SETTING UP (SOLAR) ENERGY MANAGEMENT

Using Home Assistant

Abstract

This document describes the means and methods to setup (solar) energy management using home assistant.

Marc Hauptmann, PhD

dr.marc.hauptmann@gmail.com

1 Introduction

This document describes a procedure to setup an automated (solar) energy management system with remote access capability. The document describes the required resources and provides step-by-step instructions for installation and customization of the different hard- and software components. The setup makes use of the "Home Assistant" operating system (Home Assistant (home-assistant.io))

2 Setting up the basic components and the environment

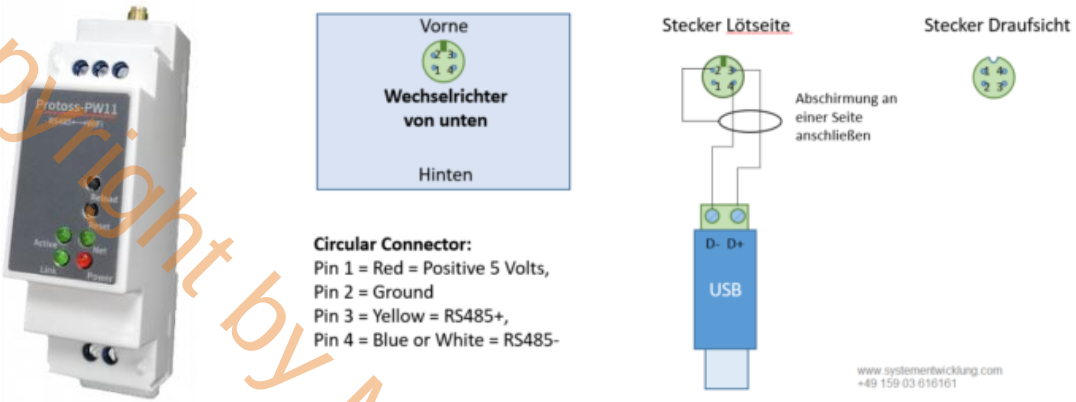
2.1 Hardware overview and sourcing

The following table summarizes the required components, where they can be obtained and their approximate pricing (based on Q3/2023 prices).

Component	Function	Sourcing	Price
Protoss PW11-H	Modbus TCP/IP bridge to connect the raspberry PI to the solar inverter. The version here is powered directly by 230V single phase via the electrical cabinet and connects to the raspberry PI via WiFi.	AliExpress	32,31 €
Sencys VOB electric cables 4mm² x3	Cables to connect the protoss interface directly to 230V AC (hence the thickness)	Brico	7,19 € per 5m
RS485 Modbus cable 1pr 24 AWG 10m	2 pole modbus cable to connect the TCP/IP bridge to the inverter.	Bol.com	29,95 €
Electric wire ferrules. (0.5 – 2.5mm)	It is highly recommended to use these for connecting the wires inside the modbus cable to the protoss TCP/IP bridge	Brico	9,99 € per set of 50pcs
RS485 connector for Solis inverter (Exceed-conn EC04681-2014-BF bundle)	Connector to connect the cable to the COM port of the Solis inverter. Note that for other inverters, other connectors maybe required.	Ebay (avarixgmbh)	13,99 €
Vemico Raspberry PI4 4GB Starter KIT	Raspberry PI is the center piece of the installation running the Home Assistant OS and communicating with all smart components. The 4th version is recommended to run Home Wizzard properly, and the starter kit contains all basic components, such as a case, cables and an SD card storage medium.	Amazon (Vemico Webshop)	249,99 €
***Optional (recommended)* SSD update for RPi4**	This upgrade contains an 128GB SSD, an SSD to SATA adapter to connect the SSD to the RPi4 via its USB3 connectors, and a 3D printed case that can fit both the RPi4 as well as the SSD. SSD usage is recommended to prevent "sudden death" of the SD card.	Bol.com (freva.com webshop)	49,15 €
***Optional for digital meters:* Home Wizard WiFi P1 Meter**	WiFi Dongle that can be connected to the digital electricity meter's P1 port, and can be used to monitor electric consumption incl. peak power as well as gas consumption.	HomeWizard Energy webshop	29,95 €
Optional (for Z-Wave connectivity): **Aeotec Z-stick Gen 5+**	Z-wave is an open standard for the wireless connection of devices like fire alarms. **Important: only order model ZW090-C!** ***When used in combination with an SSD, an USB extension cable might be required.***	Bol.com	54,90

2.2 Installing and setting up the Modbus TCP/IP bridge

The PW11-H version of the Protoss TCP/IP modbus bridge can be installed directly into the electronic cabinet. The bottom 2 connectors are for supplying the AC voltage via the L and N wires, respectively. The top 3 connectors are the modbus +, - and ground connectors, respectively. On the inverter site, the modbus wires are soldered to the exceedconn connector as depicted below, the connector itself being attached to the inverter's COM-port.



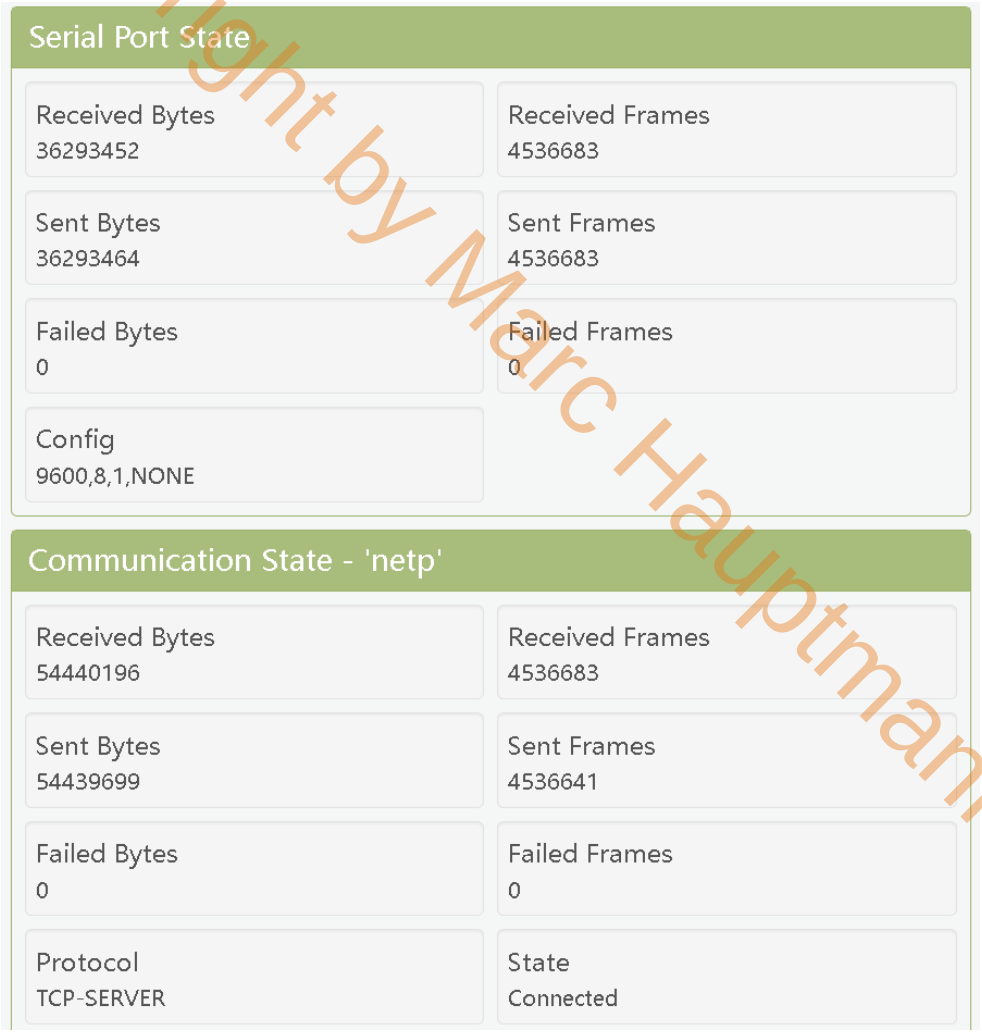
The protoss bridge can be configured via its intranet landing page. Before the TCP/IP bridge is setup for the first time, one has to connect to the PW11 wifi network to be able to access the configuration screen. The landing page can be accessed for the first time by the default IP 10.10.100.254, using the default user name and password *admin.* The following settings are recommended/required to enable communication between both the Protoss and Raspberry PI, as well as the Raspberry PI and the inverter:

- WAN settings should be configured to DHCP ON, with a fixed WAN IP and subnet mask. To identify the right settings, let the Protoss first set up its (external) IP and mask automatically (it might be required to manually restart the protoss via the reload button in order for it to "latch on"), then fix them via DHCP ON. The latter is important to ensure home assistant's connection to the TCP/IP-modbus bridge is stable.
    - For some routers (e.g. Mobile Vikings / Proximus Internet Box) it might be wise to also configure DHCP reservations using the IP and (WiFi!) MAC addresses visible on the intranet landing page.
- LAN settings should remain with DHCP Server OFF and standard internal IP (10.10.100.254) and mask (255.255.255.0).
- The Protoss should be configured in AP+STA mode. In this configuration, the Modbus bridge is connected both to home network via the router (meaning it is accessible within the home WiFi using its external IP) as well as accessible via the direct WiFi connection using the default IP. Use the *scan* feature in order to find the home network SSID and enter the STA KEY to connect.
- In communication settings, setup the TCP server. This is used to access modbus communication from the intranet site. Local port should be set to 8899, *buffersize* to 512 bytes*, keep alive* to 60s and *timeout* to 30s. Set *Protocol settings max accept* to 3, disable *Security* and set *Route* to Uart.
- In serial port settings, configure the serial side modbus connection (the one that goes into the inverter). Set baud rate to the baud rate of the inverter (can be checked at the inverter, but is

usually 9600), and configure the data bit, stop bit and parity (usually 8, 1 and None). *Buffer size* again to 512, *gap time* to 100, *Flow control* to Half Duplex, *Cli* to always, *waiting time* to 300, and *Protocol Settings* to Modbus.

At this point, it makes sense to verify whether the connection to the inverter is setup properly both hardware and software wise. For this purpose, a program like *Modbus Poll* can be utilized to send *Modbus* messages to the inverter via specific addresses. If e.g. the current date can be read out via its specific address, then the connection appears working. See here for the *Modbus addresses* for Solis' hybrid inverters: photovoltaikforum.com/core/attachment/202315-modbus-solis-rhi-5k-pdf/ (Please note that non-hybrid inverters will use different addresses!).

In case of successful connection and polling, the status page should show something similar to below screenshot, with non-zero number of sent and received bytes for the serial port, as well as the tcp server, which should also show the IP adress of the entity polling your inverter.



2.3 Installing and setting up the Raspberry PI with home assistant

For the assembly of the raspberry PI, follow the latest instructions on the manufacturers website. Home assistant runs as an operating system on the raspberry PI. For engineering purposes, it can also be run on a virtual machine like *Virtual Box;* however for longterm operation, installing and running it on a raspberry PI is strongly recommended. For installing home assistant onto the raspberry PI, follow the instructions outlined here: Raspberry Pi OS – Raspberry Pi.

When in the home network to which the raspberry PI is connected, home assistant can be accessed via http://homeassistant.local:8123/. Upon first start up, you will be asked to create an account, which later serves as the admin. It is advised to create multiple access accounts, e.g. in order to manage who can edit and who can only view dashboards, configurations, etc. This can be done under *Settings → Persons.*

To ensure stability especially for remote access, it is wise to fix home assistants IP address. This can be done via the router settings, which are typically accessible via the routers own landing page under its IP address. Additionally; it is recommended to fix the IP address inside home assistant via *Settings → System → Network → IPv4.* There, chose "static" and enter your IP + "/24" and leave the gateway and DNS as is. Again, for some routers (e.g. Mobile Vikings / Proximus Internet Box) it might be wise to also configure DHCP reservations using the IP and (WiFi!) MAC addresses of the RPi.

By default home assistant will show a number of systems connected to the home network, such as the router, range extenders, but also your smart phone and other smart home devices such as google chrome cast. These devices can also be typically monitored or even controlled via home assistant.

Make sure to setup basic settings such as time zones, country, units, etc. via *settings → system → general*, and enable advanced mode in your user profile in order to see all available tools and options.

2.4 Optional (but recommended): transferring storage to SSD

Tracking sensor values frequently in one of home assistant's databases might accelerate the wear of the SD card. It might be wise for these cases to have a backup SD card at hand. More importantly, in order to prevent the SD card from failing prematurely, one should consider upgrading the Raspberry Pi with an SSD.

The SSD should be larger than the SD card used. After build in, in home assistant go to: *Settings → System → Storage → transfer storage.* This will start the transfer process which may take several minutes to complete. **Keep the Raspberry Pi powered to prevent data loss** (and preferably perform a backup beforehand, see section 3.1.3 on how to setup automated cloud backup)!

In case of problems recognizing the SSD or SSD instability, disconnect other USB devices and if necessary connect an additional power supply to the USB-SATA adapter (12V).

2.5 Optional: setting up the HomeWizzard P1 meter

The home wizard P1 meter is a wifi dongle that can be connected to the P1 port of the smart electricity meter. With it, the readings of the smart electricity and connected gas meter can be readout. To connect the P1 meter to the network; follow the instructions in the companion app. (You might first have to open the P1 port on your digital electricity meter via the website Mijn Fluvius.) Once the P1 meter dongle is setup, the companion app is not needed anymore; we will monitor and store everything via home assistant. Once connected, the P1 meter should also automatically show up as an integration in home assistant. This can be checked under *Settings → Devices and Services.* There, one can also deactivate cloud connection of the P1 meter, in order to avoid data leakage (the data will stay within the home assistant environment and is completely managed by us, the end users)

2.6 Optional: Setting up a Z-Wave smoke detector (and other devices)

Home assistant allows the integration of smart devices like smoke detectors, via protocols like Z-Wave. Z-Wave has the advantage of good range due to the usage of lower frequencies than WiFi and fairly strict governance of the standard, at the expense of higher cost.

To enable Z-Wave in home assistant, one needs a Z-Wave dongle. One such device is the Z-Stick Gen 5 from Aeotec. Beware that this stick comes in 4 versions, and only 2 versions are compatible with raspberry Pi4 (amongst which version number *zw090*). ~~To setup the stick, and pair a device, follow the instructions given here: Setup Home Assistant with Z-Stick Gen5+ : Aeotec Help Desk (freshdesk.com).In short, one needs to first setup the Z-Wave JS UI, then add the Z-Wave JS integration.~~

In order to setup a z-wave stick and corresponding devices, follow the instructions on the home assistant site: Z-Wave - Home Assistant In short, add the z-wave integration in home assistant, when configuring it select the USB port of the stick and let the UI generate the security keys. Then the z-wave JS integration will be installed, and you can start adding devices to your network via its UI. By default, the home assistant z-wave integration will use the default home assistant add-on "Z-wave JS" to communicate with the z-wave stick and publish the results in home assistant entities. For better performance, and manageability (e.g. custom building of routes which comes handy with extenders), it is however though recommended to use the more powerful "z-wave JS **UI**" add-on. For instructions, how to migrate from the default to the new add-on why maintaining the configuration of the installation, follow this insightful guide: Switching Z-Wave JS Addons with Minimal Downtime! Z-Wave JS (Official) to Z-Wave JS UI (Community) - Community Guides - Home Assistant Community.

There are various smart devices available with Z-Wave compatibility such as the Heiman HS1SA-Z smoke detector (beware that only the version ending with *-Z* uses Z-Wave). Pairing instructions can be found on the internet presences of the manufacturers. In the case of the Heiman smoke detector, keep the network button pressed while inserting the battery to make it paired. Once paired, they will become available as entities within the Z-Wave JS integration (if the device has existed before, you may need to rename it to its previous name in order to be able to continue to use all of its associated entitities). To save you some hassle, make sure to properly remove and add devices via the add-ons UI. For trouble shooting by connecting the stick to your PC, you can install *Simplicity Studio* in order to simple add or remove devices outside of home assistant. See here for more detailed instructions: Z-Wave command class configuration tool download. : Aeotec Help Desk.

3 Configuring Home Assistant

3.1 Installing and configuring basic (recommended) add-ons

There are some add-ons that either enhance the functionality or usability of home assistant. Add-ons can be added under *Settings → Add-ons.* For all add-ons, make sure that after installation, automated updates and watchdog (restarts the add-on in case it is interrupted) are enabled.

3.1.1 Open Meteo

The default weather forecast in home assistant is provided by the Norwegian meteorologic institute. More accurate weather forecasts for the benelux were provided by *Buienradar,* however their API is unreliable and sometimes weather forecasts are not updated. An alternative, reliable weather forecast service is provided by *Open Meteo*. In order to add *Open Meteo* to home assistant just add the integration via *Settings → Appliances and services → add integration.*

3.1.2 File editor and advanced SSH & web terminal

The file editor is part of the official home assistant add-ons and is essential for editing files such as config.yaml. The advanced SSH & web terminal application is also part of the official add-ons, allowing the user to access the home assistant installation via the command line, e.g. in case repair is required. In both cases; follow the instructions for installation and usage. For ease of use, enable "show in home assistant sidebar" in the configuration tab. (The advanced SSH & web terminal application should be configured via the respective tab to set a userid and password. Verify the correct setup by open a

powershell terminal in windows on a PC located in the same network and type “ssh” followed by “hassio@” and your home assistant IP. Enter your password and you should see home assistants command line interface.).

More advanced user may also want to install the *Studio Code Server add-on* by Frenck, however, for development purposes it is recommended to install Studio Code Server on an external computer and then connect to home assistant and its drives using e.g. *Samba Share*, see section 3.1.7.

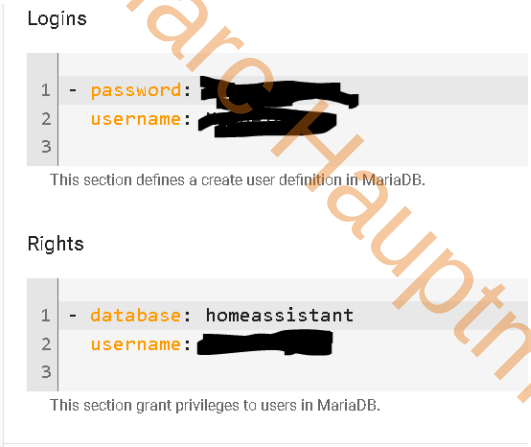
3.1.3 Home assistant google drive backup

The home assistant google drive backup is a community add-on developed by Stephen Bechem (Add-on: Home Assistant Google Drive Backup - Installation / Home Assistant OS - Home Assistant Community (home-assistant.io)). It can not only be used to schedule local backups on a regular basis, but also to backup these backups to your personal Google Drive. For detailed instructions follow the website. For ease of use, enable “show in home assistant sidebar” in the configuration tab.

3.1.4 Maria DB

If you’re working with an SD card as the primary storage medium in your raspberry PI, then it is highly recommended to switch storage of the home assistant history from the default SQLite database to Maria DB. Maria DB is available via the official home assistant add-ons collection.

Before starting Maria DB, it should be configured via the corresponding tab. Enter the credentials of the main account according to the screenshot below. Here username in the logins and rights section refers to the same username. After configuring start up Maria DB.



As a next step, go to file editor and open *secrets.yaml.* We will reference this file later in the *config.yaml* in order to enable database access and storage in MariaDB. In *secrets.yaml*, add this line: `mariadb_url:'mysql://username:yourpassword@core-mariadb/homeassistant?charset=utf8'`
Replace username with the username, and *yourpassword* with the password you configured MariaDB with. **Please note, when switching from SQLite to Maria DB, your previous history will be lost.**

3.1.5 Influx DB

Influx DB is an add on that allows to store time series data over longer periods and visualize them with e.g. Grafana. The DB runs in parallel to the home assistant DB that is used to store the short term history of the integrations and sensors.

Influx DB can be installed the easiest using the community add-on made by Franck Nijhof (GitHub - hassio-addons/addon-influxdb: InfluxDB - Home Assistant Community Add-ons). After installing the add-on, start it and open the web UI (you can pin the web UI to the left side bar in home assistant via the configuration tab). Next, in the UI, go to *Influx DB admin,* and create a new database named e.g.

"home_assistent". Here, you can also adapt the retention setting or you leave it at infinite, which is the default. In the *Users* tab, create a new user and set its password. Make sure you provide read and write rights to the home assistant DB you have just created. Lastly, add the username and password to the *secrets.yaml,* following this example:

```
influxdb_user: username
influxdb_pass: password
```

3.1.6 Jupyterlab

Jupyterlab is another add-on by Franck Nijhof (GitHub - hassio-addons/addon-jupyterlab: JupyterLab Lite - Home Assistant Community Add-ons), that allows to run python code e.g. in jupyter notebooks. It is essential if we want to perform advanced data analytics or even machine learning based on data collected in the home assistant databases.

Similar to influx DB, Jupyterlab can be installed via the add-on store community add-on collection. It is recommended to pin Jupyterlab to the left side bar via the configuration tab of the add-on for ease of access.

3.1.7 Samba Share

Samba Share is another add-on by Franck Nijhof (addons/samba at master · home-assistant/addons · GitHub), that allows to connect to home assistant's file system on an external Windows (e.g. desktop) computer via SMB. It is essential if we want to develop add-ons or "Apps" for specific automations such as solar forecasting.

Similar to influx DB, Samba Share can be installed via the add-on store community add-on collection.

In the configuration tab, input a username and password. Name the work group "WORKGROUP". On your windows computer, enter the home assistant IP address (e.g. \\192.168.1.48), then the username and password you configured, and the home assistant filesystem becomes accessible. Check "remember my credentials" to not having to enter the username and password each time you connect, and also consider making a short cut on your desktop for easy connection.

3.1.8 App Daemon

App Daemon is yet another add-on by Franck Nijhof (Home Assistant Community Add-on: AppDaemon 4 - Installation / Home Assistant OS - Home Assistant Community (home-assistant.io)), that allows to run python code as "Apps" for the purpose of advanced automations and services.

Similar to influx DB, App Daemon can be installed via the add-on store community add-on collection. The general configuration of App Daemon consists of 3 steps: 1. Specifying the Alpine Linux Packages that should be installed as prerequisites. This will be installed each time App Daemon is restarted. This can be a way to install python packages such as *scikit learn* and *pandas* that require compilation and contain a lot of dependencies, which would otherwise be challenging. 2. Specifying the python packages that are required to run the code (e.g. *beautifulsoup4*, *pysolar*). 3. Set the webserver port to 5050.

The usage and python coding principles within App Daemon will be explained in more detail in section 5.3.

3.1.9 HACS and apexcharts-card

There is a home assistant community store, which provides many more custom made integrations, as well as front-end elements. For installation instruction, please check out this website: Home Assistant Community Store | HACS.

One useful front end element is apexcharts, which implements the apexchart.js library. It can be used to make interactive graphs and add them directly into home assistant dashboards. The add-on allows a lot of flexibility in configuring the graphs, including advanced transformation functionalities for sensor data, which we can use to e.g. visualize solar forecasts.

3.2 Basic configuration via config.yaml

The basic configuration of home assistant is defined by the content of *config.yaml*. It consists of these parts:

1. The header contains mostly standard things regarding scenes; automations, voice activation.
2. In the *modbus* section, one can configure sensors that represent Modbus readouts via the specified addresses. An example is given here:

```
modbus:
  - name: "Solis_Inverter"
    type: tcp
    host: 192.168.1.9
    port: 8899
    sensors:
      - name: House load power
        data_type: uint16
        slave: 1
        address: 33147
        input_type: input
        count: 1
        unit_of_measurement: W
        state_class: measurement
        scan_interval:                                    20
```

   Here, *host* is the IP address that was configured for the Protoss TCP/IP modbus bridge, and *port* refers to the internal port of the TCP server that we set up in section 2.2. A sensor is typically configured with a name, date type; address and measurement unit according to the specification of the inverter. For Solis hybrid inverters, these specifications can be found in https://ginlongsolis.freshdesk.com/helpdesk/attachments/36112313359.
3. In *scrape*, we can define sensors that represent information that is scraped from the web, such as energy prices. An example is given here:

```
scrape:
# Example configuration.yaml entry
  - resource:
https://callmepower.be/nl/energie/leveranciers/eneco/tarieven/zon-en-wind-flex
    scan_interval: 3600
    sensor:
      - name: stroomprijs_dag
        unique_id: priceday
        select: "td"
        index: 10
        unit_of_measurement: EUR/kWh
        value_template: '{{ value | replace (",", ".") | float
* 0.01 }}'
```

   Resource represents the website from which the information is to be scraped. The scrape utility uses functionality of the *beautiful soup* package. Select and index are identical to the *select* method and indexing used by *beautiful soup* to retrieve information out of e.g. tables

on website. *Value template* can be used to format the retrieved information into home assistants internal formats.

4. The *template* section lets you define template sensors, which combine and transform readings of -often multiple- existing sensor, often times using Boolean conditions. This can be used to e.g define energy tariffs for which the prices change over the course of a day. Some example is shown here:

```
template:
  - sensor:
      - name: stroomprijs
        unit_of_measurement: EUR/kWh
        state: >
          {% set tariff = { "HT":
states('sensor.stroomprijs_dag'), "LT":
states('sensor.stroomprijs_nacht'), "DN":
states('sensor.p1_meter_5c2faf0f2bfc_active_tariff') | int} %}
          {% if (tariff.DN<2) %}
            {{ tariff.HT }}
          {% else %}
            {{ tariff.LT }}
          {% endif %}
```

Here, we make use of the P1 smart meter's own tariff indication, distinguishing day and night tariffs automatically, but they same can be also done time based using the internal clock feature of home assistant.

Another example are template sensor that can be used to calculate the required load and unload current for a battery to keep grid input within the "peak tariff" limit of 2.5 kW:

```
template:
- name: unload_current
        unit_of_measurement: A
        state: >
          {% set s1 = states('sensor.net_consumption') | float
%}
          {{(((min(max(0,(s1-1700)/5),1000)/50) | round(0,
default=0))*50) | round(0, default=0)}}
- name: load_current
        unit_of_measurement: A
        state: >
          {% set s1 = states('sensor.net_consumption') | float
%}
          {{(((max(0,(1700-s1)/5)/50)| round(0, default=0))*50)
| round(0, default=0)}}
```

Finally, we may want to setup a binary sensor, that triggers when the battery charge state is below or above a certain threshold. Compared to the binary sensor that can be configured via the helper section in home assistant (see section 3.3), the threshold can be defined via another input sensor and hence made configurable. The sensor additionally is configured via a hardcoded hysteresis to prevent rapid triggering when used in automations:

```
- binary_sensor:
      - name: batt_thresh_trigg
        #icon: mdi:thermometer-low
        availability: |
```

```
                {{ not false in
                   [states('sensor.battery_charge'),

states('input_number.Battery_management_threshold')]|map('is_nu
mber')
                }}
          state: |
                {% set
entity=states('sensor.battery_charge')|float(0) %}
                {% set
limit=states('input_number.Battery_management_threshold')|float
(0) + (4 if this.state=='on' else -4)  %}
                {{ entity < limit }}
```

5. Platform sensors are a useful addition, for instance to provide statistics of an sensor such as active meter power, see here for an example:

```
sensor:
  - platform: statistics
    name: "Active power minute average"
    entity_id: sensor.p1_meter_5c2faf0f2bfc_active_power
    state_characteristic: mean
    max_age:
      minutes: 3
```

6. Z-Wave smoke detectors can be used to trigger alarms and notifications. It is even possible to push high priority alarms to iOS and Android devices with the companion app installed. An example of such an alarm configuration is shown here:

alert:

```
  brandalarm_batterij:
         name: Rook bij batterij
         message: Er is rook bij de batterij!!
         done_message: Alarm uitgeschakeld.
         entity_id:
binary_sensor.brandalarm_batterij_smoke_detected
         state: "on"
         repeat: 1
         can_acknowledge: false
         skip_first: false
         notifiers:
             - mobile_app_iphone_van_delphine_2
             - mobile_app_sm_a505fn
         data:
                 push:
                   sound:
                     name: default
                     critical: 1
                     volume: 1
                     ttl: 0
                 priority: high
                 color: "red"
                 vibrationPattern: "100, 1000, 100, 1000, 100,
1000, 100, 1000"
```

```
                    importance: max
                    ledColor: "red"
                    channel: "alarm_ha"
```

Repeat can be used to repeat the notification according to the specified interval(s). The notifiers are the entities with the home assistant companion app installed, and data can be used to modify the notifications to e.g. always trigger a sound even the phone is in silent mode. Similarly, alarms can be set e.g. when the battery level drops or too much power is consumed. Every alarm also creates an entity, which can be used to acknowledge and subsequently toggle off the alarm. On android, above code creates a new channel the first time with above settings. Setting changes afterwards can be ignored. In order to remove a notification channel, use a service with message "remove_channel". Consequently, a service can also be used to initialize a new channel with above settings.

7. In the *recorder* section it can be specified for which sensors historical data should be stored in the *Maria DB* database for diagnostic purposes. An example of a *recorder* configuration is shown here:

```
recorder:
    db_url: !secret mariadb_url
    commit_interval: 60
    purge_keep_days: 30
    auto_purge: true
    include:
        domains:
            - automation
            - updater
        entities:
            - sensor.battery_total_power_dir
            - sensor.battery_charge
            - sensor.house_load_power
```

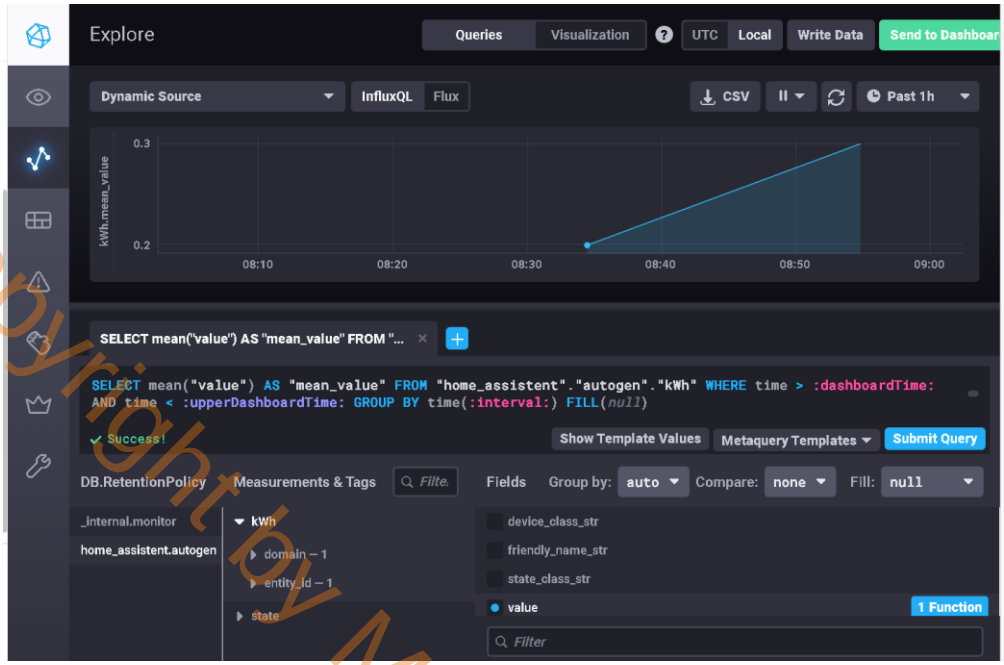
db_url is the url incl. credentials that was configured in section 3.1.4; When using an SD card as the storage medium, commit interval should be set to higher values (e.g. 60s) in order to avoid deteriorating the medium, while *purge_keep_days* should be kept to moderate number of days as to avoid running out of capacity. Under *include → entities*, the sensor for which historical data should be recorded can be defined. Alternatively, sensors can explicitly be excluded using *exclude,* though it makes more sense to keep the number of recorded sensor to a strict *need to have* base.

8. Similarly to *MariaDB,* InfluxDB can be configured to only record certain entities and sensors. This occurs very much similar to how MariaDB is configured. We just have to provide the host (equivalent to the home assistant IP) and port (typically 8086, but check the InfluxDB web-UI), as well as the credentials as setup in section 3.1.5. Here you can find a configuration example:

```
influxdb:
      host: 192.168.1.48
      port: 8086
      database: home_assistent
      username: !secret influxdb_user
      password: !secret influxdb_pass
      max_retries: 3
      default_measurement: state
      include:
        entities:
```

```
    - sensor.daily_power_generated
    - sensor.solar_forecast_hourly
```

In order to check, whether *InfluxDB* was correctly configured, checked the home assistant logbook for connection errors and check the data logger in its web UI for available data:



3.3 Setting up and "calibrating" utility meters

Utility meters are useful in order to track e.g. energy production or consumption over fixed periods (i.e. daily). That means their value will be reset at the end of the period, and a new cycle begins with each subsequent period. Utility meter like readings are also needed to track energy production and consumption via home assistant's energy dashboard.

While utility meters can be setup via *config.yaml* as well (e.g. `utility_meter:`

```
    gas_daily:
        name: Daily gas energy consumption
        source: sensor.gas_kWh
        cycle: daily),
```

it is often more convenient to set them up via *settings → appliances and services →helpers.* There one can set the source sensor, the meter period, customize the reset point, etc. By default, a utility meter starts its first cycle after setup with a count of zero. However in some situations it might be required to set the utility meter to a starting value that e.g. reflects the reading of the underlying sensor at the time of setup. This can be done via *development tools → services.* In service, select *Utility meter: calibrate*, select the target sensor, set the desired target value and execute.

3.4 Setting up dashboards

Dashboards are the front end by which sensor readings can be combined, visualized and organized. They can be setup and pre-configured via *settings → dashboards,* here e.g it can be configured if the dashboard is visible in the side bar and if they can only be edited by the admin. Dashboards can be edited, by clicking on the 3 points in the top right corner of the respective dashboard, and selecting *configure UI*. It is recommended to use these basic 3 types of dashboards:

1. The standard home assistant “landing page” dashboard. This is the default dashboard, with a default arrangement that depends on the registered integrations and entities. It is useful to add selected utility meter readings to the top of the dashboard as “badges”. Furthermore, widgets can be added for weather and sun forecasts as well as control of smart home integrations such as Google Chromecast. Lastly, one can make cards providing overview of the states and settings of multiple entities of the same device, e.g. Z-Wave smoke detectors, as well as alarms that allow the possibility to acknowledge and toggle off.
2. The energy dashboard is a preconfigured dashboard that is very useful to track power consumption, generation and other related statistics, for selected time intervals. The dashboard readings are updated on an hourly basis, so it is less suited for real time monitoring but more for collecting long term statistics over consumption and generation. The main configuration items are the utility meters that are used to track power consumption, generation, battery charge etc. The tracking is not limit to electricity but also e:g: gas consumption can be tracked as well. There is even -albeit functionally limited- functionality to forecast solar production. Do note that when a new input is configured, it may take up to 2 hours until the readings become visible in the dashboard. It is also possible to add energy tariffs (e.g. as configured in section 3.1.6.3) to track costs and profits. In order for the costs and profits to become visible in the dashboard it is not sufficient to only add the tariffs; but the cost “sensor” that is subsequently generated by the dashboard also needs to be added to the included entities in the *recorder* configuration (see section 3.1.6.5). Usually this “sensor” carries the name of the utility meter used to track consumption/generation with an additional “_cost” added.
3. In order to complement the energy dashboard, an additional dashboard for real time monitoring of power levels and daily power generation / consumption (divided over multiple tabs) is recommended. The power readings can also be used to check for the balance of currently consumed and generated power in order to not exceed peak levels imposed by the governing entities. Needle meters are useful visualizations for real time readouts, which allow for indicating different levels of desirability, while history plots of daily utility meters allow to track the development of consumption / generation through out the day.
4. Additional, customized dashboards maybe setup in order to visualize solar predictions (see chapter 5.3) using the apexcharts card from HACS (see section 3.1.9), and in order to manually trigger automations. The latter should not be made available to all users, but only administrators who know what they are doing. 😉

3.5 Setting up remote access via cloudflared

A very nice feature of home assistant is the companion app. It is available for both Android and iOS, and offers the possibility to access and even configure home assistant from mobile devices such as smartphones, tablets, etc, and it even can track the location of connected mobile devices (in case location access is enabled). The companion app can easily be configured when the mobile device is connected to the same network as home assistant, and will autodetect any home assistant instance during installation and setup.

The added value of the companion app is greatly enhanced when home assistant is configured for remote access, since then the companion app can be used to monitor and control home assistant “on the go”. There are several ways to make home assistant accessible remotely, one of the safest, most convenient and cheapest one is via *Cloudflare*. In order to setup remote access via *Cloudflare,* the following steps have to be taken:

1. Open a new account on https://dash.cloudflare.com .

2. Register an internet domain that will serve as an entry port to home assistant from remote. Unfortunately it is not possible anymore to register free domains with *Cloudflare*, instead the most convenient way is to purchase and register a domain (e.g. .uk) on *Cloudflare* itself, via *Domain Registration → Register Domains.*The internet domain you just registered should show up under *websites,* if not press *add a site* and follow the instructions. (Make sure to select the free plan when asked.)
   The *DNS* settings of your website should show the nameservers that *Cloudflare* uses to tunnel to home assistant via your domain. *SSL* settings should be set to flexible by default.
3. Install the *Cloudflare add-on* on home assistant via *Settings → Add-ons → Add-on store.* Install the add-on via the github repository `https://github.com/brenner-tobias/ha-addons`. The add-on should be configured before start, by adding the host name that was setup in step 2 to the config page.
4. The *Config.yaml* needs some additional modification as well. Make sure the header still contains *default_config:*. Additionally, add the following lines to the *config.yaml:*

```
http:
      use_x_forwarded_for: true
      trusted_proxies:
      -IP
```

   In order to configure the right *IP,* try to access home assistant by typing your domain in a browser. Most likely you will get a *400* or similar error. Now check the logbook of home assistant via the *settings* menu (under *system*). It will show some error :/ warning related to an external IP trying to access home assistant and being refused. The IP in question is the one to be added to the *config.yaml*.
5. In your companion app, under *settings → system → network*, configure the home assistant urls for internal and external access. To verify home assistant access works, disconnect your mobile device from the home WLAN and try to access home assistant via the 4G connection of your phone. (Alternatively, try to access home assistant via your internet browser and the domain you setup in step 2).
6. It is recommended to enable two-factor authentication for maximum security. For this, go to your user profile, and enable the option. A QR code will be displayed that you can use to register your totp authenticator app (for instance google authenticator).

If you plan to access home assistant with multiple people, then it is wise to setup multiple accounts via *settings →persons.* In this way you can keep admin/editing rights confined to one key user (usually yourself), and limit other users rights on a strict "need-to" base. If applicable, the companion app of other non-admin users should then be setup using their own respective home assistant user accounts, including their own two factor authentication.

4 Enabling scripts and automations

Home assistant offers the possibility to automatically trigger processes or state changes of devices upon certain events and under certain preconditions. One potential usecase is smart energy management: In Belgium, "peak consumption" is penalized by the grid owner (Fluvius), while at the same time the "electrification" of private households through electric cars and heat pumps is promoted. Home batteries may offer a way out, but may not be sufficiently charged through solar energy generation and / or have insufficient storage capacity when energy demand is high. Automations can be used to overcome these 2 issues: batteries can be charged from the grid when excess power is available (i.e. when overall energy consumption in the house is low), and battery discharge current can be limited when demand is high in order to retain battery charge as much as possible while "assisting" electrical

consumption from the grid in order to stay below peak tariff thresholds. In the future, this usecase could be extended to take advantage of dynamic energy tariffs by dynamic charging (and potential later injection back into the grid as a form of "arbitrage trading").

We will look into this example usecase in more detail in the next sections.

4.1 Preparation and precondition

To enable grid charging on the inverter side, the inverter should be pre-configured with a time interval (ideally for full flexibility from 12AM until 11.59PM) and charging of the battery from grid should be enabled. On Solis hybrid inverters, the former can be done via *Advanced Settings->Storage Energy Set->Storage Mode Select->Self Use → Time of Use,* while the latter can be done via *Advanced Settings->Storage Energy Set->Storage Mode Select->Self Use-> Charge from grid->Allow.*

Furthermore, it is handy to monitor the applied charging settings via modbus. For this, add another modbus sensor according to section 3.2.2, with the register containing that information. On Solis inverters, the register ID is 33132. It contains 16 bits of information with the sensor readout being converted into an integer number. For instance, the setting with overnight charging disabled has bits 0 and 5 true (to generally enable battery charging) and bit 1 disabled (to disable time of use charging) resulting in output 2^0+2^5=33. Overnight charging enabled will have bit 1 enabled, resulting in output 2^0+2^1+2^5=35. We'll remember these settings for the automation. Additional modbus sensors to potentially add in order to monitor the battery's charge and discharge currents, since we are going to manipulate these settings in our automation:

- Register 33206: Maximum grid charge current of the battery
- Register 33143: Maximum battery charge current (including solar)
- Register 33144: Maximum battery discharge current

4.2 Setting up the automation flow

In our automation example, we would like to achieve 2 things:

1. Switch between battery charge and discharge mode depending on the current power consumption in the house and the solar power generation.
2. Modulate the charge / discharge current according to excess power capacity available and power assistance capacity required, respectively.

The generic flow for this situation is as follows:

1. An automation will be triggered when the (un)load current calculated in section 3.2, example 4 as the delta of energy consumption in the house, solar energy generation and peak tariff thresholds, divided by the nominal battery voltage is larger than zero.
2. The automation kicks off a script, which repeatedly executes Modbus commands to set the battery (dis)charge current according to the value as provided by aforementioned template sensor(s).

4.2.1 Setting up the scripts

Scripts can be setup via *Settings → Automations & scenes.* A script typically consist of a sequence of actions that can be executed one time or repeatedly, the latter as long as certain conditions are fulfilled. Although scripts come with an UI editor in home assistant, it is ultimately easier to set them up using YAML. (For this, open a script, click on the 3 dots in the upper right corner and select "Edit as YAML").

A possible action that can be executed is a service call. An example of such a service is *Modbus: Write register.* It allows selecting a register to set (e.g. 43110 for changing the time of use charging setting or

43141 for adapting the charge current), the slave (typically 1), the value to be written to the register (e.g. 33/35 for 43110 or the desired current value for 43141), and finally the hub, which should be the name of the modbus connector as configured in section 3.2.2.

In our case, 2 scripts are to be prepared:

1. A script that activates battery charging from grid and continuously adapts the grid charge current according to the template sensor setup earlier. The latter is done as long as the battery charge is below a (user configurable) threshold (e.g. using the binary sensor configured in section 3.2, example 4), and the charge current is above 0. Once the condition is violated, the loop is stopped and the inverter is set back to discharge mode. In YAML, the script looks like this:

```
alias: Battery load management
sequence:
  - repeat:
      while:
        - condition: numeric_state
          entity_id: sensor.load_current
          above: 0
        - condition: state
          entity_id: binary_sensor.batt_thresh_trigg
          state: "on"
      sequence:
        - service: modbus.write_register
          data:
            address: 43110
            slave: 1
            value: 35
            hub: Solis_Inverter
        - service: modbus.write_register
          data:
            address: 43141
            slave: 1
            value: |
              {{states('sensor.load_current')}}
            hub: Solis_Inverter
        - delay:
            seconds: 20
  - service: modbus.write_register
    data:
      hub: Solis_Inverter
      address: 43110
      value: 33
      slave: 1
mode: single
icon: mdi:home-battery
```

2. Similarly, a script that activates battery discharging and continuously adapts the discharge current according to the template sensor setup earlier. The latter is done as long as the battery charge is below a (user configurable) threshold (e.g. using the binary sensor configured in section 3.2, example 4), and the discharge current is above 0. Once the condition is violated, the loop is stopped and the inverter is set back to discharge mode. In

YAML, the script looks like this:

```
alias: Battery unload management
sequence:
  - repeat:
      while:
        - condition: numeric_state
          entity_id: sensor.unload_current
          above: 0
        - condition: state
          entity_id: binary_sensor.batt_thresh_trigg
          state: "on"
      sequence:
        - service: modbus.write_register
          data:
            hub: Solis_Inverter
            slave: 1
            value: 33
            address: 43110
        - service: modbus.write_register
          data:
            address: 43118
            slave: 1
            value: |
              {{states('sensor.unload_current')}}
            hub: Solis_Inverter
        - delay:
            seconds: 20
  - service: modbus.write_register
    data:
      hub: Solis_Inverter
      address: 43118
      slave: 1
      value: 1000
mode: single
icon: mdi:home-battery-outline
```

An additional script maybe prepared in order to reliably reset the inverter to default conditions (discharge mode and default discharge current). This script repeatedly executes the corresponding modbus commands until the values of the state sensors set up in section 4.1 matches the expected values:

```
alias: Reset load / unload
sequence:
  - repeat:
      sequence:
        - service: modbus.write_register
          data:
            hub: Solis_Inverter
            address: 43110
            value: 33
            slave: 1
```

```
      until:
        - condition: numeric_state
          entity_id: sensor.gridchgstate
          above: 32
          below: 34
  - repeat:
      sequence:
        - service: modbus.write_register
          data:
            hub: Solis_Inverter
            address: 43118
            slave: 1
            value: 1000
      until:
        - condition: numeric_state
          entity_id: sensor.max_ontlaadstroom_batt
          above: 99
          below: 101
mode: single
icon: mdi:lock-reset
```

4.2.2 Setting up the automations

Automations can be setup via *Settings → Automations & scenes.* An automation typically consists of a trigger, conditions as well as an action. In order to execute the scripts set up in section 4.2.1, we prepare 2 (3) automations:

1. The first automation executes the script controlling the charging behavior of the battery. It is triggered when the template sensor of the calculated load current is larger than 0 and/or the battery charge drops below a certain threshold. Both situations serve as both triggers and conditions:

```
alias: Load management
description: triggers script to adapt battery load
trigger:
  - platform: numeric_state
    entity_id:
      - sensor.load_current
    above: 0
  - platform: state
    entity_id:
      - binary_sensor.batt_thresh_trigg
    from: "off"
    to: "on"
condition:
  - condition: numeric_state
    entity_id: sensor.load_current
    above: 0
  - condition: state
    entity_id: binary_sensor.batt_thresh_trigg
    state: "on"
action:
  - service: script.turn_on
    data: {}
```

```
    target:
      entity_id: script.battery_load_management_dupliceren
mode: single
```

2. The second automation executes the script controlling the discharging behavior of the battery. It is triggered when the template sensor of the calculated unload current is larger than 0 and/or the battery charge drops below a certain threshold. Both situations serve as both triggers and conditions:

```
alias: Unload management
description: triggers script to adapt battery unload
trigger:
  - platform: numeric_state
    entity_id:
      - sensor.unload_current
    above: 0
  - platform: state
    entity_id:
      - binary_sensor.batt_thresh_trigg
    from: "off"
    to: "on"
condition:
  - condition: numeric_state
    entity_id: sensor.unload_current
    above: 0
  - condition: state
    entity_id: binary_sensor.batt_thresh_trigg
    state: "on"
action:
  - service: script.turn_on
    data: {}
    target:
      entity_id: script.battery_management
mode: single
```

3. Optionally, a third automation triggers the reset of the inverter in case minute average power consumption from the grid (according to the sensor setup in section 3.2, example 5) is larger than 2.5kW and the battery charge is above the threshold for the previous automations. This can be used as a safety net when reset of the inverter previously failed e.g. due to connection issues:

```
alias: Reset load/unload
description: Resets load - unload state of the inverter to
default
trigger:
  - platform: numeric_state
    entity_id:
      - sensor.active_power_minute_average
    above: 2500
condition:
  - condition: state
    entity_id: binary_sensor.batt_thres_trig
    state: "off"
action:
  - service: modbus.write_register
```

```
      data:
        hub: Solis_Inverter
        address: 43110
        value: 33
        slave: 1
    - service: modbus.write_register
      data:
        hub: Solis_Inverter
        address: 43118
        value: 1000
        slave: 1
    - service: script.turn_on
      data: {}
      target:
        entity_id: script.reset_load_unload
  mode: single
```

All automations can also be triggered via action buttons in dashboard, e.g. in rare cases when automations where not kicked off due to conditions failing. Additionally, services could be utilized as part of the automation in order to send notifications that automations were triggered to users' cellphones similar to the alerts we've set up in sections 3.2.5.

The variable battery charge threshold used in these automations (and the scripts they trigger), can be used to ensure enough capacity is left in the battery for solar charging. As such, the threshold can be automatically set based on predictions of solar production. The next chapter will explain in more detailed how such solar forecasts can be setup in a customized fashion.

5 Building a solar predictor using Python and AppDaemon

In order to determine the variable battery charge threshold used in the previous chapter and in general developing an idea of near future solar production, we can use the solar yield data stored in InfluxDB, and combine it with weather forecast data from OpenMeteo before offering it to a machine learning model for training purposes.

Setting up the solar prediction engine involves three main steps:

1. Exporting solar production data from influxDB.
2. Offline model training.
3. Model deployment and forecast visualization.

Each of this steps will be described further in the next sections

5.1 Exporting solar production data from influxDB

Home assistant's databases like influx DB can be accessed and queried within home assistant using the jupyterlab add-on (see section 3.1.6).

To access the data stored in influx DB, we need to install a couple of packages via *pip install packagename.* The packages we require are pandas (should already be installed), influxdb (not influxdbclient) and datetime (should already be installed).

We load the packages and relevant classes via:

```
from datetime import timedelta
import pandas as pd
from influxdb import InfluxDBClient
```

We establish a connection to our influx database via:

```
client = InfluxDBClient(host='…', port=8086, username='…', password
='…', database ='…')
```

The host, username, password and database name correspond to what we have setup previously as described in sections 3.1.5 and 3.2.8.

We can query the influx db following below example:

```
result=client.query('SELECT value AS "value" FROM
"home_assistent"."autogen"."kWh"')
```

A good tip is to use the influx DB UI's data explorer in order to construct the right queries.

Finally, the result is a JSON dictionary that can be converted to a pandas dataframe using:

```
df = pd.DataFrame.from_dict(result.get_points(), orient='columns')
```

and exported using:

```
df.to_csv(r'vermogen_n.csv', sep=';')
```

The resulting csv file can be downloaded from the Raspberry Pi via the home assistant web interface and used for offline machine learning model training.

5.2 Offline model training

The training of the prediction model is based handled offline, in order to not deplete the limited resources of the Raspberry Pi. For that purpose, a distribution like *Anaconda* can be used.

The generic model training approach involves the following steps:

1. Preprocessing the solar production data, in order to convert it into kWh dataset per hourly interval as input for the model to train towards.
2. Querying ~~solar positioning and~~ historical weather data for the same time period, using the ~~PySolar package and~~ OpenMeteo API~~, respectively~~.
3. Training a ~~Random Forest~~ Regression model and tuning its hyperparameters, and exporting the final model for deployment to the RPi.
    a. A stochastic gradient descent based linear model with regularization delivers good performance for inference on the RPi. However, for better predictive performance, a neural network maybe employed. The required packages for training and inference do not run natively in *appdaemon / home assistant*, and hence a more complex setup is required, where the model is deployed to a cloud service like *AWS,* and called via a Web-API. The process of setting up such a cloud based inference engine is described in section 5.4.

The core idea of the solar prediction approach involves calculating "features" that represent the energy irradiated on the solar panels. These features are:

1. The global direct irradiance, which is a function of the direct normal irradiance (as provided by OpenMeteo) and the angle of incidence, which itself is a function of the tilt and azimuth of the solar panels, and the altitude and azimuth of the sun (~~as provided by PySolar~~).

a. The OpenMeteo-API offers means to calculate the global direct irradiance directly by providing the tilt angle of the solar panels. This eliminates the need for the PySolar package. However, if panels are present on 2 sides of the roof, the API needs to be called twice per forecast.

2. The global diffuse irradiance, which is a function of the diffuse horizontal irradiance (as provided by OpenMeteo) and the tilt of the solar panels.

The general concept behind these calculations is very well explained in this video: PV3x 2017 2 3 1 Calculating Irradiance based on Tilt Angle video 720 (youtube.com)

Above features are augmented with additional features such as snow depth and cloud coverage, as these may impact the working of the solar panels further and/or may subtly influence the diffuse radiation levels.

An example Jupyter notebook for the model training can be found on Github at Solar_Forecast_ML/Offline_training at main · marchauptmann1984/Solar_Forecast_ML · GitHub.

5.3 Model deployment and forecast visualization

In order to run the model and periodically provide forecasts, we can use *AppDaemon* (see 3.1.8). AppDaemon allows to run python code that interfaces with home assistants integrations, sensors, etc.

First, AppDaemon needs to be setup with the right packages that are needed to run the python code. 2 types of packages are required: 1. System packages, which are Alpine Linux packages. Many compiled Python packages with dependencies in other Python packages are available as system packages, which provides a good workaround to get them installed without too much hazzle. Packages that are required for our solar forecasting code are *py3-scikit-learn* and *py3-pandas*. 2. Python packages. Here, non-compiled packages can be installed. Packages that are required for our solar forecasting code are ~~*pysolar* and~~ *beautifulsoup4*. When specifying packages to be installed, keep in mind that these packages are downloaded, unzipped and installed each time AppDaemon is restarted. So keeping it lean will help to minimize startup time.

Secondly, we need to prepare and configure the "App" to run. This step consists of 3 parts: 1. The config.yaml for the AppDaemon apps, defining the main class according to the python script per app. 2. The model zip files, containing the trained machine learning model. 3. The actual Python script. There is extensive AppDaemon documentation available on the net, so we won't go into software engineering details here. From a functional point of view, the script does the following things: 1. Retrieving weather forecast and solar ~~position~~ irradiation information. 2. Calculating the features as used during model training. 3. Running the model for these features and retrieve its predictions of the solar generated power. 4. Exporting the model forecast into a home assistant sensor (as a timeseries attribute) for further visualization. An additional sensor contains the forecast for solar production in the next 24 hours, which can be used to determine the battery management threshold as utilized in section 4.2. The code and corresponding configuration can be found here: Solar_Forecast_ML/AppDaemon_deploy at main · marchauptmann1984/Solar_Forecast_ML · GitHub
It is best to perform the coding offline, via an IDE like VisualStudio, then copy the code and corresponding files to the RPi using *SambaShare* (see 3.1.7). The target directory for the AppDaemon apps is *addon_configs**appdaemon**apps*.

The forecast can be visualized using ApexCharts (see 3.1.9), together with the actual solar production. See below an example for configuration:

```
type: custom:apexcharts-card
graph_span: 30h
span:
  start: day
  offset: +9h
header:
  show: true
  title: Hourly solar yield measured vs. predicted
  show_states: false
  colorize_states: false
series:
  - entity: sensor.daily_power_generated
    type: column
    group_by:
      func: diff
      fill: last
      start_with_last: true
      duration: 1h
    opacity: 0.5
    show:
      in_header: before_now
    time_delta: +0.5h
  - entity: sensor.solar_forecast_hourly
    name: Solar forecast
    type: line
    color: grey
    data_generator: |
      return entity.attributes.PeakTimes.map((peak, index) => {
        return [new Date(peak).getTime(),
entity.attributes.PeakHeights[index]];
      });
    unit: kWh
    show:
      legend_value: true
      in_header: before_now
    time_delta: +0.5h
now:
  show: true
  label: now
yaxis:
  - min: 0
    max: ~2
    apex_config:
      tickAmount: 10
      title:
        text: Solar yield [kWh]
apex_config:
  dataLabels:
    enabled: true
    dropShadow:
      enabled: true
```

In this example, the power generated hourly is plotted against the forecast extracted from the sensor attribute time series. Axis and timeseries labels are added as well as an indicator for the current timing. The chart can be added natively to a dedicated dashboard for energy forecasting.

5.4 Deploying machine learning models to cloud services for cloud based inference

As explained in section 5.2, more complex models (e.g. neural networks) cannot be run anymore in home assistant / appdaemon on the RPi. Instead, we need to deploy these models to a cloud service and call them via a web-API. In this section, we describe how to do this using the example of *Amazon Web Services (AWS).*

5.4.1 Prerequisites

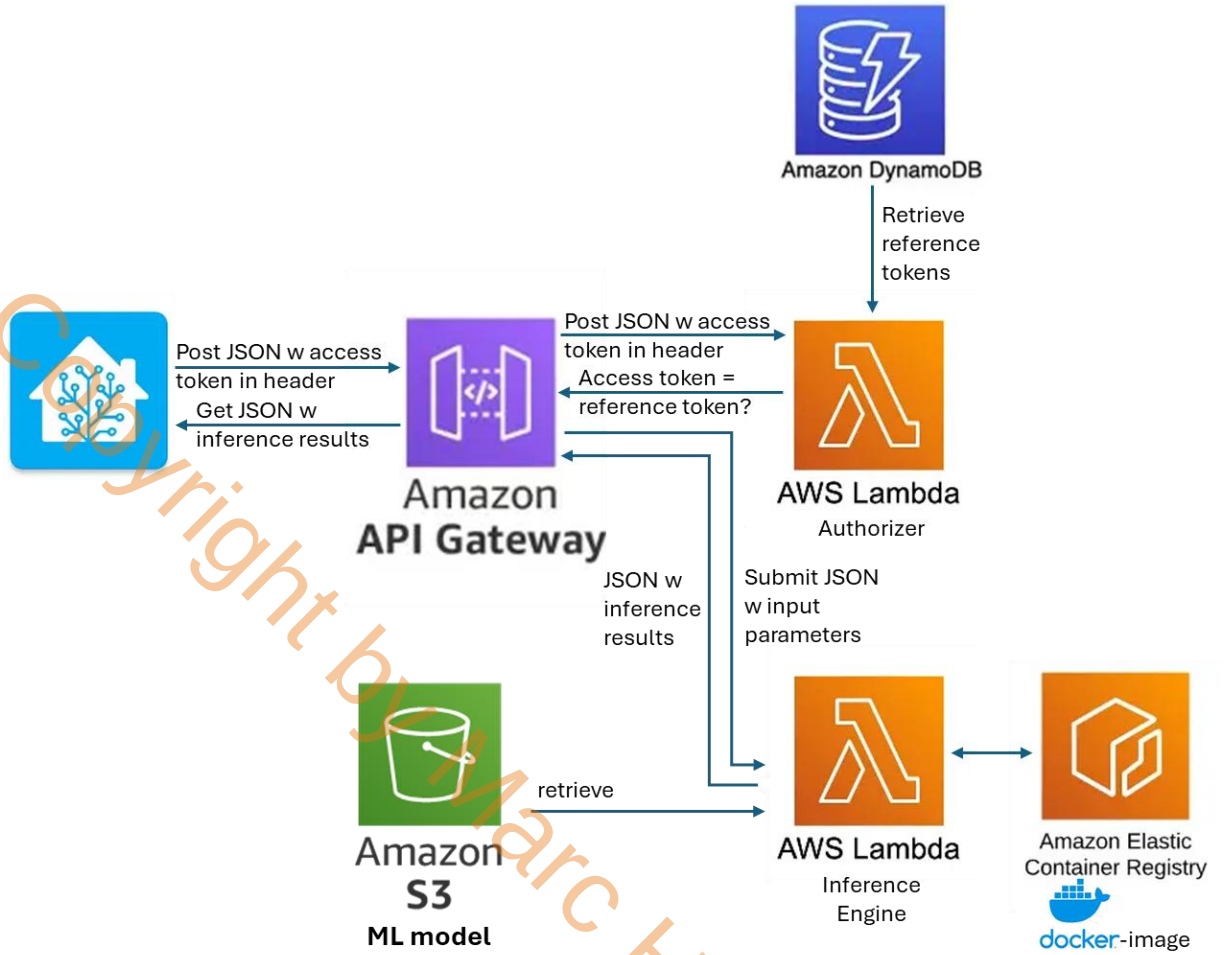
Next to the obvious prerequisite of an *AWS account,* 2 more prerequisites are *Docker* and *AWS CLI.*

Docker allows to package code in "containers" together with their dependencies (e.g. libraries), such that it can be run on any platform and in any environment. Docker can be downloaded e.g. via the Microsoft AppStore.

The AWS command line interface (*CLI*, https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-getting-started.html) is a utility that allows to automate many of the process in AWS via single line instructions. The following instructions make heavy use of the *AWS CLI* functionality. On windows, AWS CLI as well as Docker commands are best run in *powershell.*

5.4.2 Principle data flow

In order to limit cloud costs, we utilize a server less approach called "AWS lambda". *Lambda* runs short snippets of e.g. python code and only inquires charges when the code is run. It automatically scales up to the momentary usage needs, and is automatically configured and updated by AWS (no server needs to be maintained). It is lightweight and flexible and hence ideal for ML inference tasks.



We are going to use 2 *Lambda Functions*: 1 for the actual inference & 1 to regulate access to the model via the API. The latter is achieved by comparing access tokens submitted as part of the API-request header with reference tokens stored in *AWSes DynamoDB.* The *AWS API gateway* allows to post REST-like requests to the inference function and retrieve the calculated results.

For reasons of easy configurability, the *Inference Function* is running a docker container stored in *AWS ECR*, containing the required packages (e.g. *tflite, numpy*) and the actual lambda function "handler". Example code for the lambda functions, as well as the code for calling the deployed model via *Jupyter* and *AppDaemon* can be found here: Solar_Forecast_ML/AWS at main · marchauptmann1984/Solar_Forecast_ML · GitHub. For reasons of modularity and flexibility as explained in a later section, the machine learning model itself is stored in an S3-database and retrieved by the lambda function upon startup or whenever the model is updated in S3.

5.4.3 Deploying a machine learning model to AWS

5.4.3.1 Setting up AWS credentials

Credentials are needed to be able to connect with the *AWS CLI* to one's personal AWS cloud services. To configure AWS CLI, first go to *IAM → Users → Security Credentials.* Retrieve the access and secret access keys. Then in powershell type *aws configure,* and set the access and secret access keys according to the retrieved values. Set the region to your AWS region, and the output to *json.*

5.4.3.2 Building and deploying the docker image for inference

In order to create the docker image, we start by creating a *Dockerfile* containing the instructions to build the container incl. installation of dependencies, and a python file containing the actual inference

code for the lambda function handler. We put both files into one directory, and upload the machine learning model to S3. We do this by first creating an S3 bucket using this command in *powershell*:

```
aws s3 mb s3://solar-prediction --region region
```

We can check the bucket's creation by: `aws s3 ls`
We then upload the ML model via:

```
aws s3 cp model.tflite s3://solar-prediction/models/model.tflite
```

In order to keep the container lightweight, we should use lightweight dependencies like *tflite* instead of full blown *tensorflow*, which means that offline we should convert our *tensorflow* ML model to a *tflite*-version.

Next we build the container image. In order to do this, in *powershell*, we *cd* to the directory containing the Dockerfile and python code, and build the container via the following command:

```
docker build -t tflite_container .
```

Here, *tflite_container* is the name of the container image we are creating. In order to test whether the container has built correctly, we can run some simple code inside the container and verify the result (in this case the correct tflite version number should be displayed):

```
docker run --rm tflite_container python -c "import tflite_runtime as
tflite; print(tflite.__version__)"
```

When the container has built correctly, we can deploy it on the *Elastic Container Registry (ECR).* First, we need to create a new repository.

```
aws ecr create-repository --repository-name tflite_container
```

We should write down the resulting internal address (uri) to be able to use it later for our lambda function. The URI typically contains the AWS user id and the region, and has a format like *userid*.dkr.ecr.*region*.amazonaws.com/*containername*. With this URI, we're going to authenticate to the newly created repository:

```
aws ecr get-login-password | docker login --username AWS --password-
stdin userid.dkr.ecr.region.amazonaws.com/tflite_container
```

Then we tag the latest local version of our container image:

```
docker tag tflite_container:latest
userid.dkr.ecr.region.amazonaws.com/tflite_container:latest
```

and push it to the *ECR* repository:

```
docker push userid.dkr.ecr.region.amazonaws.com/
tflite_container:latest
```

As a last step we create the lambda function using the container image we've just pushed to the ECR repro. For this, we first need to create a role via *IAM* (called *iamrole* below), with the right permissions (*AWSLambdaBasicExecutionRole* and *AmazonEC2ContainerRegistryReadOnly*, respectively) to execute the lambda function and read from *ECR* (since our lambda functions uses the container image situated there). Then execute:

```
aws lambda create-function --function-name tflite_container --
package-type Image --code ImageUri= userid.dkr.ecr.region.
```

```
amazonaws.com/tflite_container:latest --role arn:aws:iam::
userid:role/iamrole
```

Since the lambda function needs to access the S3 bucket we created earlier to store the model, we need to attach *S3 read rights* to the same *IAM* role:

```
aws iam attach-role-policy --role-name role --policy-arn
arn:aws:iam::aws:policy/AmazonS3ReadOnlyAccess
```

We can test our lambda function by invoking it using a JSON input file containing some example input parameters:

```
aws lambda invoke --function-name tflite_container --payload
fileb://input.json response.json

cat response.json
```

Such an input file can for instance be generated in python using the following code:

```
import json

payload = {
    "body": json.dumps({
        "features": [
          [input values],
        ]
    })
}


with open("input.json", "w", encoding="utf-8") as f:
    json.dump(payload, f)
```

Sometimes, the lambda function may fail due to the rather low default runtime limit. This may especially be the case when some larger packages have to be loaded upon first invocation. Fortunately, the runtime limit can be adapted using:

```
aws lambda update-function-configuration --function-name
tflite_container --timeout 300
```

. Sometimes, function code needs to be updated. In principle this means rebuilding the docker container and pushing the code again to repository created earlier. Then the lambda function needs to be updated using the following command:

```
aws lambda update-function-code --function-name tflite_container --
image-uri userid.dkr.ecr.region.amazonaws.com/
tflite_container:latest
```

5.4.3.3 Connect lambda function to API gateway

In order to invoke the newly created lambda function from the outside world, we need to create an API via AWS'es API gateway and connect it to the lambda function.

The very first step, is to create the API via:

```
aws apigatewayv2 create-api --name "TFLiteAPI" --protocol-type HTTP
```

This step returns an *API_ID.* In order to delete an already existing API, the following commands can be used to retrieve the API-ID and subsequently remove the API in question via its API-ID:

```
aws apigatewayv2 get-apis
```

```
aws apigatewayv2 delete-api --api-id API_ID
```

In order to connect the API to the AWS lambda function, an integration needs to be created, using the lambda functions *ARN* from the previous section.

```
aws apigatewayv2 create-integration --api-id API_ID --integration-
type AWS_PROXY --integration-uri
arn:aws:lambda:region:userid:function:tflite_container --payload-
format-version 2.0
```

The payload version here is a parameter that becomes important later when calling the API. This command returns an *Integration_ID* that is used in the following commands. Next, we need to allow the API gateway to call the lambda function by setting the right permissions:

```
aws lambda add-permission --function-name tflite_container --
statement-id apigw-permission --action lambda:InvokeFunction --
principal apigateway.amazonaws.com --source-arn "arn:aws:execute-
api:region:userid:API_ID/*/*"
```

Here, *apigw-permission* is the name we've given to the permission. Again, permission can be removed using the following series of commands:

```
aws lambda get-policy --function-name tflite_container
```

```
aws lambda remove-permission --function-name tflite_container --
statement-id apigw-permission
```

Here, *apigw-permission* is the name we've previously given to the permission.

Now, we need to create an API endpoint (called "route" in AWS), and a versioning (called "stage" in AWS), which both define the final URL we have to call for invoking the lambda function via the API:

```
aws apigatewayv2 create-route --api-id API_ID --route-key "POST
/predict" --target "integrations/Integration_ID"
```

```
aws apigatewayv2 create-stage --api-id API_ID --stage-name prod
```

Here, "POST / predict" signifies we're invoking the function via a "POST" request, since we're submitting data to the function via the API in order for the lambda function to be able to execute the model prediction, and *prod* signifies that we're dealing with production code (though the name is chosen arbitrarily). Finally, we're ready to deploy the API via:

```
aws apigatewayv2 create-deployment --api-id API_ID --stage-name prod
```

We can test if the API works as expected by invoking it via the following command:

```
Invoke-WebRequest -Uri "https://API_ID.execute-
api.region.amazonaws.com/prod/predict" -Method POST -Headers @{
"Content-Type" = "application/json" } -Body '{"features": [[input
values]]}'
```

using e.g. the earlier used input values, and the API_ID and our AWS region. (It should be noted that the *Invoke* command only works in *Windows Powershell.* In linux, another command like *curl* is to be used. In either way, the API will be triggered from outside AWS, representing the true model inference situation when calling the inference function from Home Assistant.)

5.4.3.4 Add authentication step via authorizer function

The API set up in the previous section can in theory be used by anyone having the API-URL. If we want to control access to the API in a more granular and modular fashion, we need to add an authenticator function to AWS lambda and connect to the API created earlier.

The authenticator function then compares an access token submitted together with the input parameters to the API with reference tokens contained in a *AWS Dynamo DB* table. In order to create such a table and reference token contained in the table, we can use thes commands:

```
aws dynamodb create-table --table-name "ApiTokens" --attribute-
definitions AttributeName=token,AttributeType=S --key-schema
AttributeName=token,KeyType=HASH --billing-mode PAY_PER_REQUEST
```

```
aws dynamodb put-item --table-name ApiTokens --item
'{\"token\":{\"S\":\"token1\"}}'
```

Here *ApiTokens* is the *DynamoDB* table containing the reference tokens and *"token1"* is the reference token we created in that table. Subsequently, we need to create an additional role (called *HomeAutomationAuthRole* below) via *IAM* that can both access the table and the lambda function. The role should be created for the service *lambda* and should have the following policies attached: *AWSLambdaBasicExecutionRole, DynamoDBReadAccess*. (Note, it is possible to limit access further to only specific lambda functions and database tables but this is not shown here.) Next, we create the authorizer function, by creating a script *authorizer.py* in a directory of choice / own creation, that contains the authorization code (example see my GitHub). We then zip the python file and create a lambda function in AWS using:

```
aws lambda create-function --function-name "solar_authorizer" --
runtime python3.11 --role arn:aws:iam::userid:role/
HomeAutomationAuthRole --handler authorizer.lambda_handler --zip-
file fileb://authorizer.zip
```

Subsequently, we create an authorizer in our previously created API using:

```
aws apigatewayv2 create-authorizer --api-id API_ID --authorizer-type
REQUEST --authorizer-uri arn:aws:apigateway:region:lambda:path/
2015-03-31/functions/arn:aws:lambda:region:userid:function:
solar_authorizer/invocations --identity-source '$request.header.x-
api-key' --authorizer-payload-format-version 2.0 --enable-simple-
response --name "HomeAutomationAuthorizer" --query 'AuthorizerId' --
output text
```

Here it is important, to make sure the authorizer payload format version is consistent with the payload format version of the API integration with the inference function created earlier, and -if version 2 is used, and depending on the actual implementation of *authorizer.py*- to enable "simple response" in order to avoid triggering internal errors when calling the API. The command returns and *authorizer_id,* which will be used subsequently. We then request the *route_id* of the route (aka endpoint) created earlier and attach the authorizer to the route using both its id and the one of the route:

```
aws apigatewayv2 get-routes --api-id API_ID

aws apigatewayv2 update-route --api-id API_ID --route-id route_id --
authorizer-id authorizer_id --authorization-type CUSTOM
```

Lastly, we give the API permission to invoke the authorizer lambda function and deploy the API again:

```
aws lambda add-permission --function-name "solar_authorizer" --
statement-id apigw-auth-permission --action lambda:InvokeFunction --
principal apigateway.amazonaws.com --source-arn "arn:aws:execute-
api:region:userid:API_ID/*/POST/predict"

aws apigatewayv2 create-deployment --api-id API_ID --stage-name prod
```

Here, a new permission with name *apigw-auth-permission* is created. We can test the authorizer function within aws using:

```
aws lambda invoke --function-name solar_authorizer --payload
fileb://token.json out.json

Get-Content out.json
```

with *token.json* containing *{"headers": {"x-api-key": "token1"}}*

The whole API-authorizer setup can be tested from within *powershell* using:

```
$Url = https://API_ID.execute-api.region.amazonaws.com/prod/predict
$Headers = @{
                "Content-Type" = "application/json"
                "x-api-key" = "token1"
            }
$Body = @{
                features = @(@(input values))
            } | ConvertTo-Json -Depth 3

Invoke-RestMethod -Uri $Url -Method POST -Headers $Headers -Body
$Body
```

5.4.3.5 Calling the inference engine from home assistant / App Daemon

Now that we have successfully deployed our machine learning model to AWS and setup an API with authentication around it, we can finally call the inference function using the following python code snippet:

```
import requests
url = f"https://API_ID.execute-
api.region.amazonaws.com/prod/predict"
headers = {"Content-Type": "application/json", "x-api-key":
"token1"}
payload = {"features": [[input values]]}
response = requests.post(url, json=payload, headers=headers)
print(response.json())
```

The last line should return the correct responses from our machine learning model!

5.4.3.6 Automated retraining of the model

With the model being stored separately from the inference engine, we can retrain the model with data e.g. when the previously predicted and the actually measured solar power production deviate too much from each other. In order to achieve this, we're generating a separate lambda function containing the model training code and connect it to the same API as our inference engine.

The steps are largely the same as outlined in sections 5.4.3.2 and 5.4.3.3. The main difference is that the container image should contain *tensorflow* instead of *tflite,* since *tflite* only allows for inference. *Tensorflow* is a bigger dependency than *tflite*, but that should be okay since we're running model retraining much less frequently than inference (once a day vs. every 10 min).

In order to be able to retrain the model, we need to do 2 additional things:

1. We need to upload the original "keras" version of the machine learning model:
   `aws s3 cp model.keras s3://solar-prediction/models/model.keras`
   Internally, the retrain lambda function will download the keras model, retrain it with the received input data, convert the retrained model to a *tflite* version, and upload both model versions to S3.
2. We need to grant additional rights to the previously utilized IAM role, in order to enable upload of the model files to S3:
   `aws iam attach-role-policy --role-name role --policy-arn arn:aws:iam::aws:policy/AmazonS3FullAccess`

Since the container image of the retrain function is larger than that of the inference function due to the usage of *tensorflow,* we need to adapt the configuration of its lambda container such that it can run properly. More specifically, we need to increase the timeout setting, and increase the working and storage memory sizes:

```
aws lambda update-function-configuration --function-name
retrain_container --memory-size 4096

aws lambda update-function-configuration --function-name
retrain_container --timeout 900

aws lambda update-function-configuration --function-name
retrain_container --ephemeral-storage '{\"Size\": 10240 }'
```

We connect the retrain lambda function to the same API as setup in section 5.4.3.3. We do create a separate route for it, e.g. `"POST /retrain"`. If for some reason, the deployment of the API fails, then this likely has to do with *cloudwatch* monitoring pipelines that have previously been created for this API. In that case, in the AWS management console, go to *API gateway → API → logging → disable,* then deploy the API and then reactive the login using the API's urn and deploy that again as well.

In order to invoke the retrain code from within e.g. *Appdaemon*, we can use a similar code snippet as in section 5.4.3.5:

```
import requests
url = f"https://API_ID.execute-
api.region.amazonaws.com/prod/retrain"
headers = {"Content-Type": "application/json", "x-api-key":
"token1"}
payload = {"features": [[input values]], "targets": [[expected
```

```
values]]}
response = requests.post(url, json=payload, headers=headers)
print(response.json())
```

The main difference here is the use of the “retrain” route in the API call, as well as the adapted json-body of the API payload, that has been extended with the target values required for model training, representing the actually measured solar-generated power.

6 Integrating heating circuits into home assistant

A major energy consumer with a lot of energy saving potential is the heating system. Heat pumps consume electrical energy with an efficiency that depends on factors such as supply temperature, outside temperature, desired temperature etc. Furthermore, syncing solar energy production and heat / warmwater generation can allow to utilize excess solar energy to generate warmwater for later consumption without taking power from the grid. In this chapter we’re going to discuss how to setup both the monitoring and control of your heating system in home assistant, as well as integrating it into the energy management of our house using automations.

6.1 Prerequisites

6.1.1 Ebus adapter

In order to control our heating system from within home assistant, we first need to establish a connection between it and our RPi. Fortunately, there is a standard with which most heating systems communicate amongst their component – ebus. This protocol shares some similarities with modbus, which we covered earlier in section 2.2.

We can then communicate with the heating system from our RPi using an ebus adapter. There are some adapters commercially available, but the recommended one is the one designed and sold by the ebus daemon (ebusd) initiative. It can be purchase here: Reserve-my-adapter – ebusd. This adapter offers enough configuration options and is also supported by the same team which maintains the ebusd repository.

In order to ensure a stable communication, it is further more recommended to connect the ebus adapter to the RPi via ethernet. For this purpose, an additional Ethernet shield is required such as a USR-ES1 Modul w W5500 (see https://adapter.ebusd.eu/v5/#ethernet for more details).

6.1.2 Ebus add-on

In order to communicate with the heating system from within home assistant via the ebus adapter, the ebusd add-on by Lucas Grebe is required (ha-addons/ebusd/DOCS.md at main · LukasGrebe/ha-addons · GitHub). It is a community add-on that provides a container to run the ebus daemon software within home assistant. Next to that it also provides a command line interface for testing and debugging purposes. It can be installed similar to other add-ons as e.g. described in section 3.1.

6.1.3 MQTT

MQTT is a messaging protocol that allows to convert poll readings from e.g. *ebusd* to be converted into home assistant entities like sensors, switches, etc. *MQTT* can be added to home assistant via *Settings → Appliances and services → Integration → Add integration*.

6.2 Configuration

6.2.1 Ebus adapter

The ebus adapter should be connected to the heating system using 2 non-twisted electrical wires (0.5 mm2), close to one of the units like the thermostat. A tight, reliable connection ensures signals to and

from the heating system are probably sent and received. The firmware should be updated to the latest version using the instructions on this site: Start using the adapter - eBUS Adapter Shield v5.

Especially with Vaillant heating systems, even after properly connecting the adapter to it, the received signals might still contain a lot of "SYN received" or similar errors. This is due to the fact that the baud rate is not exactly at the intended frequency of 2400 baud, and arbitration is not properly set up. Both can be adjusted via the configuration page of the ebus-adapter, which can be reached via its IP address. (Again; here it is recommended to fix the adapter's IP address in the router). In the configuration page, open a terminal via *Repl* and type *ebus -v*. The adapter will scan ebus traffic and display the baudrate and arbitration. Repeat this step multiple times to confirm baudrate stability. In the ebus subsection, the baudrate offset can be adjusted to match the measured baud rate, and the arbitration delay should be set to the minimum detected value.

6.2.2 Ebusd add-on

An important step of the integration is the configuration of the ebusd addon in home assistant. Configuration starts with setting the IP address to the address configured for the daemon on the ethernet. Latency settings of 50ms should be appropriate to ensure proper signal capture, and poll interval for the data can be set to 1s, to ensure frequent updates of the parameter values to be monitored without overloading the bus. Polling is done by the ebus daemon through actively sending out polling messages to the heating system. The addon allows to further provide command line options to ebusd, in order to e.g. set the number of retries & timeout durations (-- *acquiretimeout=100* -- *acquireretries=10 --sendretries=10 --receivetimeout=100*). Setting the access level option to "*" will enable to control parameters that on the appliances themselves are stored in the installer menus. *Mqtt* and *log* levels can follow the default settings (*Mqtt* as explained later should be enabled to provide the ebusd readouts as sensors to home assistant).

One important group of settings concerns the configuration files (.csv) required for translating the binary signals into readable (and writeable) messages. To start with; when *scan config* is on, *ebusd* will scan the bus for identifiers of the individual heating system components; then retrieve the corresponding configuration files from the pre-configured location. If no location is specified, then the files are directly pulled from the official github repro (the files are named according to the heating circuit appliances, with one file per appliance). However, in order to have more control over the configuration files used (and allow them to be modified), it is advised to -once that the components have been identified- disable the scan setting; and provide only the desired configuration files via a local directory on the RPi (e.g. */config/ebusd/ebusd-2.1.x/en*). Here it is important to maintain the same subfolder structure as in the github repo.

The files can be adapted to configure which parameters to poll, which to keep read only or configurable (see Home · john30/ebusd Wiki · GitHub for more details on how this works). The parameters are then polled by ebusd and added as entities to home assistants via *mqtt*. Which parameters are added how (i.e. as what entity types) to home assistant can be configured in the *mqtt-hassio.cfg* file located in the base directory of the (local) configuration file repository (e.g. */config/ebusd/*). The most important settings here are:

- *filter-circuit:* limit which appliances in the cv-system are readout and added to home assistant.
- *filter-name:* limit which type of parameters are included in home assistant(by the configured parameter name). Comment this out if you want to include everything that is polled.
- *Filter-direction:* configure if only readable or also writeable messages are included in home assistant. In order to include writeable messages as switches, input fields, numerical etc., specify *filter-direction = r|u|^w*. Note that in this case, all changes to switches, numerical input

etc. that are created by *MQTT* will directly change the parameters of the heating circuit, so handle with care.

Lastly, it can be handy to add parameters yourself that are not yet preconfigured in the available configuration files. For this some basic understanding of the structuring of ebus-messages is required, see here for reference: HowTos · john30/ebusd Wiki · GitHub. An easy, straightforward approach to f ind "hidden" parameters:

1. Go to the ebusd addon via *settings→ addons*, and go to the logbook page. There, if properly configured, messages sent over the ebus and received by the ebus-adapter are displayed. Unconfigured messages look something like this: 1026b5230106 / 1078016d01008000805a0100800080503c. The part before the "/" indicates the appliance in the heating system, so look for correspondences with similar identifiers in your configuration files (e.g. "b523" in this example, often this are contained in specific "headerlines" at the beginning of subsections in the csv related to a specific circuit in the heating system), as well as the type of parameter (typically the following numbers like 0106). The part after the "/" contains the actual message, here some trial and error converting back and forth from hexadecimal to decimal can help to reveal the meaning, by comparing the values to values displayed e.g. in the thermostat.
2. Next, edit your configuration files to include the new messages by using the identifies found in step 1. Often times it is recommended to copy an existing message definition from the same section and adapting the identifier as well as the parameter naming (e.g. *"z1FlowTemperature"*). If step 1 did not yield conclusive results, one can also try to "sweep" through identifiers by increasing the identifier numbers with respect to previous entries in the configuration file.
3. Lastly, activate and check the new parameters in *ebusd*. Here, we can us the *SSH web terminal* as configured in section 3.1.2. There, we have to first log into the docker container running the addon by typing: *docker exec -it `docker ps | grep ebusd | awk '{print $1}'` /bin/bash* . (Check if the ebusd addon is running if this is not working). Then reload the configuration files by typing *ebusctl reload* and then *ebusctl -c heatingcircuit parametername*. Here, *heatingcircuitname* is the name of the appliance according to the name of the csv file, and *parametername* the name of the newly configured parameter. The configuration was successful if no *ERR* messages returned, but sensible readings.

6.3 Integration

6.3.1 Dashboard

The sensors generated by ebusd can be utilized to visualize crucial parameters of the heating system such as environmental and storage temperatures, current operational mode of the heatpump, diagnostic settings of the heating system, statistics of the heat generation etc. Here, conditional cards can be used to only display parameters that are relevant to the current operational mode of the heat pump. Furthermore, automations can be enabled and/or triggered from the respective dashboard pages. For more information on setting up dashboards refer to section 3.4.

6.3.2 Automations

If *MQTT* is configured for *ebusd* in write mode, then settings of the heating system can easily be changed. This also allows for easy setup of more complex automations. Some examples include:

- Triggering the absence mode by detecting the presence/absence of cellphones of family members.

- Heating up warm water at elevated temperature when excess solar power is available (i.e. battery is nearly full and solar forecast is above a certain threshold).
- Triggering warm water generation manually.
- Setting room temperature set points.

For proper integration with the heating system, it is also important to foresee automations that react to changes in writeable parameters outside home assistant (e.g. due to adjustments done in the thermostat). Automations are also a good way to limit e.g. temperature settings to reasonable ranges. See here for one example:

```
alias: Cooling temperature select
description: Pass down cooling temperature from selector upon
certain conditions
mode: single
triggers:
  - entity_id:
      - input_number.cooltemp
    trigger: state
  - entity_id:
      - sensor.ebusd_ctlv2_z1actualheatingroomtempdesired_tempv
    trigger: state
conditions: []
actions:
  - choose:
      - conditions:
          - condition: numeric_state
            entity_id: input_number.cooltemp
            above:
sensor.ebusd_ctlv2_z1actualheatingroomtempdesired_tempv

        sequence:
          - metadata: {}
            data:
              value: "{{states('input_number.cooltemp')}}"
            target:
              entity_id:
number.ebusd_ctlv2_z1coolingroomtempdesiredtimecontrolled_tempv
            action: number.set_value
      - conditions:
          - condition: numeric_state
            entity_id: input_number.cooltemp
            below:
sensor.ebusd_ctlv2_z1actualheatingroomtempdesired_tempv
        sequence:
          - target:
              entity_id: input_number.cooltemp
            data:
              value: >-
{{states('sensor.ebusd_ctlv2_z1actualheatingroomtempdesired_tempv')}
}
            action: input_number.set_value
          - metadata: {}
```

```
            data:
              value: >-
{{states('sensor.ebusd_ctlv2_z1actualheatingroomtempdesired_tempv')}
}
            target:
              entity_id:
number.ebusd_ctlv2_z1coolingroomtempdesiredtimecontrolled_tempv
            action: number.set_value
```

In this automation, the cooling temperature setpoint is adapted whenever the value of a numerical input changes. The numerical input is configured such that only sensible temperature values can be chosen. Additionally conditions in the automation safeguard that the cooling temperature cannot be set below the heating temperature, even in case the heating temperature is increased after initially setting the cooling temperature. Lastly, some parameters cannot be changed directly via *MQTT* inputs (selection fields, switches), but rather need to be modified using service calls. One example, where an absence time is set to the value of a datetime input, is shown below:

```
service: mqtt.publish
data:
   qos: "1"
   retain: false
   topic: ebusd/ctlv2/z1QuickVetoEndTime/set
   payload_template: "{{states('input_datetime.absence')}}"
```

7 Next steps

This is a living document describing the key steps in order to setup home assistant for energy automation and monitoring purposes. It is aimed to continuously expand this document with new insights and ideas. Some items on the “backlog” include:

1. Adding zones, areas, rooms (incl. automated notifications), and other smart devices such as switches.

43110 → charge from grid (35 -on 33-off)

43141 → charge current (in 0.1A)

43118 → discharge current (in 0.1A)

Automation

Solar forecast (first forecast, then automation)

Additional automation → make lower limit depending energy consumption depending temperatures

Make list of reauirements for warmtepomp