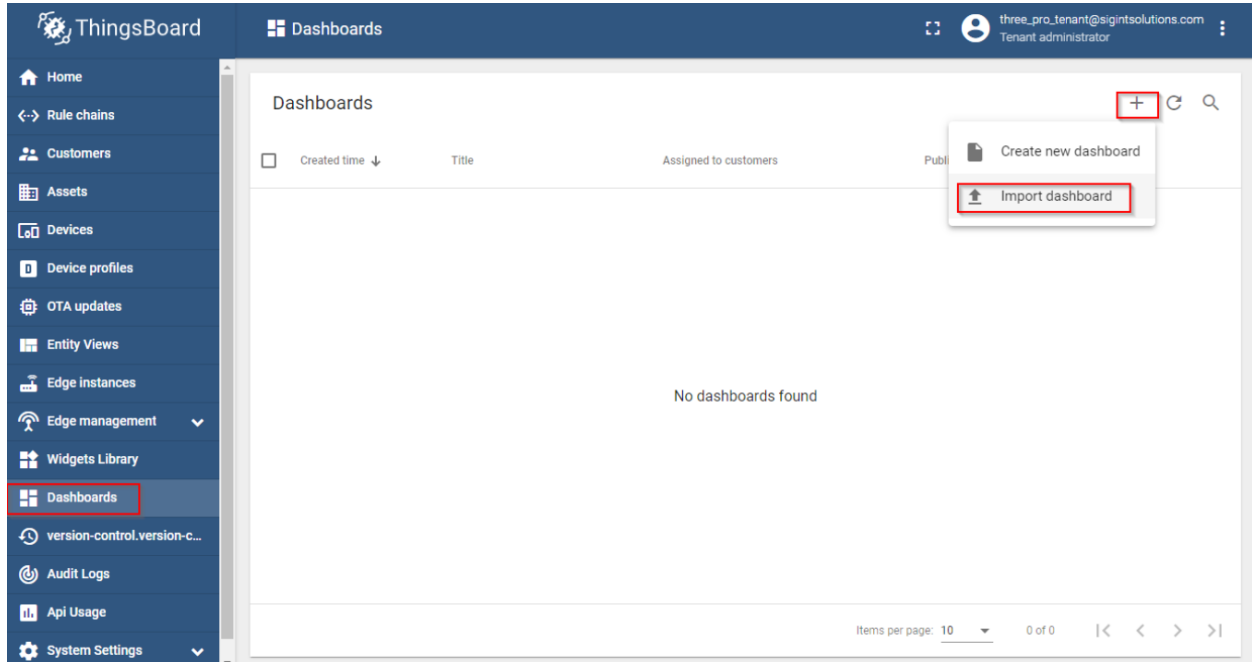
The widgets should appear on the widget packages list

2022-10-27 15:47:58	Control widgets	<input checked="" type="checkbox"/>			
<input type="checkbox"/>	2022-11-03 10:19:10 Custom	<input type="checkbox"/>			
2022-10-27 15:47:58	Date	<input checked="" type="checkbox"/>			



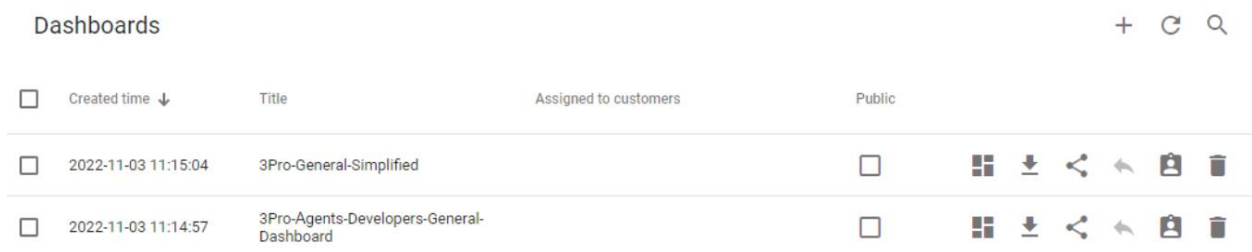
## Importing dashboards

To import the dashboard, press on the `Dashboards` button on the left, then press the `+` symbol, and finally on the `import dashboard`



Import the files `3Pro-Agents-Developers-General-Dashboard.json` and the file `3Pro-General-Simplified.json`.

The dashboards should appear on the Dashboards list.



To open a dashboard you can press on the following symbol as indicated on the picture





Dashboards					+ ↻ 🔍	
<input type="checkbox"/>	Created time ↓	Title	Assigned to customers	Public		
<input type="checkbox"/>	2022-11-03 11:15:04	3Pro-General-Simplified		<input type="checkbox"/>		
<input type="checkbox"/>	2022-11-03 11:14:57	3Pro-Agents-Developers-General-Dashboard		<input type="checkbox"/>		

though at this point there are no devices installed

### Entities

Entity name ↑	Entity type	Label	Type
No entities found			

0 of 0 |< < > >|

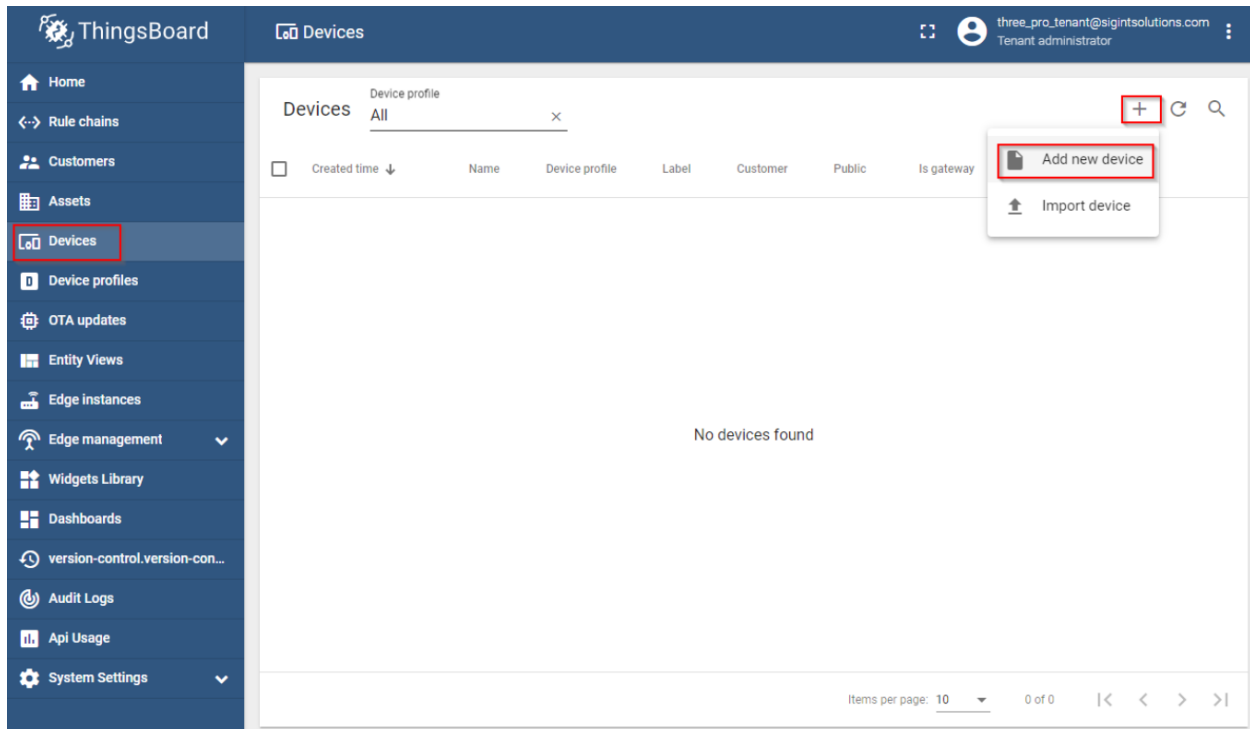
### Device Locations

Leaflet | © OpenStreetMap contributors

*Adding devices.*

To add a 3pro device, press on the `Devices` button, then on the `+` symbol, and finally on the `add new device`





Name the device with a suitable unique name, and select the correct device profile which should be `3Pro Device`.





### Add new device ? ×

**1** Device details **2** Credentials Optional **3** Customer Optional

Name \*  
3Pro test device

Label

Select existing device profile Device profile \* 3Pro Device ×

Create new device profile

Is gateway

Description

Press `add` to finish the device installation.

On the new device on the list press on the device name.

A new page will open with some details about the device.

2 very important tabs in this new page is the `Attributes` `(1)` as well as the `Latest Telemetry` `(2)`



## 3Pro test device

Device details 1 2

< Details **Attributes** Latest telemetry Alarms Events Relations

Client attributes Entity attributes scope  
Client attributes

<input type="checkbox"/>	Last update time	Key ↑	Value
--------------------------	------------------	-------	-------

`Attributes` are device attributes that can be used from the server for postprocessing the transmitted data, and `Latest Telemetry` is displaying the latest telemetry that was received from the device. Information about the device configuration and well the transmission method will be described later in this document.

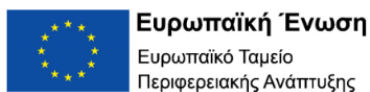
### *Connecting devices*

On the new device on the list press on the device name.

A new page will open with some details about the device.

The most important button at this point is the `Copy access token`. This will copy on the clipboard a token in the following form `NWYs8rdilrTq8JZsWB31`.

This will be used by the device to transmit the data and will be referred to later as \$ACCESS\_TOKEN





The screenshot displays the '3Pro test device' interface. On the left, a sidebar lists devices with a 'Created time' column. A red box highlights the timestamp '2022-11-03 11:48:39'. The main panel shows the device details for '3Pro test device', including buttons for 'Open details page', 'Make device public', 'Copy device Id', and 'Copy access token'. The 'Copy access token' button is highlighted with a red box.

### Connectivity Options

The device can be connected either via `Mqtt` or via `Http` requests.

MQTT is a lightweight publish-subscribe messaging protocol which probably makes it the most suitable for various IoT devices

### MQTT Connect

To connect the application needs to send MQTT CONNECT message with username that contains \$ACCESS\_TOKEN.

Possible return codes, and their reasons during connect sequence:





```
0x00 Connected - Successfully connected to ThingsBoard MQTT server.  
0x04 Connection Refused, bad username or password - Username is empty.  
0x05 Connection Refused, not authorized - Username contains invalid  
$ACCESS_TOKEN.
```

In order to publish telemetry data to ThingsBoard server node, send PUBLISH message to the following topic:

**v1/devices/me/telemetry**

[Http connect](#)

HTTP is a general-purpose network protocol that can be used in IoT applications. You can find more information about HTTP here. HTTP protocol is TCP based and uses request-response model.

In order to publish telemetry data to ThingsBoard server node, send POST request to the following URL:

**[http\(s\)://host:port/api/v1/\\$ACCESS\\_TOKEN/telemetry](http(s)://host:port/api/v1/$ACCESS_TOKEN/telemetry)**

*Device format*

The simplest supported data formats are:

For a single telemetry point:

```
{  
  "key1": "value1",  
}
```

For multiple telemetry points

```
{  
  "key1": "value1",  
  "key2": "value2"  
}
```

In this case the server-side timestamp will be assigned to the uploaded telemetry. For the calculations to be performed correctly, all the measurements for each measurement cycle of the device must be transmitted at the same time, so all the transmissions have the same timestamp.



In case this is not possible, or in case that historical data must be transmitted, the following format is also possible.

```
{
  "ts":1451649600512,
  "values": {
    "key1":"value1",
    "key2":"value2"
  }
}
```

In the example above, we assume that “1451649600512” is a unix timestamp with milliseconds precision. For example, the value ‘1451649600512’ corresponds to ‘Fri, 01 Jan 2016 12:00:00.512 GMT’

*Mqtt transmission example.*

To demonstrate the above, below its an example transmission from a windows machine.

```
~$ mosquitto_pub -d -q 1 -h argus01.sigintsolutions.com -p 8883 -t "v1/devices/me/telemetry"
-u "NWYs8rdilrTq8JZsWB31" -m "{\"ts\":1451649600512,\"values\": {\"key1\": \"value1\", \"key2\": \"value2\"}}" --cafile "/mnt/c/1/STAR_sigintsolutions_com/AAACertificateServices.crt"
Client mosq-0RhbsZqQSMCvEB7xbq sending CONNECT
Client mosq-0RhbsZqQSMCvEB7xbq received CONNACK (0)
Client mosq-0RhbsZqQSMCvEB7xbq sending PUBLISH (d0, q1, r0, m1, 'v1/devices/me/telemetry', ... (65 bytes))
Client mosq-0RhbsZqQSMCvEB7xbq received PUBACK (Mid: 1, RC:0)
Client mosq-0RhbsZqQSMCvEB7xbq sending DISCONNECT
```

and the resulting telemetry on the device page, **latest telemetry**

3Pro test device  
Device details

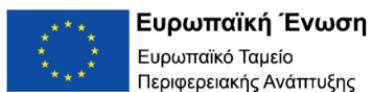
Details Attributes **Latest telemetry** Alarms Events Relations Audit Logs version-control.version-control

Latest telemetry

<input type="checkbox"/>	Last update time	Key ↑	Value
<input type="checkbox"/>	2016-01-01 14:00:00	key1	value1
<input type="checkbox"/>	2016-01-01 14:00:00	key2	value2

### Initializing devices

As mentioned multiple times in this document, for the 3Pro postprocessing to work correctly, a list of server attributes must be set on the device. These can be configured on the device either from the device dashboard itself, or using the 3ProConfigurator Application which can be found on another repository.





### Required parameters.

A new device needs the following parameters.

- **dischargeRate**: Discharge of the irrigation supply system (one or more drippers or sprinklers) over the specified field area in L/h
- **elevation**: The elevation of the of the location of the device. Right now it is not used in any calculation.
- **enableBatteryAlarm**: Boolean values that controls if the creation of alarms for low battery on the device will be active.
- **batteryLimit**: The value that will act as a limit to trigger the batteryAlarm
- **enableInactivityAlarm**: Boolean value that control the creation of alarm for device inactivity. If true inactivity alarm will be triggered when the device doesnt send any information in a time period larger than **inactivityTimeoutMinutes**
- **fieldCapacity**: Volumetric soil water content after 24-48 hour drainage of saturated soil (cm<sup>3</sup>\_water/cm<sup>3</sup>\_soil), expressed as percentage, e.g., 24
- **fieldName**: name of the field
- **latitude**: The latitude of the device location in decimal degrees(ddg)
- **longitude**: The longitude of the device location in decimal degrees(ddg)
- **soilMoistureThreshold**: The threshold that will be used to display warning message on the soil moisture.
- **wettedArea**: The area wetted by the irrigation system (m<sup>2</sup>). This area should be smaller than the field area, unless the whole field is flooded.
- **wiltingPoint**: is the point at which soil moisture becomes so low that plants are no longer able to extract water from the soil, and as a result, their leaves wilt. At this point, the soil is considered to be at its lowest level of plant-available water, and it becomes difficult for plants to maintain their growth and health. The wilting point is an important parameter in agriculture and environmental science, as it can be used to determine the amount of available water in the soil and to make decisions about irrigation and water management practices. Soil moisture sensors can be used to measure the moisture content of the soil and determine the wilting point.
- **fieldArea**: This is the field area (m<sup>2</sup>) used for the irrigation system discharge. It could be the area of the full field, of a terrace, or of a single tree.
- **inactivityTimeoutMinutes**: is used with conjunction with **enableInactivityAlarm** to trigger an alarm.



- **rainPerTick**: in case a device has water tipping bucket, this is used to calculate the actual value of the rain.
- **cropCoefficientInitial**: Crop coefficient at the start of green up (trees) or between planting and 10% field cover (field crops). The crop coefficients are used for checking the irrigation water needs.
- **cropCoefficientMid**: Crop coefficient for the full maturity stage, starting from near full canopy cover till the aging of the leaves (drying, yellowing).
- **cropCoefficientEnd**: Crop coefficient at the end of the growing season when transpiration stops, such as leaf drop (trees) or harvest (field crops).
- **dayStartInitialStage**: For field crops from planting to 10% field cover, can be skipped for trees, so date is same as start development stage
- **dayStartDevelopmentStage**: For field crops at 10% cover, for trees at leaf out
- **dayStartMidSeason**: Start of effective full cover (70-80%) or heading/flowering for field crops
- **dayEndMidSeason**: Start of crop maturity or leaf drying/coloring
- **dayEndLateSeason**: For field crops at harvest, for trees start of leaf drop, end of irrigation season
- **transmissionDuration\_m**: the logger trasmits telemetry in a predetermined period. This value is used to determine that sensors have transmitted at that predetermined period. This is not related to the inactivity alarm, because the logger maybe active, with some of the critical sensors being faulty.
- **nameOfTemperatureKey**: the name of the telemetry key that is used to calculate the min, max, and mean temperatures. This is used because there is a possibility that the logger has more than one temperature sensors, so a way is needed to differentiate the temperature telemetry kies.
- **nameOfHumidityKey**. Similarly but for the humidity.
- **tempLowLimit**: A temperature threshold. Temperature values lower than that threshold will trigger an alarm to the user. Indicating that the sensor may be faulty (for example if the sensor indicates -50 °C), or, depending on the settings, some extreme conditions that the owner of the logger wants to monitor (for example if the limit was set at -0°C) the soil will be damaged by frost.
- **tempHighLimit**: A high temperature threshold. Similar to the above description.
- **humidityLowLimit**: Similarly but for low humidity threshold
- **humidityHighLimit**: Similarly but for high humidity threshold
- **soilMoistureLowLimit**: Similarly but for low soil moisture threshold

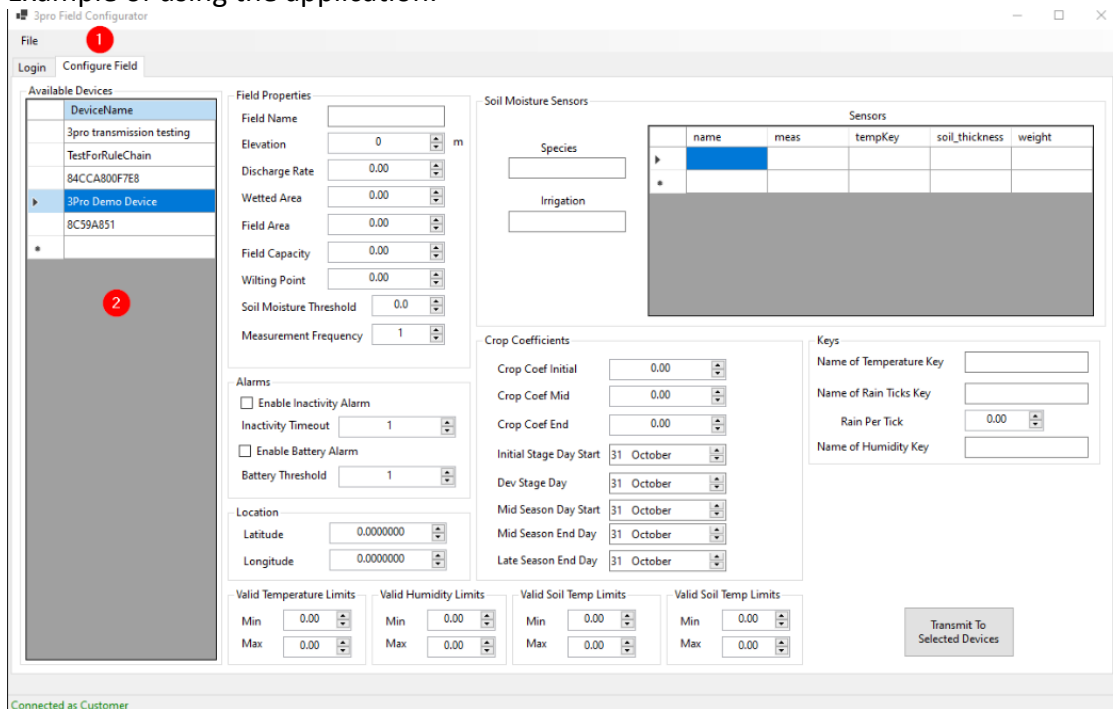


- **soilMoistureHighLimit**: similarly but for high soil moisture threshold
- **soilTemperatureLowLimit**: similarly but for low soil temperature threshold.
- **soilTemperatureHighLimit**: similarly but for high soil temperature threshold
- **fcErrorMargin**:
- **etcErrorMargin**:
- **fieldConfiguration** field configuration is a **JSON** structure that holds information about which telemetry keys are used for the calculation of the soil moisture properties. This is used because for the calculation, each sensor needs additional information that needs to be configured and to be used on the calculation, like the sensor depth. that way the sensor names can be used as variables.

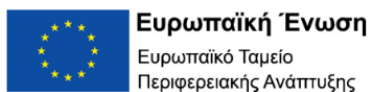
### Using the 3Pro field configurator

3Pro field configurator has been developed, to make the initial configuration of a device easy to a user / agent / researcher.

Example of using the application.



Further instructions can be found on its specific application repository.







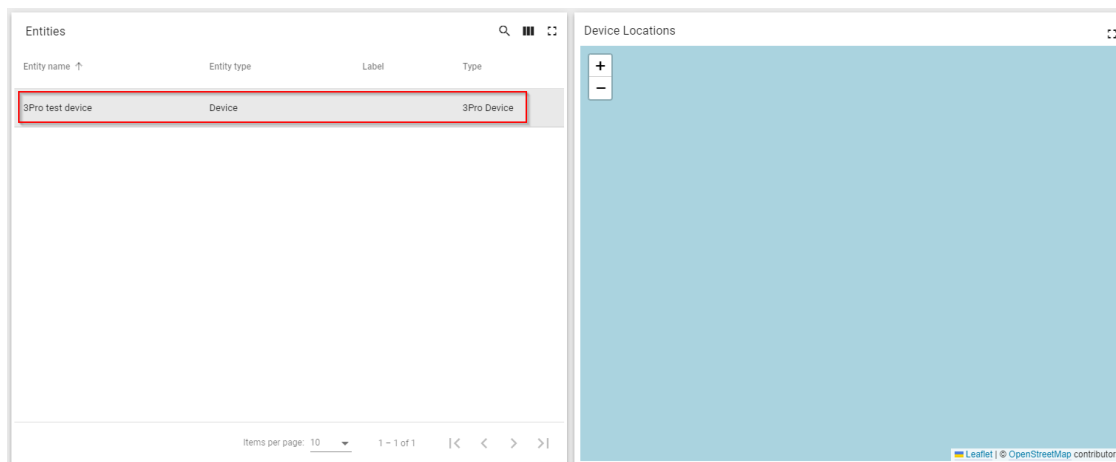
A sample settings file, that can be loaded directly on the 3Pro field configurator can be found on this repository under the folder Settings Sample

### *Using the Dashboard*

Instructions of using the dashboard for the initial configuration of the device can be found in the next chapter.

### The 3Pro dashboard

After devices with the correct type have been installed in the platform, they will be displayed in the dashboard pages.



Warning: The device will only be displayed if it is the correct device type.

Clicking on top of the device name will transition to another dashboard page with a lot of information about the device, charts, etc.

The following dashboard page has various data that are displayed.

On the top of the page, there is the name of the dashboard, the name of the device, an the time-range that is displayed in the dashboard. On the following example data that have been transmitted from the device for the last 7 days will be visible.



3Pro-Agents-Developers-General-Das... ▾ 3Pro test device History - last 7 days ↓

Realtime History

Last  
Last 7 days Advanced

Time period

Interval

Data aggregation function  
None ▾

Max values  25000

Timezone  
Browser Time (UTC+02:00) ×

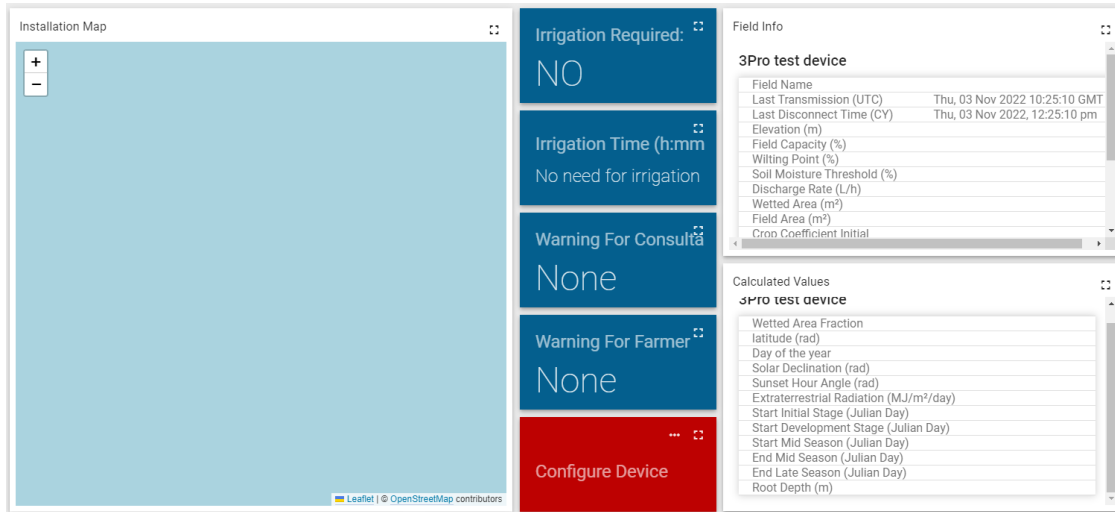
Cancel Update

This time can be modified accordingly and all the charts will update to accommodate that new time range.

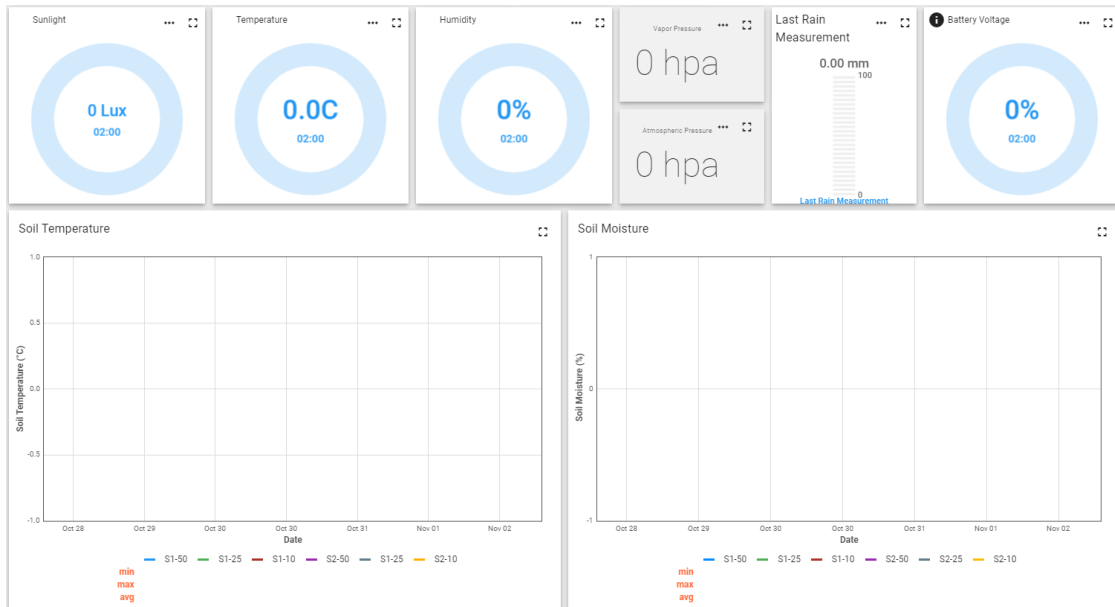




On the first portion of the dashboard, we have the map display, a list of warning panels, the configuration panel, and finally some fields with the values that have been configured for that particular device.

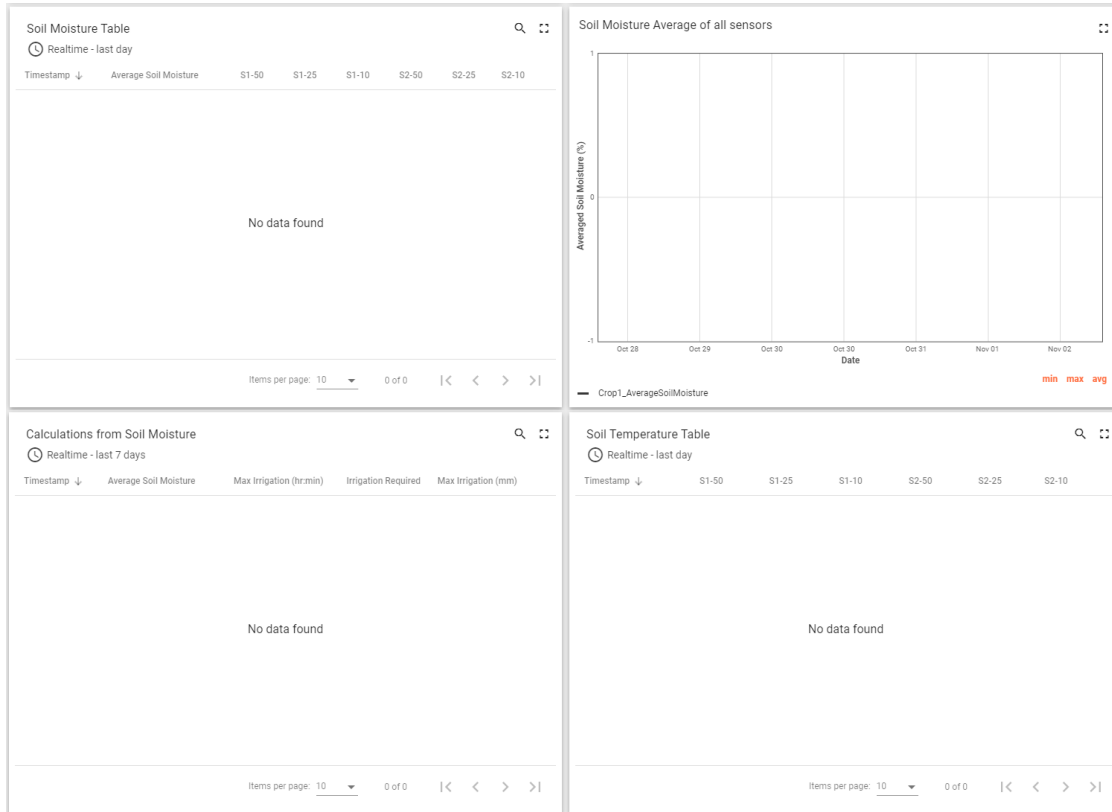


This section has environmental information as well as Charts for soil temperature and soil moisture.

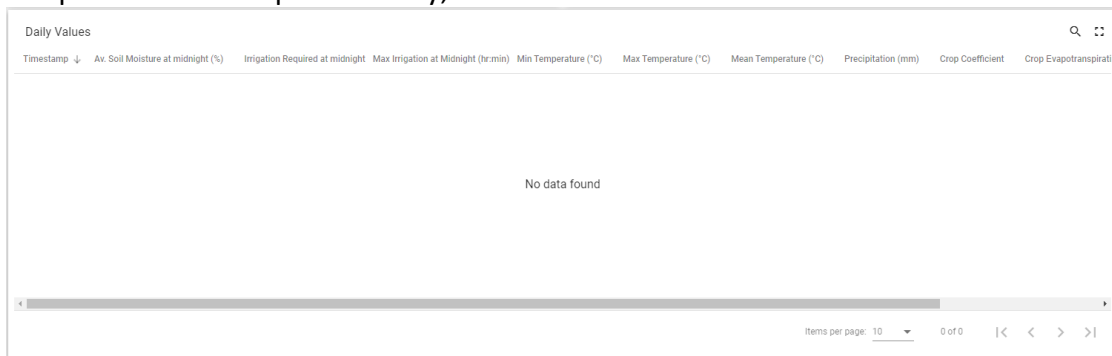




In this section we have the same information as the above, but in a table format. An additional chart is also on this section with the calculated soil average from all the sensors (which is processed from the rule engine.)



This section has values that have calculated for each day, like the minimum and maximum temperature for the previous day, etc.

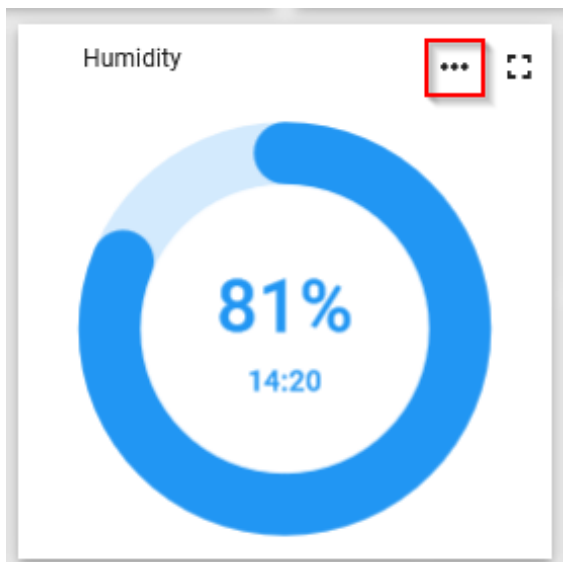




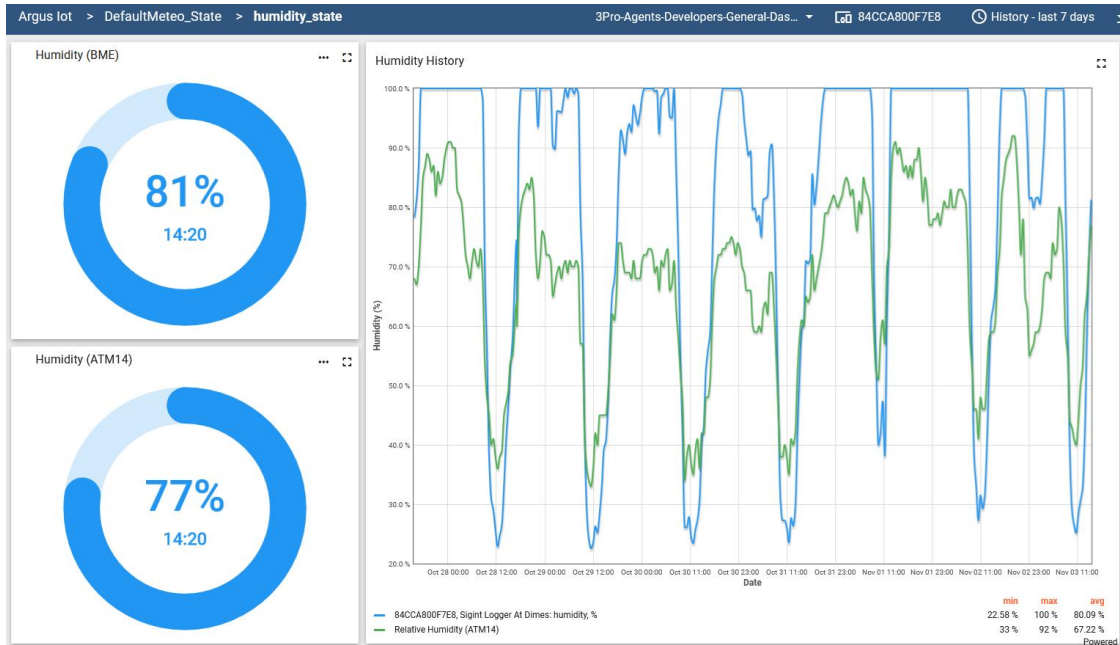
Finally this section has device error codes that can be used with ARGUS IOT logger, as well as alarms relevant to the device.

The screenshot shows two panels. The left panel, titled 'Esp Status Codes', has columns for 'Timestamp', 'ESP Status Code', and 'SDI Status Codes'. It displays 'No data found'. The right panel, titled 'Alarms', has a 'Realtime - last day' filter and a table with columns: 'Created time', 'Originator', 'Type', 'Severity', and 'Status'. It lists three identical warning alarms from '3Pro test device' at '2022-11-03 12:25:10' with the type 'Incorrect Configuration Alarm Alarm' and severity 'Warning'. Each alarm entry has a checkbox, a three-dot menu, a checkmark, and an 'X' icon.

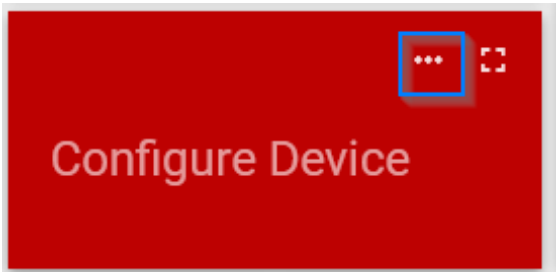
Besides the main screen, most widgets have additional information that can be displayed by pressing on the . . . symbol on top of the widget (if it exists)



This will transition to another dashboard state with more information and charts about that particular telemetry



Similarly the device can be configured by pressing on the . . . symbol on the top of the red configure device widget.



This will transition to an additional dashboard page where the required values for that device can be put, as was described on the section Required parameters.



Device Selection > DefaultMeteo\_State > config\_device\_state 3Pro-Agents-Developers-General-Das... 3

<b>Device Location</b> Latitude * 0 Longitude * 0	<b>Field Name</b> Value *	<b>Field Info</b> <b>3Pro test device</b> Field Name Last Transmission (UTC) Thu, 03 Nov 2022 10:25:10 GMT Last Disconnect Time (CY) Thu, 03 Nov 2022, 12:25:10 pm Elevation (m) Field Capacity (%) Wilting Point (%) Soil Moisture Threshold (%) Discharge Rate (L/h) Wetted Area (m <sup>2</sup> ) Field Area (m <sup>2</sup> ) Crop Coefficient Initial Crop Coefficient Mid Crop Coefficient End Day Start Initial Stage NaN undefined Day Start Development Stage NaN undefined Day Start Mid Season NaN undefined Day End Mid Season NaN undefined Day End Late Season NaN undefined	
<b>Enable Battery Alarm</b> <input type="checkbox"/> false	<b>Battery Percentage Limit (%)</b> Value * 0	<b>Calculated Values</b> <b>3Pro test device</b> Wetted Area Fraction latitude (rad) Day of the year Solar Declination (rad) Sunset Hour Angle (rad) Extraterrestrial Radiation (MJ / m <sup>2</sup> / day) Start Initial Stage (Julian Day) Start Development Stage (Julian Day)	
<b>Enable Inactivity Alarm</b> <input type="checkbox"/> false	<b>Inactivity Timeout (m)</b> Value * 0		
<b>Elevation (m)</b> Value * 0	<b>Soil Moisture Threshold (%)</b> Value * 0	<b>Field Capacity (%)</b> Value * 0	
<b>Discharge Rate (L / Hr)</b> Value * 0	<b>Wilting Point (%)</b> Value * 0		
<b>Field Capacity (%)</b> Value * 0	<b>Wetted Area (m<sup>2</sup>)</b> Value * 0	<b>Field Area (m<sup>2</sup>)</b> Value * 0	<b>Transmission Duration (m)</b> Value * 0

While populating the relevant values, these will be displayed on the **Field inform** widget. Any calculated values that are derived from the data are calculated immediately and are displayed on the **Calculated Values** card



## Field Info

### 3Pro test device

Field Name	
Last Transmission (UTC)	Thu, 03 Nov 2022 10:25:10 GMT
Last Disconnect Time (CY)	Thu, 03 Nov 2022, 12:25:10 pm
Elevation (m)	
Field Capacity (%)	
Wilting Point (%)	
Soil Moisture Threshold (%)	
Discharge Rate (L/h)	
Wetted Area (m <sup>2</sup> )	
Field Area (m <sup>2</sup> )	
Crop Coefficient Initial	
Crop Coefficient Mid	
Crop Coefficient End	
Day Start Initial Stage	NaN undefined
Day Start Development Stage	NaN undefined
Day Start Mid Season	NaN undefined
Day End Mid Season	NaN undefined
Day End Late Season	NaN undefined

## Calculated Values

### 3Pro test device

Wetted Area Fraction	
latitude (rad)	
Day of the year	
Solar Declination (rad)	
Sunset Hour Angle (rad)	
Extraterrestrial Radiation (MJ / m <sup>2</sup> / day)	
Start Initial Stage (Julian Day)	
Start Development Stage (Julian Day)	







### *Names of the keys for the dashboard.*

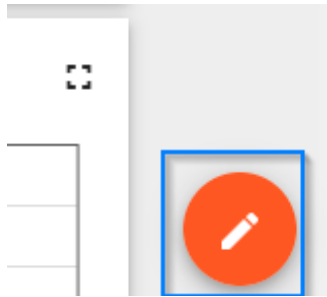
While the name of the keys of critical values for the rule engine can be adjusted on the configuration, this is not the case for the keys that are displayed on the dashboard.

On the list below its a list of the default keys that are used on the dashboard.

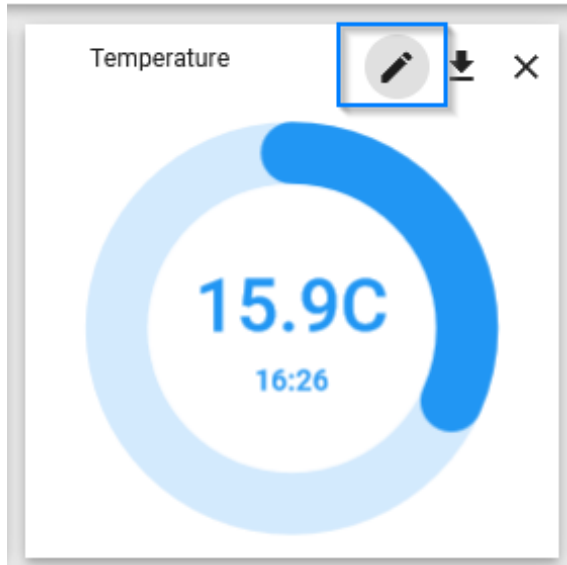
```
'Altitude'  
'Ambient Temperature'  
'Atmospheric Pressure'  
'Battery Percentage'  
'Battery Voltage'  
'Humidity'  
'Luminosity'  
'Pressure'  
'Pulse 0 Ticks' //For rain measurements  
'Relative Humidity'  
'Soil Temperature ##' //Soil temperature 01 - 06  
'Soil Vol Water Content ##' //Soil moisture 01 - 06  
'Vapor Pressure'
```

In case different keys names are going to be transmitted from the device, and change is not possible, then these keys must be modified on the dashboard.

To modify the keys for a widget, enter edit mode



Press the pencil icon on the widget



Press the pencil icon again on the key name

Type	Parameters
1. Entity	Entity alias * Selected Device Filter

Maximum 1 timeseries/attribute is allowed.

and change the key on the new page.






### Data key configuration ✕

Key \*  
Ambient Temperature ✕

Label \* Ambient Temperature

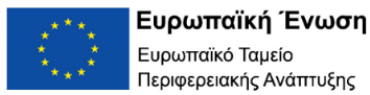
Color \*  #2196f3

Special symbol to show next to value

Number of digits after floating point  ⌵

Use data post-processing function

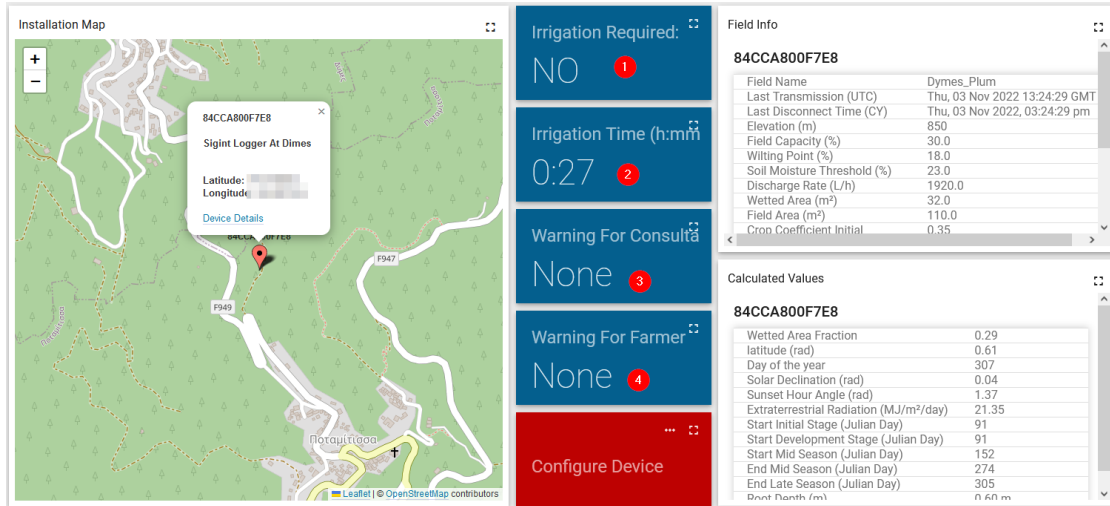
Cancel Save





### Example

To give a more concrete example the following dashboard is from an already configured device that already transmits data.



As is visible from the picture, the map displays the location of the device.

Additionally the calculations display a message on widget (1), with Yes or No if the farmer needs to irrigate.

Even if the answer is NO, it will still display on widget (2) the calculated time for the irrigation. In the case of the example, the farmer should irrigate for 0.27 time, but, at least for the time being, the moisture soil levels are not low enough to warrant the irrigation.

Widget (3) displays the following messages if they are applicable

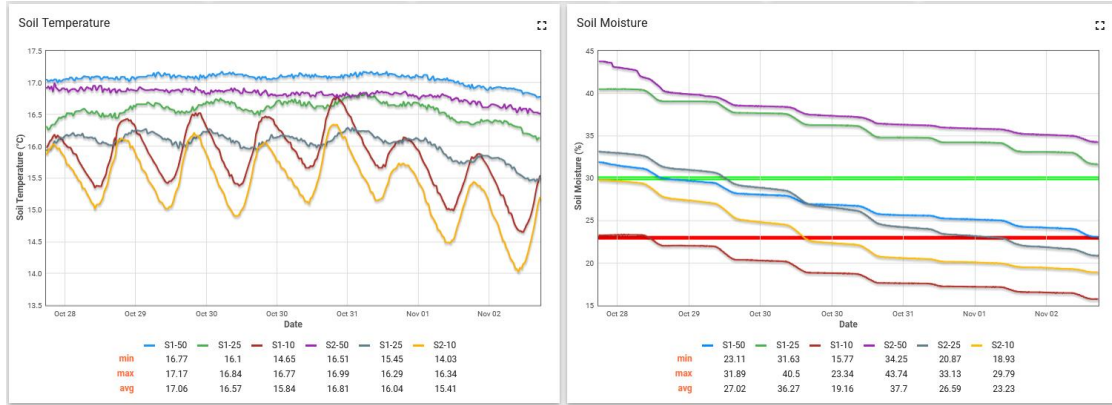
Kc too low or WA\_fraction too large.  
Kc to High or WA-Fraction too small.

Finally on widget (4) a warning Drainage losses, reduce irrigation duration will be displayed if the farmer over-irrigates the field.

This particular field has 6 soil sensors. On the first chart with can see the chart for the temperature of the soil in this 6 particular sensors. On the right we can see another chart which

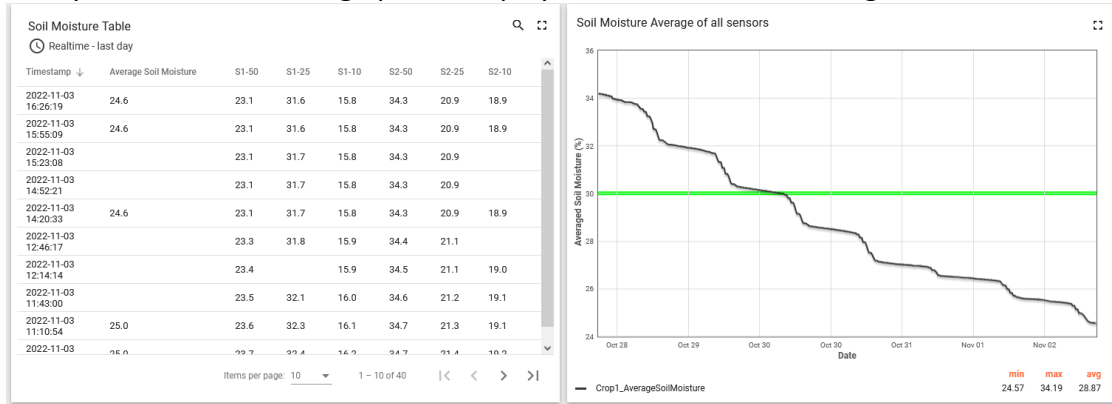


show the soil moisture levels for the same 6 sensors.



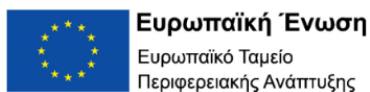
On the next section we have the Soil Moisture Table which shows the soil moisture measurements for each sensor. The soil average for the field can be calculated only if the measurements all the sensors on the field are available at the same time.

In any case, the soil average plot is displayed on the chart to the right.



On the next section we have all the values that are reported once per day

Timestamp	Av. Soil Moisture at midnight (%)	Irrigation Required at midnight	Max Irrigation at Midnight (hr:min)	Min Temperature (°C)	Max Temperature (°C)	Mean Temperature (°C)	Precipitation (mm)	Crop Coefficient	Crop Evapotranspiration
2022-11-03 02:43:36	25.5	false	0:27	8.7	21.3	15.0	0	0.00	0.0
2022-11-02 02:29:37	26.4	false	0:21	9.1	21.2	15.2	0	0.50	1.2
2022-11-01 02:14:06	27.0	false	0:18	11.0	23.3	17.1	0	0.51	1.3
2022-10-31 03:01:29	28.5	false	0:09	10.5	23.3	16.9	0	0.52	1.4
2022-10-30 03:46:53	30.1	false	0:00	10.8	23.0	16.9	0	0.53	1.4
2022-10-29 03:32:57	31.9	false	0:00	10.8	23.6	17.2	0	0.55	1.4
2022-10-28 04:21:02	33.9	false	0:00	11.6	24.0	17.8	0	0.56	1.5



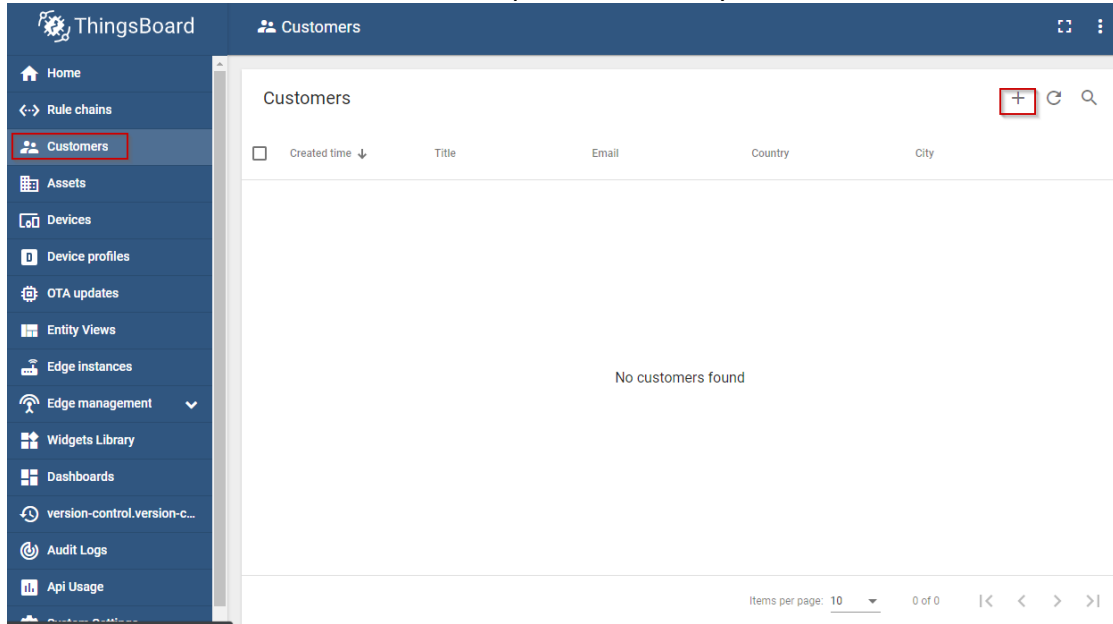


## Creating farmer entities and assigning devices

Tenant has the ability to create device and modify dashboards. This should not be the case for a single farmer that just needs to see his soil moisture levels and get some warnings.

A farmer customer entity can be created with the following way.

Press on the button Customers, then press on the + symbol.



On the new page add the relevant information for the farmer. The only mandatory field is the Title Then press Add to complete the process.



### Add Customer

**Title \***  
New Farmer

Description

Country

City      State / Province      Zip / Postal Code

Address

Address 2

Phone

Cancel    Add

To add a user to this new customer press on the manage customer User icon

Created time ↓	Title	Email	Country	City
2022-11-03 17:00:46	New Farmer			

On this press on the + symbol and on the new page, add the relevant email and press Add



This will create a new activation link which can be used by that customer.

After the activation link has accessed a new page will be displayed, so that the user can create a new password.





## Create Password

Password

Password again

**Create Password** Cancel

On the tenant side of things, to assign the device to that customer, press on the devices tab  
Then press the button Assign to customer

Created time ↓	Name	Device profile	Label	Customer	Public	Is gateway	
2022-11-03 11:48:39	3Pro test device	3Pro Device			<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>

Similarly the dashboard must also be assigned to the customer.

to do that press on the Dashboards Tab, then press on the dashboard that needs to be shared.

In this case the Simplified dashboard will be shared with the user. To do that press on the manage assigned dashboards button



Created time ↓	Title	Assigned to customers	Public
2022-11-03 11:15:04	3Pro-General-Simplified	<input type="checkbox"/>	<input type="checkbox"/>
2022-11-03 11:14:57	3Pro-Agents-Developers-General-Dashboard	<input type="checkbox"/>	<input type="checkbox"/>

And on the new page, select the customer name from the entity list.

## Manage assigned customers

Assigned customers

Entity list

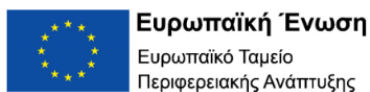
New Farmer

Cancel Update

After the dashboard is assigned to customer, it can be assigned as the default dashboard, so it can be displayed directly at his home screen when he logs on the platform.

To do this, press on the customers when logged in as a tenant.

Then press on the name of the farmer, and on the new page that will open press the pencil icon to edit the details of the customer.





On the section Home Dashboard select the dashboard name, and disable the check box Hide home dashboard toolbar

**New Farmer**  
Customer details

Title \*  
New Farmer

Description

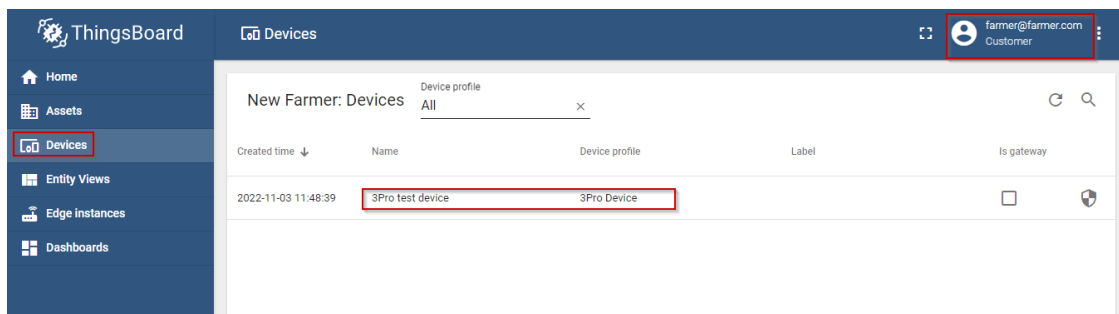
Home dashboard  
3Pro-General-Simplified

Country

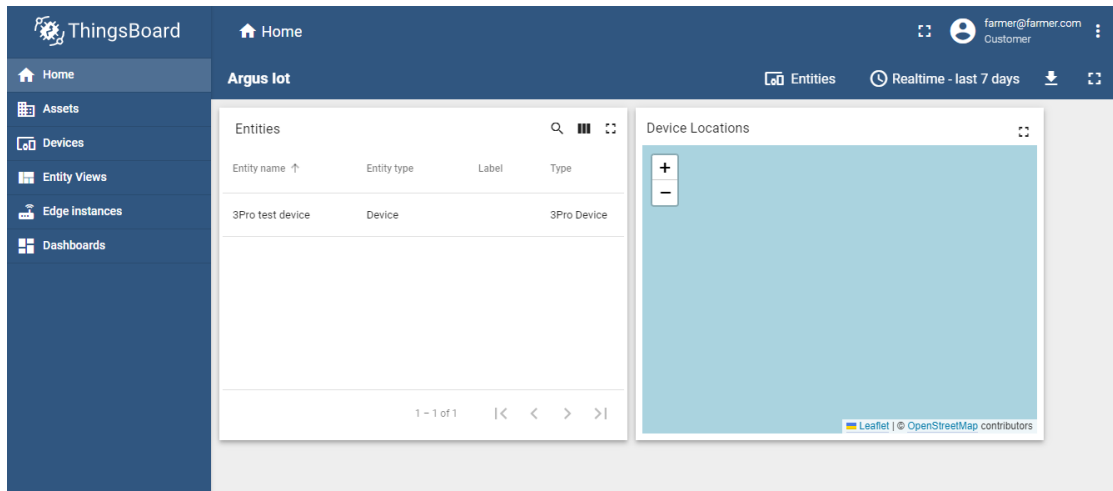
City State / Province Zip / Postal Code

Hide home dashboard toolbar

The farmer should be able to access his devices from the device list, and see his data from the dashboards.



If a default dashboard was selected for that user, then it will be immediately visible on the Home tab



Alternative if a default dashboard wasn't selected, the dashboards can be found on the Dashboards tab.

### Downloading Data

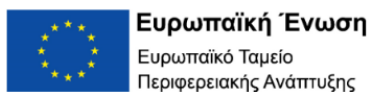
To reduce the potential for abuse the source code for that particular tool is not provided, as it can be used to perform DoS attacks to thingsboard installations.

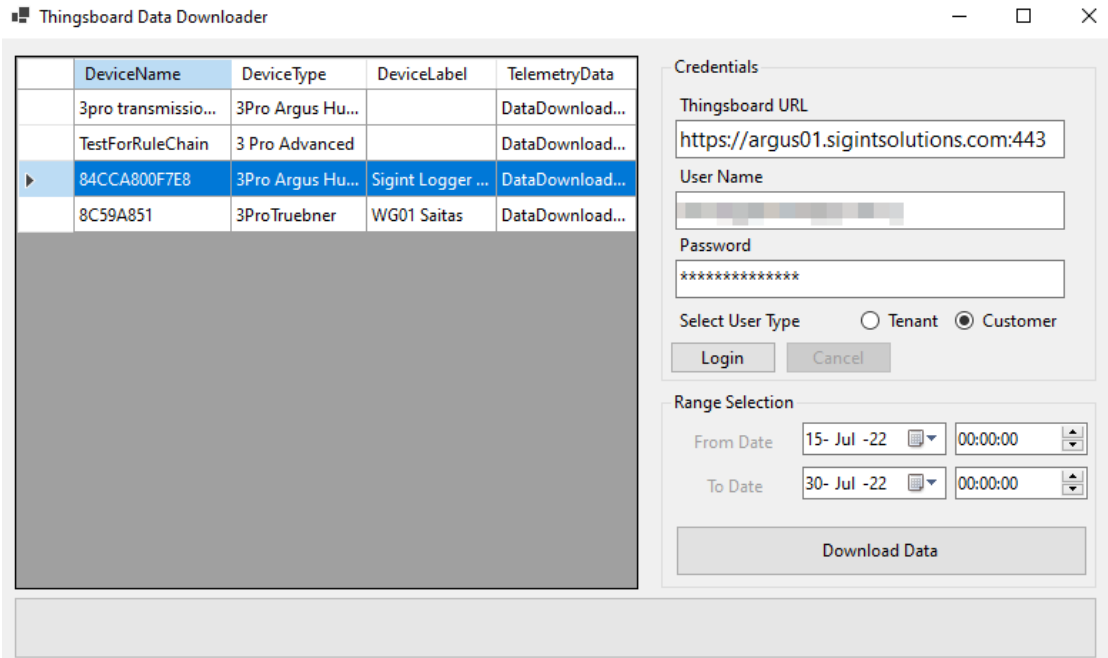
The downloader application can be found on the following repository [3pro Data Downloader](#)

To download data run the thingsboard Data Downloader and input the necessary credentials.

After logging in, a list of the available devices will be available.

Select one or more devices and select the time range.





Press the download data,

**Warning:** The download process can take up to 30 minutes to complete for devices with a lot of data and extended time ranges. (i.e., 2 years of data)

## Description of the other utility applications

### *Data post processor application*

The 3 pro data post processor has been developed to give the researcher the ability to process the data from a field that is monitored retroactively, using different settings.

The data post processor can be found on the following link. [Data Post Processor](#)

During normal operation of the platform, every data point that is transmitted is used to calculate the status of the soil and is used to make predictions. For these calculations to work the devices must be correctly configured using specific server attributes. For the specific server attributes as well as description of them refer to 3 pro field configurator git, as well on the main project git.

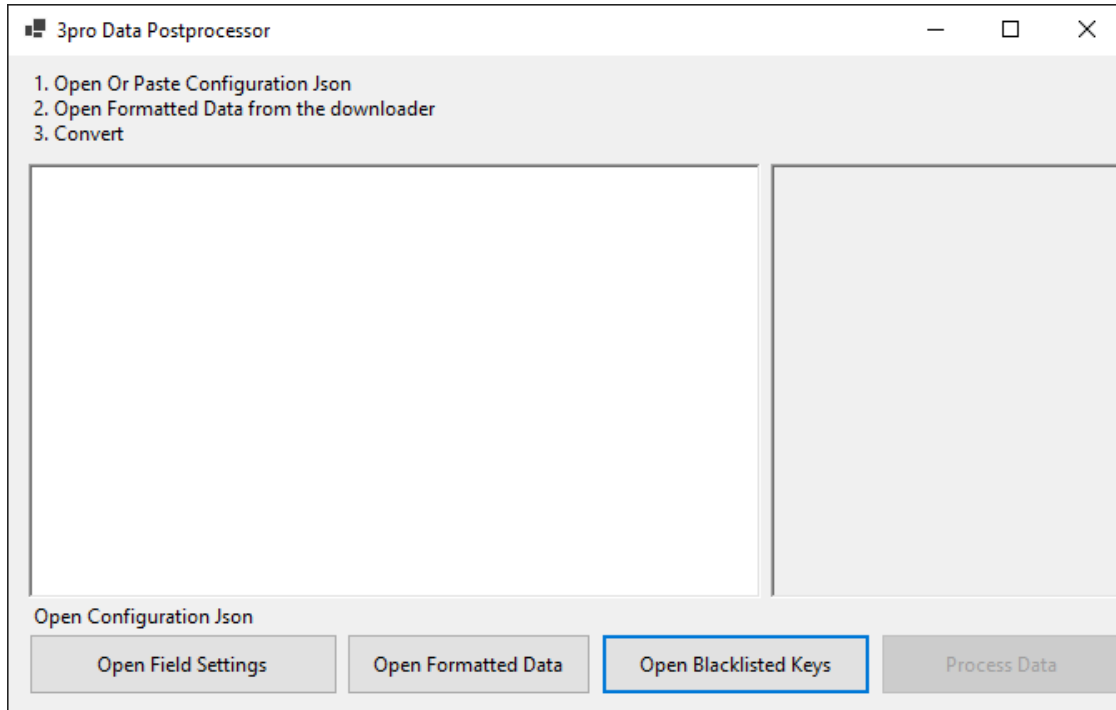
The problem with that is that if a researcher changes one of the server attribute, this will only be taken into effect after the point where the value changed. While this functionality could



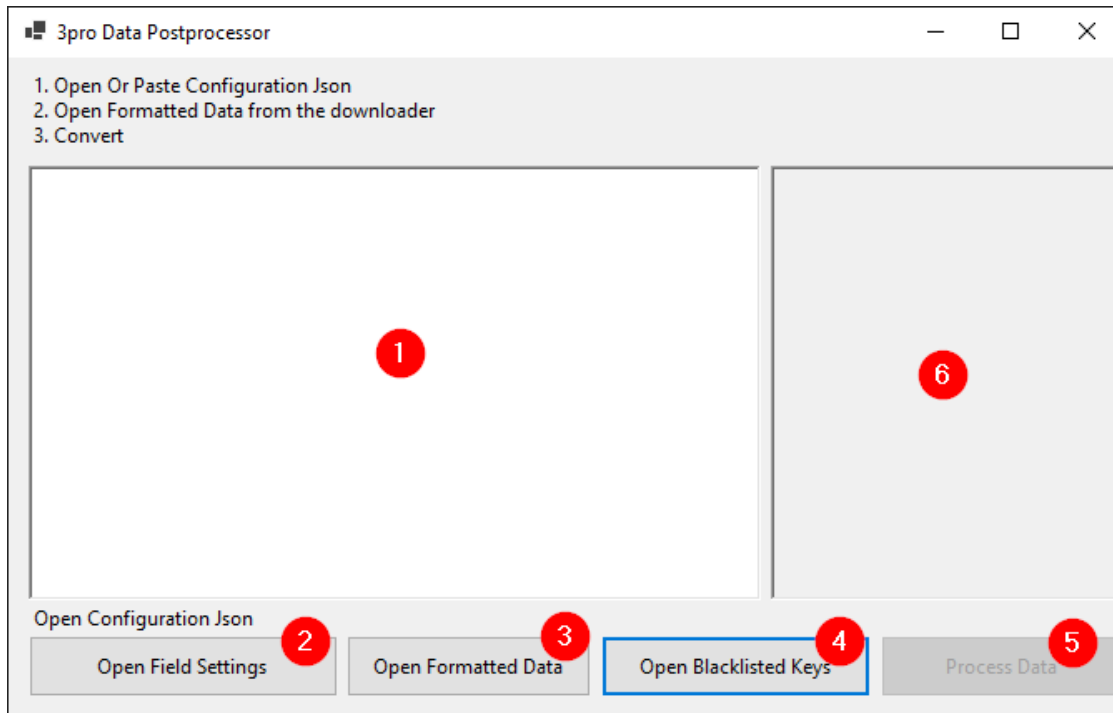
have been added in the platform, irresponsible use of this functionality (especially if a device has data points going back years), will put a major strain on the server.

This application solves this issue by running calculations on a local, downloaded copy of the data (as described on the section about downloading data.)

Main image of the application.



The application consists of 2 sections, as well as 4 buttons.



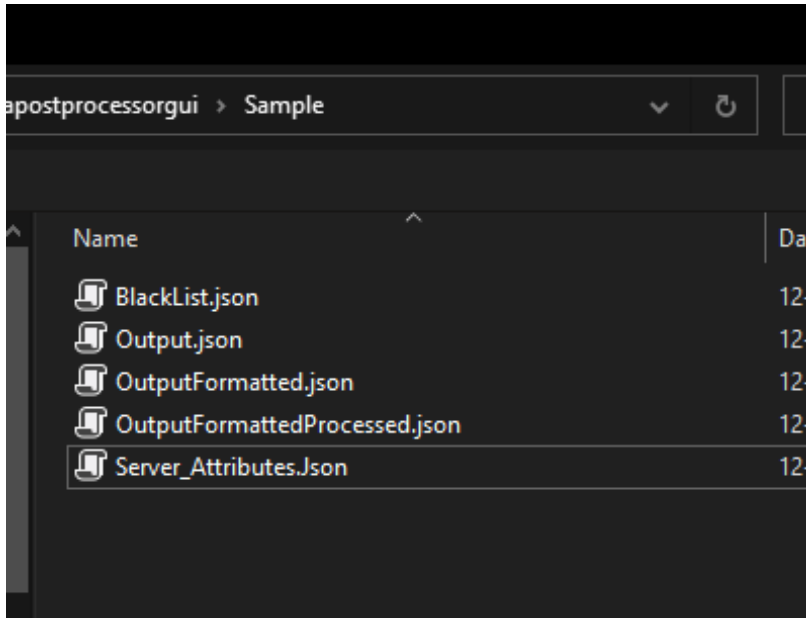
(1). The device configuration json can be pasted here. Alternative it can be loaded by the use of button (2) a valid device configuration example file can be found in this repository.

(3) This button open a file selector box that can be used to select a file with the data from the device. The data from the device must have been downloaded by the DataDownloader application.

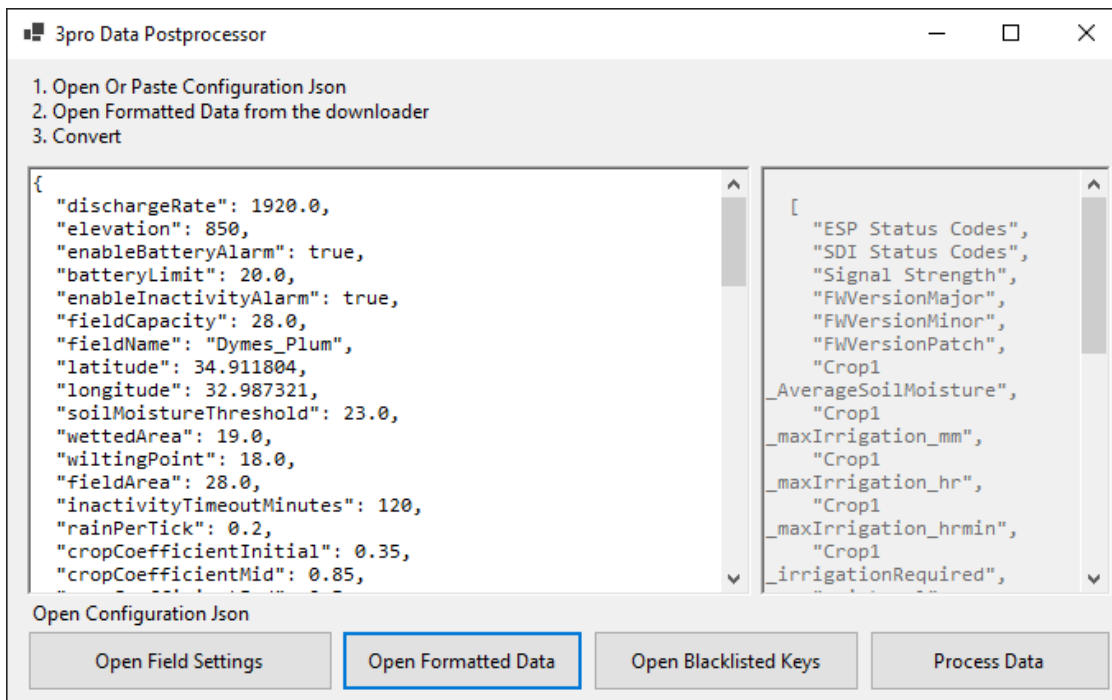
(4) Open blacklisted keys. This opens a file selector box that can be used to select a json with some blacklisted keys. These keys will be removed from the output file.

[Example usage](#)

For demonstration sample files are included in this repository.



After opening all the files the Process Data button will become enabled







```

C:\Users\g.michalis\source\repos\3pro\3prodata\postprocessor\gui\bin\Debug\net6.0-windows\3ProDataPostProcessorGUI.exe
Calculated CropCoefficient: 0.85, ReferenceEvapotranspiration: 5.374168751093255, Evapotranspiration: 5.374168751093255
Calculated Daly solar properties JDN: 244, Solar Declination: 0.13555909450453021, SunsetHourAngle: 1.6661333533114584,
  Extraterrestrial Radiation: 34.67405852377178
Calculated CropCoefficient: 0.85, ReferenceEvapotranspiration: 5.1488468957195375, Evapotranspiration: 5.148846895719537
5
Calculated Daly Soil Moisture Average 30.825416666666666, and MaxIrrigation_mm: 0
Calculated Daly solar properties JDN: 245, Solar Declination: 0.12889158683329247, SunsetHourAngle: 1.6613771198462273,
  Extraterrestrial Radiation: 34.48531524440577
Calculated CropCoefficient: 0.85, ReferenceEvapotranspiration: 5.262669968588263, Evapotranspiration: 5.262669968588263
Calculated Daly solar properties JDN: 246, Solar Declination: 0.12218588580316808, SunsetHourAngle: 1.6566039906702923,
  Extraterrestrial Radiation: 34.29447842093969
Calculated CropCoefficient: 0.85, ReferenceEvapotranspiration: 5.212249001001342, Evapotranspiration: 5.212249001001342
Calculated Daly Soil Moisture Average 32.702291666666666, and MaxIrrigation_mm: 0
Calculated Daly solar properties JDN: 247, Solar Declination: 0.1154439784580925, SunsetHourAngle: 1.651814976981595, E
xtraterrestrial Radiation: 34.10159159458551
Calculated CropCoefficient: 0.85, ReferenceEvapotranspiration: 5.134720136521941, Evapotranspiration: 5.134720136521941
Calculated Daly Soil Moisture Average 28.552708333333328, and MaxIrrigation_mm: 0
Calculated Daly solar properties JDN: 248, Solar Declination: 0.10866786257071502, SunsetHourAngle: 1.6470110734526846,
  Extraterrestrial Radiation: 33.90670012732307
Calculated CropCoefficient: 0.85, ReferenceEvapotranspiration: 4.944501019649049, Evapotranspiration: 4.944501019649049
Calculated Daly Soil Moisture Average 26.398333333333333, and MaxIrrigation_mm: 6.52107142857143
Calculated Daly solar properties JDN: 249, Solar Declination: 0.10185954605041293, SunsetHourAngle: 1.6421932589961006,
  Extraterrestrial Radiation: 33.70085120337561
Calculated CropCoefficient: 0.85, ReferenceEvapotranspiration: 4.514273340815335, Evapotranspiration: 4.514273340815335
Calculated Daly solar properties JDN: 250, Solar Declination: 0.09502104634830706, SunsetHourAngle: 1.6373624975614627,
  Extraterrestrial Radiation: 33.51109382770784
Calculated CropCoefficient: 0.85, ReferenceEvapotranspiration: 4.672788639841552, Evapotranspiration: 4.672788639841552
[12:38:34 INF] Saved json file
[12:38:37 INF] Saved cvs file

```

After pressing the button, the application should process the data and save the result as the with the Processed suffix on the file name. By default the application will generate another .json as well as .csv file.

### Device configurator application

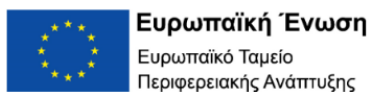
3Pro field configurator has been developed to make the initial configuration of a device easy to a user / agent / researcher.

To perform the necessary research a number of wireless sensors had to be used to take various measurements of the state of the soil as well as the environmental conditions in an area. To keep all these data easily accessible, a sensor observation IoT platform had to be developed.

On the Sensor Platform, device entities of type “3Pro Device” can be added. This device entity will record the sensor data that is transmitted from the device, and will post process the data accordingly to give some predictions. But for the data to be post processed successfully some specific parameters about a device must be set correctly.

### Description of server attributes.

- **dischargeRate:** Discharge of the irrigation supply system (one or more drippers or sprinklers) over the specified field area in L/h
- **elevation:** The elevation of the of the location of the device. Right now it is not used in any calculation.





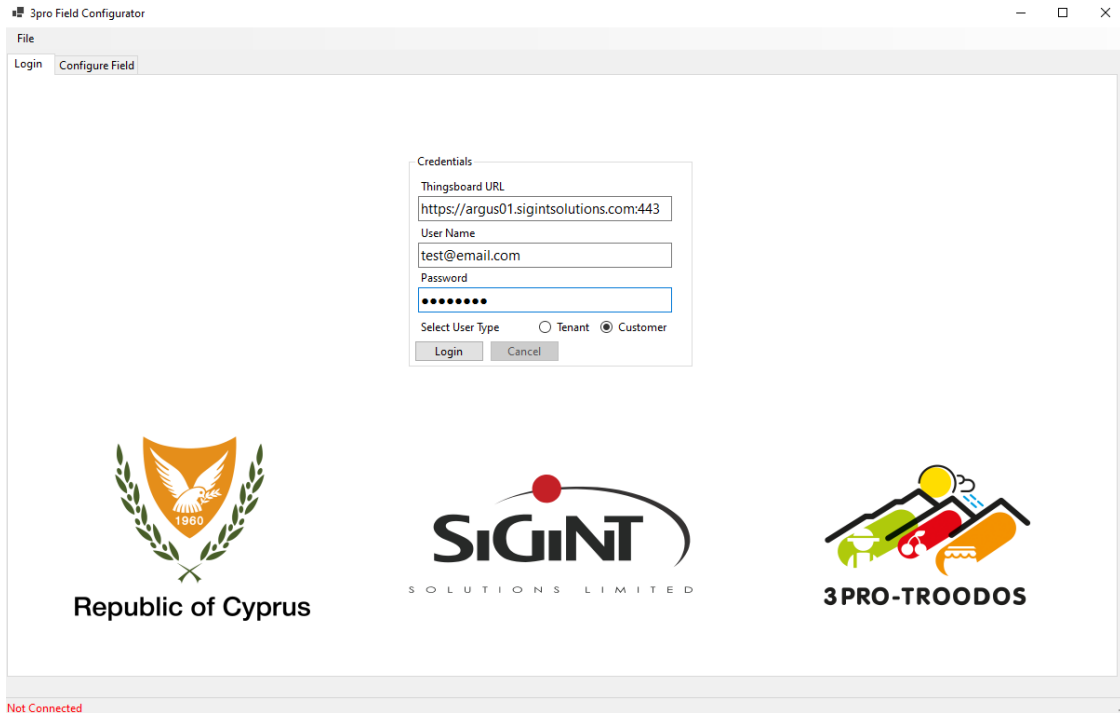
- **enableBatteryAlarm**: Boolean values that controls if the creation of alarms for low battery on the device will be active.
- **batteryLimit**: The value that will act as a limit to trigger the **batteryAlarm**
- **enableInactivityAlarm**: Boolean value that control the creation of alarm for device inactivity. If true inactivity alarm will be triggered when the device doesnt send any information in a time period larger than **inactivityTimeoutMinutes`**
- **fieldCapacity**: Volumetric soil water content after 24-48 hour drainage of saturated soil ( $\text{cm}^3_{\text{water}}/\text{cm}^3_{\text{soil}}$ ), expressed as percentage, e.g., 24
- **fieldName**: name of the field
- **latitude**: The latitude of the device location in decimal degrees(ddg)
- **longitude**: The longitude of the device location in decimal degrees(ddg)
- **soilMoistureThreshold**: The threshold that will be used to display warning message on the soil moisture.
- **wettedArea**: The area wetted by the irrigation system ( $\text{m}^2$ ). This area should be smaller than the field area, unless the whole field is flooded.
- **wiltingPoint**: ASK ADRIANA
- **fieldArea**: This is the field area ( $\text{m}^2$ ) used for the irrigation system discharge. It could be the area of the full field, of a terrace, or of a single tree.
- **inactivityTimeoutMinutes**: is used with conjunction with **enableInactivityAlarm** to trigger an alarm.
- **rainPerTick**: in case a device has water tipping bucket, this is used to calculate the actual value of the rain.
- **cropCoefficientInitial**: Crop coefficient at the start of green up (trees) or between planting and 10% field cover (field crops). The crop coefficients are used for checking the irrigation water needs.
- **cropCoefficientMid**: Crop coefficient for the full maturity stage, starting from near full canopy cover till the aging of the leafs (drying, yellowing).
- **cropCoefficientEnd**: Crop coefficient at the end of the growing season when transpiration stops, such as leaf drop (trees) or harvest (field crops).
- **dayStartInitialStage**: For field crops from planting to 10% field cover, can be skipped for trees, so date is same as start development stage
- **dayStartDevelopmentStage**: For field crops at 10% cover, for trees at leaf out
- **dayStartMidSeason**: Start of effective full cover (70-80%) or heading/flowering for field crops
- **dayEndMidSeason**: Start of crop maturity or leaf drying/coloring



- **dayEndLateSeason:** For field crops at harvest, for trees start of leaf drop, end of irrigation season
- **transmissionDuration\_m:** the logger transmits telemetry in a predetermined period. This value is used to determine that sensors have transmitted at that predetermined period. This is not related to the inactivity alarm, because the logger maybe active, with some of the critical sensors being faulty.
- **nameOfTemperatureKey:** the name of the telemetry key that is used to calculate the min, max, and mean temperatures. This is used because there is a possibility that the logger has more than one temperature sensors, so a way is needed to differentiate the temperature telemetry keys.
- **nameOfHumidityKey.** Similarly but for the humidity.
- **tempLowLimit:** A temperature threshold. Temperature values lower than that threshold will trigger an alarm to the user. Indicating that the sensor may be faulty (for example if the sensor indicates -50 °C), or, depending on the settings, some extreme conditions that the owner of the logger wants to monitor (for example if the limit was set at -0°C) the soil will be damaged by frost.
- **tempHighLimit:** A high temperature threshold. Similar to the above description.
- **humidityLowLimit:** Similarly but for low humidity threshold
- **humidityHighLimit:** Similarly but for high humidity threshold
- **soilMoistureLowLimit:** Similarly but for low soil moisture threshold
- **soilMoistureHighLimit:** similarly but for high soil moisture threshold
- **soilTemperatureLowLimit:** similarly but for low soil temperature threshold.
- **soilTemperatureHighLimit:** similarly but for high soil temperature threshold
- **fcErrorMargin:**
- **etcErrorMargin:**
- **fieldConfiguration** field configuration is a JSON structure that holds information about which telemetry keys are used for the calculation of the soil moisture properties. This is used because for the calculation, each sensor needs additional information that needs to be configured and to be used on the calculation, like the sensor depth. that way the sensor names can be used as variables.

[Instructions for application use](#)

The initial screen of the program consists of the login screen.



The user can input the url in the first field (1) in the format `https://server.com:port_number`

The user must also select the user type by checkbox (2). For more information about the user types please refer to the main project repository.

Credentials

Thingsboard URL  
https://argus01.sigintsolutions.com:443

User Name  
test@email.com

Password  
●●●●●●●●

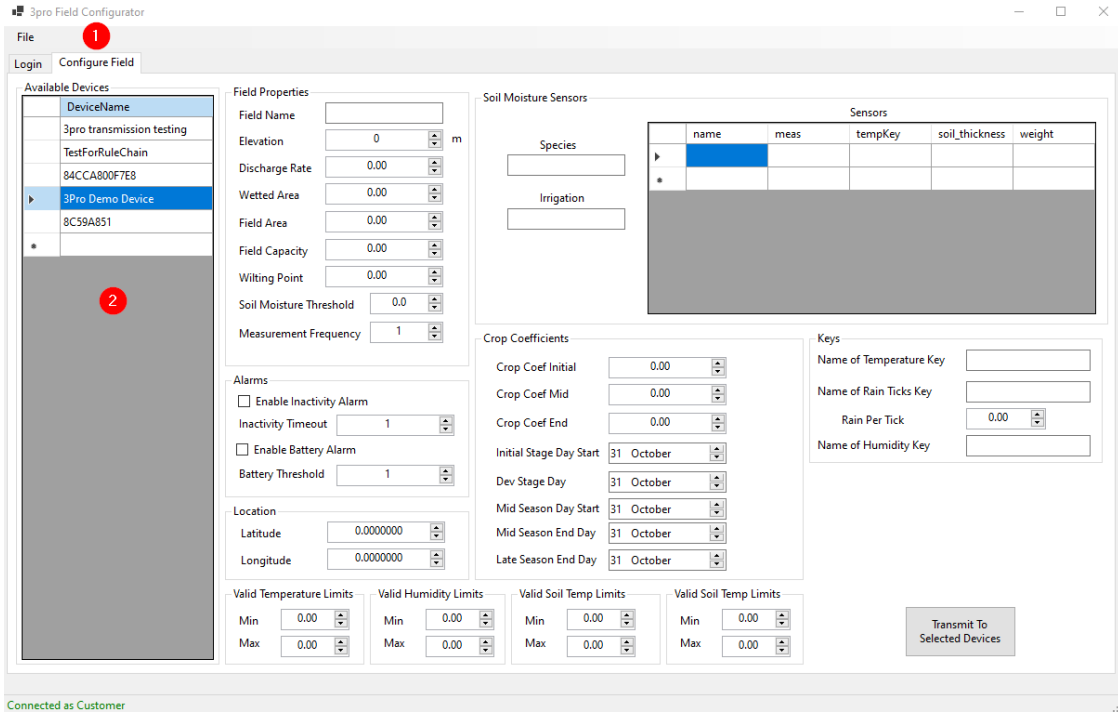
Select User Type     Tenant     Customer

Login    Cancel



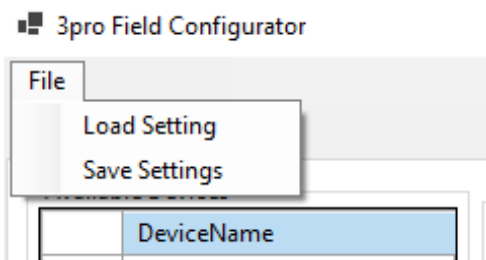


after successful login on the second tab (1) a list of the available devices for modification (2) will appear.



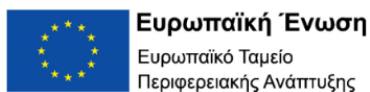
Additionally on the same tab there are fields for all the settings that were described above and needs to be modified.

The current settings can be loaded via the file menu. Similarly the currently displayed settings can be saved.



A sample for a valid file can be found in this repository.

Additionally, a valid file follows the following format.





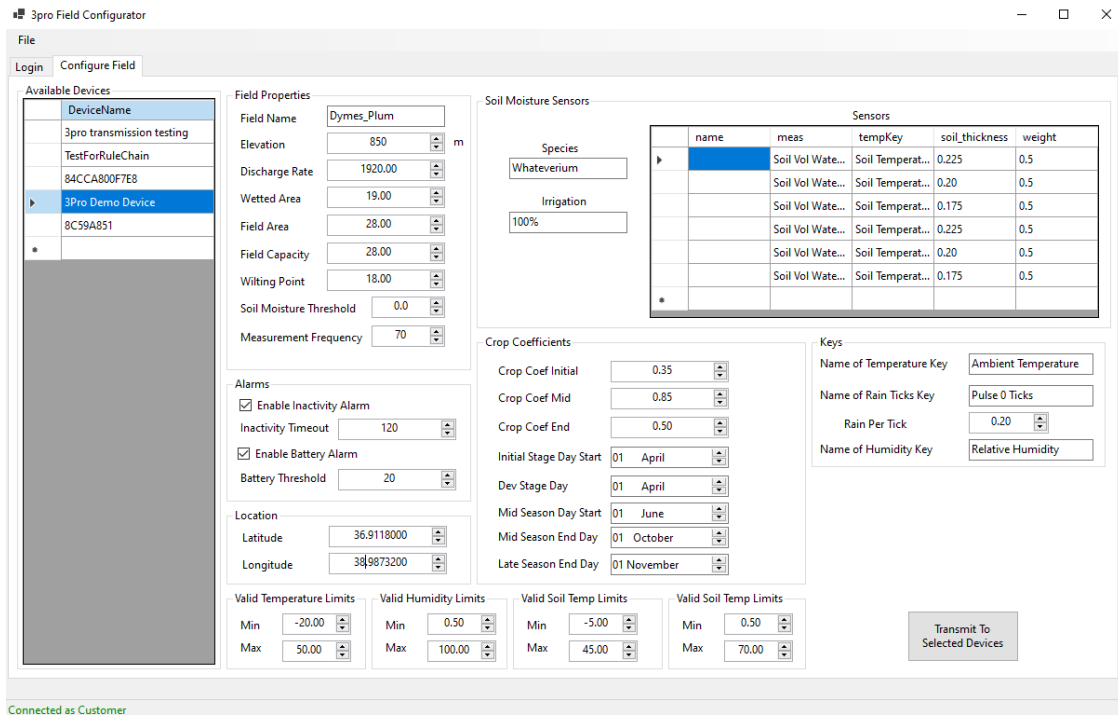
```
{
  "dischargeRate": 1920,
  "elevation": 850,
  "enableBatteryAlarm": true,
  "batteryLimit": 20.0,
  "enableInactivityAlarm": true,
  "fieldCapacity": 28.0,
  "fieldName": "Dymes_Plum",
  "latitude": 34.000000,
  "longitude": 32.000000,
  "soilMoistureThreshold": 23.0,
  "wettedArea": 19,
  "wiltingPoint": 18.0,
  "fieldArea": 28.0,
  "inactivityTimeoutMinutes": 120,
  "rainPerTick": 0.2,
  "cropCoefficientInitial": 0.35,
  "cropCoefficientMid": 0.85,
  "cropCoefficientEnd": 0.5,
  "dayStartInitialStage": 1648798140000,
  "dayStartDevelopmentStage": 1648798140000,
  "dayStartMidSeason": 1654068540000,
  "dayEndMidSeason": 1664609340000,
  "dayEndLateSeason": 1667291340000,
  "fcErrorMargin": 2.0,
  "etcErrorMargin": 0.2,
  "transmissionDuration_m": 70,
  "tempLowLimit": -20.0,
  "tempHighLimit": 50.0,
  "humidityLowLimit": 0.5,
  "humidityHighLimit": 100.0,
  "soilMoistureLowLimit": 0.5,
  "soilMoistureHighLimit": 70.0,
  "soilTemperatureLowLimit": -5.0,
  "soilTemperatureHighLimit": 45.0,
  "nameOfTemperatureKey": "Ambient Temperature",
  "nameOfHumidityKey": "Relative Humidity",
  "nameOfRainTicksKey": "Pulse 0 Ticks",
  "fieldConfiguration": {
    "groupsArray": [
      {
        "Crop1": {
          "species": "Whateverium",
          "irrigation": "100%",
```



```
"sensorArray": [  
  {  
    "name": "",  
    "meas": "Soil Vol Water Content 01",  
    "tempKey": "Soil Temperature 01",  
    "soil_thickness": "0.225",  
    "weight": "0.5"  
  },  
  {  
    "name": "",  
    "meas": "Soil Vol Water Content 02",  
    "tempKey": "Soil Temperature 02",  
    "soil_thickness": "0.20",  
    "weight": "0.5"  
  },  
  {  
    "name": "",  
    "meas": "Soil Vol Water Content 03",  
    "tempKey": "Soil Temperature 03",  
    "soil_thickness": "0.175",  
    "weight": "0.5"  
  },  
  {  
    "name": "",  
    "meas": "Soil Vol Water Content 04",  
    "tempKey": "Soil Temperature 04",  
    "soil_thickness": "0.225",  
    "weight": "0.5"  
  },  
  {  
    "name": "",  
    "meas": "Soil Vol Water Content 05",  
    "tempKey": "Soil Temperature 05",  
    "soil_thickness": "0.20",  
    "weight": "0.5"  
  },  
  {  
    "name": "",  
    "meas": "Soil Vol Water Content 06",  
    "tempKey": "Soil Temperature 06",  
    "soil_thickness": "0.175",  
    "weight": "0.5"  
  }  
]
```



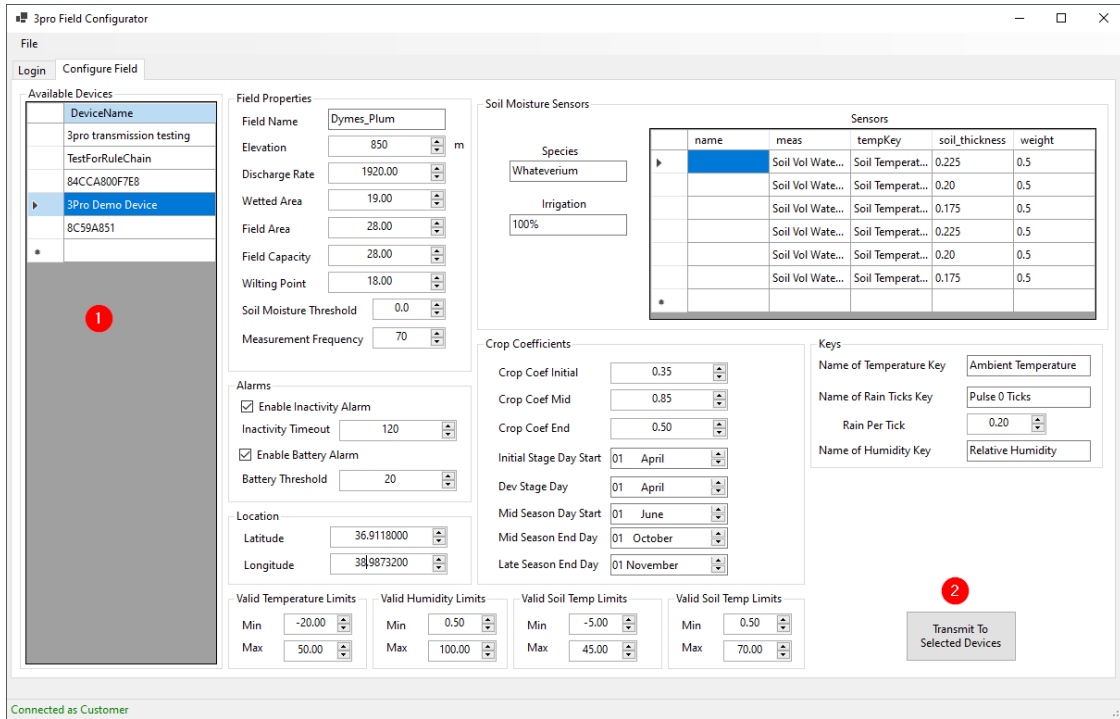
After loading the settings file, all the relevant fields will be updated with the loaded values.



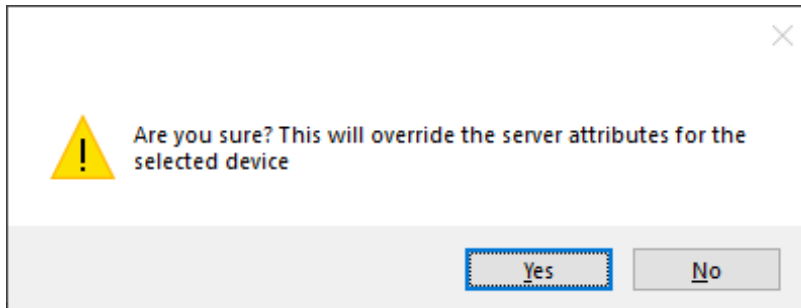
Now the settings can be transmitted to the relevant devices by selecting the devices from the device list (1) and pressing the button (2)







Upon pressing of the transmit button, a warning will be displayed



Care must be taken upon the device selection because any transmission will overwrite any settings on that device with the same attribute name

Before Transmission:



### 3Pro Demo Device

Device details

Details | **Attributes** | Latest telemetry | Alarms | Events | Relations

Entity attributes scope: Server attributes

<input type="checkbox"/>	Last update time	Key ↑	Value	
<input type="checkbox"/>	2022-10-31 13:29:53	active	false	
<input type="checkbox"/>	2022-10-31 13:29:53	inactivityAlarmTime	1667215793179	

After Transmission.

### 3Pro Demo Device

Device details

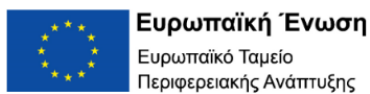
Details | **Attributes** | Latest telemetry | Alarms | Events | Relations

Entity attributes scope: Server attributes

<input type="checkbox"/>	Last update time	Key ↑	Value	
<input type="checkbox"/>	2022-10-31 15:07:02	cropCoefficientEnd	0.5	
<input type="checkbox"/>	2022-10-31 15:07:02	cropCoefficientInitial	0.35	
<input type="checkbox"/>	2022-10-31 15:07:02	cropCoefficientMid	0.85	
<input type="checkbox"/>	2022-10-31 15:07:02	dayEndLateSeason	1667291340000	
<input type="checkbox"/>	2022-10-31 15:07:02	dayEndLateSeason_JDN	305	
<input type="checkbox"/>	2022-10-31 15:07:02	dayEndMidSeason	1664609340000	
<input type="checkbox"/>	2022-10-31 15:07:02	dayEndMidSeason_JDN	274	
<input type="checkbox"/>	2022-10-31 15:07:02	dayStartDevelopmentStage	1648798140000	

Running the application

Unzip the latest release and run the executable. This application requires net core runtime to be installed. <https://dotnet.microsoft.com/en-us/download>





## Algorithms and implementation

### Introduction on post processing

While the platform can receive telemetry (ie sensor data points) from any kind of device, there are specific requirements and algorithms that are used to calculate the state of the soil. To run these calculations a complicated post processing algorithm was implemented and is executed every time that telemetry information is received from a 3pro device. The current document describes the details of that implementation.

### Terminology

#### *Rule engine*

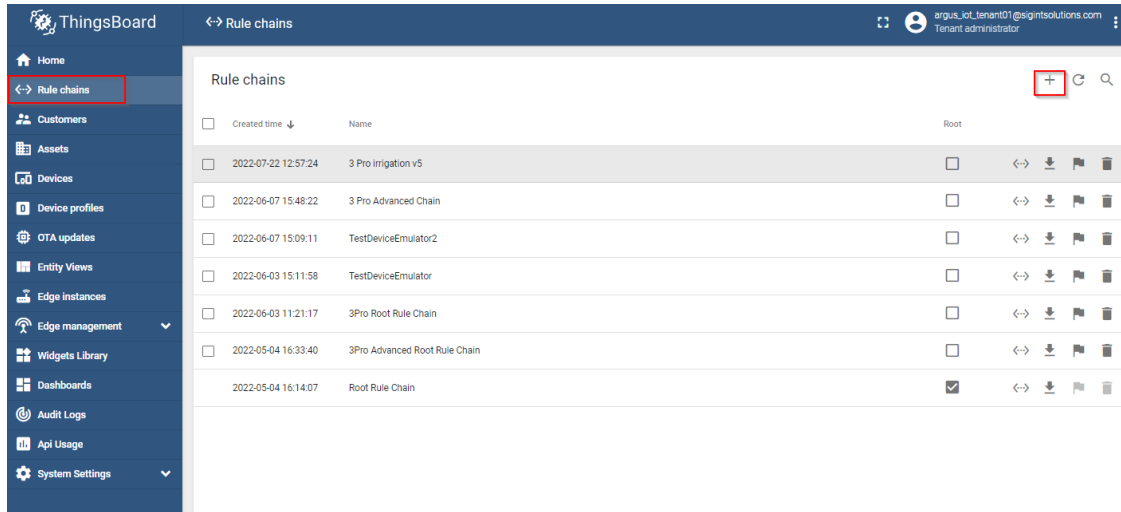
Rule engine is a post processing engine that events from the device or the dashboard or various other inputs are processed. Each processing engine is bound to a specific device profile.

The screenshot shows the ThingsBoard interface with the 'Device profiles' tab selected. The table below lists the device profiles:

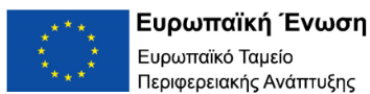
Created time	Name	Profile type
2022-07-18 14:46:16	3 pro irrigation v4	Default
2022-06-29 14:18:57	CE certification Testing	Default
2022-06-09 08:34:07	3 Pro Advanced	Default
2022-05-24 14:29:26	3ProDeviceAdvanced	Default
2022-05-04 16:33:55	3ProDevice	Default
2022-05-04 16:14:07	default	Default

The details for the '3 pro irrigation v4' profile are shown on the right. The 'Rule chain' field is set to '3 Pro irrigation v5'.

Rule chains can be created or set from the Rule chains tab



Rule engine consists of various nodes with various functionalities including script nodes. With the correct combination of nodes the capabilities of the platform can be greatly expanded.



←→ Rule chains > ↔ 3

Search nodes

Filter

- check alarm status
- check existence fields
- check relation
- gps geofencing filter
- message type
- message type switch
- originator type
- originator type switch
- script
- switch

Enrichment

- calculate delta
- customer attributes
- customer details
- originator attributes
- originator fields
- originator telemetry
- related attributes
- related device attribut...



## Mqtt

MQTT is an OASIS standard messaging protocol for the Internet of Things (IoT). It is designed as an extremely lightweight publish/subscribe messaging transport that is ideal for connecting remote devices with a small code footprint and minimal network bandwidth.

The platform support telemetry transmission using the mqtt protocol, as well as http requests.

## Device Properties

### Device profile

A tenant administrator is able to configure common settings for multiple devices using Device profiles. The device profiles can be used in this case to assign specific rule chains to a group of devices.

### Static properties

A Device has various properties. Some of the more mandane properties is the \* device name \* device profile \* device label \* access token. \* ## Device telemetry Besides these static values, the device has telemetry, which corresponds to a sensor measurement.

A telemetry data point is a single measurement of a sensor and has the following properties



## 3 Pro Irrigation V4

Device details

Details   Attributes   **Latest telemetry**   Alarms   Events   Relations   Audit Logs

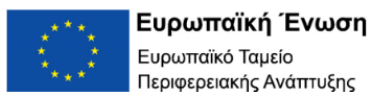
### Latest telemetry

<input type="checkbox"/>	Last update time	Key ↑	Value
<input type="checkbox"/>	2022-07-26 15:52:55	Ambient Temperature	38.5
<input type="checkbox"/>	2022-07-26 15:52:55	Atmospheric Pressure	61.8
<input type="checkbox"/>	2022-07-26 15:52:50	averageSoilMoistureAtDayChange	9.52
<input type="checkbox"/>	2022-07-26 15:52:55	Crop1_AverageSoilMoisture	31.227
<input type="checkbox"/>	2022-07-26 15:52:55	Crop1_IrrigationRequired	false
<input type="checkbox"/>	2022-07-26 15:52:55	Crop1_maxIrrigation_hr	0
<input type="checkbox"/>	2022-07-26 15:52:55	Crop1_maxIrrigation_hrmin	0
<input type="checkbox"/>	2022-07-26 15:52:55	Crop1_maxIrrigation_mm	0
<input type="checkbox"/>	2022-07-26 15:52:50	cropEvapotranspiration_mm	6.97

- **timestamp** The timestamp is saved internally in unix standard time in milliseconds format.
- **key** The key is the name of of the measurement of the sensor. This has to be unique per device. For example this name can be Temperature, or something equivalent.
- **value:** Value is the value of the measurement. internally is stored as string

This values are transmitted from the device using either the mqtt protocol, or http format. The format that needs to be transmitted is described in a following section.

Besides the telemetry that is transmitted from the device, there are also computed values that are saved as a telemetry from the post processing chain.





## Device Attributes

Besides the telemetry, device has also some variables that are called attributes. This come in 3 different flavors: server attributes, client attributes, shared attributes.

All of them are basically static values that are stored in the device entity in the platform, and they can be used to store some constants that are used for calculations on the post processing rule chains, or to display some information to the end user.

Like the telemetry they also have the following properties \* key. The key is the name of the attribute \* value the value of the attribute. In contrast with the telemetry values with are stored internally only as strings, the value of a server attribute can be in different formats like boolean, string, double, etc

Example of the formats that can be used.

```
{
  "firmwareVersion": "v2.3.1",
  "booleanParameter": true,
  "doubleParameter": 42.0,
  "longParameter": 73,
  "configuration": {
    "someNumber": 42,
    "someArray": [1,2,3],
    "someNestedObject": {"key": "value"}
  }
}
```

For 3Pro only server attributes are used.





### 3 Pro Irrigation V4

Device details

Details | Attributes | **Latest telemetry** | Alarms | Events | Relations | Audit Logs

Entity attributes scope  
Server attributes

<input type="checkbox"/>	Last update time	Key ↑	Value	
<input type="checkbox"/>	2022-07-26 16:03:28	active	false	
<input type="checkbox"/>	2022-07-26 15:52:50	averageSoilMoistureAtPreviousDayChange	9.52	
<input type="checkbox"/>	2022-07-22 15:08:36	batteryLimit	30	
<input type="checkbox"/>	2022-07-19 16:07:09	cropCoefficient	0.5	
<input type="checkbox"/>	2022-07-18 16:41:20	cropCoefficientEnd	0.5	
<input type="checkbox"/>	2022-07-21 17:21:30	cropCoefficientInitial	0.36	
<input type="checkbox"/>	2022-07-18 16:41:20	cropCoefficientMid	0.85	
<input type="checkbox"/>	2022-07-18 16:41:20	dayEndLateSeason	1667260800000	

## Dashboard

A dashboard is an interface that is used to display information using to a user using widgets. The widgets can display plots, or static values taken from any type of attributes, or telemetry data points. A dashboard can be shared with one or multiple users, and can display information from one or multiple devices.

## 3pro device requirements

For the post processing engine and the dashboard to process the correct information the device needs a specific configuration, with specific names on the server attributes, as well as the telemetry key names.

## Required server attributes

For the device to operate it needs the following server attributes upon creation. The values that are on the following table are only displayed for example.





Attribute name	Data type	Value	Units	Description
dischargeRate	double	150	L/hr	
elevation	integer	650	m	
enableBatteryAlarm	boolean	TRUE		
batteryLimit	double	20	%	
enableInactivityAlarm	boolean	TRUE		
fieldCapacity	double	28	%	
fieldName	string	Dymes_Plum		
latitude	string		ddg	
longitude	string		ddg	
soilMoistureThreshold	double	23	%	
wettedArea	double	19.63	m <sup>2</sup>	
wiltingPoint	double	18	%	
fieldArea	double	25	m <sup>2</sup>	
inactivityTimeoutMinutes	integer	120	m	
rainPerTick	double	0.2	mm	
cropCoefficientInitial	double	0.35		
cropCoefficientMid	double	0.85		
cropCoefficientEnd	double	0.5		
dayStartInitialStage	integer	1648771200000	unix standard time in ms	
dayStartDevelopmentStage	integer	1648771200000	unix standard time in ms	
dayStartMidSeason	integer	1654041600000	unix standard time in ms	



Attribute name	Data type	Value	Units	Description
dayEndMidSeason	integer	1664582400000	unix standard time in ms	
dayEndLateSeason	integer	1667260800000	unix standard time in ms	
transmissionDuration_m	integer	70		
nameOfTemperatureKey	string	Ambient Temperature		This represents the name of the temperature key that will be used on the calculation of the min and max temperatures.
fieldConfiguration	JSON			A data structure that stores the name of the soil moisture keys that will be used on the calculation of the averages

#### Description of server attributes.

- **dischargeRate:** Discharge of the irrigation supply system (one or more drippers or sprinklers) over the specified field area in L/h
- **elevation:** The elevation of the of the location of the device. Right now it is not used in any calculation.
- **enableBatteryAlarm:** Boolean values that controls if the creation of alarms for low battery on the device will be active.
- **batteryLimit:** The value that will act as a limit to trigger the batteryAlarm
- **enableInactivityAlarm:** Boolean value that control the creation of alarm for device inactivity. If true inactivity alarm will be triggered when the device doesnt send any information in a time period larger than `inactivityTimeoutMinutes``



- **fieldCapacity:** Volumetric soil water content after 24-48 hour drainage of saturated soil (cm<sup>3</sup>\_water/cm<sup>3</sup>\_soil), expressed as percentage, e.g., 24
- **fieldName:** name of the field
- **latitude:** The latitude of the device location in decimal degrees(ddg)
- **longitude:** The longitude of the device location in decimal degrees(ddg)
- **soilMoistureThreshold:** The threshold that will be used to display warning message on the soil moisture.
- **wettedArea:** The area wetted by the irrigation system (m<sup>2</sup>). This area should be smaller than the field area, unless the whole field is flooded.
- **wiltingPoint:** ASK ADRIANA
- **fieldArea:** This is the field area (m<sup>2</sup>) used for the irrigation system discharge. It could be the area of the full field, of a terrace, or of a single tree.
- **inactivityTimeoutMinutes:** is used with conjunction with **enableInactivityAlarm** to trigger an alarm.
- **rainPerTick:** in case a device has water tipping bucket, this is used to calculate the actual value of the rain.
- **cropCoefficientInitial:** Crop coefficient at the start of green up (trees) or between planting and 10% field cover (field crops). The crop coefficients are used for checking the irrigation water needs.
- **cropCoefficientMid:** Crop coefficient for the full maturity stage, starting from near full canopy cover till the aging of the leaves (drying, yellowing).
- **cropCoefficientEnd:** Crop coefficient at the end of the growing season when transpiration stops, such as leaf drop (trees) or harvest (field crops).
- **dayStartInitialStage:** For field crops from planting to 10% field cover, can be skipped for trees, so date is same as start development stage
- **dayStartDevelopmentStage:** For field crops at 10% cover, for trees at leaf out
- **dayStartMidSeason:** Start of effective full cover (70-80%) or heading/flowering for field crops
- **dayEndMidSeason:** Start of crop maturity or leaf drying/coloring
- **dayEndLateSeason:** For field crops at harvest, for trees start of leaf drop, end of irrigation season
- **transmissionDuration\_m:** the logger trasmits telemetry in a predetermined period. This value is used to determine that sensors have transmitted at that predetermined period. This is not related to the inactivity alarm, because the logger maybe active, with some of the critical sensors being faulty.



- `nameOfTemperatureKey`: the name of the telemetry key that is used to calculate the min, max, and mean temperatures.
- `fieldConfiguration` field configuration is a JSON structure that holds information about which telemetry keys are used for the calculation of the soil moisture properties. This is used because for the calculation, each sensor needs additional information that needs to be configured and to be used on the calculation, like the sensor depth. that way the sensor names can be used as variables.

```
{
  "fieldConfiguration": {
    "groupsArray": [
      {
        "Crop1": {
          "species": "Dubium",
          "irrigation": "100%",
          "sensorArray": [
            {
              "name": "",
              "meas": "Soil Vol Water Content 01",
              "soil_thickness": "0.225",
              "weight": "0.5"
            },
            {
              "name": "",
              "meas": "Soil Vol Water Content 02",
              "soil_thickness": "0.20",
              "weight": "0.5"
            },
            {
              "name": "",
              "meas": "Soil Vol Water Content 03",
              "soil_thickness": "0.175",
              "weight": "0.5"
            },
            {
              "name": "",
              "meas": "Soil Vol Water Content 04",
              "soil_thickness": "0.25",
              "weight": "0.5"
            },
            {
              "name": "",
              "meas": "Soil Vol Water Content 05",
```



```

        "soil_thickness": "0.20",
        "weight": "0.5"
    },
    {
        "name": "",
        "meas": "Soil Vol Water Content 06",
        "soil_thickness": "0.175",
        "weight": "0.5"
    }
]
}

```

while the keys can be generated manually via the device page, the easiest way to add them, is to use a REST request with the following format.

The displayed values are only used for example.

```

{
  "cropCoefficient": 0.5,
  "dischargeRate": 150.0,
  "elevation": 650,
  "enableBatteryAlarm": true,
  "batteryLimit": 20.0,
  "enableInactivityAlarm": true,
  "fieldCapacity": 28.0,
  "fieldName": "Dymes_Plum",
  "latitude": 34.911804,
  "longitude": 32.987321,
  "soilMoistureThreshold": 23.0,
  "wettedArea": 19.63,
  "wiltingPoint": 18.0,
  "fieldArea": 25.0,
  "inactivityTimeoutMinutes": 120,
  "rainPerTick":0.2,
  "cropCoefficientInitial":0.35,
  "cropCoefficientMid":0.85,
  "cropCoefficientEnd":0.5,
  "dayStartInitialStage": 1648771200000,
  "dayStartDevelopmentStage": 1648771200000,
  "dayStartMidSeason": 1654041600000,

```



```
"dayEndMidSeason": 1664582400000,
"dayEndLateSeason": 1667260800000,
"transmissionDuration": 70,
"nameOfTemperatureKey": "Ambient Temperature",
"fieldConfiguration": {
  "groupsArray": [
    {
      "Crop1": {
        "species": "Dubium",
        "irrigation": "100%",
        "sensorArray": [
          {
            "name": "",
            "meas": "Soil Vol Water Content 01",
            "soil_thickness": "0.225",
            "weight": "0.5"
          },
          {
            "name": "",
            "meas": "Soil Vol Water Content 02",
            "soil_thickness": "0.20",
            "weight": "0.5"
          },
          {
            "name": "",
            "meas": "Soil Vol Water Content 03",
            "soil_thickness": "0.175",
            "weight": "0.5"
          },
          {
            "name": "",
            "meas": "Soil Vol Water Content 04",
            "soil_thickness": "0.25",
            "weight": "0.5"
          },
          {
            "name": "",
            "meas": "Soil Vol Water Content 05",
            "soil_thickness": "0.20",
            "weight": "0.5"
          },
          {
            "name": "",
            "meas": "Soil Vol Water Content 06",
```





```
"soil_thickness": "0.175",  
"weight": "0.5"  
}  
]  
}  
}  
]  
}  
}
```

[Auto generated server attributes and telemetry.](#)

Besides the attributes referenced above, during operation the rule chain will generate a series of additional server attributes, as well as a series of telemetry

These keys are the following.

- **wettedAreaFraction**: Calculated from  $\frac{wettedArea}{fieldArea}$
- **dayStartInitialStage\_JDN**. Internal conversion from the dates that the user inputs to Day of the year format.
- **dayStartDevelopmentStage\_JDN**
- **dayStartMidSeason\_JDN**
- **dayEndMidSeason\_JDN**
- **dayEndLateSeason\_JDN**
- **rootDepthSum\_m**: The root depth summary. Calculated by the sensor information on the **fieldConfiguration**
- **latitude\_rad**: Internal converted variable of the **latitude ddg**
- **solarDeclination\_rad**: solar declination angle for the current day in rad
- **sunsetHourNumber\_rad**: Hour Angle at Sunset in rad
- **extraterrestrialRadiation\_MJm2day**: the radiation produced from the sun at the top of the atmosphere in units  $MJm^{-2}day^{-1}$
- **memory**: internal value that is used to find the min and max temperature
- **julianDayNumber**: the current day of the year
- **lastSoilMoistureAverageTimestamp**: the timestamp in unix standard time milliseconds of the last time that the averaging of the soil moisture sensor is performed. The reason that this variable is needed is because in some loggers the data are transmitted one at a time.





- `averageSoilMoistureAtPreviousDayChange`: variable that keeps the average soil moisture at the time of change of the previous day. This is used for various calculations in conjunction with the current value.

#### Configuring device

Refer to the configuration application.

#### Telemetry Transmission formats

While thingsboard supports various transmission formats, with or without timestamps, because of the post processing which expects data in a specific format, only 3 formats are available to used.

##### Format A

Format 1 requires all the telemetry data to be packaged on a same json. The following example uses only 2 telemetry keys 'temperature' and 'humidity', but up to 200 telemetry keys can be used at the same time.

Since the timestamp is not included on the package, the message will receive the timestamp from time that the server has.

```
{
  "temperature": 42.2,
  "humidity": 70,
}
```

##### Format B

same as format A, but in this case the timestamp is included on the message from the logger.

```
{
  "ts": 1527863043000,
  "values": {
    "temperature": 42.2,
    "humidity": 70
  }
}
```

##### Format C

In situations where the logger cannot transmit all telemetry values at the same time, A series of telemetry messages can be transmitted as far as they have **the same timestamp**.





The timestamp must be the same or else the rule chain will not be able to recognize that the soil moisture telemetry values belong in same group and thus, can be averaged together

```
{
  "ts": 1527863043000,
  "values": {
    "humidity": 70
  }
}
{
  "ts": 1527863043000,
  "values": {
    "temperature": 42.2,
  }
}
```

[Transmission example](#)

Transmission example using Mqtt using format A, using mosquitto application

```
mosquitto_pub -d -q 1 -h "3PRO_URL" -p 8883 -t "v1/devices/me/telemetry" -u "
DEVICE_ACCESS_TOKEN" -m '{"temperature":40.21,"latitude":35.110492,"longi
tude":33.342518}' --cafile "/mnt/c/1/STAR_sigintsolutions_com/AAACertificat
eServices.crt"
```

This will transmit telemetry data points for 'temperature', 'latitude', 'longitude'

[Setting the device to use the rule chain](#)

Refer to Initializing devices section

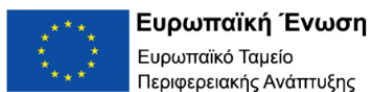
[Mathematical calculations / Theory](#)

[Constants](#)

To perform the calculation there are some specific information that are known for the field.

These are:

- Latitude as Location of station, in decimal degrees
- Longitude as Location of station, in decimal degrees





- `Field_capacity` Field capacity of soil
- `Wilting_point` Wilting point of soil
- `Soil_moisture_threshold`: Soil moisture below which crop becomes stressed, should be displayed in graphs
- `Discharge_rate`: Irrigation discharge rate over field area (1 tree or multiple trees)
- `Field_area` Field area (Dymes: full terrace)
- `Wetted_area`: Field area wetted by irrigation
- `Crop_coefficient_for_initial_stage`: Kc1
- `Crop_coefficient_for_mid_season`: Kc2
- `Crop_coefficient_at_end_season`: Kc3
- `Start_initial_stage` For field crops from planting to 10% field cover, can be skipped for trees, so date is same as start development stage
- `Start_development_stage`: For field crops at 10% cover, for trees at leaf out
- `Start_mid_season`: Start of effective full cover (70-80%) or heading/flowering for field crops represented as day of the year number
- `End_mid_season`: Start of crop maturity or leaf drying/coloring, represented as day of the year number
- `End_late_season`: For field crops at harvest, for trees start of leaf drop, end of irrigation season. represented as day of the year number

There is also the series of sensors that are installed on the field. Each sensor has the following additional properties \* `Soil_Thickness` in m, which represents the thickness of the soil layer that represents the sensor measurement (cm) \* `Sensor_weight` ( $\theta - 1$ )

for example, on a field with 3 sensors on 3 different depths we would have the following properties

`"name": "Sensor 1", "soil_thickness": "0.225", "weight": "0.5"`

`"name": "Sensor 2", "soil_thickness": "0.20", "weight": "0.5"`

`"name": "Sensor 3", "soil_thickness": "0.20", "weight": "0.5"`



### Calculations

- **Latitude** is converted from decimal degrees to rad

$$latitude_{rad} = \frac{\pi}{180} * latitude$$

- **Root zone depth** RootDepth is calculated by the summation of all the sensor properties

$$RootDepth = \sum_{i=1}^n (ST_i * W_i)$$

where:

ST is the Soil thickness

W is the weight of the sensor.

- **Minimum temperature** MinTemp and **Maximum temperature** MaxTemp is the temperature minimum and maximum respectively in the field in 24 hours period
- **Mean Temperature** TemperatureMean is calculated as:
- **Wetted Area Fraction** WettedAreaFraction is calculated as

$$WettedAreaFraction = \frac{WettedArea}{FieldArea}$$

- **Soil Moisture Average** SoilMoistureAverage is calculated using the following equation.

$$SoilMoistureAverage = \frac{\sum_{i=1}^n (ST_i * W_i * SM_i)}{RootDepth}$$

- **Max Irrigation mm** MaxIrrigation\_mm is calculated using the following equation

$$MaxIrrigation_{mm} = 0.1 * (FieldCapacity - SoilMoistureAverage) * RootDepth * 100$$

- **Max Irrigation Hr** MaxIrrigation\_Hris calculated using the following equation

$$MaxIrrigation_{hr} = MaxIrrigation_{mm} * \frac{FieldArea}{DischargeRate}$$

- **Max Irrigation Hr.Min** MaxIrrigation\_HrMin is calculated using the following equation.



$$\begin{aligned}
 & \text{MaxIrrigation}_{HrMin} \\
 &= \text{MaxIrrigation}_{hr} * [\text{MaxIrrugation}_{hr}] \\
 &+ (\text{maxIrrigation}_{hr} - [\text{MaxIrrugation}_{hr}]) * \frac{60}{100}
 \end{aligned}$$

- **Solar Declination**  $SolarDeclination$  or  $\delta$  is calculated using the following equation

$$\delta = -23.45^\circ \times \cos\left(\frac{360}{365} \times (d + 10)\right)$$

where:

- the  $d$  is the number of days since the start of the year (julian day number)
- The declination angle equals zero at the equinoxes (March 22 and September 22), positive during the summer in northern hemisphere and negative during winter in the northern hemisphere. The declination reaches a maximum angle on June 22 which is  $23.45^\circ$  (the northern hemisphere summer solstice) and a minimum angle on December 21-22 which is of  $-23.45^\circ$  (the northern hemisphere winter solstice).
- In the above equation, the +10 is due to the fact that the winter solstice occurs before the start of the year.
- The equation also assumes the orbit of the sun to be a perfect circle and the fraction of  $360/365$  converts the number of days to the position in the orbit.
- The apparent northward movement of the Sun during the northern spring, reaching the celestial equator during the March equinox. The declination reaches a maximum angle equal to the axial tilt of the Earth's axial tilt ( $23.44^\circ$ ) on the June solstice, then starts decreasing until reaching its minimum ( $-23.44^\circ$ ) on the December solstice, where its value is equal to the negative of the axial tilt. Seasons are a direct product of this variation.

It is then converted to rad using

$$SolarDeclination_{rad} = \frac{\pi}{180} * SolarDeclination$$

- **Sunset hour angle** is  $SunsetHourAngle_{rad}$  calculated using the following equation

$$SunsetHourAngle_{rad} = \arccos(-\tan(latitude_{rad}) * \tan(solarDeclination_{rad}))$$

- Extraterrestrial radiation  $ExtRadiation$  is calculated using the following equation.



$$ExtRadiation = (24 * \frac{60}{\pi}) * solarConstant * InverseRelativeDistanceEarthSun * ((\sin(SunsetHourAngle_{rad}) * \sin(latitude_{rad}) * \sin(solarDeclination_{rad})) + (\cos(latitude_{rad}) * \cos(solarDeclination_{rad}) * \sin(SunsetHourAngle_{rad})));$$

where:

$$SolarConstant = 0.082MJM^{-2}min^{-1}$$

$$InverseRelativeDistanceEarthSun = 1 + 0.033 * \cos\left(\frac{2\pi}{365} * DayNumber\right)$$

- **Actual Crop Coefficient**, kc is calculated using the following equation

if DayNumber is less than Start initial stage or more than End late season, then the crop coefficient is assumed as zero because there is no growth of the crop.

When is between Start initial stage and Start development stage, then Kc1 is used. Between Start development stage and Start mid season, a linear interpolation is performed between Kc1 and Kc2. When is between Start mid season and End mid season, then Kc2 is used. Finally when between End mid season and End late season, a linear interpolation is performed between Kc2 and Kc3.

- **reference evapotranspiration**  $ET_{o\_mm}$  is calculated using the following equation

$$ET_{o\_mm} = 0.0023 * (TemperatureMean + 17.8) * \sqrt{temperatureMax - temperatureMin} * 0.408 * ExtRadiation;$$

- **crop evapotranspiration**  $ET_{c\_mm}$  is calculated using the following equation.

$$ET_{c\_mm} = Kc * ET_{o\_mm}$$

## Description of implementation

Since not all the values are available at a single time in the actual platform, there are various workarounds and tricks that were used to implement the above algorithm. The next section discusses details about this implementation so any potential developer can modify the chains in the future.

### Explanations of nodes

#### Switch node

```
function Switch(msg, metadata, msgType): string[]
```



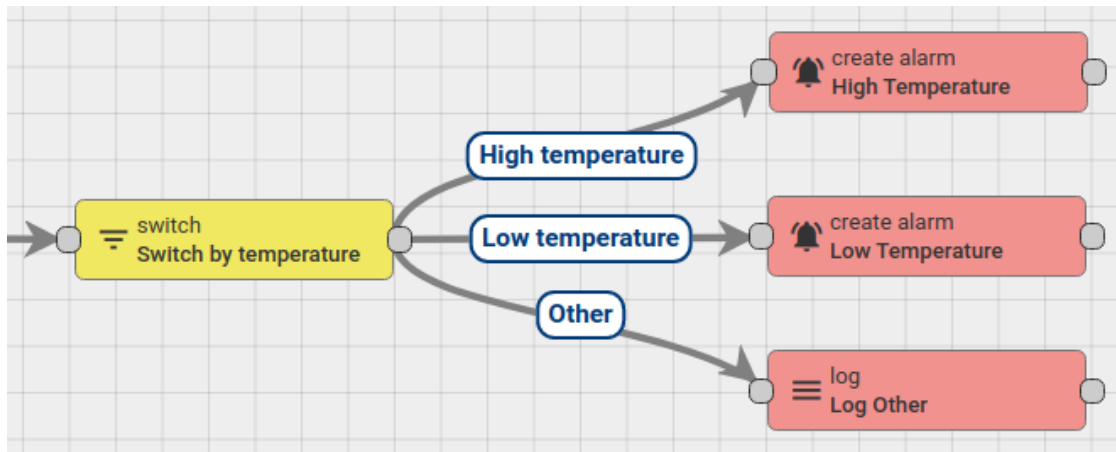


JavaScript function computing an array of Link names to forward the incoming Message.

**Returns:** Should return an array of string values presenting link names that the Rule Engine should use to further route the incoming Message.

**Example** Forward all messages with temperature value greater than 30 to the 'High temperature' chain, with temperature value lower than 20 to the 'Low temperature' chain and all other messages to the 'Other' chain:

```
if (msg.temperature > 30)
{
  return ['High temperature'];
} else if (msg.temperature < 20)
{
  return ['Low temperature'];
} else
{
  return ['Other'];
}
```



[Script node](#)

```
function Filter(msg, metadata, msgType): boolean
```

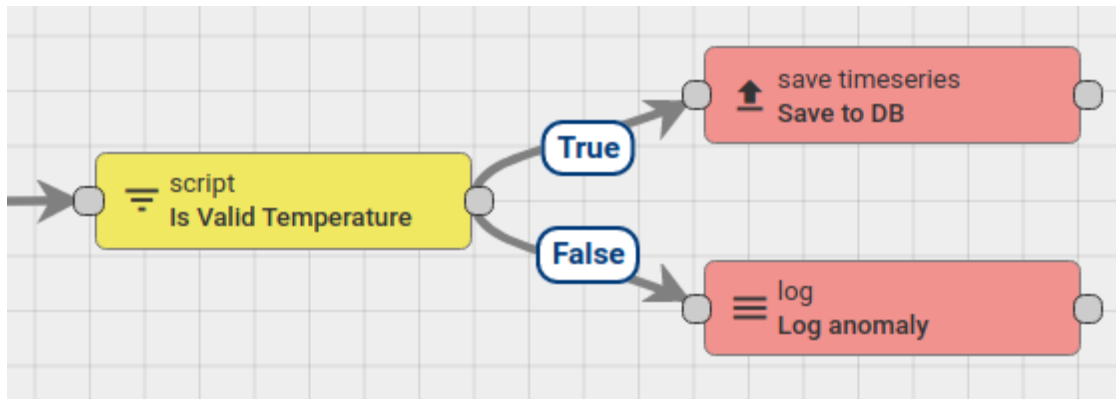
JavaScript function defines a boolean expression based on the incoming Message and Metadata.

**Returns:** a boolean value. If true - routes Message to subsequent rule nodes that are related via True link, otherwise sends Message to rule nodes related via False link. Uses 'Failure' link in case of any failures to evaluate the expression.



**Example** Forward all messages with temperature value greater than 20 to the True link and all other messages to the False link. Assumes that incoming messages always contain the 'temperature' field:

```
return msg.temperature > 20;
```



#### Transform Script Node

```
function Transform(msg, metadata, msgType): {msg: object, metadata: object, msgType: string}
```

The JavaScript function to transform input Message payload, Metadata and/or Message type to the output message.

**Returns:** Should return the object with the following structure:

```
{
  msg?: {[key: string]: any},
  metadata?: {[key: string]: string},
  msgType?: string
}
```

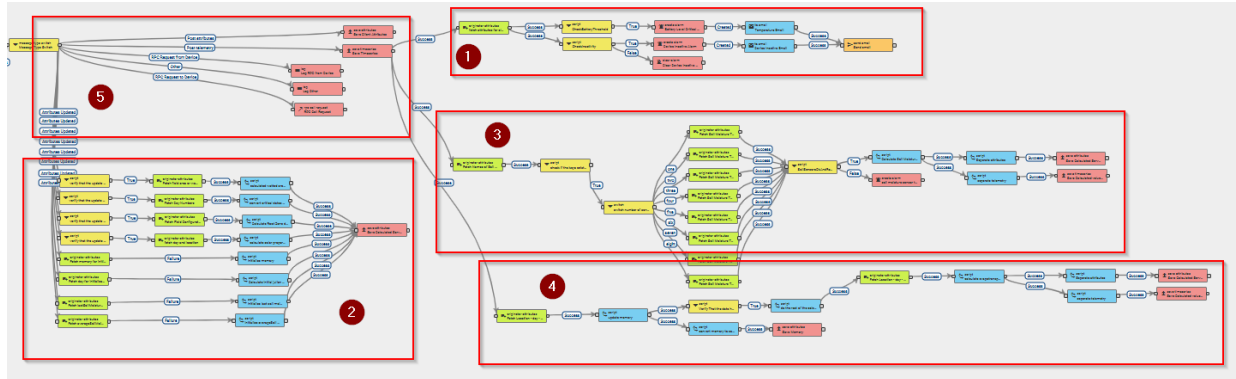
All fields in resulting object are optional and will be taken from original message if not specified.

**Example** Transform value of the 'temperature' field from °F to °C:

```
msg.temperature = (msg.temperature - 32) * 5 / 9;
return {msg: msg};
```



## Implementation

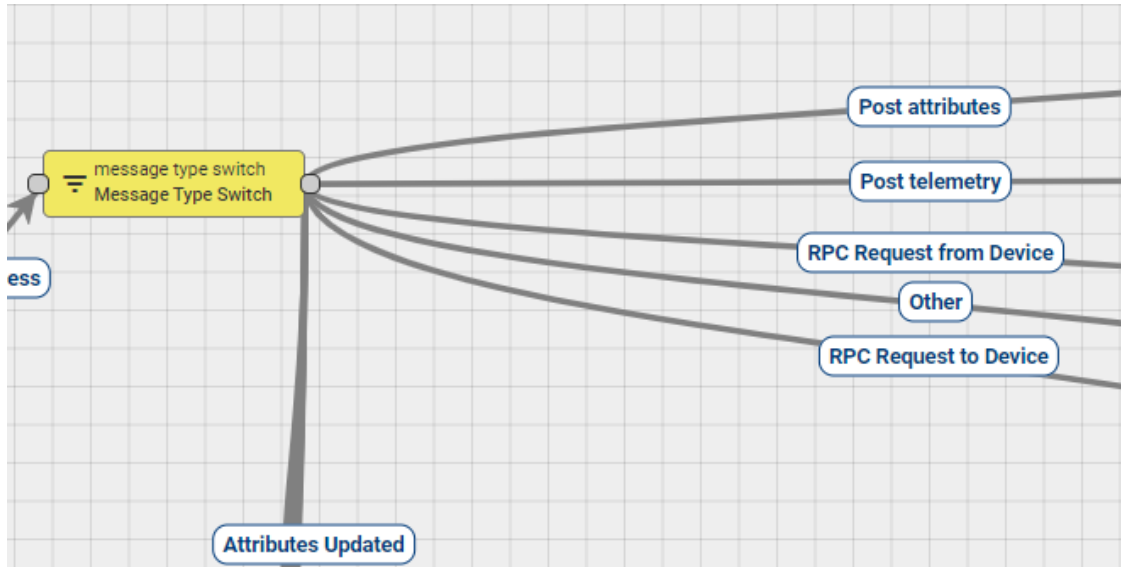


The rule chain consists of 5 different group of nodes that perform various operations.

All the messages have 3 basic properties as they come into the rule chain. \* msg. This is in a json format, and can contain telemetry data or other information depending the message format. \* msgType The type of the message. For example when telemetry is incoming the message type will be POST\_TELEMETRY, and when the attributes are updated from the user the message type will be ATTRIBUTES\_UPDATED \* metadata various additional information that can be included on the message, like the timestamp and the device name.

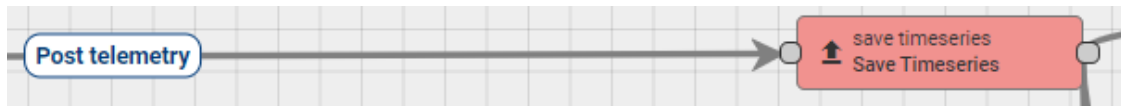
### Group 5, Input

Not a lot need to be set for this particular block. At the first level we have the input node, which takes all the messages that originate for a particular device. These messages can be telemetry that originates from the actual physical device, or it can be an attribute update that originates from the device dashboard and is triggered by the user. After the message has been received, there is a node that switches the message direction and which nodes will be triggered downstream, depending the message type.



The most critical options in this situation is the POST\_TELEMETRY and the ATTRIBUTES\_UPDATED.

When the message type is POST\_TELEMETRY the message is guided in the save timeseries node



The telemetry values are then stored in the database for the amount of time that it has been indicated on the node settings

# Save Timeseries

Action - save timeseries

Details Events Help

Name \*

Save Timeseries

Default TTL in seconds \*

63072000

Skip latest persistence

Use server ts

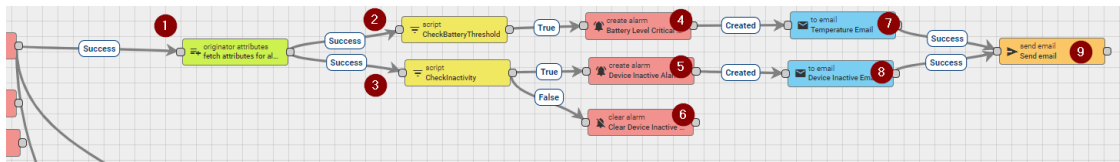
Enable this setting to use the timestamp of the message processing (ir etc).

Description

In this case it is 63072000 seconds which corresponds to 2 years of historical data. This means that 2 years after the record of a datapoint, that datapoint will get purged.

### Group 1, Alarms

Group 1 is a collection of nodes which generates some alarms for the device, like alarm for device inactivity (ie, device is offline)



### Originator attributes [1] node

After the values are saved in the database, an originator attributes [1] node, fetches specific server attributes from the device.



more specifically it fetches \* enableBatteryAlarm \* enableInactivityAlarm \* batteryLimit \* inactivityTimeoutMinutes

from the server attributes that were described in the section mentioning the Required server attributes

### fetch attributes for alarms

Enrichment - originator attributes

Details Events Help

Name \*

fetch attributes for alarms

Tell Failure

If at least one selected key doesn't exist the outbound message will report "Failure".

Client attributes

Client attributes

Hint: use `#{metadataKey}` for value from metadata. `#{messageKey}` for value from message body

Shared attributes

Shared attributes

Hint: use `#{metadataKey}` for value from metadata. `#{messageKey}` for value from message body

Server attributes

enableBatteryAlarm × enableInactivityAlarm × batteryLimit × inactivityTimeoutMinutes × Server attributes

Hint: use `#{metadataKey}` for value from metadata. `#{messageKey}` for value from message body

Latest timeseries

Latest timeseries

Hint: use `#{metadataKey}` for value from metadata. `#{messageKey}` for value from message body

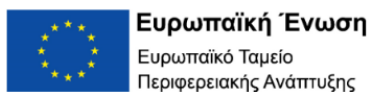
If the attributes are not present in the device, the node will report failure. In this case this should happen only if the device is misconfigured during initialization.

These nodes are used throughout the rule chain to pull various information when needed. and after they are pulled, they are stored on the metadata object.

To give an example, all the messages that enter a node have a type IN and a type of OUT during the exit.

in this case, on input the message had the following properties.

```
{  
  "msg": {  
    "Ambient Temperature": 25.9,  
  }  
}
```





```

    "Atmospheric Pressure": 88,
    "Luminosity": 51.3,
    "Soil Temperature 01": 26,
    "Soil Temperature 02": 20.2,
    "Soil Temperature 03": 24.9,
    "Soil Temperature 04": 10.9,
    "Soil Temperature 05": 18,
    "Soil Temperature 06": 19.8,
    "Soil Vol Water Content 01": 20.2,
    "Soil Vol Water Content 02": 23.7,
    "Soil Vol Water Content 03": 36.6,
    "Soil Vol Water Content 04": 3.5,
    "Soil Vol Water Content 05": 18.5,
    "Soil Vol Water Content 06": 40.1
  },
  "metadata" : {
    "deviceName": "3 Pro Irrigation V4",
    "deviceType": "3 pro irrigation v4",
    "ts": "1658923046911"
  }
}

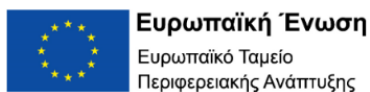
```

on output the server attributes has been pulled successfully from the database or the device properties, and the message is modified accordingly.

```

{
  "msg": {
    "Ambient Temperature": 25.9,
    "Atmospheric Pressure": 88,
    "Luminosity": 51.3,
    "Soil Temperature 01": 26,
    "Soil Temperature 02": 20.2,
    "Soil Temperature 03": 24.9,
    "Soil Temperature 04": 10.9,
    "Soil Temperature 05": 18,
    "Soil Temperature 06": 19.8,
    "Soil Vol Water Content 01": 20.2,
    "Soil Vol Water Content 02": 23.7,
    "Soil Vol Water Content 03": 36.6,
    "Soil Vol Water Content 04": 3.5,
    "Soil Vol Water Content 05": 18.5,
    "Soil Vol Water Content 06": 40.1
  },
  "metadata" : {

```





```
"deviceName": "3 Pro Irrigation V4",  
"deviceType": "3 pro irrigation v4",  
"ss_batteryLimit": "30",  
"ss_enableBatteryAlarm": "true",  
"ss_enableInactivityAlarm": "true",  
"ss_inactivityTimeoutMinutes": "120",  
"ts": "1658923046911"  
}  
}
```

one thing to note, is the `ss_` suffix. This indicates that this was a server attribute, and they are stored that way automatically on the metadata.

The in and out values can be examined by enabling the debug mode on each node.

fetch attributes for alarms  
Enrichment - originator attributes

Details Events Help

Name \*  
fetch attributes for alarms

Debug mode

Tell Failure  
If at least one selected key doesn't exist the outbound message will report "Failure".

Then on next transmission can be examined on the events tab

fetch attributes for alarms  
Enrichment - originator attributes

Details **Events** Help

Event type  
Debug

last day

Event time	Server	Type	Entity type	Message Id	Message Type	Relation Type	Data	Metadata	Error
2022-07-27 14:57:26	c65f422a7d17	IN	DEVICE	55a46b83-7526-4f15-b...	POST_TELEMETRY_RE...		...	...	
2022-07-27 14:57:26	c65f422a7d17	OUT	DEVICE	55a46b83-7526-4f15-b...	POST_TELEMETRY_RE...	Success	...	...	

## Script node [2]

A script node has the ability to execute custom logical operations on the incoming data, and returns a true or false, which can be used to trigger other nodes downstream.

Script node executers javascript code.



### CheckBatteryThreshold

Filter - script

Details Events Help

Name \*

CheckBatteryThreshold

Filter

```
function Filter(msg, metadata, msgType) {  
  1 function stringToBool(str)  
  2 {  
  3   return (String(str).toLowerCase() == "true");  
  4 }  
  5  
  6  
  7 //alarm is stored as string "true". We need to parse it first.  
  8 if (stringToBool(metadata.ss_enableBatteryAlarm))  
  9 {  
 10   //if message doesn't have that key property then there is no comparison that can be done.  
 11   if (msg.hasOwnProperty('Battery Percentage'))  
 12  
 13 }  
}
```

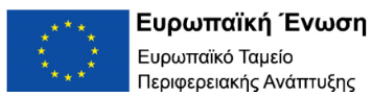
Test filter function

Description

in this case the script node is connected via a success relation on the previous node [1] and runs a check to check if the battery is in a critical level.

The script is the following.

```
function stringToBool(str)  
{  
  return (String(str).toLowerCase() == "true");  
}  
  
//alarm is stored as string "true". We need to parse it first.  
if (stringToBool(metadata.ss_enableBatteryAlarm))  
{  
  //if message doesn't have that key property then there is no comparison that can be done.  
  if (msg.hasOwnProperty('Battery Percentage'))  
  {  
    if (parseFloat(msg['Battery Percentage']) <= parseFloat(metadata.ss_batteryLimit))  
    {  
      return true;  
    }  
  }  
}
```





```
}  
return false;
```

if the node returns true it means that an alarm has to be created. Nodes [4], [7], [8] finish the creation of the alarm as well as the email message that will be transmitted.

#### Script node [3]

Similarly script node 3 will be triggered after the originator attribute [1] node, and will execute a script that will return true when the device is inactive for a period longer than the configured device timeout.

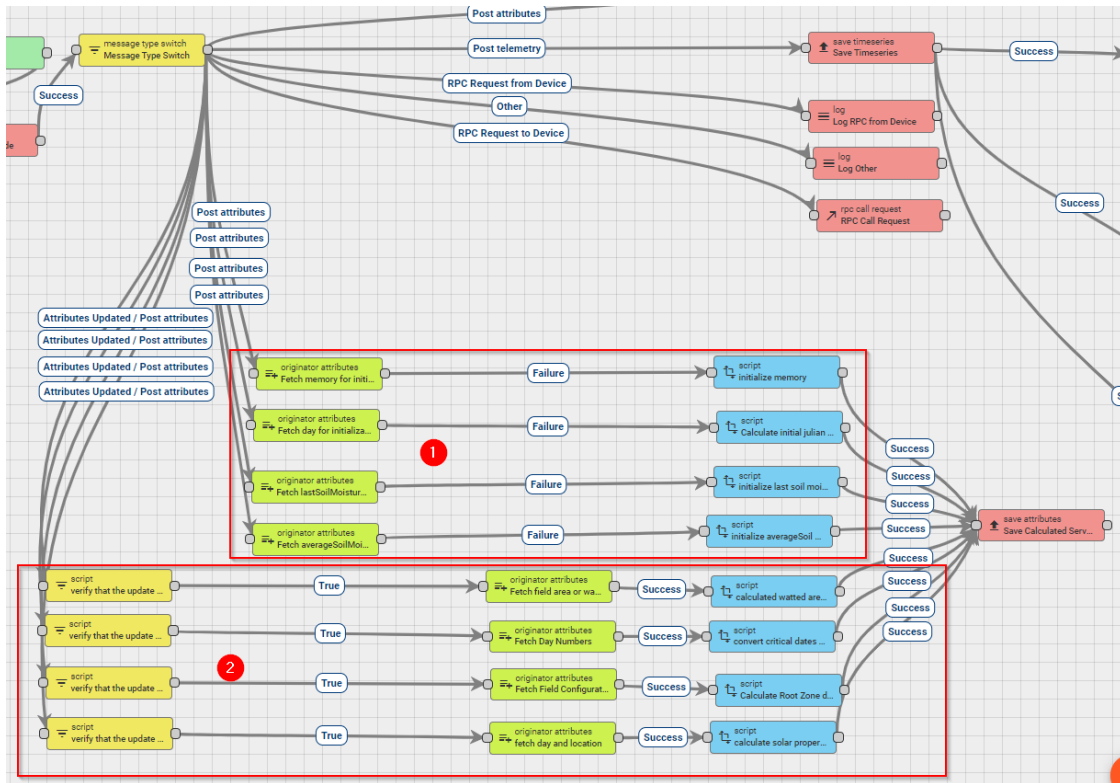
```
function stringToBool(str)  
{  
    return (String(str).toLowerCase() == "true");  
}  
  
if (stringToBool(metadata.ss_enableInactivityAlarm))  
{  
    //one minute in ms is 60000. Date.now() as well as the rest of the timest  
    //amps are stored is unix standard time in milliseconds internally.  
    if ((Date.now() + (parseFloat(metadata.ss_inactivityTimeoutMinutes) * 60  
000)) > parseFloat(metadata.ss_lastActivityTime))  
    {  
        return true;  
    }  
}  
return false;
```

if the node returns true it means that an alarm has to be created. Nodes [5], [6], [8], [9] finish the creation of the alarm as well as the email message that will be transmitted.

#### Group 2, Server attributes from dashboard configuration node

The second group is related to the update of the calculated server attributes. The update of the attributes must be executed when the device is initially configured, as well as when the user changes a relevant setting from the dashboard. The message types will be POST\_ATTRIBUTES & ATTRIBUTES\_UPDATED respectfully. In both situations the nodes in Group 2 must be triggered, with some exception which would be explained later in this section.





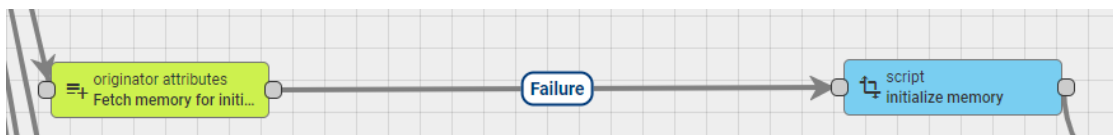
This Group is also divided in 2 different groups.

Subgroup 2.1, is triggered when the message is of type POST\_ATTRIBUTES. This type of message will only happen on device configuration, where all the server attributes for the device will be posted to the device using a helper application.

In this case, this means that several of teh calculations that calculate the additional server attributes must be performed.

[Subgroup 2.1 description.](#)

[Memory server attribute initialization.](#)



An originator attributes node, is trying to fetch the memory attribute. Since the device is just initialized, this means that this server attribute is not available on the database, and the node will report failure.



upon failure, it will trigger a node transformation script, which will create that variable. The transformation script runs scripts in javascript language.

## initialize memory

Transformation - script

Details Events Help

Name \*  
initialize memory

Transform

```
function Transform(msg, metadata, msgType) {
```

```
1 var newMsg = {};  
2  
3 //change type to post attributes  
4 var msgType = "POST_ATTRIBUTES_REQUEST";  
5  
6  
7 newMsg.memory = {  
8   "currentTemperatureMin": 100,  
9   "currentTemperatureMax": -100,  
10  "rainTicksSummary": 0,  
11 };  
12
```

Test transformer function

Description

```
var newMsg = {};
```

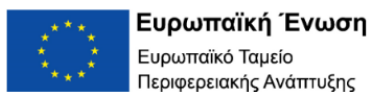
```
//change type to post attributes so that it can be stored as attribute downstream.
```

```
var msgType = "POST_ATTRIBUTES_REQUEST";
```

```
//this will create the memory server attribute with the subproperties initialized to unreasonable values.
```

```
//they will be overwritten with actual values at the first telemetry transmission.
```

```
newMsg.memory = {  
  "currentTemperatureMin": 100,  
  "currentTemperatureMax": -100,
```





```
"rainTicksSummary": 0,  
};  
  
return {msg: newMsg, metadata: metadata, msgType: msgType};
```

This will return a message like this.

```
{  
  "msg": {  
    "currentTemperatureMin": 100,  
    "currentTemperatureMax": -100,  
    "rainTicksSummary": 0,  
  },  
  "metadata" : {}  
}
```

that message will be transferred to a `save attributes` node, and since the type of the message that was just created is of type `POST_ATTRIBUTES_REQUEST`, it will be stored as server attribute on that device.



`julianDayNumber` server attribute initialization.



An `originator attributes` node, is trying to fetch the `julianDayNumber` attribute. Since the device is just initialized, this means that this server attribute is not available on the database, and the node will report failure.

upon failure, it will trigger a node transformation `script`, which will create that variable.

below is the code for that transformation script

```
var newMsg = {};  
  
//if the previous node failed, that means that julian day number wasn't avail  
able,
```



```
//ie, this is the first time that the device is ininialized.

//helper functions
//returns true if the year is Leap.
Date.prototype.isLeapYear = function ()
{
    var year = this.getFullYear();
    if ((year & 3) != 0) return false;
    return ((year % 100) != 0 || (year % 400) == 0);
};

// Get Day of Year
Date.prototype.getDOY = function ()
{
    //array that holds the day count on each month
    var dayCount = [0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334];
    var mn = this.getMonth();
    var dn = this.getDate();
    var dayOfYear = dayCount[mn] + dn;
    //increase by one if the year is Leap.
    if (mn > 1 && this.isLeapYear())
    {
        dayOfYear++;
    }
    return dayOfYear;
};

function DegToRad(degrees)
{
    return degrees * (Math.PI / 180.0);
}

var currentDate = new Date();
newMsg.julianDayNumber = currentDate.getDOY();

//change type to post attributes
var msgType = "POST_ATTRIBUTES_REQUEST";

return {msg: newMsg, metadata: metadata, msgType: msgType};
```

lastSoilMoistureAverageTimestamp server attribute initialization.

An originator attributes node, is trying to fetch the lastSoilMoistureAverageTimestamp attribute. Since the device is just initialized, this



means that this server attribute is not available on the database, and the node will report failure. which will trigger the execution of another script node.



```
var newMsg = {};
```

```
//change type to post attributes
```

```
var msgType = "POST_ATTRIBUTES_REQUEST";
```

```
newMsg.lastSoilMoistureAverageTimestamp = 0;
```

```
return {msg: newMsg, metadata: metadata, msgType: msgType};
```

averageSoilMoistureAtPreviousDayChange server attribute initialization.

An originator attributes node, is trying to fetch the averageSoilMoistureAtPreviousDayChange attribute. Since the device is just initialized, this means that this server attribute is not available on the database, and the node will report failure. which will trigger the execution of another script node.

```
var newMsg = {};
```

```
//change type to post attributes
```

```
var msgType = "POST_ATTRIBUTES_REQUEST";
```

```
newMsg.averageSoilMoistureAtPreviousDayChange = 0;
```

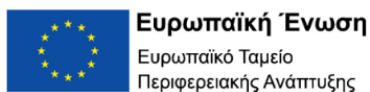
```
return {msg: newMsg, metadata: metadata, msgType: msgType};
```

### Subgroup 2.2 description

Subgroup 2.2 is also triggered upon the first initialization of the device; it calculates the constant server attributes which depend on the configurable server attributes.

Besides that first initialization though, the same calculations have to be performed upon when the user changes these values in the dashboard.

### Calculating wetted area fraction





Calculating wetted area fraction requires the server property `fieldArea` or `wettedArea`. So the chain starts by checking if the `ATTRIBUTES_UPDATED` message has a change on these 2 properties.

The chain starts with a script node with the following code.

```
if (msg.hasOwnProperty('fieldArea') || msg.hasOwnProperty('wettedArea'))
{
    return true;
}
return false;
```

On true it will trigger an originator node which will pull both of these field from the server, and will place them on the metadata. The reason why the values are pulled from the database instead of using them directly from the message is because if the user updated these properties, the message contains one or the other, not both of them. One other note is that since the event is `ATTRIBUTES_UPDATED`, the values have already been updated on the server at that point. (so the values that will be pulled by the originator attributes node are already up to date. )

the originator attributes node triggers a transformation script node on success with the following code.

```
//requires the field configuration data as well as
//ss_fieldCapacity
//ss_wettedArea
//ss_soilMoistureThreshhold
//ss_dischargeRate
var newMsg = {};

//---calculate wet area fraction
newMsg.wettedAreaFraction = parseFloat(metadata.ss_wettedArea) / parseFloat(metadata.ss_fieldArea);
newMsg.wettedAreaFraction = parseFloat(newMsg.wettedAreaFraction.toFixed(3));

//change type to post attributes
var msgType = "POST_ATTRIBUTES_REQUEST";
return {msg: newMsg, metadata: metadata, msgType: msgType};
```

this creates or updates the server attribute `wettedAreaFraction`



## Calculating critical dates



To avoid any issues of parsing dates, or making the user do calculations, the critical dates that correspond to the beginning of the crops, are stored internally as unix standard time in ms, and they are selected by the user using standard DateTimeSelectors widget.



<b>Start initial stage</b> Date * 01/04/2022 ✓ ✕
<b>Start development stage</b> Date * 01/04/2022 ✓ ✕
<b>Start mid season</b> Date * 01/06/2022 ✓ ✕
<b>End mid season</b> Date * 01/10/2022 ✓ ✕
<b>End late season</b> Date * 01/11/2022 ✓ ✕

This means that every time that any of these values change by the user, they have to be converted to Number of day of the year format







This is accomplished by a script node with the following code.

```
if (msg.hasOwnProperty('dayStartInitialStage') ||
    msg.hasOwnProperty('dayStartDevelopmentStage') ||
    msg.hasOwnProperty('dayStartMidSeason') ||
    msg.hasOwnProperty('dayEndMidSeason') ||
    msg.hasOwnProperty('dayEndLateSeason'))
{
    return true;
}
return false;
```

this return true when a relevant property is updated.

This will trigger an originator attributes node which will pull all 5 server attributes.

which in turn, will trigger another transformation script node with the following code.

```
var newMsg = {};

//---calculate the dates as Julian date numbers

//calculate the julian dates for the important dates
var tempDate = new Date(parseInt(metadata.ss_dayStartInitialStage));
newMsg.dayStartInitialStage_JDN = tempDate.getDOY();

tempDate = new Date(parseInt(metadata.ss_dayStartDevelopmentStage));
newMsg.dayStartDevelopmentStage_JDN = tempDate.getDOY();

tempDate = new Date(parseInt(metadata.ss_dayStartMidSeason));
newMsg.dayStartMidSeason_JDN = tempDate.getDOY();

tempDate = new Date(parseInt(metadata.ss_dayEndMidSeason));
newMsg.dayEndMidSeason_JDN = tempDate.getDOY();

tempDate = new Date(parseInt(metadata.ss_dayEndLateSeason));
newMsg.dayEndLateSeason_JDN = tempDate.getDOY();

//change type to post attributes
var msgType = "POST_ATTRIBUTES_REQUEST";

return {msg: newMsg, metadata: metadata, msgType: msgType};
```



which will update (or create) the following properties. \* dayStartInitialStage\_JDN \* dayStartDevelopmentStage\_JDN \* dayStartMidSeason\_JDN \* dayEndMidSeason\_JDN \* dayEndLateSeason\_JDN

### Calculating root zone depth



To calculate the root zone depth, all the information about the soil moisture sensor on the field are needed.

A script node checks if the update is related to the fieldConfiguration which contains the sensor information.

```
if (msg.hasOwnProperty('fieldConfiguration'))
{
    return true;
}
return false;
```

on true an originator attributes node is fetching the fieldConfiguration server attribute from the database.

on success, a transformation script is triggered which loops though all the sensor properties and calculates the rootZoneDepth.

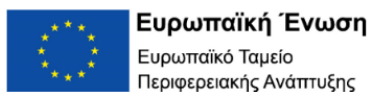
```
var newMsg = {};

//change type to post attributes
var msgType = "POST_ATTRIBUTES_REQUEST";

var fieldConfiguration = JSON.parse(metadata.ss_fieldConfiguration);

var index = 0;
var name = Object.keys(fieldConfiguration.groupsArray[index]);
var fieldConfiguration = JSON.parse(metadata.ss_fieldConfiguration);
var numberOfSensors = Object.keys(fieldConfiguration.groupsArray[index][name].sensorArray).length;
var rootDepthSum_m = 0;

for (var sensorIndex = 0; sensorIndex < numberOfSensors; sensorIndex++)
{
    var sensor = fieldConfiguration.groupsArray[index][name].sensorArray[sens
```





```
orIndex];
    rootDepthSum_m += sensor.soil_thickness * sensor.weight;
}

newMsg.rootDepthSum_m = rootDepthSum_m;

return {msg: newMsg, metadata: metadata, msgType: msgType};
```

This creates or updates the rootDepthSum\_m server attribute.

### Calculating solar properties

The solar properties are dependent on the location of the field. More specifically they are dependent on the latitude of the sensors.

a script checks that an update has been performed on the latitude server attribute

```
if (msg.hasOwnProperty('latitude'))
{
    return true;
}
return false;
```

on true an originator attributes node fetches the julianDayNumber as well as the latitude from the database.

on success a transformation script node is triggered with the following code.

```
function DegToRad(degrees)
{
    return degrees * (Math.PI / 180.0);
}
```

```
var newMsg = {};
```

```
//also calculate the solar declination and some other things
//---calculate solar properties
//The following equation can be used to calculate the declination angle:  $\delta = -23.45^\circ \times \cos(360/365 \times (d+10))$ 
// where the d is the number of days since the start of the year (ie julian day number) The declination angle equals zero at the equinoxes
 //(March 22 and September 22), positive during the summer in northern hemisphere and negative during winter in the northern hemisphere.
//The declination reaches a maximum angle on June 22 which is 23.45° (the no
```



```
rthern hemisphere summer solstice) and a minimum angle
//on December 21-22 which is of  $-23.45^\circ$  (the northern hemisphere winter solst
ice).
//In the above equation, the +10 is due to the fact that the winter solstice
occurs before the start of the year.
// The equation also assumes the orbit of the sun to be a perfect circle and
the fraction of 360/365
//converts the number of days to the position in the orbit. The apparent nort
hward movement of the Sun during the northern spring,
// reaching the celestial equator during the March equinox. The declination r
eaches a maximum angle equal to the axial
//tilt of the Earth's axial tilt ( $23.44^\circ$ ) on the June solstice, then starts d
ecreasing until reaching its minimum ( $-23.44^\circ$ )
//on the December solstice, where its value is equal to the negative of the a
xial tilt. Seasons are a direct product of this variation.

//---calculate latitude in rad
var latitude_rad = DegToRad(parseFloat(metadata.ss_latitude));

var solarDeclination_rad = DegToRad(-23.45) * Math.cos(DegToRad(360.0 / 365.0
* (parseInt(metadata.ss_julianDayNumber) + 10)));
var sunsetHourNumber_rad = Math.acos(-Math.tan(latitude_rad) * Math.tan(solar
Declination_rad));

var solarConstant_MJM2min = 0.082;
var inverseRelativeDistanceEarthSun = 1 + 0.033 * Math.cos(((2 * Math.PI) / 3
65) * parseInt(metadata.ss_julianDayNumber));

var extraterrestrialRadiation_MJM2day = (24 * 60 / Math.PI) * solarConstant_M
JM2min * inverseRelativeDistanceEarthSun *
((sunsetHourNumber_rad * Math.sin(latitude_rad) * Math.sin(solarDeclinati
on_rad)) +
(Math.cos(latitude_rad) * Math.cos(solarDeclination_rad) * Math.sin(s
unsetHourNumber_rad)));

//add them on message
newMsg.latitude_rad = parseFloat(latitude_rad.toFixed(4));
newMsg.solarDeclination_rad = parseFloat(solarDeclination_rad.toFixed(4));
newMsg.sunsetHourNumber_rad = parseFloat(sunsetHourNumber_rad.toFixed(4));
newMsg.extraterrestrialRadiation_MJM2day = parseFloat(extraterrestrialRadiati
on_MJM2day.toFixed(4));

//change type to post attributes
var msgType = "POST_ATTRIBUTES_REQUEST";
```

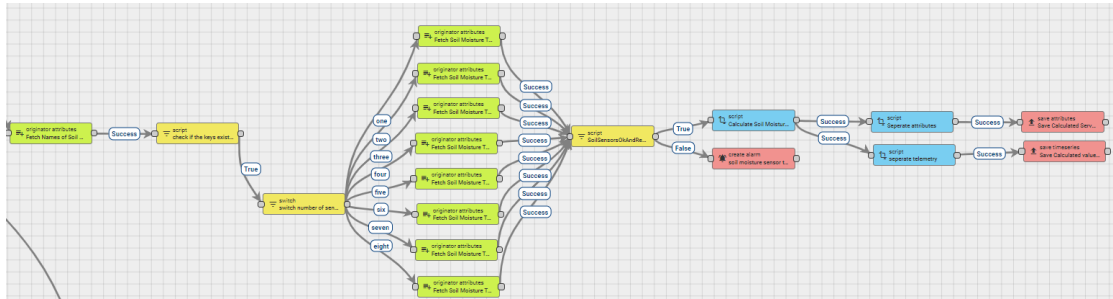
```
return {msg: newMsg, metadata: metadata, msgType: msgType};
```

This will create or update the following server attributes.

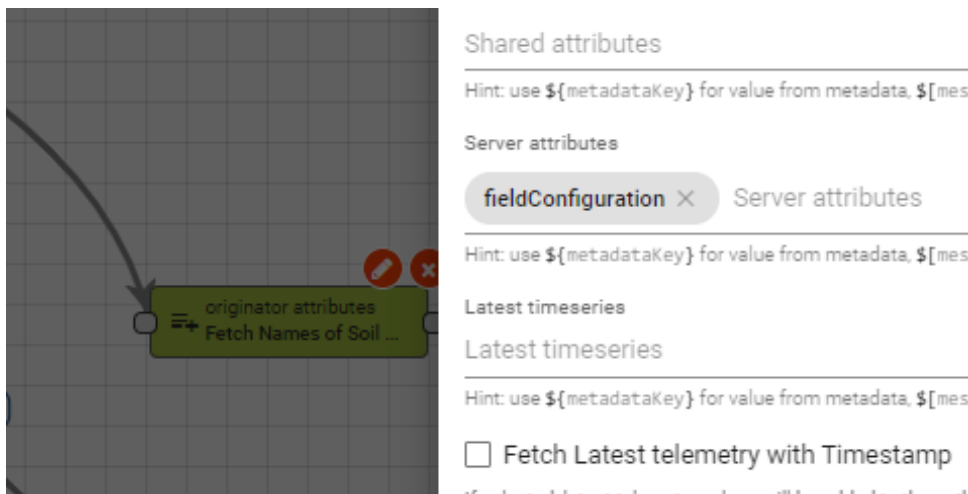
- latitude\_rad
- solarDeclination\_rad
- sunsetHourNumber\_rad
- extraterrestrialRadiation\_MJm2day

### Group 3, Soil average calculation

Group 3 is related to the calculation of the averaging of the soil moisture sensors.



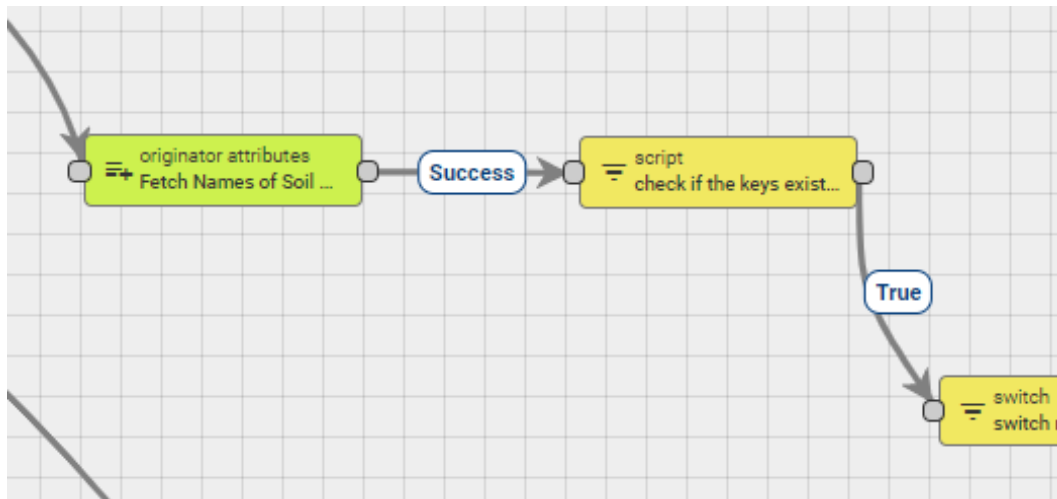
The calculation starts with a `originator attributes` node that executes every time that a value is saved on the database and fetches the sensor attribute `fieldConfiguration`. This cannot be avoided because the rule chain needs to find the name of the keys that will be used to trigger the rest of the calculation.





After the fieldConfiguration is pulled, the metadata has the following information

```
{
  "deviceName": "3 Pro Irrigation V4",
  "deviceType": "3 pro irrigation v4",
  "ss_fieldConfiguration": "{\\"groupsArray\\":[{\\"Crop1\\":{\\"species\\":\\"Dubium\\",\\"irrigation\\":\\"100%\\",\\"sensorArray\\":[{\\"name\\":\\"\\",\\"meas\\":\\"Soil Vol Water Content 01\\",\\"soil_thickness\\":\\"0.225\\",\\"weight\\":\\"0.5\\"}],{\\"name\\":\\"\\",\\"meas\\":\\"Soil Vol Water Content 02\\",\\"soil_thickness\\":\\"0.20\\",\\"weight\\":\\"0.5\\"}],{\\"name\\":\\"\\",\\"meas\\":\\"Soil Vol Water Content 03\\",\\"soil_thickness\\":\\"0.175\\",\\"weight\\":\\"0.5\\"}],{\\"name\\":\\"\\",\\"meas\\":\\"Soil Vol Water Content 04\\",\\"soil_thickness\\":\\"0.25\\",\\"weight\\":\\"0.5\\"}],{\\"name\\":\\"\\",\\"meas\\":\\"Soil Vol Water Content 05\\",\\"soil_thickness\\":\\"0.20\\",\\"weight\\":\\"0.5\\"}],{\\"name\\":\\"\\",\\"meas\\":\\"Soil Vol Water Content 06\\",\\"soil_thickness\\":\\"0.175\\",\\"weight\\":\\"0.5\\"}]}}]",
  "ts": "1658923046911"
}
```



On success relation, a script node is triggered with the following code.

```
//fetch the name of the sensors
var fieldConfiguration = JSON.parse(metadata.ss_fieldConfiguration);

//this works only for the first crop right now
var index = 0;

//get the name of the root object. it will be used to select the sensor array
var name = Object.keys(fieldConfiguration.groupsArray[index]);
```



```
//get number of sensors on that group.
var numberOfSensors = Object.keys(fieldConfiguration.groupsArray[index][name]
.sensorArray).length;

//first Lets check if the keys are available
for (var sensorIndex = 0; sensorIndex < numberOfSensors; sensorIndex++)
{
    var sensor = fieldConfiguration.groupsArray[index][name].sensorArray[sens
orIndex];
    //check if the sensor key exists on the message
    //if the message has the sensor key, we can do more post processing downs
tream.
    if (msg.hasOwnProperty(sensor.meas))
    {
        return true;
    }
}
return false;
```

This script returns true, only if the message telemetry contains one of the keys that are related to the calculation of the soil moisture average since there is a possibility that the messages will be received one at a time, and the logger will have an unknown number of messages, its better to only do the rest of the post processing if the message that came has the relevant telemetry. In any other way the postprocessing would have to trigger the same number of times as the number of telemetry messages that will come on each transmission cycle from the logger.

on true a switch node is triggered. The switch with the following script:

```
function nextRelation(metadata, msg)
{

    //fetch the name of the sensors
    var fieldConfiguration = JSON.parse(metadata.ss_fieldConfiguration);

    //this works only for the first crop right now
    var index = 0;

    //get the name of the root object. it will be used to select the sensor a
rray
    var name = Object.keys(fieldConfiguration.groupsArray[index]);

    //get number of sensors on that group.
```



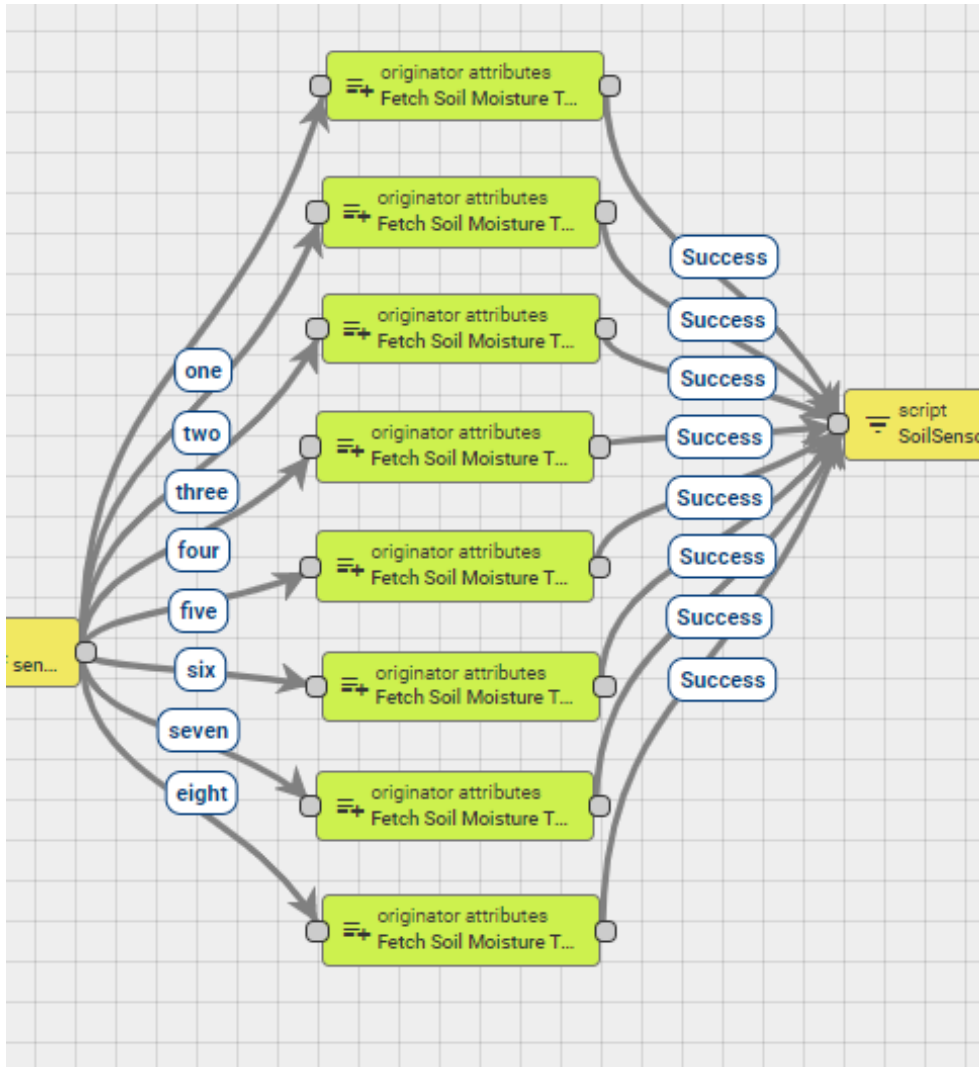
```
    var numberOfSensors = Object.keys(fieldConfiguration.groupsArray[index][name].sensorArray).length;
    return switchResult (numberOfSensors);
}

function switchResult(value)
{
    switch(value)
    {
        case 1:
            return ['one'];
        case 2:
            return ['two'];
        case 3:
            return ['three'];
        case 4:
            return ['four'];
        case 5:
            return ['five'];
        case 6:
            return ['six'];
        case 7:
            return ['seven'];
        case 8:
            return ['eight'];
        case 9:
            return ['nine'];
        case 10:
            return ['ten'];
        case 11:
            return ['eleven'];
        case 12:
            return ['twelve'];
        default:
            return ['error'];
    }
}

return nextRelation(metadata, msg);
```

This script counts how many soil moisture sensors are in the field configuration and forwards the message to a relevant originator attribute node.





The reason is that originator attribute node needs to fetch the relevant telemetry for the database.

For example, the fieldConfiguration that is used on this example, is using 6 different sensors with names of Soil Vol Water Content ## with numbers from '01' to '06'

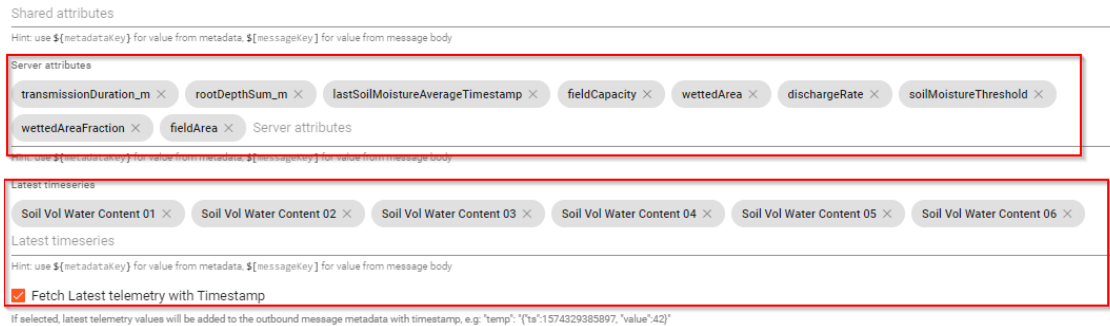
At this point a high level explanation of why this is needed. When a logger sends all the telemetry keys on the same package, then averaging is straightforward because the calculation is able to be performed directly on the values that are on the incoming message. In case the logger transmits the messages one at the time. (which can happen if the logger doesn't have support for nesting of telemetry, or if any historical data are transmitted). The problem is that it



is not possible to know that all the telemetry keys of the same group. So every time that a new message with the Soil Vol Water Content ## key is received, all the latest messages of the group are pulled from the database, and they are checked that they have the same timestamp on later nodes before proceeding.

Similarly, since the keys has to be pulled from the database individually, and the originator attributes node will report failure if the keys are not available, this means that depending the number of soil moisture sensors, different originator nodes must handle the pulling of the database.

Hense, if the number of sensors in the group is six, the originator attributes node with relation six will be pulled, and that node will pull the 6 latest relevant telemetry-timeseries keys from the database. Note the use of the Fetch latest telemetry with timestamp checkbox.



## Soil Vol Water Content XX

As well as a number of server attributes that will be used for the calculations downstream. \* transmissionDuration\_m \* rootDepthSum\_m \* lastSoilMoistureAverageTimestamp \* fieldCapacity \* wettedArea \* dischargeRate \* soilMoistureThreshold \* wettedAreaFraction \* fieldArea

After originator attributes node reports success, the output will have the following form

```
{
  "msg":{
    "Soil Temperature 01": 26, //or whatever key was received
  },
  "metadata":{
    "Soil Vol Water Content 01": "{ \"ts\":1658923046911, \"value\":20.2}",
    "Soil Vol Water Content 02": "{ \"ts\":1658923046911, \"value\":23.7}",
    "Soil Vol Water Content 03": "{ \"ts\":1658923046911, \"value\":36.6}",
  }
}
```





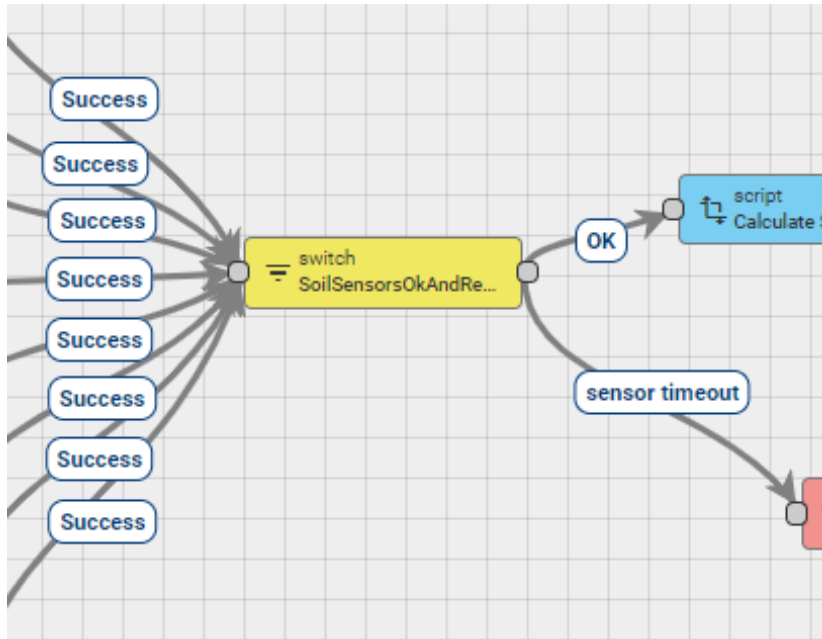
```

"Soil Vol Water Content 04": "{ \"ts\":1658923046911, \"value\":3.5}",
"Soil Vol Water Content 05": "{ \"ts\":1658923046911, \"value\":18.5}",
"Soil Vol Water Content 06": "{ \"ts\":1658923046911, \"value\":40.1}",
"deviceName": "3 Pro Irrigation V4",
"deviceType": "3 pro irrigation v4",
"ss_dischargeRate": "1920",
"ss_fieldArea": "110",
"ss_fieldCapacity": "28.0",
"ss_fieldConfiguration": "{ \"groupsArray\": [{ \"Crop1\": { \"species\": \"Dubium\", \"irrigation\": \"100%\", \"sensorArray\": [ { \"name\": \"\", \"meas\": \"Soil Vol Water Content 01\", \"soil_thickness\": \"0.225\", \"weight\": \"0.5\" }, { \"name\": \"\", \"meas\": \"Soil Vol Water Content 02\", \"soil_thickness\": \"0.20\", \"weight\": \"0.5\" }, { \"name\": \"\", \"meas\": \"Soil Vol Water Content 03\", \"soil_thickness\": \"0.175\", \"weight\": \"0.5\" }, { \"name\": \"\", \"meas\": \"Soil Vol Water Content 04\", \"soil_thickness\": \"0.25\", \"weight\": \"0.5\" }, { \"name\": \"\", \"meas\": \"Soil Vol Water Content 05\", \"soil_thickness\": \"0.20\", \"weight\": \"0.5\" }, { \"name\": \"\", \"meas\": \"Soil Vol Water Content 06\", \"soil_thickness\": \"0.175\", \"weight\": \"0.5\" } ] } ] }",
"ss_lastSoilMoistureAverageTimestamp": "0",
"ss_rootDepthSum_m": "0.6125",
"ss_soilMoistureThreshold": "21",
"ss_transmissionDuration_m": "70",
"ss_wettedArea": "39",
"ss_wettedAreaFraction": "0.355",
"ts": "1658923046911"
}
}

```

Note that the timeseries that were pulled from the database dont include any prefix, and that the timestamp is included because of the checkbox on the settings of the originator attribute node.





Downstream a switch node is executed with the following code

```
function nextRelation(metadata, msg)
{
    var fieldConfiguration = JSON.parse(metadata.ss_fieldConfiguration);
    var minutesOffset = 5;
    var transmissionDurationLimit_ms = (parseFloat(metadata.ss_transmissionDuration_m) + minutesOffset) * 60000;
    var lastSoilMoistureAverageTimestamp_s = Math.round(parseInt(metadata.ss_lastSoilMoistureAverageTimestamp) / 1000);

    //this works only for the first crop right now
    var index = 0;

    //get the name of the root object. it will be used to select the sensor array
    var name = Object.keys(fieldConfiguration.groupsArray[index]);

    //get number of sensors on that group.
    var numberOfSensors = Object.keys(fieldConfiguration.groupsArray[index][name].sensorArray).length;

    var timestamp = 0;
```



```

//first Lets check if the keys are available
for (var sensorIndex = 0; sensorIndex < numberOfSensors; sensorIndex++)
{
    var sensor = fieldConfiguration.groupsArray[index][name].sensorArray[
sensorIndex];
    //check if the sensor key exists on the metadata.
    //It should be there 100% since we have success status on the previous
node as a requirement to execute this node,
    //but better be extra safe
    if (!metadata.hasOwnProperty(sensor.meas))
    {
        return ['doesnt exist']
    }

    //from the first sensor onward start checking the timestamp by comparing
it with the previous sensor.
    if (sensorIndex >= 1)
    {
        //the name of the sensor exists at the metadata at this point 100
% so fetch it

        //var sensorA = JSON.parse("{\"ts\":1654681764417,\"value\":22.3}");
        var sensorA = JSON.parse(metadata[sensor.meas]);

        //also fetch the previous sensor for comparison.
        var sensorTemp = fieldConfiguration.groupsArray[index][name].sensorArray[
sensorIndex - 1];
        var sensorB = JSON.parse(metadata[sensorTemp.meas]);
        var timestampA = Math.round(parseInt(sensorA.ts) / 1000); //avoid
an ms comparison. convert them to seconds using int division.
        var timestampB = Math.round(parseInt(sensorB.ts) / 1000);
        if (timestampA !== timestampB)
        {
            return ['different timestamps']
        }
        //save the timestamp so we can use it out of the loop.
        timestamp = timestampA;
    }
}
//if we reached that point, all the sensors have the same timestamp, so
//check that the timestamp is recent enough
var currentDate = new Date()
var currentTimeDifference = currentDate.getTime() - (timestamp * 1000);

```





```
if (currentTimeDifference >= transmissionDurationLimit_ms)
{
    return ['sensor timeout'];
}

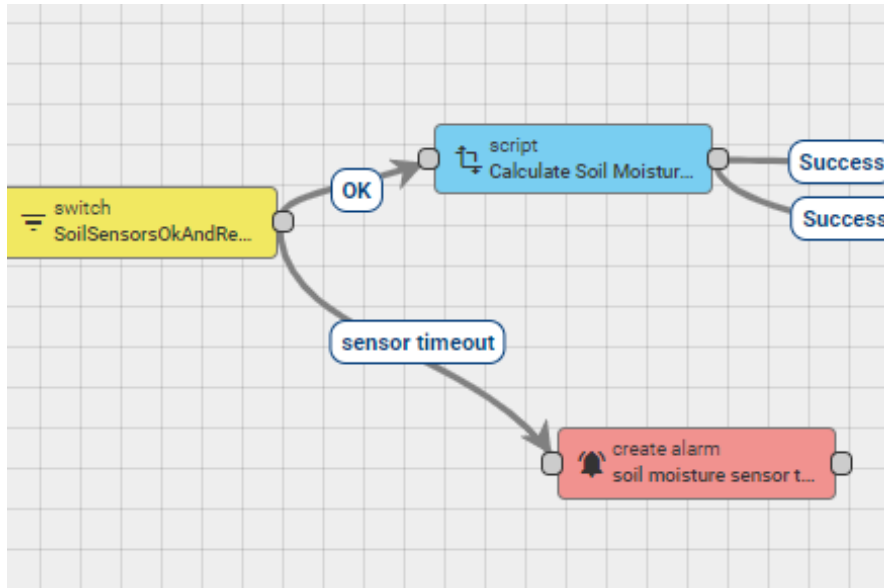
//finally check that this average hasn't been already stored
//if the timestamp is the same (or less) than the last time that the aver
age has been stored
//it means that the value has already been stored, so return false
if (timestamp <= lastSoilMoistureAverageTimestamp_s)
{
    return ['already performed'];
}

return ['OK'];
}

return nextRelation(metadata, msg);
```

this function will return OK only if 3 things are true \* A. The Soil Moisture keys have all the same timestamp. \* B. The timestamp is different than the last time that the average is performed. \* C. The timestamp of the telemetry happened in the last configured period.

A is because, to have a meaningful calculation you need all the measurements at the same time. B, is because without this check, on this example that uses 6 different sensors, the average would have been computed 6 different times C. If the timestamp is older than current time - transmissionDurationLimit, then the sensors have an issue and an alarm must be triggered.



The alarm will be triggered using a sensor timeout relation on the exit of that node. On OK relation a transformation script node will be executed which will calculate the soil average as well as additional attributes

The script has the following code

```

//requires the field configuration data as well as
//ss_fieldCapacity
//ss_wettedArea
//ss_soilMoistureThresshold
//ss_dischargeRate
var m_to_cm = 100;

var fieldConfiguration = JSON.parse(metadata.ss_fieldConfiguration);
var rootzoneDepth_cm = parseFloat(metadata.ss_rootDepthSum_m) * m_to_cm;
var latestSoilMoistureTimestamp = 0;

var newMsg = {};

var cropsNumber = Object.keys(fieldConfiguration.groupsArray).length;
//only one group right now
var index = 0;

var averageSoilMoisture = 0; //will store the multiplication of the thicknes
s of the soil
  
```



```
var averageSoilMoistureSummary = 0; //will store the sum of the averageSoilMoisture

//get the name of the root object. it will be used to select the sensor array
var name = Object.keys(fieldConfiguration.groupsArray[index]);

//get number of sensors on that group.
var numberOfSensors = Object.keys(fieldConfiguration.groupsArray[index][name].sensorArray).length;

//loop to calculate the averages
for (var sensorIndex = 0; sensorIndex < numberOfSensors; sensorIndex++)
{
    var sensor = fieldConfiguration.groupsArray[index][name].sensorArray[sensorIndex];
    //get the name of the measurement. it should correspond with the name of the measurement from the Logger.
    var measurementName = sensor.meas;

    //no need to check that the property exists, since this is guaranteed from the previous node.
    var sensorSoilThickness = parseFloat(sensor.soil_thickness) * m_to_cm * parseFloat(sensor.weight);

    //this will pull the "SoilMoisture01": "{\\"ts\\":1654681764417,\\"value\\":2.3}" format
    var curSensorFromMetadata = JSON.parse(metadata[measurementName]);

    averageSoilMoisture += sensorSoilThickness * parseFloat(curSensorFromMetadata.value)

    //store also the timestamp
    latestSoilMoistureTimestamp = parseInt(curSensorFromMetadata.ts);
}

//summarise the total of all the measurements of each sensors in this group.
averageSoilMoistureSummary += averageSoilMoisture;

//and calculate the summary dividing by the total thickness.
var averageSM = averageSoilMoistureSummary / rootzoneDepth_cm;

//finally generate the new entry
```





```

newMsg[name + "_AverageSoilMoisture"] = parseFloat(averageSM.toFixed(3));
//-----process 3

var maxIrrigation_mm = 0;

var temp = 0.1 * (parseFloat(metadata.ss_fieldCapacity) - averageSM) * rootzoneDepth_cm;
if (temp < 0)
{
    maxIrrigation_mm = 0;
}
else
{
    maxIrrigation_mm = temp * (parseFloat(metadata.ss_wettedAreaFraction));
}

var maxIrrigation_hr = maxIrrigation_mm * parseFloat(metadata.ss_fieldArea) /
parseFloat(metadata.ss_dischargeRate);
var roundMaxIrrugation_hr = Math.floor(maxIrrigation_hr);
newMsg.roundMaxIrrugation_hr = roundMaxIrrugation_hr;
var maxIrrigation_hrmin = roundMaxIrrugation_hr + (maxIrrigation_hr - roundMaxIrrugation_hr) * (60.0 / 100.0);
var irrigationRequired = averageSM <= parseFloat(metadata.ss_soilMoistureThreshold);

newMsg[name + "_maxIrrigation_mm"] = parseFloat(maxIrrigation_mm.toFixed(3));
newMsg[name + "_maxIrrigation_hr"] = parseFloat(maxIrrigation_hr.toFixed(3));
newMsg[name + "_maxIrrigation_hrmin"] = parseFloat(maxIrrigation_hrmin.toFixed(3));
newMsg[name + "_irrigationRequired"] = irrigationRequired;

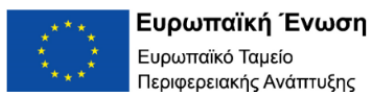
//also store the timestamp
newMsg.latestSoilMoistureTimestamp = latestSoilMoistureTimestamp;

return {msg: newMsg, metadata: metadata, msgType: msgType};

return {msg: newMsg, metadata: metadata, msgType: msgType};

```

The above script calculates and places it on the message. \* Crop1\_AverageSoilMoisture \* Crop1\_maxIrrigation\_mm \* Crop1\_maxIrrigation\_hr \* Crop1\_maxIrrigation\_hrmin \* Crop1\_irrigationRequired \* latestSoilMoistureTimestamp





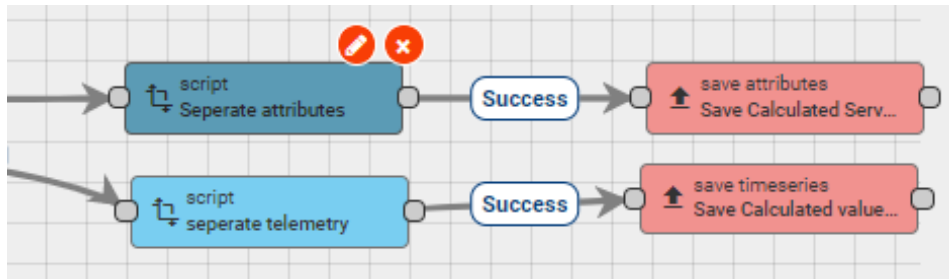
The resulting message output on node success is:

```
{
  "msg":{
    "Crop1_AverageSoilMoisture": 22.271,
    "Crop1_maxIrrigation_mm": 12.456,
    "Crop1_maxIrrigation_hr": 0.714,
    "Crop1_maxIrrigation_hrmin": 0.428,
    "Crop1_irrigationRequired": false,
    "latestSoilMoistureTimestamp": 1658923046911
  },
  "metadata":{
    "Soil Vol Water Content 01": "{\"ts\":1658923046911,\"value\":20.2}\",
    "Soil Vol Water Content 02": "{\"ts\":1658923046911,\"value\":23.7}\",
    "Soil Vol Water Content 03": "{\"ts\":1658923046911,\"value\":36.6}\",
    "Soil Vol Water Content 04": "{\"ts\":1658923046911,\"value\":3.5}\",
    "Soil Vol Water Content 05": "{\"ts\":1658923046911,\"value\":18.5}\",
    "Soil Vol Water Content 06": "{\"ts\":1658923046911,\"value\":40.1}\",
    "deviceName": "3 Pro Irrigation V4",
    "deviceType": "3 pro irrigation v4",
    "ss_dischargeRate": "1920",
    "ss_fieldArea": "110",
    "ss_fieldCapacity": "28.0",
    "ss_fieldConfiguration": "{\"groupsArray\": [{\"Crop1\": {\"species\": \"Dubium\", \"irrigation\": \"100%\", \"sensorArray\": [{\"name\": \"\", \"meas\": \"Soil Vol Water Content 01\", \"soil_thickness\": \"0.225\", \"weight\": \"0.5\"}, {\"name\": \"\", \"meas\": \"Soil Vol Water Content 02\", \"soil_thickness\": \"0.20\", \"weight\": \"0.5\"}, {\"name\": \"\", \"meas\": \"Soil Vol Water Content 03\", \"soil_thickness\": \"0.175\", \"weight\": \"0.5\"}, {\"name\": \"\", \"meas\": \"Soil Vol Water Content 04\", \"soil_thickness\": \"0.25\", \"weight\": \"0.5\"}, {\"name\": \"\", \"meas\": \"Soil Vol Water Content 05\", \"soil_thickness\": \"0.20\", \"weight\": \"0.5\"}, {\"name\": \"\", \"meas\": \"Soil Vol Water Content 06\", \"soil_thickness\": \"0.175\", \"weight\": \"0.5\"}]}]}]}\",
    "ss_lastSoilMoistureAverageTimestamp": "0",
    "ss_rootDepthSum_m": "0.6125",
    "ss_soilMoistureThreshold": "21",
    "ss_transmissionDuration_m": "70",
    "ss_wettedArea": "39",
    "ss_wettedAreaFraction": "0.355",
    "ts": "1658923046911"
  }
}
```



Of that message the latestSoilMoistureTimestamp must be saved as server attribute, since there is no need for plotting or keeping historical data for that value. It is only used on the next cycle of the calculation. The other values must be saved as telemetry / timeseries, so they can be plotted if need.

To accomplish this on success, 2 additional transformation script nodes are executed.



Script 1 seperates the attributes from the message and has the following code

```
//create an empty object which will be used to store the final result.
var newMsg = {}; //empty object

newMsg.julianDayNumber = msg.julianDayNumber;
newMsg.latitude_rad = msg.latitude_rad;
newMsg.sunsetHourNumber_rad = msg.sunsetHourNumber_rad;
newMsg.extraterrestrialRadiation_MJm2day = msg.extraterrestrialRadiation_MJm2day;
//also save this as server attributes so that it is used on the next Loop the next day
newMsg.averageSoilMoistureAtPreviousDayChange = msg.averageSoilMoistureAtDayChange;

var msgType = "POST_ATTRIBUTES_REQUEST";

return {msg: newMsg, metadata: metadata, msgType: msgType};
```

and the following output

```
{
  "latestSoilMoistureTimestamp": 1658923046911
}
```

Script 2 seperates the timeseries from the message and has the following code

```
//create an empty object which will be used to store the final result.
var newMsg = {}; //empty object
```





```

newMsg = msg;
//only remove the latestSoilMoistureTimestamp and keep the rest of the properties
delete newMsg.latestSoilMoistureTimestamp;
var msgType = "POST_TELEMETRY_REQUEST";
return {msg: newMsg, metadata: metadata, msgType: msgType};

```

and the following output.

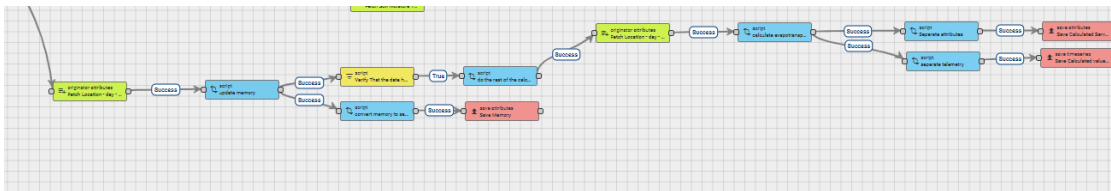
```

msg: {
  "Crop1_AverageSoilMoisture": 22.271,
  "Crop1_maxIrrigation_mm": 12.456,
  "Crop1_maxIrrigation_hr": 0.714,
  "Crop1_maxIrrigation_hrmin": 0.428,
  "Crop1_irrigationRequired": false
}

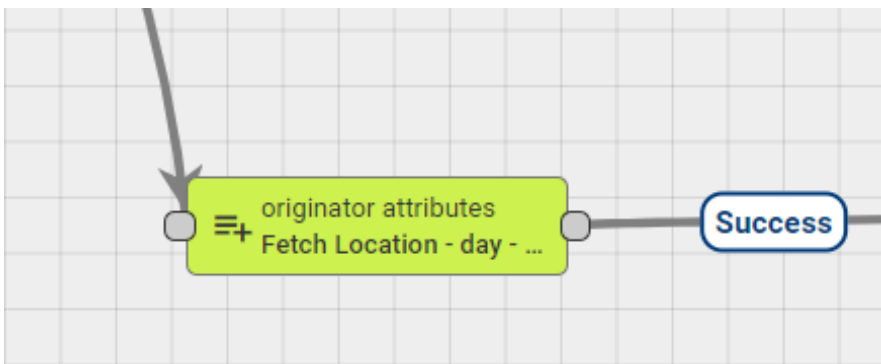
```

#### Group 4, Day change calculations

Group4 of nodes, is responsible of finding the min and max temperature, checking if the day has changed, and calculate the various daily properties on day change.



every time a new telemetry message is saved on a database an originator attribute fetches some relevant server attributes from the database.



- julianDayNumber
- memory





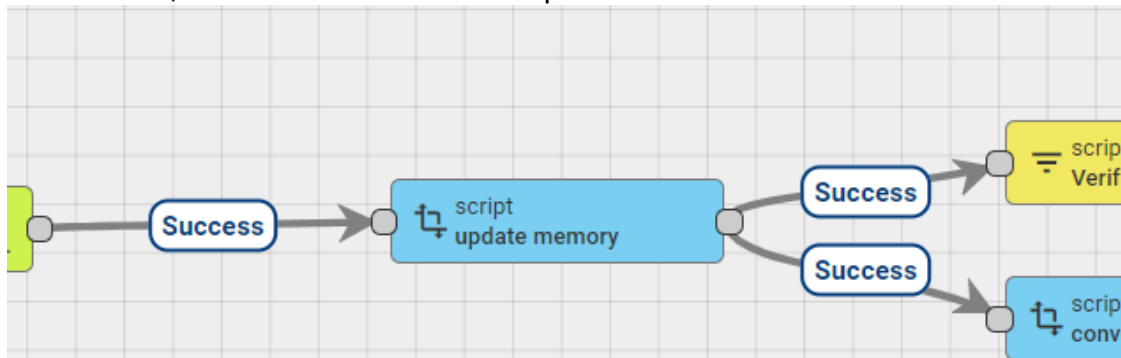
- nameOfTemperatureKey
- latitude
- rainPerTick

Not all attributes will be used immediately.

The resulting message will be something like that

```
{
  "msg": {
    not important.
  },
  "metadata": {
    "deviceName": "3 Pro Irrigation V4",
    "deviceType": "3 pro irrigation v4",
    "ss_julianDayNumber": "208",
    "ss_latitude": "34.9118",
    "ss_memory": "{\"currentTemperatureMin\":18,\"currentTemperatureMax\":39.9,\"rainTicksSummary\":0}",
    "ss_nameOfTemperatureKey": "Ambient Temperature",
    "ss_rainPerTick": "0.2",
    "ts": "1658923046911"
  }
}
```

on success, a transformation script is executed



The script has the following code

```
//create an empty object which will be used to store the final result.
var newMsg = {}; //empty object

//-----calculate day of the year-----
```



```
//helper functions
//returns true if the year is Leap.
Date.prototype.isLeapYear = function ()
{
    var year = this.getFullYear();
    if ((year & 3) != 0) return false;
    return ((year % 100) != 0 || (year % 400) == 0);
};

// Get Day of Year
Date.prototype.getDOY = function ()
{
    //array that holds the day count on each month
    var dayCount = [0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334];
    var mn = this.getMonth();
    var dn = this.getDate();
    var dayOfYear = dayCount[mn] + dn;
    //increase by one if the year is Leap.
    if (mn > 1 && this.isLeapYear())
    {
        dayOfYear++;
    }
    return dayOfYear;
};

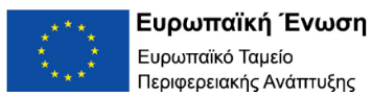
function DegToRad(degrees)
{
    return degrees * (Math.PI / 180.0);
}

var currentDate = new Date();
newMsg.julianDayNumber = currentDate.getDOY();

//-----update memory-----

var nameOfTemperatureKey = metadata.ss_nameOfTemperatureKey;
var nameOfRainTicksKey = metadata.ss_nameOfRainTicksKey;

//select the temperature either from the meteorological sensor, or the bme
if ((msg.hasOwnProperty(nameOfTemperatureKey) || msg.hasOwnProperty(nameOfRainTicksKey)))
{
    var memoryJSON = JSON.parse(metadata.ss_memory);
    //update the values for the next cycle
```





```
//start by updating the temperature when that telemetry arrives
if (msg.hasOwnProperty(nameOfTemperatureKey))
{
    var temperature = parseFloat(msg[nameOfTemperatureKey]);
    //update the min and max values for the next loop.
    if (temperature < parseFloat(memoryJSON.currentTemperatureMin))
    {
        memoryJSON.currentTemperatureMin = temperature;
    }

    if (temperature > parseFloat(memoryJSON.currentTemperatureMax))
    {
        memoryJSON.currentTemperatureMax = temperature;
    }

    //also store the last measurement. This is because when we reset
    //the memory on the day change, we would not know which value to use.
    memoryJSON.lastTemperatureMeasurement = temperature;
}
//continue by updating the rain
if (msg.hasOwnProperty(nameOfRainTicksKey))
{
    var ticks = parseFloat(msg[nameOfRainTicksKey]);
    memoryJSON.rainTicksSummary = parseFloat(memoryJSON.rainTicksSummary)
+ ticks;
}

//stringify so you can store it.
var memory = JSON.stringify(memoryJSON);
newMsg.memory = memory;
}

return {msg: newMsg, metadata: metadata, msgType: msgType};
```

The script calculates the: julianDayNumber. If the incoming message has a temperature key, it also updates the currentTemperatureMin and currentTemperatureMax in the memory server attribute.

Assuming that the previous message had a temperature key, and that the day has changed:

```
{
  "msg": {
    "temperature": 40,
  },
}
```



```

"metadata": {
}
}

```

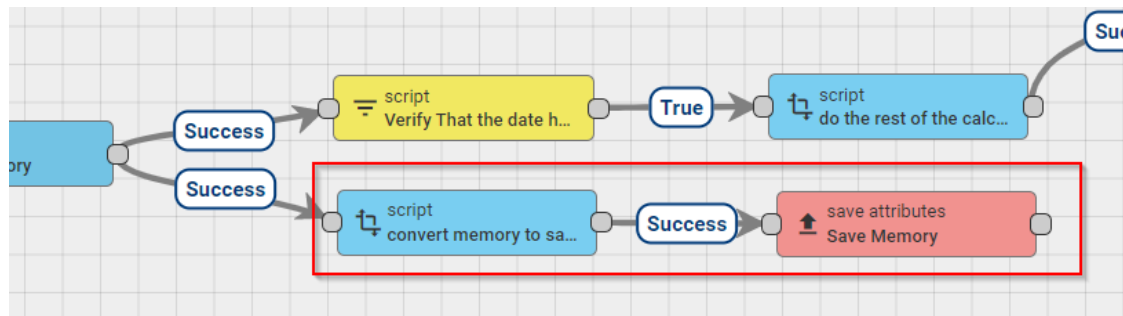
The resulting output will be something like that. Note the updated currentTemperatureMax, and the updated `julianDayNumber`

```

{
  "msg":{
    "julianDayNumber": 209,
    "memory": "{ \"currentTemperatureMin\":18, \"currentTemperatureMax\":40.0, \"rainTicksSummary\":0}"
  }
  "metadata":{
    "deviceName": "3 Pro Irrigation V4",
    "deviceType": "3 pro irrigation v4",
    "ss_julianDayNumber": "208",
    "ss_latitude": "34.9118",
    "ss_memory": "{ \"currentTemperatureMin\":18, \"currentTemperatureMax\":39.9, \"rainTicksSummary\":0}",
    "ss_nameOfTemperatureKey": "Ambient Temperature",
    "ss_rainPerTick": "0.2",
    "ts": "1658923046911"
  }
}

```

On success 2 different node chains will be executed.



starting from the bottom chain, a transformation script node is executed with the following code

```

//create an empty object which will be used to store the final result.
var newMsg = {}; //empty object

//extract the memory from the message

```







```
newMsg.memory = msg.memory;  
var msgType = "POST_ATTRIBUTES_REQUEST";  
  
return {msg: newMsg, metadata: metadata, msgType: msgType};
```

The message has to be converted to a POST\_ATTRIBUTES\_REQUEST to be update, and that node does exactly that.

The resulting output is

```
{  
  "msg":{  
    "memory": "{ \"currentTemperatureMin\":18, \"currentTemperatureMax\":40  
.0, \"rainTicksSummary\":0}"  
  },  
}
```

and the save attributes node saves the memory as server attribute.

On the top chain, a script is executed checking if the day has changed, so it can triggered the next series of calculations.

the script has the following code

```
//return true if the day has changed in comparison to the saved date  
if (parseInt(metadata.ss_julianDayNumber) != parseInt(msg.julianDayNumber))  
{  
  return true;  
}  
return false;
```

as indicated on the transformation script section above, when the day changes, then the julianDayNumber on the msg and was just calculated will be different than the ss\_julianDayNumber which was stored on the database. This script returns true when that change is detected.

on true, an additional transformation script is executed with the following code

```
function DegToRad(degrees)  
{  
  return degrees * (Math.PI / 180.0);  
}
```

```
//create an empty object which will be used to store the final result.
```





```
var newMsg = {}; //empty object

//at this point we have the memory as well as the julian day number on the message, as well as some metadata values.

//keep the new day number so we can update that attribute later.
newMsg.julianDayNumber = msg.julianDayNumber;

//also calculate the solar declination and some other things
//---calculate solar properties
//The following equation can be used to calculate the declination angle:  $\delta = -23.45^\circ \times \cos(360/365 \times (d+10))$ 
// where the d is the number of days since the start of the year The declination angle equals zero at the equinoxes
//(March 22 and September 22), positive during the summer in northern hemisphere and negative during winter in the northern hemisphere.
//The declination reaches a maximum angle on June 22 which is  $23.45^\circ$  (the northern hemisphere summer solstice) and a minimum angle
//on December 21-22 which is of  $-23.45^\circ$  (the northern hemisphere winter solstice).
//In the above equation, the +10 is due to the fact that the winter solstice occurs before the start of the year.
// The equation also assumes the orbit of the sun to be a perfect circle and the fraction of 360/365
//converts the number of days to the position in the orbit. The apparent northward movement of the Sun during the northern spring,
// reaching the celestial equator during the March equinox. The declination reaches a maximum angle equal to the axial
//tilt of the Earth's axial tilt ( $23.44^\circ$ ) on the June solstice, then starts decreasing until reaching its minimum ( $-23.44^\circ$ )
//on the December solstice, where its value is equal to the negative of the axial tilt. Seasons are a direct product of this variation.

//---calculate latitude in rad
var latitude_rad = DegToRad(parseFloat(metadata.ss_latitude));

var solarDeclination_rad = DegToRad(-23.45) * Math.cos(DegToRad(360.0 / 365.0 * (newMsg.julianDayNumber + 10)));
var sunsetHourNumber_rad = Math.acos(-Math.tan(latitude_rad) * Math.tan(solarDeclination_rad));

var solarConstant_MJM2min = 0.082;
var inverseRelativeDistanceEarthSun = 1 + 0.033 * Math.cos(((2 * Math.PI) / 365) * newMsg.julianDayNumber);
```



```
var extraterrestrialRadiation_MJm2day = (24 * 60 / Math.PI) * solarConstant_MJm2min * inverseRelativeDistanceEarthSun * ((sunsetHourNumber_rad * Math.sin(latitude_rad) * Math.sin(solarDeclination_rad)) + (Math.cos(latitude_rad) * Math.cos(solarDeclination_rad) * Math.sin(sunsetHourNumber_rad)));

//add them on message
newMsg.solarDeclination_rad = parseFloat(solarDeclination_rad.toFixed(4));
newMsg.sunsetHourNumber_rad = parseFloat(sunsetHourNumber_rad.toFixed(4));
newMsg.extraterrestrialRadiation_MJm2day = parseFloat(extraterrestrialRadiation_MJm2day.toFixed(4));
//also add the julian date

/*now its the time to calculate the min max temperatures.*/
//since the day has changed, the only thing to do is to pull out the current min and max values.
var memoryJSON = JSON.parse(metadata.ss_memory);

newMsg.minTemperature = parseFloat(memoryJSON.currentTemperatureMin);
newMsg.minTemperature = parseFloat(newMsg.minTemperature.toFixed(2));

newMsg.maxTemperature = parseFloat(memoryJSON.currentTemperatureMax);
newMsg.maxTemperature = parseFloat(newMsg.maxTemperature.toFixed(2));

newMsg.temperatureMean = (newMsg.minTemperature + newMsg.maxTemperature) / 2.0;
newMsg.temperatureMean = parseFloat(newMsg.temperatureMean.toFixed(2));

//update precipitation level per day.
newMsg.precipitationLevel_mm = parseFloat(memoryJSON.rainTicksSummary) * parseFloat(metadata.ss_rainPerTick);
newMsg.precipitationLevel_mm = parseFloat(newMsg.precipitationLevel_mm.toFixed(2));

//cleanup
newMsg.minTemperature = parseFloat(newMsg.minTemperature.toFixed(2));
newMsg.maxTemperature = parseFloat(newMsg.maxTemperature.toFixed(2));
newMsg.temperatureMean = parseFloat(newMsg.temperatureMean.toFixed(2));
newMsg.precipitationLevel_mm = parseFloat(newMsg.precipitationLevel_mm.toFixed(2));

//finally since the day has changed, we need to reset the memoryJSON
```



```
newMsg.memory = {
    //initialize as the last measurement, so even if no measurements were performed the rest of the day
    //it will not report something absurd like -100 or 100°C,
    "currentTemperatureMin": parseFloat(memoryJSON.lastTemperatureMeasurement),
    "currentTemperatureMax": parseFloat(memoryJSON.lastTemperatureMeasurement),
    "rainTicksSummary": 0,
    "lastTemperatureMeasurement": parseFloat(memoryJSON.lastTemperatureMeasurement)
};

//at this point we have the new message that has the following attributes
//julianDayNumber
//sunsetHourNumber_rad
//extraterrestrialRadiation_MJm2day

//as well as the following values that need to be stored as telemetry
//minTemperature
//maxTemperature
//temperatureMean
//precipitationLevel_mm
return {msg: newMsg, metadata: metadata, msgType: msgType};
```

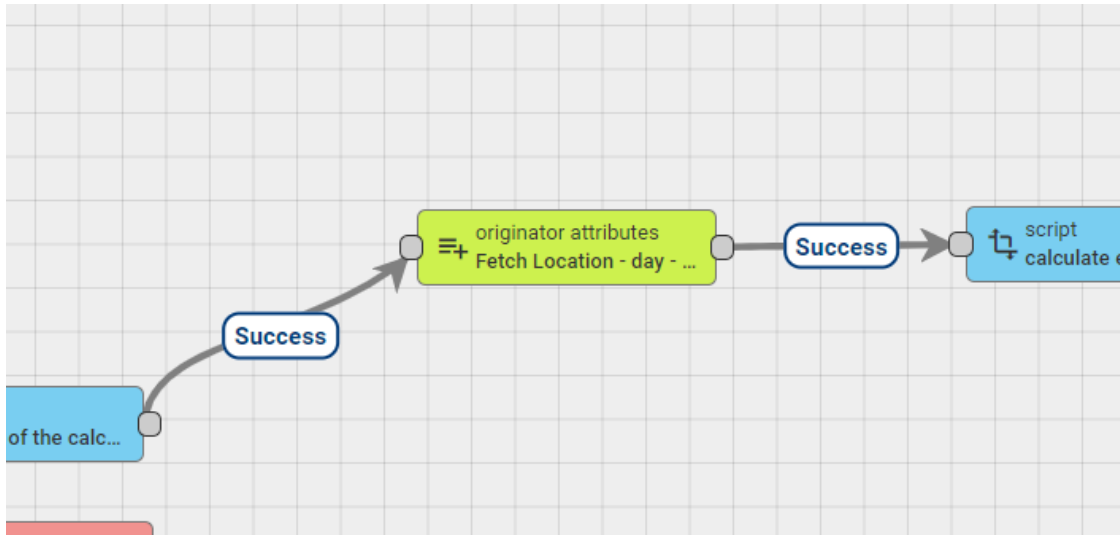
Since the day has changed, this script stores the currently max and min value of the temperature, as well as it calculates various properties.

More specifically it calculates - minTemperature - maxTemperature - temperatureMean - precipitationLevel\_mm

```
{
  "msg": {
    "julianDayNumber": 208,
    "latitude_rad": 0.6093,
    "solarDeclination_rad": 0.3352,
    "sunsetHourNumber_rad": 1.8164,
    "extraterrestrialRadiation_MJm2day": 39.8591,
    "minTemperature": 18,
    "maxTemperature": 39.9,
    "temperatureMean": 28.95,
    "precipitationLevel_mm": 0,
    "referenceEvapotranspiration": 8.18
  },
}
```



```
"metadata" : {
  "deviceName": "3 Pro Irrigation V4",
  "deviceType": "3 pro irrigation v4",
  "ss_julianDayNumber": "207",
  "ss_latitude": "34.9118",
  "ss_memory": "{ \"currentTemperatureMin\":18, \"currentTemperatureMax\":39.9, \"rainTicksSummary\":0}",
  "ss_nameOfTemperatureKey": "Ambient Temperature",
  "ss_rainPerTick": "0.2",
  "ts": "1658921188647"
}
```



On true an originator node is executed which pulls various values. This operation is performed only one time per day.



Name \*  
Fetch

Debug mode

Tell Failure

If at least one selected key doesn't exist the outbound message will report "Failure".

Client attributes

Client attributes

Hint: use `#{metadataKey}` for value from metadata, `#{messageKey}` for value from message body

Shared attributes

Shared attributes

Hint: use `#{metadataKey}` for value from metadata, `#{messageKey}` for value from message body

Server attributes

solarDeclination\_rad × sunsetHourNumber\_rad × extraterrestrialRadiation\_MJm2day × julianDayNumber ×  
dayStartInitialStage\_JDN × dayStartDevelopmentStage\_JDN × dayStartMidSeason\_JDN × dayEndMidSeason\_JDN ×  
dayEndLateSeason\_JDN × cropCoefficientInitial × cropCoefficientMid × cropCoefficientEnd ×  
averageSoilMoistureAtPreviousDayChange × wettedAreaFraction × rootDepthSum\_m × Server attributes

Hint: use `#{metadataKey}` for value from metadata, `#{messageKey}` for value from message body

Latest timeseries

Crop1\_AverageSoilMoisture × Crop1\_maxIrrigation\_hr × Crop1\_maxIrrigation\_hrmin × Crop1\_irrigationRequired ×

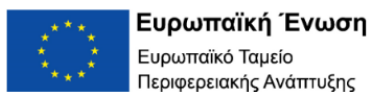
Latest timeseries

The node pulls \* solarDeclination\_rad \* sunsetHourNumber\_rad \* extraterrestrialRadiation\_MJm2day \* julianDayNumber \* dayStartInitialStage\_JDN \* dayStartDevelopmentStage\_JDN \* dayStartMidSeason\_JDN \* dayEndMidSeason\_JDN \* dayEndLateSeason\_JDN \* cropCoefficientInitial \* cropCoefficientMid \* cropCoefficientEnd \* averageSoilMoisture \* AtPreviousDayChange \* wettedAreaFraction \* rootDepthSum\_m

as well as the latest timeseries \* Crop\_AverageSoilMoisture \* Crop\_maxIrrigation\_hr \* Crop\_maxIrrugation\_hrmin \* Crop\_irrigationRequired

These values will have been calculated in the last inactivityTimeoutMinutes minutes on the Group 3 nodes, and are assumed as "instant" values that will be used in the daily calculations.

On success the output will be:





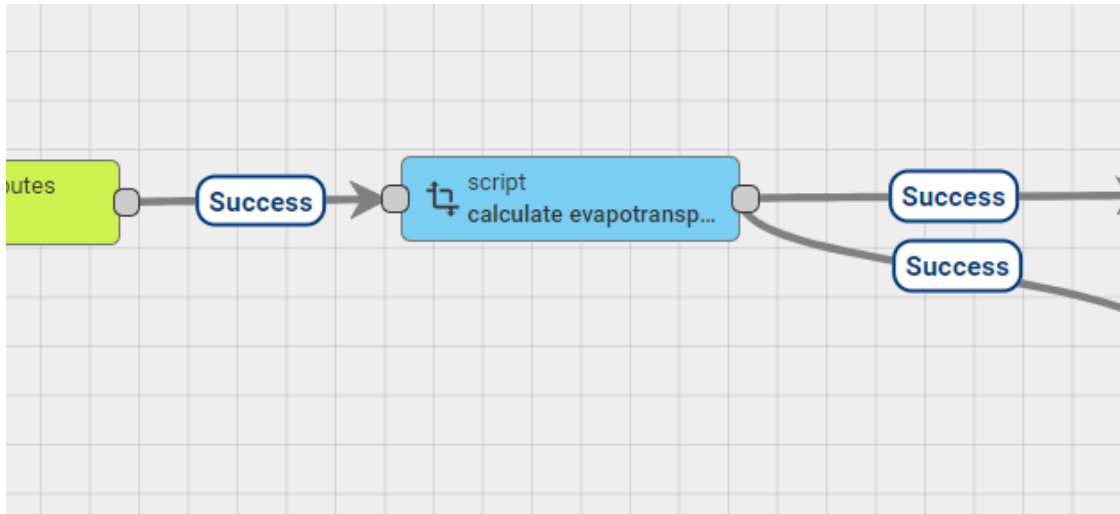
```

{
  "msg": {
    "julianDayNumber": 208,
    "solarDeclination_rad": 0.3352,
    "sunsetHourNumber_rad": 1.8164,
    "extraterrestrialRadiation_MJm2day": 39.8591,
    "minTemperature": 18,
    "maxTemperature": 39.9,
    "temperatureMean": 28.95,
    "precipitationLevel_mm": 0,
    "referenceEvapotranspiration": 8.18
  },
  "metadata" : {
    "Crop1_AverageSoilMoisture": "{\"ts\":\"1658839975558\",\"value\":31.227}",
    "Crop1_irrigationRequired": "{\"ts\":\"1658839975558\",\"value\":false}",
    "Crop1_maxIrrigation_hr": "{\"ts\":\"1658839975558\",\"value\":0}",
    "Crop1_maxIrrigation_hrmin": "{\"ts\":\"1658839975558\",\"value\":0}",
    "deviceName": "3 Pro Irrigation V4",
    "deviceType": "3 pro irrigation v4",
    "ss_averageSoilMoistureAtPreviousDayChange": "9.52",
    "ss_cropCoefficientEnd": "0.5",
    "ss_cropCoefficientInitial": "0.36",
    "ss_cropCoefficientMid": "0.85",
    "ss_dayEndLateSeason_JDN": "305",
    "ss_dayEndMidSeason_JDN": "274",
    "ss_dayStartDevelopmentStage_JDN": "91",
    "ss_dayStartInitialStage_JDN": "91",
    "ss_dayStartMidSeason_JDN": "152",
    "ss_extraterrestrialRadiation_MJm2day": "39.954",
    "ss_julianDayNumber": "207",
    "ss_latitude": "34.9118",
    "ss_memory": "{\"currentTemperatureMin\":18,\"currentTemperatureMax\":39.9,\"rainTicksSummary\":0}",
    "ss_nameOfTemperatureKey": "Ambient Temperature",
    "ss_rainPerTick": "0.2",
    "ss_rootDepthSum_m": "0.6125",
    "ss_solarDeclination_rad": "0.3431",
    "ss_sunsetHourNumber_rad": "1.8196",
    "ss_wettedAreaFraction": "0.355",
    "ts": "1658921188647"
  }
}

```



On success an transformation script is executed



with the following code

```

/* at this point we have a message on the chain which has has the following p
roperties
{
  "julianDayNumber": 206,
  "solarDeclination_rad": 0.3431,
  "sunsetHourNumber_rad": 1.8228,
  "extraterrestrialRadiation_MJm2day": 40.0462,
  "minTemperature": 18.2,
  "maxTemperature": 39.3,
  "temperatureMean": 28.75,
  "precipitationLevel_mm": null,
}

```

we also have the following metadata.

since we pulled the julianDayNumber and the extretterestial radiation from th  
e database now before saving the updated values,  
we have the calculated values from the previous day, so we can use them on th  
e calculations.

```

{
  "deviceName": "3 Pro Irrigation V4",
  "deviceType": "3 pro irrigation v4",
  "ss_memory": "{\"currentTemperatureMin\":18.2, \"currentTemperatureMax\":39.3, \"rainTicksSummary\":0}",
  "ss_nameOfTemperatureKey": "Ambient Temperature",
}

```







```
"ts": "1658757779401"

"Crop1_AverageSoilMoisture": "{ \"ts\":1658757779401, \"value\":26.773}",
"ss_extraterrestrialRadiation_MJm2day": "40.4648",
"ss_julianDayNumber": "205",
"ss_latitude": "34.9118",
"ss_solarDeclination_rad": "0.3431",
"ss_sunsetHourNumber_rad": "1.8289",
"ss_dayStartInitialStage_JDN": "",
"ss_dayStartDevelopmentStage_JDN": "",
"ss_dayStartMidSeason_JDN": "",
"ss_dayEndMidSeason_JDN": "",
"ss_dayEndLateSeason_JDN": "",
"ss_cropCoefficientInitial": "",
"ss_cropCoefficientMid": "",
"ss_cropCoefficientEnd": "",
"ss_previousAverageSoilMoisture": "",
"ss_rootDepthSum_m": "",
"ss_wettedAreaFraction": "";
}
```

we also pulled the  
`Crop1_AverageSoilMoisture`

we need to also find the solar declination rad and sunset hour number rad, as well as the extraterrestrial radiation for for the previous day.  
we have pulled this from the database before saving the new values.

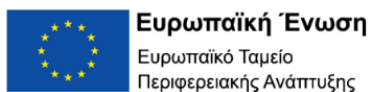
\*/

```
function linearInterpolation(x, x1, x2, y1, y2)
{
    return (y1 + (x - x1) * ( (y2 - y1) / (x2 - x1) ));
}
```

```
function digitsFormat(value, numberOfDigits)
{
    return parseFloat(value.toFixed(numberOfDigits));
}
```

//calculate `ETo_mm`

```
var extraterrestrialRadiationPreviousDay_MJm2day = parseFloat(metadata.ss_extraterrestrialRadiation_MJm2day);
```





```
var temperatureMean = parseFloat(msg.temperatureMean);
var temperatureMax = parseFloat(msg.maxTemperature);
var temperatureMin = parseFloat(msg.minTemperature);

var ETo_mm = 0.0023 * (temperatureMean + 17.8) *
    Math.sqrt(temperatureMax - temperatureMin) *
    0.408 * extraterrestrialRadiationPreviousDay_MJm2day;

//calculate actual crop resultCropCoefficient
//parsed variables
var previous_JDN = parseInt(metadata.ss_julianDayNumber);

var dayStartInitialStage_JDN = parseInt(metadata.ss_dayStartInitialStage_JDN)
;
var dayStartDevelopmentStage_JDN = parseInt(metadata.ss_dayStartDevelopmentStage_JDN);
var dayStartMidSeason_JDN = parseInt(metadata.ss_dayStartMidSeason_JDN);
var dayEndMidSeason_JDN = parseInt(metadata.ss_dayEndMidSeason_JDN);
var dayEndLateSeason_JDN = parseInt(metadata.ss_dayEndLateSeason_JDN);
var cropCoefficientInitial = parseFloat(metadata.ss_cropCoefficientInitial);
var cropCoefficientMid = parseFloat(metadata.ss_cropCoefficientMid);
var cropCoefficientEnd = parseFloat(metadata.ss_cropCoefficientEnd);

var resultCropCoefficient = 0;

if (previous_JDN < dayStartInitialStage_JDN)
{
    resultCropCoefficient = 0;
}
else if (previous_JDN <= dayStartDevelopmentStage_JDN)
{
    resultCropCoefficient = cropCoefficientInitial;
}
else if (previous_JDN <= dayStartMidSeason_JDN)
{
    resultCropCoefficient = linearInterpolation(previous_JDN, dayStartDevelopmentStage_JDN, dayStartMidSeason_JDN, cropCoefficientInitial, cropCoefficientMid);
}
else if (previous_JDN <= dayEndMidSeason_JDN)
{
    resultCropCoefficient = cropCoefficientMid;
}
}
```



```
else if (previous_JDN <= dayEndLateSeason_JDN)
{
    resultCropCoefficient = linearInterpolation(previous_JDN, dayEndMidSeason_JDN, dayEndLateSeason_JDN, cropCoefficientMid, cropCoefficientEnd);
}
else
{
    resultCropCoefficient = 0;
}

//calculate ETc_mm, whatever is this
var ETc_mm = resultCropCoefficient * ETo_mm;

//add the results to the message

msg.cropEvapotranspiration_mm = parseFloat(ETc_mm.toFixed(2));
msg.referenceEvapotranspiration_mm = parseFloat(ETo_mm.toFixed(2));
msg.resultCropCoefficient = parseFloat(resultCropCoefficient.toFixed(2));

var averageSoilMoistureAtDayChange;
var maxIrrigation_hrAtDayChange;
var maxIrrigation_hrminAtDayChange;
var irrigationRequiredStatusAtDayChange;

//pull the daily soil moisture summary
if (metadata.hasOwnProperty('Crop1_AverageSoilMoisture'))
{
    averageSoilMoistureAtDayChange = JSON.parse(metadata.Crop1_AverageSoilMoisture);
    msg.averageSoilMoistureAtDayChange = parseFloat(averageSoilMoistureAtDayChange.value.toFixed(2));
}

if (metadata.hasOwnProperty('Crop1_maxIrrigation_hr'))
{
    maxIrrigation_hrAtDayChange = JSON.parse(metadata.Crop1_maxIrrigation_hr);
    ;
    msg.maxIrrigation_hrAtDayChange = parseFloat(maxIrrigation_hrAtDayChange.value.toFixed(2));
}

if (metadata.hasOwnProperty('Crop1_maxIrrigation_hrmin'))
{
    maxIrrigation_hrminAtDayChange = JSON.parse(metadata.Crop1_maxIrrigation_
```



```
hrmin);
    msg.maxIrrigation_hrminAtDayChange = parseFloat(maxIrrigation_hrminAtDayC
hange.value.toFixed(2));
}

if (metadata.hasOwnProperty('Crop1_irrigationRequired'))
{
    irrigationRequiredStatusAtDayChange = JSON.parse(metadata.Crop1_irrigatio
nRequired);
    msg.irrigationRequiredStatusAtDayChange = irrigationRequiredStatusAtDayCh
ange.value;
}

//calculate deltaSoilMoisture
var averageSoilMoistureAtPreviousDayChange = parseFloat(metadata.ss_averageSo
ilMoistureAtPreviousDayChange);
var deltaSoilMoisture = 0.1 * (msg.averageSoilMoistureAtDayChange - averageSo
ilMoistureAtPreviousDayChange) * parseFloat(metadata.ss_rootDepthSum_m) * par
seFloat(metadata.ss_wettedAreaFraction);

msg.deltaSoilMoisture = parseFloat(deltaSoilMoisture.toFixed(2));

//calculate warning farmer
//calculate warning concultant

return {msg: msg, metadata: metadata, msgType: msgType};
```

this script calculates \* cropEvapotranspiration\_mm \*  
referenceEvapotranspiration\_mm \* resultCropCoefficient \*  
averageSoilMoistureAtDayChange \* maxIrrigation\_hrAtDayChange \*  
maxIrrigation\_hrminAtDayChange \* irrigationRequiredStatusAtDayChange  
deltaSoilMoisture

the resulting message will be

```
{ "msg": {
  "julianDayNumber": 208,
  "latitude_rad": 0.6093,
  "solarDeclination_rad": 0.3352,
  "sunsetHourNumber_rad": 1.8164,
  "extraterrestrialRadiation_MJm2day": 39.8591,
  "minTemperature": 18,
  "maxTemperature": 39.9,
  "temperatureMean": 28.95,
```

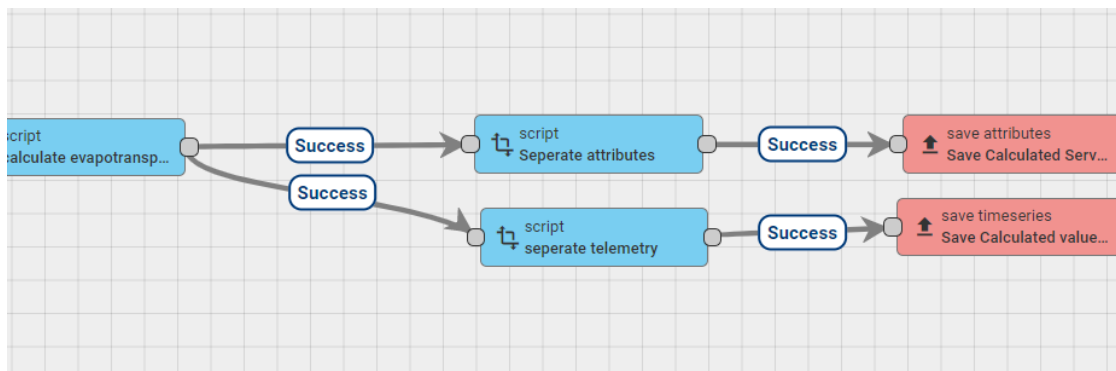


```

"precipitationLevel_mm": 0,
"cropEvapotranspiration_mm": 6.97,
"referenceEvapotranspiration_mm": 8.2,
"resultCropCoefficient": 0.85,
"averageSoilMoistureAtDayChange": 31.23,
"maxIrrigation_hrAtDayChange": 0,
"maxIrrigation_hrminAtDayChange": 0,
"irrigationRequiredStatusAtDayChange": false,
"deltaSoilMoisture": 0.47
},
"metadata": {
  not important
}
}

```

Some of these values need to be saved as telemetry / timeseries, and some needs to be saved as server attributes. So upon successfull calculations 2 transformation scripts are needed to seperate the timeseries and the attributes



Script 1 is a has the following code

```

//create an empty object which will be used to store the final result.
var newMsg = {}; //empty object

newMsg.julianDayNumber = msg.julianDayNumber;
newMsg.sunsetHourNumber_rad = msg.sunsetHourNumber_rad;
newMsg.extraterrestrialRadiation_MJm2day = msg.extraterrestrialRadiation_MJm2
day;
//also save this as server attributes so that it is used on the next Loop the
next day
newMsg.averageSoilMoistureAtPreviousDayChange = msg.averageSoilMoistureAtDayC

```



```
hange;  
//update the memory  
newMsg.memory = msg.memory;  
  
var msgType = "POST_ATTRIBUTES_REQUEST";  
  
return {msg: newMsg, metadata: metadata, msgType: msgType};
```

more specifically this separates the: - julianDayNumber which has been update on day change. - latitude\_rad - sunsetHourNumber\_rad - extraterrestrialRadiation\_MJm2day - averageSoilMoistureAtPreviousDayChange needs to be stored for calculation of the delta for the next loop.

the resulting message is

```
{  
  "msg": {  
    "julianDayNumber": 208,  
    "latitude_rad": 0.6093,  
    "sunsetHourNumber_rad": 1.8164,  
    "extraterrestrialRadiation_MJm2day": 39.8591,  
    "averageSoilMoistureAtPreviousDayChange": 31.23  
  },  
  "metadata":{  
    not important  
  }  
}
```

Similarly script 2 separates the telemetry / timeseries and has the following code .

```
//create an empty object which will be used to store the final result.  
var newMsg = {}; //empty object  
  
//this are the values that are calculated and we need to store as telemetry  
newMsg.minTemperature = msg.minTemperature;  
newMsg.maxTemperature = msg.maxTemperature;  
newMsg.temperatureMean = msg.temperatureMean;  
newMsg.precipitationLevel_mm = msg.precipitationLevel_mm;  
  
newMsg.cropEvapotranspiration_mm = msg.cropEvapotranspiration_mm;  
newMsg.referenceEvapotranspiration_mm = msg.referenceEvapotranspiration_mm;
```



```
newMsg.resultCropCoefficient = msg.resultCropCoefficient;
newMsg.warningFarmer = msg.warningFarmer;

if (msg.hasOwnProperty('warningFarmer'))
{
    newMsg.warningFarmer = msg.warningFarmer;
}

if (msg.hasOwnProperty('warningConsultant'))
{
    newMsg.warningConsultant = msg.warningConsultant;
}

if (msg.hasOwnProperty('averageSoilMoistureAtDayChange'))
{
    newMsg.averageSoilMoistureAtDayChange = msg.averageSoilMoistureAtDayChange;
}

if (msg.hasOwnProperty('maxIrrigation_hrAtDayChange'))
{
    newMsg.maxIrrigation_hrAtDayChange = msg.maxIrrigation_hrAtDayChange;
}

if (msg.hasOwnProperty('maxIrrigation_hrminAtDayChange'))
{
    newMsg.maxIrrigation_hrminAtDayChange = msg.maxIrrigation_hrminAtDayChange;
}

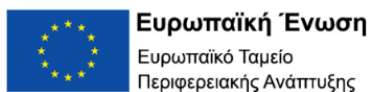
if (msg.hasOwnProperty('irrigationRequiredStatusAtDayChange'))
{
    newMsg.irrigationRequiredStatusAtDayChange = msg.irrigationRequiredStatusAtDayChange;
}

var msgType = "POST_TELEMETRY_REQUEST";

return {msg: newMsg, metadata: metadata, msgType: msgType};
```

with the resulting message being

```
{
  "msg" : {
```





```
"minTemperature": 18,  
"maxTemperature": 39.9,  
"temperatureMean": 28.95,  
"precipitationLevel_mm": 0,  
"referenceEvapotranspiration": 8.18,  
"cropEvapotranspiration_mm": 6.97,  
"referenceEvapotranspiration_mm": 8.2,  
"resultCropCoefficient": 0.85,  
"averageSoilMoistureAtDayChange": 31.23,  
"maxIrrigation_hrAtDayChange": 0,  
"maxIrrigation_hrminAtDayChange": 0,  
"irrigationRequiredStatusAtDayChange": false  
},  
"metadata": {  
  not important  
}  
}
```

## Conclusion

In conclusion, our IoT platform for collecting data from environmental sensors has proven to be a successful solution. The platform achieved satisfactory results, meeting our expectations in terms of battery life and sensor observation system functionality. The platform has demonstrated its ability to collect, process and analyze data from a range of sensors, providing valuable insights into the environmental conditions of the study area.

The decision to use an open-source platform was a wise choice on the second iteration of the platform, as it allowed us to focus our efforts on customizing and optimizing the platform to meet our specific requirements, while also benefiting from the ongoing support and maintenance of the wider developer community.

Overall, the success of the project has demonstrated the potential of IoT technologies in environmental monitoring and data collection. The platform offers a reliable and efficient solution for collecting and analyzing environmental data, with potential applications in a wide range of fields, including agriculture, natural resource management, and climate science.