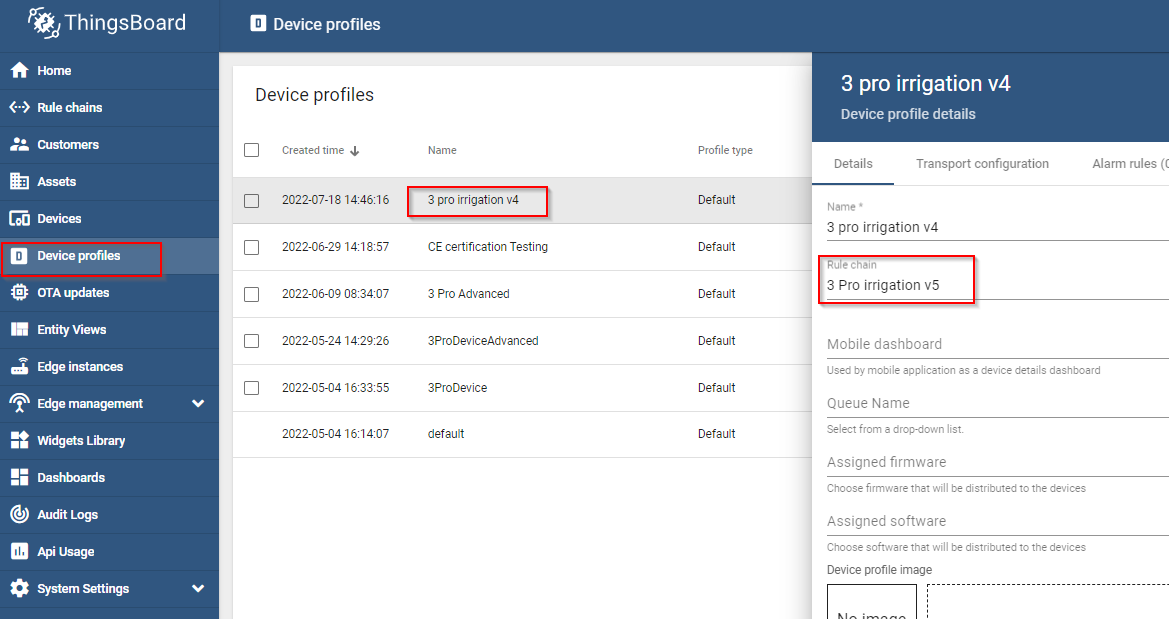
# Introduction on post processing

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 3 pro device. The current document describes the details of that implementation.

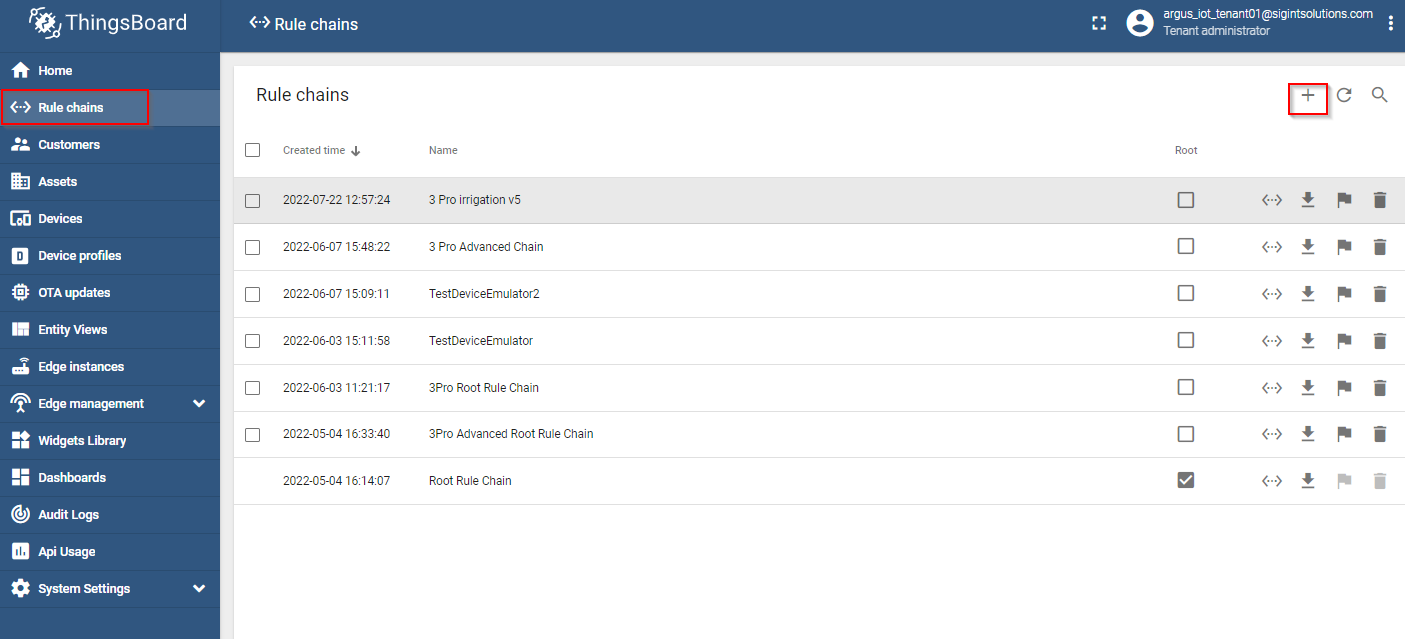
# Terminology

## Rule engine

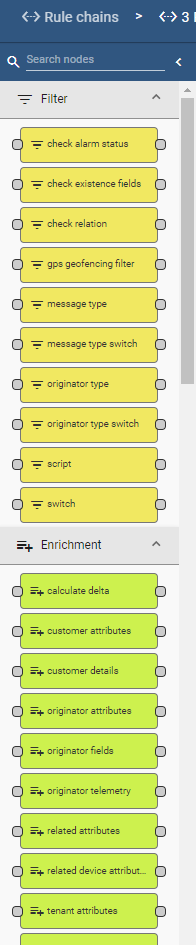
Rule engine is a post processing engine that events from the device or the dashboard or various other inputs are processed. Each processing engine is bound to a specific device profile.



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.



## 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 transmision using the mqtt protocol, as well as http requests.

# device Properties.

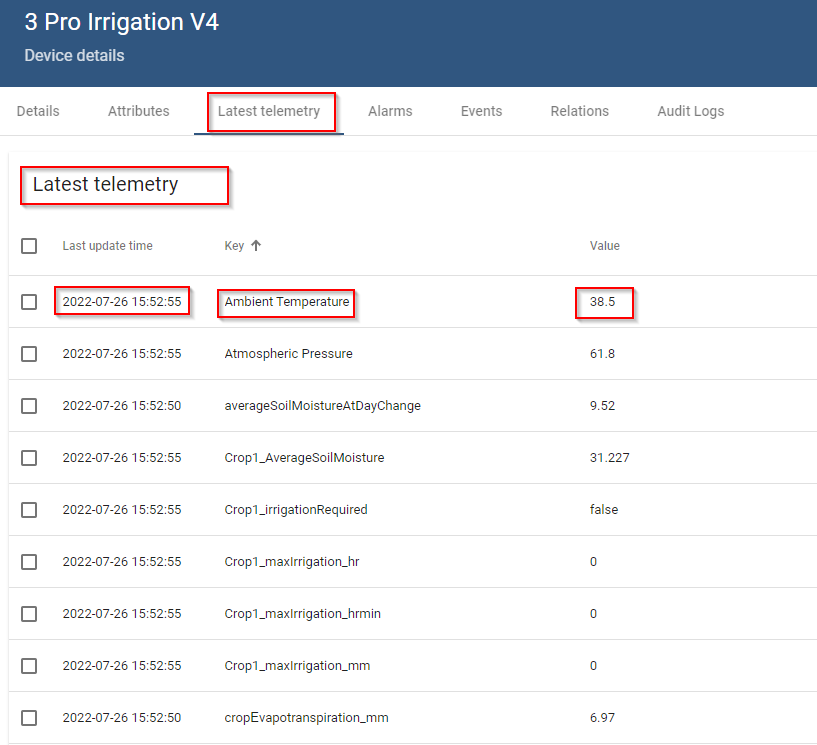
## Device profile

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 measurerement.

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



* 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

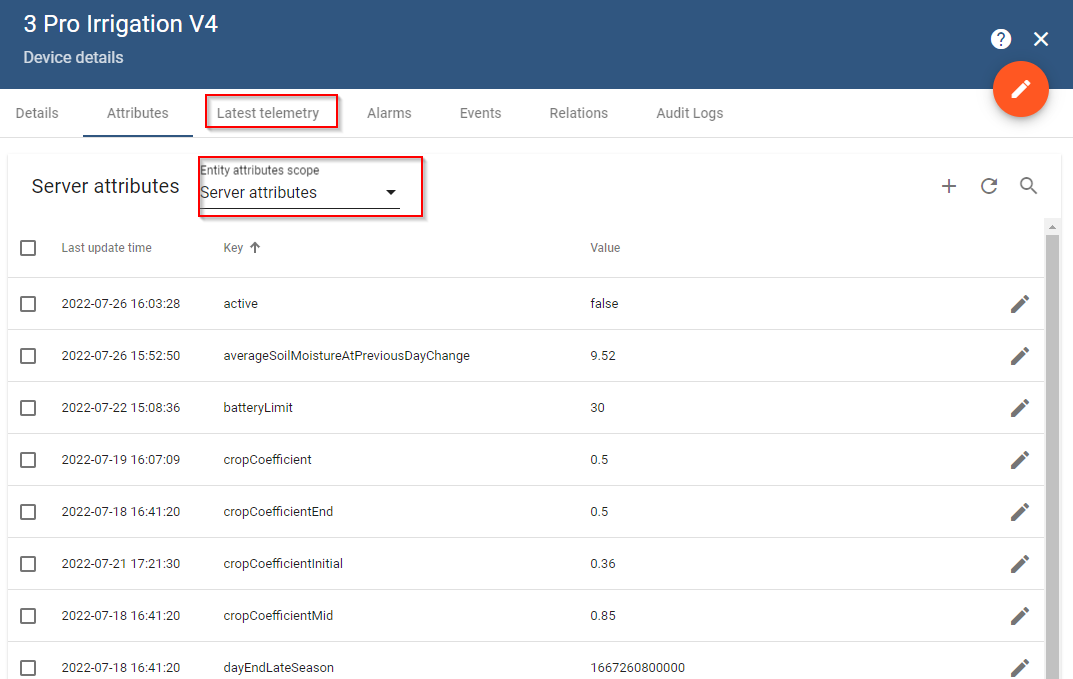
All of them are basically static values that are stored in the device entity in the platform, and they can be used to store some constants that are used for calculations on the post processing rule chains, or to display some information to the end user.

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.



## 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.

# 3 pro 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^2 |  |
| wiltingPoint | double | 18 | % |  |
| fieldArea | double | 25 | m^2 |  |
| 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 |  |
| 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 thaninactivityTimeoutMinutes`
* fieldCapacity: Volumetric soil water content after 24-48 hour drainage of saturated soil (cm3\_water/cm3\_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 (m2). This area should be smaller than the field area, unless the whole field is flooded.
* wiltingPoint: ASK ADRIANA
* fieldArea: This is the field area (m2) 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 conjuction 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 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
* 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
* 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 conjuction with the current value.

### Configuring device

* TODO

## 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","longitude":"33.342518"}" --cafile "/mnt/c/1/STAR\_sigintsolutions\_com/AAACertificateServices.crt"

This will transmit telemetry data points for ‘temperature’, ‘latitude’, ‘longitude’

## Setting the device to use the rule chain.

TODO

# 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 (0 - 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
* **Root zone depth** RootDepth is calculated by the summation of all the sensor properties

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
* **Soil Moisture Average** SoilMoistureAverage is calculated using the following equation.
* **Max Irrigation mm** MaxIrrigation\_mm is calculated using the following equation
* **Max Irrigation Hr** MaxIrrigation\_Hris calculated using the following equation
* **Max Irrigation Hr.Min** MaxIrrigation\_HrMin Is calculated using the following equation.
* **Solar Declination** SolarDeclination or δ is calculated using the following equation
* 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° (the northern hemisphere summer solstice) and a minimum angle  
  on December 21-22 which is of -23.45° (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°) on the June solstice, then starts decreasing until reaching its minimum (−23.44°) 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

* **Sunset hour angle** is SunsetHourAngle\_rad calculated using the following equation
* Extraterrestial radiation ExtRadiation is calculated using the following equation.

$$ ExtRadiation = (24 \* \frac{60}{\pi}) \* solarConstant \* InverseRelativeDistanceEarthSun \*\\
((SunsetHourAngle\_{rad} \* \sin(latitude\_{rad}) \* \sin(solarDeclination\_{rad})) +
(\cos(latitude\_{rad}) \* \\\cos(solarDeclination\_{rad}) \* \sin(SunsetHourAngle\_{rad}))); $$

where:

* **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. Finnaly when between End mid season and End late season, a linear interpolation is performed between Kc2 and Kc3

* **reference evapotranspiration** ETO\_mm is calculated using the following equation
* **crop evapotranspiration** ETc\_mm is calculated using the following equation.

# 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.

## Explainations of nodes

### Switch node

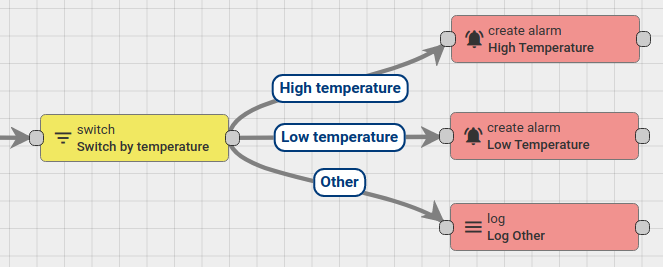
function Switch(msg, metadata, msgType): string[]

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

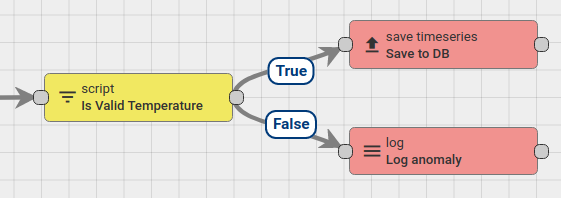
function Filter(msg, metadata, msgType): boolean

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:

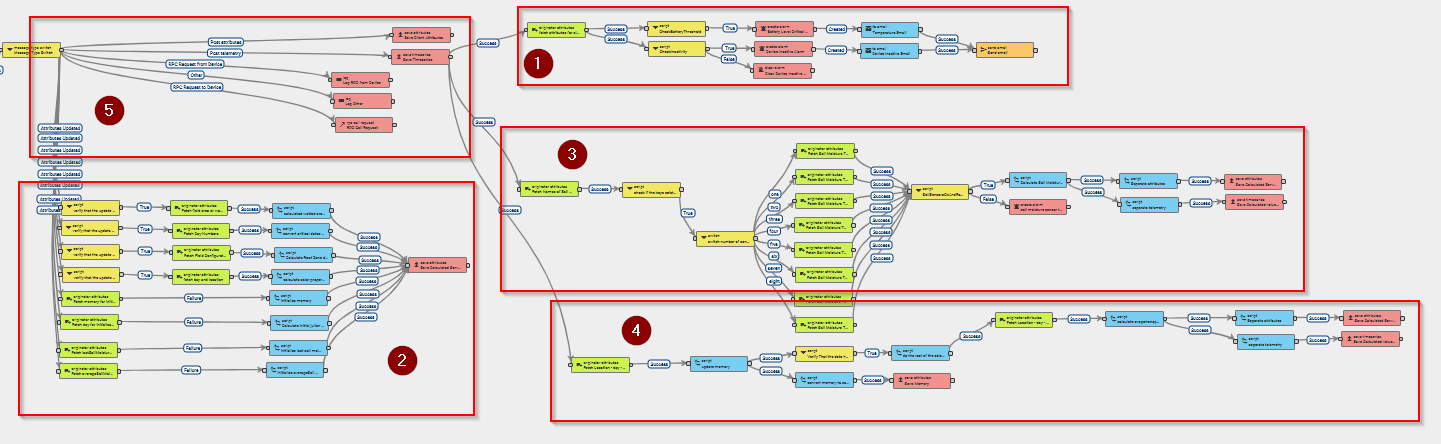
{   
 msg?: {[key: string]: any},  
 metadata?: {[key: string]: string},  
 msgType?: string  
}

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.

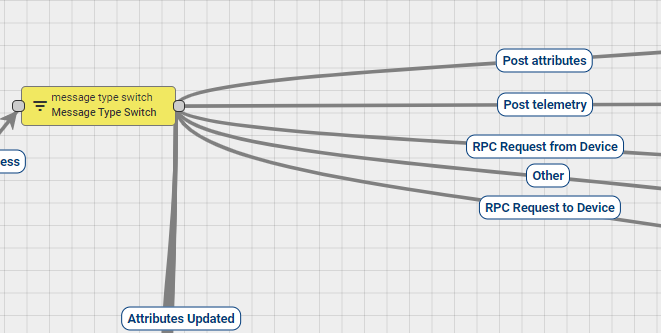


The rule chain consists of 5 different group of nodes that perform various operations.

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.

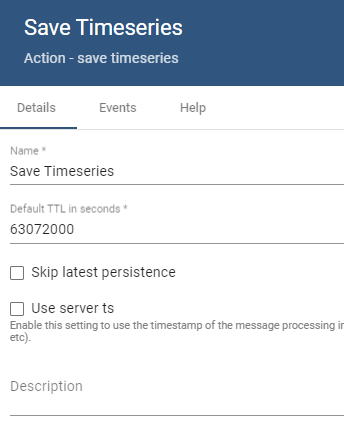


The most critical options in this situation is the POST\_TELEMETRY and the ATTRIBUTES\_UPDATED.

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



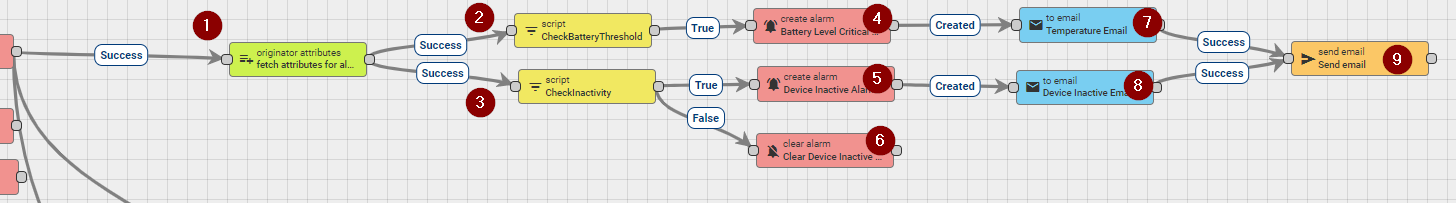
The telemetry values are then stored in the database for the amount of time that it has been indicated on the node settings



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)

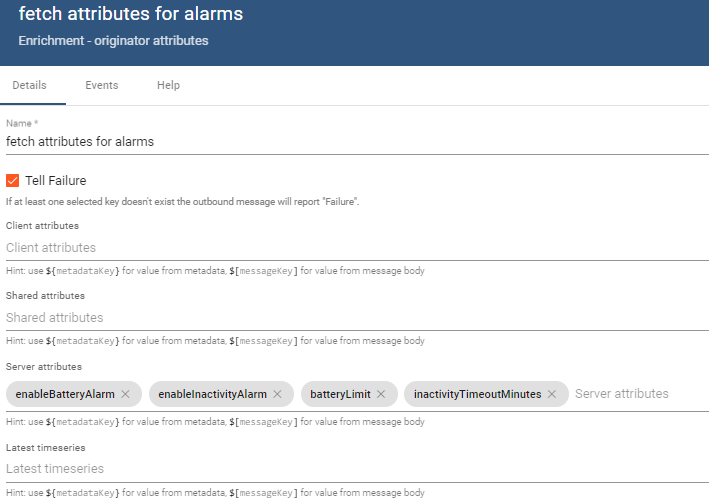


### **originator attributes [1] node**

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



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 thoughout 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.

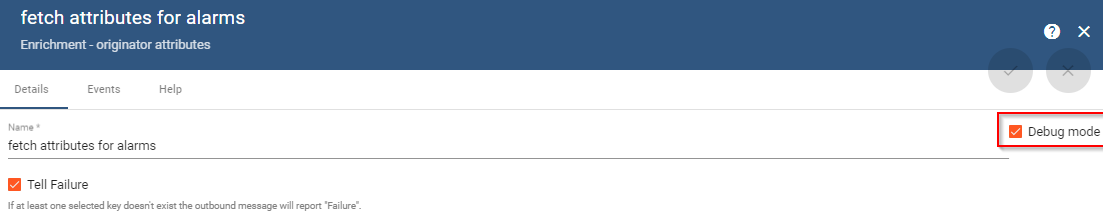
{  
 "msg": {  
 "Ambient Temperature": 25.9,  
 "Atmospheric Pressure": 88,  
 "Luminosity": 51.3,  
 "Soil Temperature 01": 26,  
 "Soil Temperature 02": 20.2,  
 "Soil Temperature 03": 24.9,  
 "Soil Temperature 04": 10.9,  
 "Soil Temperature 05": 18,  
 "Soil Temperature 06": 19.8,  
 "Soil Vol 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.

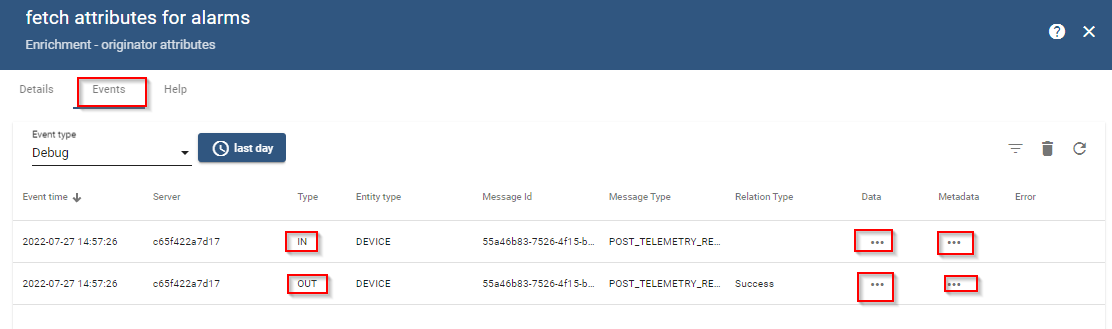
{  
 "msg": {  
 "Ambient Temperature": 25.9,  
 "Atmospheric Pressure": 88,  
 "Luminosity": 51.3,  
 "Soil Temperature 01": 26,  
 "Soil Temperature 02": 20.2,  
 "Soil Temperature 03": 24.9,  
 "Soil Temperature 04": 10.9,  
 "Soil Temperature 05": 18,  
 "Soil Temperature 06": 19.8,  
 "Soil Vol 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.



Then on next transmission can be examined on the events tab



### **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.



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]**

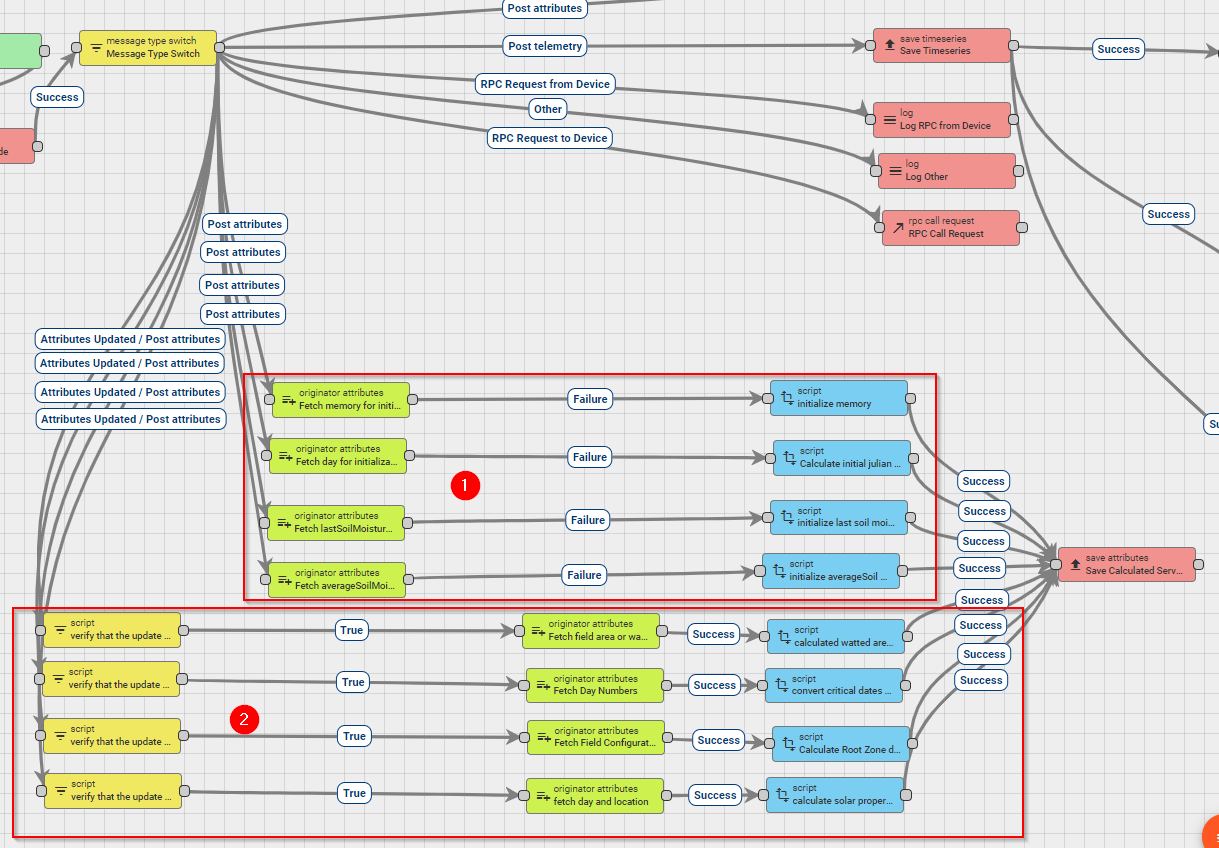
Similarly script node 3 will be triggered after the originator attribute [1] node, and will execute a script that will return true when the device is inactive for a period longer than the configured device timeout.

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 timestamps are stored is unix standard time in milliseconds internally.   
 if ((Date.now() + (parseFloat(metadata.ss\_inactivityTimeoutMinutes) \* 60000)) > 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 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.

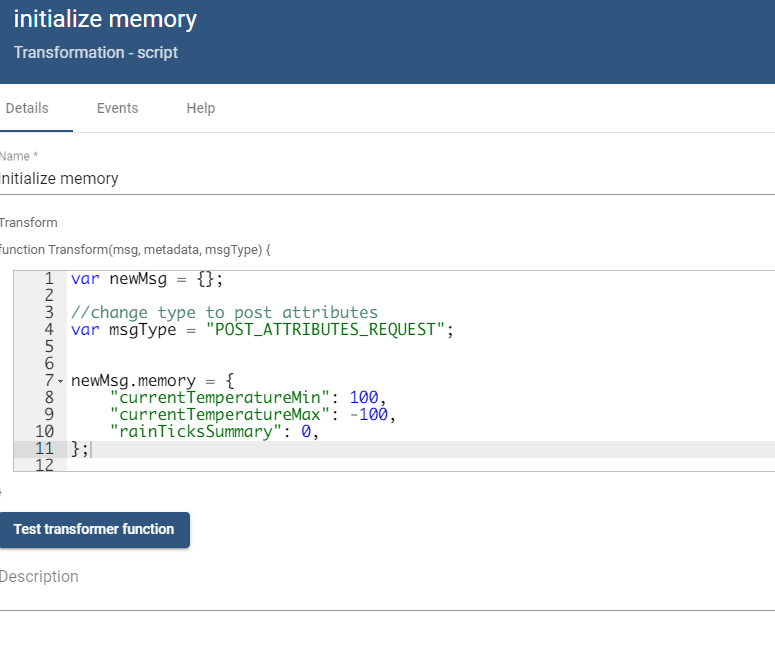
### subgroup 1 description.

#### 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 tranformation script, which will create that variable. The transformation script runs scripts in javascript language.



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 transfered 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 tranformation 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 available,   
//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 description.

Subgroup 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.

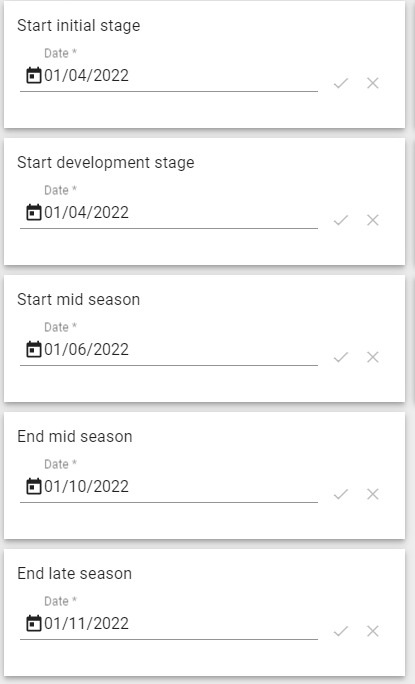
//requires the field configuration data as well as  
//ss\_fieldCapacity  
//ss\_wettedArea  
//ss\_soilMoistureThresshold  
//ss\_dischargeRate  
var newMsg = {};  
  
//---calculate wet area fraction  
newMsg.wettedAreaFraction = parseFloat(metadata.ss\_wettedArea) / parseFloat(metadata.ss\_fieldArea);  
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 beggining of the crops, are stored internally as unix standard time in ms, and they are selected by the user using standard DateTimeSelectors widget.



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[sensorIndex];  
 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 dependend 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 teh following code.

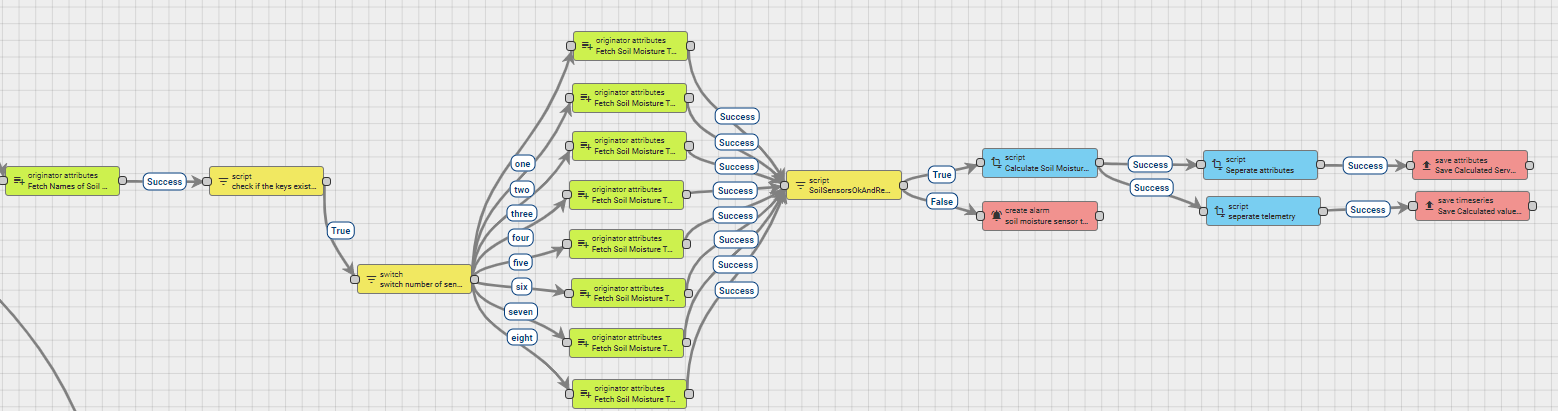
function DegToRad(degrees)  
{  
 return degrees \* (Math.PI / 180.0);  
}  
  
var newMsg = {};  
  
  
//also calculate the solar declination and some other things  
//---calculate solar properties  
//The following equation can be used to calculate the declination angle: δ=−23.45°×cos(360/365×(d+10))   
// 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 northern hemisphere summer solstice) and a minimum angle   
//on December 21-22 which is of -23.45° (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°) on the June solstice, then starts decreasing until reaching its minimum (−23.44°)   
//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 \* (parseInt(metadata.ss\_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) \* parseInt(metadata.ss\_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.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(extraterrestrialRadiation\_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.

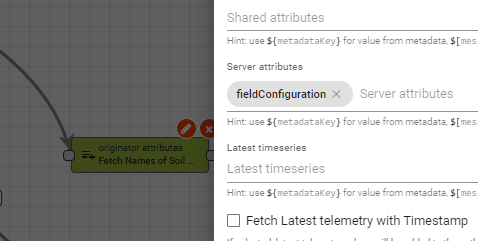
* latitude\_rad
* solarDeclination\_rad
* sunsetHourNumber\_rad
* extraterrestrialRadiation\_MJm2day

## Group 3, Soil average calculation

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

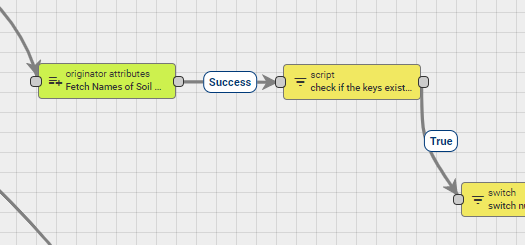


The calculation starts with a originator attributes node that executes every time that a value is saved on the database and fetches the sensor attribute fieldConfiguration This cannot be avoided because the rule chain needs to find the name of the keys that will be used to trigger the rest of the calculation.



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.

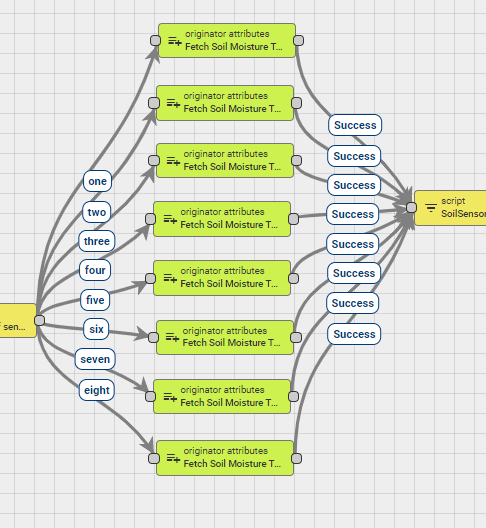
//fetch the name of the sensors  
var fieldConfiguration = JSON.parse(metadata.ss\_fieldConfiguration);  
  
//this works only for the first crop right now  
var index = 0;  
  
//get the name of the root object. it will be used to select the sensor array  
var name = Object.keys(fieldConfiguration.groupsArray[index]);  
  
//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[sensorIndex];  
 //check if the sensor key exists on the message  
 //if the message has the sensor key, we can do more post processing downstream.  
 if (msg.hasOwnProperty(sensor.meas))  
 {  
 return true;  
 }  
}  
return false;

This scripts 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 messaged 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 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;  
 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.



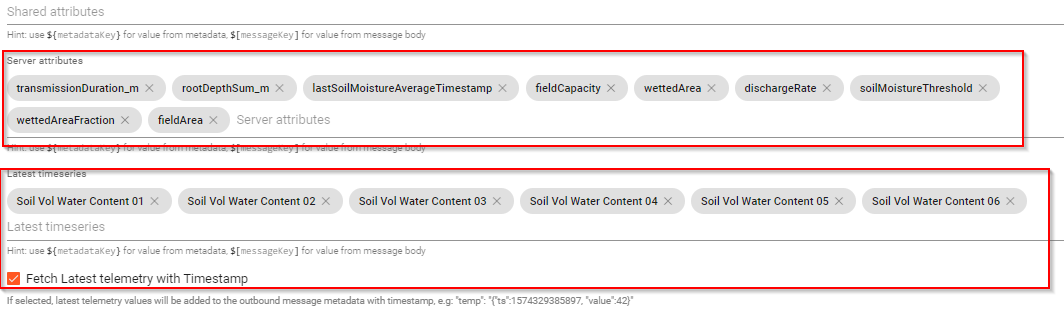
The reason is that originator attribute node needs to fetch the relevant telemetry for the database.

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 explaination of why this is needed. When a logger sends all the telemetry keys on the same package, then averaging is straightfoward 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.



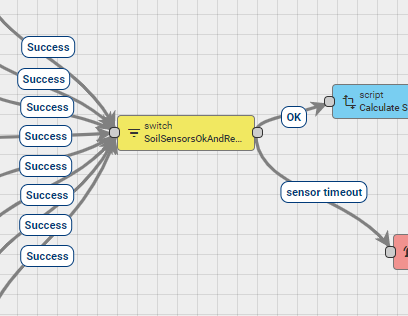
Soil Vol Water Content XX

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.

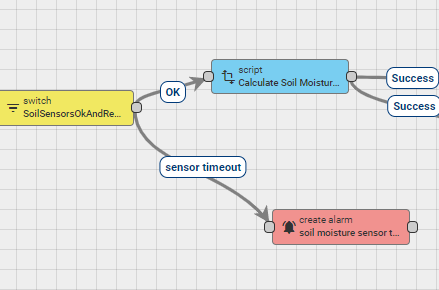


Downstream a switch node is executed with the following code

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'];  
 }  
   
 //finaly check that this average hasn't been already stored  
 //if the timestamp is the same (or less) than the last time that the average 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 meaningfull 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 thickness 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 guarranted from the previous node.   
 var sensorSoilThickness = parseFloat(sensor.soil\_thickness) \* m\_to\_cm \* parseFloat(sensor.weight);  
   
 //this will pull the "SoilMoisture01": "{\"ts\":1654681764417,\"value\":22.3}" 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;  
  
//finaly 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};

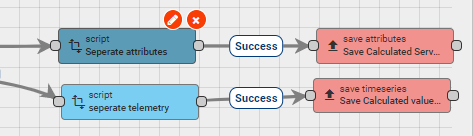
The above script calculates andp places it on the message. \* Crop1\_AverageSoilMoisture \* Crop1\_maxIrrigation\_mm \* Crop1\_maxIrrigation\_hr \* Crop1\_maxIrrigation\_hrmin \* Crop1\_irrigationRequired \* latestSoilMoistureTimestamp

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 accomplice 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 separates the timeseries from the message and has the following code

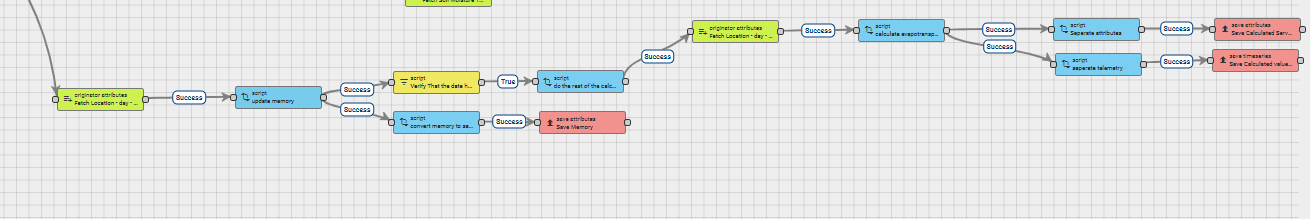
//create an empty object which will be used to store the final result.   
var newMsg = {}; //empty object  
newMsg = msg;  
//only remove the latestSoilMoistureTimestamp and keep the rest of the properties  
delete newMsg.latestSoilMoistureTimestamp;  
var msgType = "POST\_TELEMETRY\_REQUEST";  
return {msg: newMsg, metadata: metadata, msgType: msgType};

and the following output.

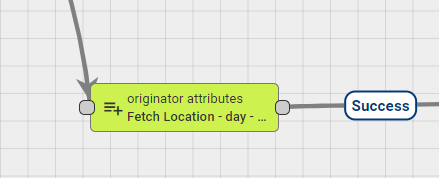
msg: {  
 "Crop1\_AverageSoilMoisture": 22.271,  
 "Crop1\_maxIrrigation\_mm": 12.456,  
 "Crop1\_maxIrrigation\_hr": 0.714,  
 "Crop1\_maxIrrigation\_hrmin": 0.428,  
 "Crop1\_irrigationRequired": false  
}

## 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.

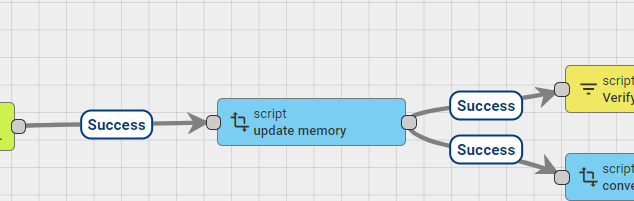


* julianDayNumber
* memory
* nameOfTemperatureKey
* latitude
* rainPerTick

Not all attributes will be used immidiately.

The resulting message will me 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};

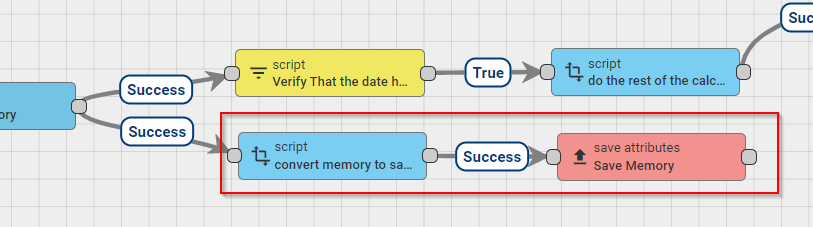
The script calculates the: julianDayNumber. If the incoming message has a temperature key, it also updates the currentTemperatureMin and currentTemperatureMax in the memory server attribute.

assuming tha 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.

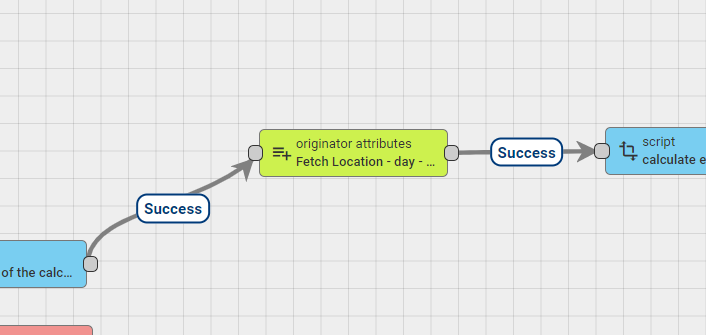
on true, an additional transformation script is executed with the following code

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 daty 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: δ=−23.45°×cos(360/365×(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° (the northern hemisphere summer solstice) and a minimum angle   
//on December 21-22 which is of -23.45° (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°) on the June solstice, then starts decreasing until reaching its minimum (−23.44°)   
//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));  
  
//finaly since the day has changed, we need to reset the memoryJSON  
newMsg.memory = {  
 //initialize as the last measurement, so even if no measurents 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};

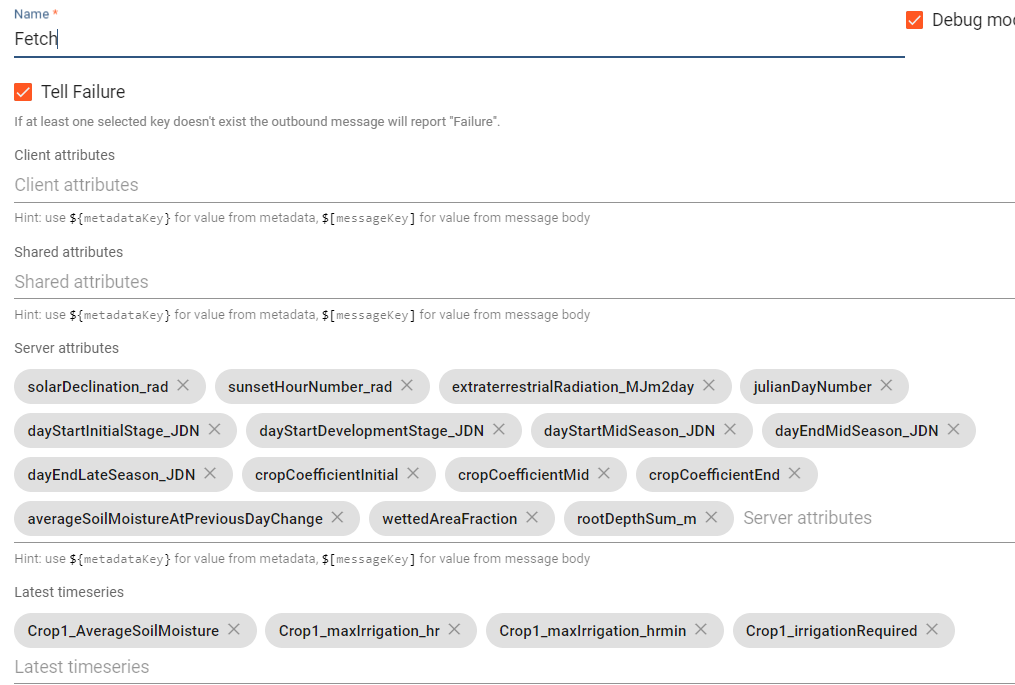
Since the day has changed, this script stores the currently max and min value of the temperature, as will as it calculates various properties.

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.



The node pulls \* solarDeclination\_rad \* sunsetHourNumber\_rad \* extratterestialRadiation\_MJm2day \* julianDayNumber \* dayStartInitialStage\_JDN \* dayStartDevelopmentStage\_JDN \* dayStartMidSeason\_JDN \* dayEndMidSeanon\_JDN \* dayEndLateSeanon\_JDN \* cropCoefficientInitial \* cropCoefficientMid \* cropCoefficientEnd \* averageSoilMoisture \* AtPreviousDayChange \* wettedAreaFraction \* rootDepthSum\_m

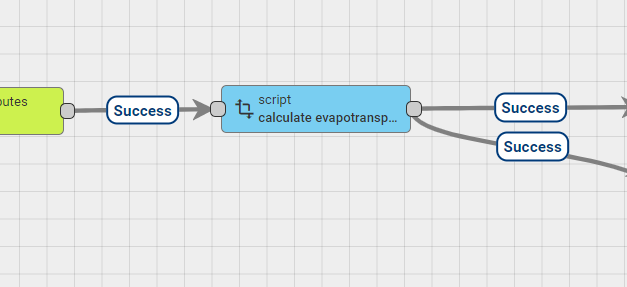
as well as the latest timeseries \* Crop\_AverageSoilMoisture \* Crop\_maxIrrigation\_hr \* Crop\_maxIrrugation\_hrmin \* Crop\_irrigationRequired

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

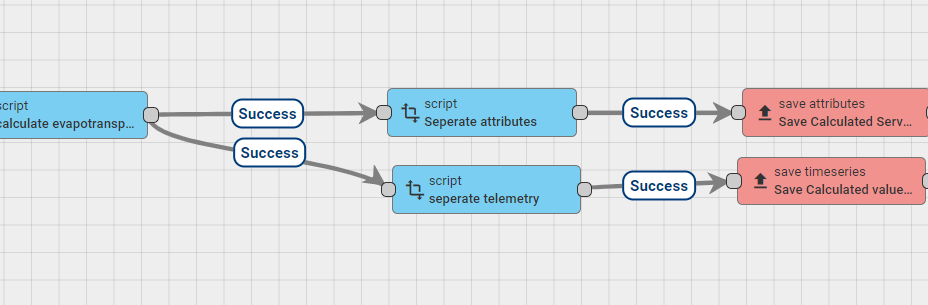
/\* at this point we have a message on the chain which has has the following properties  
{  
 "julianDayNumber": 206,  
 "solarDeclination\_rad": 0.3431,  
 "sunsetHourNumber\_rad": 1.8228,  
 "extraterrestrialRadiation\_MJm2day": 40.0462,  
 "minTemperature": 18.2,  
 "maxTemperature": 39.3,  
 "temperatureMean": 28.75,  
 "precipitationLevel\_mm": null,  
}  
  
we also have the following metadata.   
since we pulled the julianDayNumber and the extretterestial radiation from the database now before saving the updated values,   
we have the calculated values from the previous day, so we can use them on the 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 extraterrestial 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.cropΕvapotranspiration\_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\_hrminAtDayChange.value.toFixed(2));  
}  
  
if (metadata.hasOwnProperty('Crop1\_irrigationRequired'))  
{  
 irrigationRequiredStatusAtDayChange = JSON.parse(metadata.Crop1\_irrigationRequired);   
 msg.irrigationRequiredStatusAtDayChange = irrigationRequiredStatusAtDayChange.value;  
}  
  
//calculate deltaSoilMoisture  
var averageSoilMoistureAtPreviousDayChange = parseFloat(metadata.ss\_averageSoilMoistureAtPreviousDayChange);  
var deltaSoilMoisture = 0.1 \* (msg.averageSoilMoistureAtDayChange - averageSoilMoistureAtPreviousDayChange) \* parseFloat(metadata.ss\_rootDepthSum\_m) \* parseFloat(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 \* cropΕvapotranspiration\_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,  
 "cropΕvapotranspiration\_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\_MJm2day;  
//also save this as server attributes so that it is used on the next loop the next day  
newMsg.averageSoilMoistureAtPreviousDayChange = msg.averageSoilMoistureAtDayChange;  
//update the memory  
newMsg.memory = msg.memory;  
  
var msgType = "POST\_ATTRIBUTES\_REQUEST";  
  
return {msg: newMsg, metadata: metadata, msgType: msgType};

more specifically this seperates the: - julianDayNumber whic has been updade 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 seperates 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,  
 "cropΕvapotranspiration\_mm": 6.97,  
 "referenceEvapotranspiration\_mm": 8.2,  
 "resultCropCoefficient": 0.85,  
 "averageSoilMoistureAtDayChange": 31.23,  
 "maxIrrigation\_hrAtDayChange": 0,  
 "maxIrrigation\_hrminAtDayChange": 0,  
 "irrigationRequiredStatusAtDayChange": false  
 },  
 "metadata": {  
 not important  
 }  
}

TODO. add the ability to add more crop calculations seperating the field configuration. remove the update of latitude\_rad on group 4 remove the old evapotransporation calculation on group 4.