



Ein Batterie Management System für Victron ESS  
Systeme mit (zukünftiger) Möglichkeit zur  
Stromkostenoptimierung

# Inhaltsverzeichnis

Einleitung.....	3
essBATT controller.....	4
essBATT watchdog.....	4
essBATT smartcharger.....	4
Zielgruppe von essBATT.....	5
Für wen ist essBATT NICHT?.....	5
Zielgruppe dieses Dokuments.....	5
Einrichtung.....	6
Voraussetzungen an das ESS.....	6
Voraussetzung um dieser Einrichtungsbeschreibung folgen zu können.....	6
Installation von Rasberry OS.....	7
Einrichten von VSCode Remote.....	8
Einrichten aller Komponenten mit Docker Compose.....	11
Klonen der essBatt Git Repositories.....	15
Docker Compose file.....	16
MQTT Server Konfiguration.....	17
Konfiguration des essBATT Controllers.....	20
Konfiguration des Cerbo / VenusOS.....	22
Konfiguration der Multis.....	22
Starten der Docker Container.....	22
MQTT Server Testing.....	24
Das essBATT Controller Logfile.....	25
Nachträgliche Änderung von essBATT Controller Einstellungen.....	26
Konfiguration vom essBATT Controller.....	28
Eigene Weiterentwicklungen und Debugging vom essBATT Controller.....	28
Konfiguration des Victron ESS.....	29
Einstellugen im Cerbo / VenusOS.....	29
Einstellungen in den Multis.....	30
Konfiguration des (physischen / übergeordnetem) BMS und Balancers.....	30
NEEY Balancer.....	31
Einrichten von iobroker.....	31
6. Python Skript entwickeln im Docker Container.....	35
Changelog.....	37

# Einleitung

Die Notwendigkeit zur Entwicklung von essBATT hat sich ergeben, da ich unzufrieden mit dem Gesamtsystem bestehend aus Victron ESS und Batrium BMS war. Beim Victron System fehlte mir die Möglichkeit die Batterie zu laden wenn der Strom günstig ist (Stichwort „dynamischer Stromtarif“) und bei Bedarf ein Top Balancing des Akkus durchzuführen und beim Zusammenspiel aus Victron und Batrium fehlte mir der Schutz der Zelle mit der niedrigsten Spannung beim Entladen und der Zelle mit der höchsten Spannung beim Laden. Außerdem habe ich beobachtet, dass die Batterie teilweise trotzdem geladen wurde, obwohl der maximale Ladestrom auf 0A begrenzt war. Dies hat zur Überladung einer Zelle geführt, was ich durch die Einstellungen in Batrium eigentlich für unmöglich gehalten habe. Außerdem ändert sich durch den DIY Charakter meines Systems ständig die Systemkonfiguration und es können sich Fehler einschleichen. Wenn ein Fehler auftritt der nicht automatisch erkannt wird (wie z.B. die überladene Zelle), wollte ich auch aus der Ferne die Möglichkeit haben das System zu steuern. Über das VRM Portal oder die VenusOS GUI war dies nicht in der Form möglich, wie ich es gerne hätte. Ein weiterer Punkt war, dass sich im Winter das System durch den geringen PV Ertrag anders verhalten soll als im Sommer, ausgelöst durch den chronisch niedrigen Ladezustand der Batterie. Um Strom zu sparen sollen die Multis fast immer abgeschaltet sein, da es teilweise mehrere Tage dauern kann bis die PV wieder nennenswert Strom in den Akku geladen hat. Daraus haben sich für mich folgende Zielsetzungen für essBATT ergeben:

- 100% Schutz der einzelnen Zellen (mit Einschränkungen bereits implementiert)
- Zeitgesteuerte Ladung der Batterie (Datum, Uhrzeit, maximaler Ladestrom, Zielladestand)
- (ferngesteuertes) Top Balancing der Batterie
- (ferngesteuertes) Abschalten vom Charger und/oder Inverter der Multis
- Anpassung des Systemverhaltens im Winter
- Automatisierte und optimierte Ladestrategie bei dynamischen Stromtarifen (noch nicht implementiert)
- Steuerung mit dem Smartphone

Stand Frühjahr 2024 wurden die meisten Punkte bereits erreicht. Ausnahmen:

- Für einen 100% Schutz der Zellen fehlt noch der essBATT watchdog. Der soll das Victron ESS in einen sicheren Zustand bringen wenn der essBATT controller ausfällt. Außerdem soll eine Warnung des Nutzers über Telegram und Mail ermöglicht werden. Der hier genannte „100% Schutz“ bezieht sich auf die Ansteuerung des Victron Systems und umfasst natürlich nicht die elektrische Sicherheit des Gesamtsystems!

- Das Skript zur optimierten Ladesteuerung existiert auch noch nicht

essBATT besteht im Wesentlichen aus drei Komponenten bzw. drei unterschiedlichen Pythonskripten die miteinander über MQTT interagieren, aber auf unterschiedlichen Hardwarekomponenten ausgeführt werden können/müssen: dem controller, dem watchdog und dem smartcharger (sigh!). Der Controller und der Watchdog sind die Kernkomponenten und der Smartcharger ist optional für Leute mit dynamischem Stromtarif.

## **essBATT controller**

Der Controller agiert im Wesentlichen wie ein Batteriemanagementsystem und schützt die Einzelzellen bzw. hält das gesamte ESS System innerhalb seiner Systemgrenzen (maximaler Strom, maximale Zellspannung,...). Das Gesamtsystem hat somit zwei BMS: das an der Batterie verbaute BMS und zusätzlich noch den essBATT Controller. Aus dem Grund ist es extrem wichtig die essBATT Konfiguration mit der Konfiguration des physischen BMS abzustimmen. Ziel bei der Abstimmung ist, dass das physische BMS nie eingreift (also die Grenzen für z.B. maximale Zellspannung weiter gefasst sind als in der essBATT Controller Konfiguration), aber trotzdem noch einen „Basisschutz“ bietet. Eigentlich genau so wie beim physischen BMS und der Multi ESS Konfiguration auch: die ESS Settings im Multi haben eine höhere Priorität als die des physischen BMS. Und die Settings im physischen BMS haben eine höhere Priorität als die von essBATT. Letztendlich hat man dann drei separate Schutzsysteme für seinen Akku – was ja auch nicht schlecht ist.

## **essBATT watchdog**

Der Watchdog dient letztlich nur zur Überwachung des Controllers und bringt im Falle eines Ausfalls des Controllers das Victron System in einen sicheren Zustand. Wenn der essBATT Controller auf einem Raspberry Pi läuft und dieser kaputt geht oder zu Wartungs-/Updatezwecken runtergefahren werden muss, greift der Watchdog ein. Außerdem dient der Watchdog zur Benachrichtigung des Nutzers, wenn irgendwelche gefährlichen Systemzustände detektiert werden. Z.B. könnte die Kommunikation mit dem Cerbo/VenusOS abreissen und der Watchdog würde dann (solange es noch eine Verbindung des Watchdogs mit dem Internet gibt und nicht das gesamte Netzwerk ausgefallen ist) eine Mail oder eine Telegram Botschaft raussenden. Der Watchdog ist aktuell noch nicht implementiert. Den essBATT Controller OHNE den Watchdog zu betreiben ist extrem gefährlich – ich habe schon Berichte gelesen wo mit einem Victron Mode 2/3 System der Akku beschädigt wurde.

## **essBATT smartcharger**

Sorry für den Namen, aber etwas nichtssagenderes ist mir nicht eingefallen. Der smartcharger ist ein Projekt für die Zukunft und existiert noch nicht. Ziel ist einen dynamischen Stromtarif so zu nutzen, das bei Vorhersage des morgigen Verbrauchs und der morgigen PV Erzeugung bei gegebenem Ladestand der Batterie (preis-)optimal die Batterie geladen wird (unter Berücksichtigung der Lade-

und Entladeverluste). Stay Tuned!

## Zielgruppe von essBATT

essBATT ist potentiell interessant für technisch versierte DIY PV- und Batteriebastler die ein Victron ESS System (inkl. Batteriespeicher) betreiben. Der Batteriespeicher kann selbstgebaut sein, hauptsache er überträgt den SOC, die minimale und maximale Zellspannung, die Gesamtspannung der Batterie und den Batteriestrom (korrekt) an das VenusOS (z.B. Cerbo oder VenusOS auf einem Raspberry). Wenn ihr zusätzlich noch Bedarf an einem der oben genannten Features habt oder ihr eine Absprungbasis für ganz andere verrückte Ideen braucht, könnte essBATT für euch interessant sein. Wenn euer System die von Victron [hier](#) definierten Anforderungen für ein ESS System erfüllt, kommt essBATT für euch in Frage. Für die Ansteuerung nutzt essBATT aktuell (Stand Frühjahr 2024) nur den Funktionsumfang vom [ESS Mode 2](#).

## Für wen ist essBATT NICHT?

essBATT ist ein privates Projekt, das ich in meiner Freizeit entwickle. Dabei erfüllt die Software nicht die Ansprüche in Bezug auf Softwarequalität und Testing, die professionelle Softwarelösungen bieten. Wenn man die Funktionalität von essBATT kaufen könnte hätte ich das getan. Wenn ihr Software mit Garantie und Support benötigt ist essBATT nicht für euch! Außerdem ist essBATT nichts für euch wenn ihr kein Grundwissen und Interesse in Computern/Programmierung mitbringt. Aus diesem Grunde werde ich auch nie einen Installer bauen, der essBATT mit einem Mausklick/Kommandozeilenbefehl installiert. Der manuelle Installationsprozess ist ein Feature ;-)

## Zielgruppe dieses Dokuments

Dieses Dokument beschreibt was das essBATT System an Features bietet, aus welchen Komponenten es besteht und wie man **eine Realisierungsmöglichkeit des Gesamtsystem** einrichtet. Die selbe Anleitung liegt jeder Softwarekomponente von essBATT bei (controller, smartcharger und watchdog), weil der Fokus dieses Dokuments darauf liegt dem interessierten Nutzer das gesamte Ökosystem vorzustellen und benötigte Skills zu vermitteln. Zielgruppe für die essBATT Dokumentation sind technisch interessierte Nutzer aus dem DIY Umfeld, denen an der einen oder anderen Stelle aber das Wissen fehlt bestimmte Ideen umzusetzen. Profis suchen sich einfach geeignete Hardware mit Netzwerkzugang auf denen Python läuft und starten die zwei/drei essBATT Skripte – denn mehr ist es im Prinzip auch nicht.

# Einrichtung

Wie bereits in der Einleitung angekündigt wird im Kapitel „Einrichtung“ der vollständige Installationsprozess einer Realisierungsmöglichkeit für essBATT beschrieben. Um es potentiellen Nutzern und mir einfach zu machen beschreibe ich hier die Einrichtung auf einem neuen/ungenutzten Raspberry Pi 4b für den essBATT controller (bzw. smartcharger) und einem weiteren ungenutzten Raspberry für den essBATT watchdog. Wenn ihr essBATT auf Komponenten nutzen wollt die ihr bereits für andere Aufgaben nutzt müsst ihr diese Anleitung selbständig auf eure konkrete Situation anpassen.

## Voraussetzungen an das ESS

- ESS System wie von Victron spezifiziert:  
[https://www.victronenergy.com/media/pg/Energy\\_Storage\\_System/de/ess-introduction---features.html](https://www.victronenergy.com/media/pg/Energy_Storage_System/de/ess-introduction---features.html)
- **Separates (Hardware-) BMS** (z.B. das von mir verwendete Batrium) das im Victron CCGX System SOC, minimale und maximale Einzelzellspannung, Batteriestrom, Batteriespannung und Batterieleistung korrekt bereitstellt (sodass die Werte unter „battery“ eingetragen werden). Dies ist vermutlich bei allen BMS der Fall, die per CAN mit dem Cerbo/VenusOS verbunden sind und von Haus aus eine Victron Unterstützung mitbringen. Ob das auch für die China BMS → Victron Anbindung gilt kann ich aktuell nur vermuten.
- **Netzzähler** der die Leistung auf L1, L2, L3 und die Gesamtleistung im CCGX System unter „Grid“ bereitstellt. Ich nutze einen EM24 dafür. Wenn man ein „Multi only“ ESS hat darf man die Multis vermutlich nicht in den Stand By Modus schalten, da man sonst keine Information mehr über den Stromverbrauch im Haus erhält. essBATT unterstützt deswegen aktuell nur den ESS Betrieb MIT separatem Netzzähler. Sollte es Bedarf geben könnte man einen Betrieb ohne Netzzähler vermutlich mit wenig Aufwand ermöglichen.
- Der Akku benötigt einen **Batterie Balancer** ([z.B. den NEEY](#))! Dieser muss mit den Einstellungen in essBATT zum Balancing abgestimmt werden. (z.B. muss die Startspannung des Batterie Balancers unterhalb der essBATT „Zielspannung“ für das Balancing liegen).

## Voraussetzung um dieser Einrichtungsbeschreibung folgen zu können

- Raspberry Pi 4b
- Sandisk Max Endurance SD Karte mit 64GB

- Ein Windows Rechner von dem aus die ganze Konfiguration gemacht wird. Dieser muss im selben WIFI/Netz hängen wie der Raspberry Pi
- Euer Victron Cerbo/VenusOS befindet sich im selben Ethernet Netzwerk wie die beiden Boards auf denen die essBATT Komponenten laufen sollen (dargestellt im Schaubild)

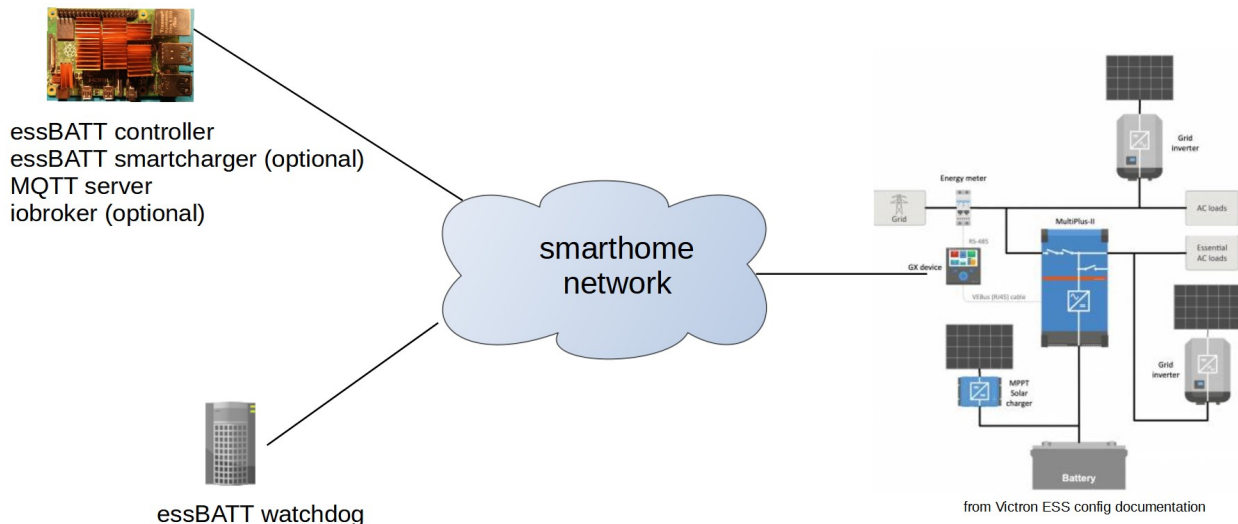


Schaubild 1: essBATT Netzwerktopologie - Welche Komponenten werden benötigt und was läuft wo

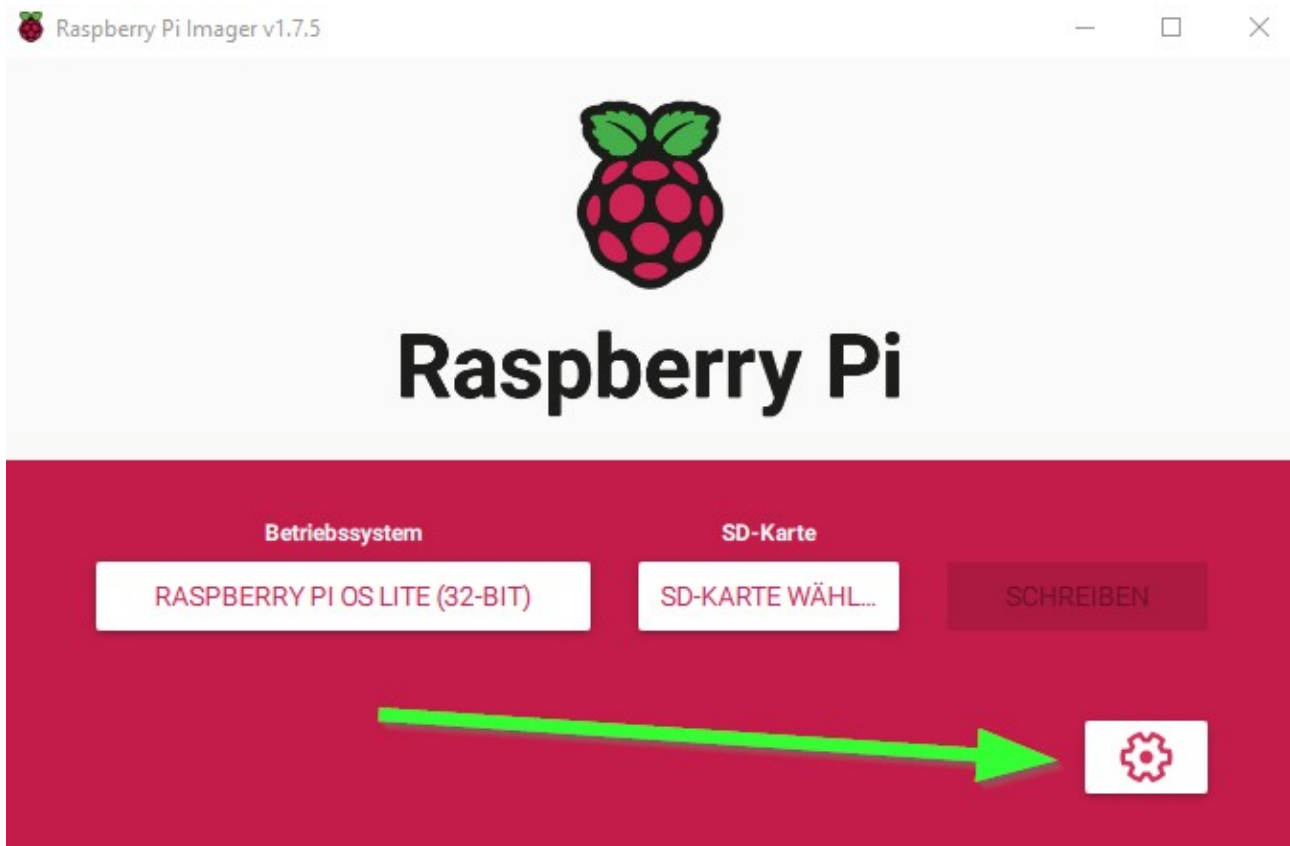
## Installation von Rasberry OS

Zuerst besorgt man sich das Tool „Raspberry Pi Imager“ von der offiziellen Seite:

<https://www.raspberrypi.com/software/>

Man folgt der Anleitung zum Erstellen des Image und wählt als OS das Raspberry 32bit Lite aus. 32 Bit weil es weniger Inkompatibilitäten bei den Bibliotheken als bei 64bit gibt. Lite, weil wir immer nur von einem externen Rechner auf den Pi zugreifen und keine Desktop Umgebung brauchen.

Wichtig beim Einrichten ist folgender Button:



Unbedingt folgende Einstellungen vornehmen:

- Hostname: Gib einen individuellen Namen – meiner heißt nach seiner Aufgabe „raspismarhome“
- SSH: unbedingt aktivieren da man sich später fast ausschließlich über SSH verbinde
- WIFI: Auch wenn man den PI später per LAN Kabel anschließt würde ich auch erstmal die WIFI Daten eintragen, damit man sich dann auch entspannt per SSH verbinden kann
- Benutzername und Passwort setzen
- TODO: Wie/wann die Timezone setzen???

Dann das Image auf die SD Karte flashen. Ich würde dringend zu einer **Sandisk max endurance** Karte raten und gleich 64GB nehmen. Die normalen SD Karten sterben wie die Fliegen in einem Raspi – einfach mal bei Amazon die Kommentare lesen und dann NICHT bei Amazon bestellen ;-)

## Einrichten von VSCode Remote

Mit VSCode Remote kann ziemlich bequem vom Desktop Rechner aus auf dem Pi gearbeitet werden. Man kann Ordner anlegen und Python Skripte ausführen sowie Dateien bearbeiten. Das ist teilweise extrem hilfreich, da das mit der Kommandozeile über SSH auf dem Raspi sehr mühsam



ist.

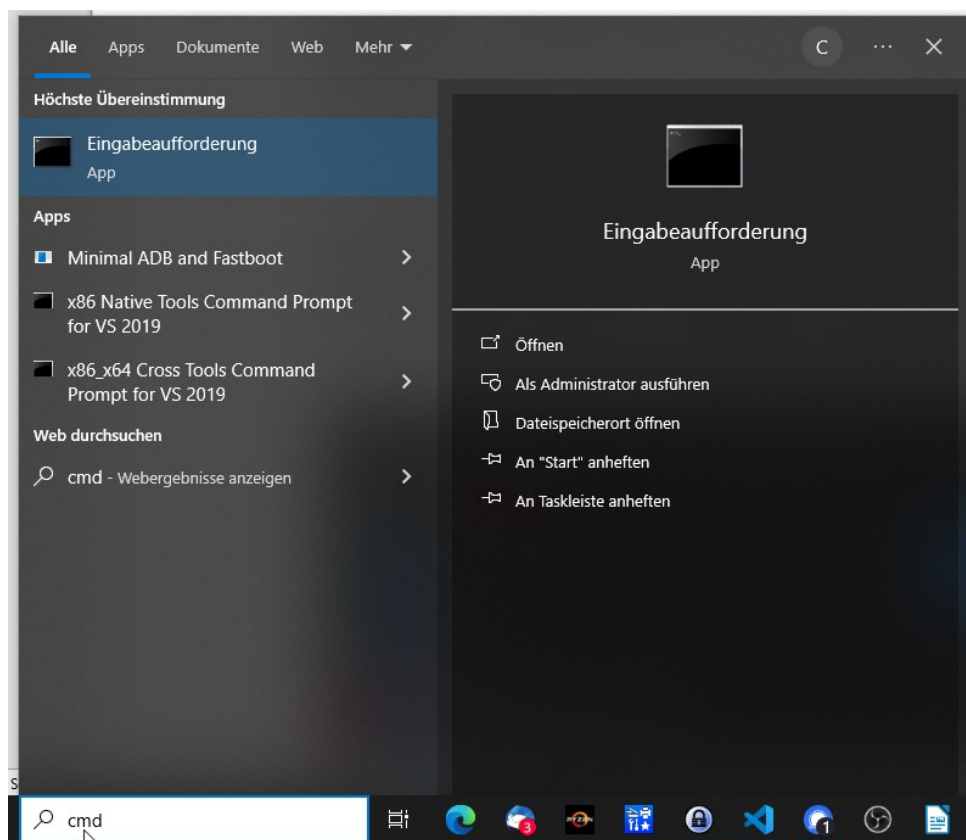
Dazu Download von VSCode für Windows hier:

<https://code.visualstudio.com/download>

VSCode als portable einrichten (optional aber ganz praktisch manchmal) so:

<https://code.visualstudio.com/docs/editor/portable>

Jetzt noch den Raspberry mit Strom versorgen und starten. WLAN und SSH haben wir ja schon bei der Erstellung des Betriebssystems auf dem Raspi aktiviert. Zum Testen ob SSH grundsätzlich funktioniert kann man das auch erstmal in der Kommandozeile testen. Dazu in Windows Suche „CMD“ eintippen und eine Kommandozeile starten:



Dort dann „ssh [benutzername@hostname](#)“ eintippen. Bei mir sieht das so aus:

```
christoph@raspismarhome: ~
Connection to raspismarhome closed.

C:\Users\Christoph>ssh christoph@raspismarhome
christoph@raspismarhome's password:
Linux raspismarhome 6.1.21-v8+ #1642 SMP PREEMPT Mon Apr  3 17:24:16 BST 2023 aarch64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

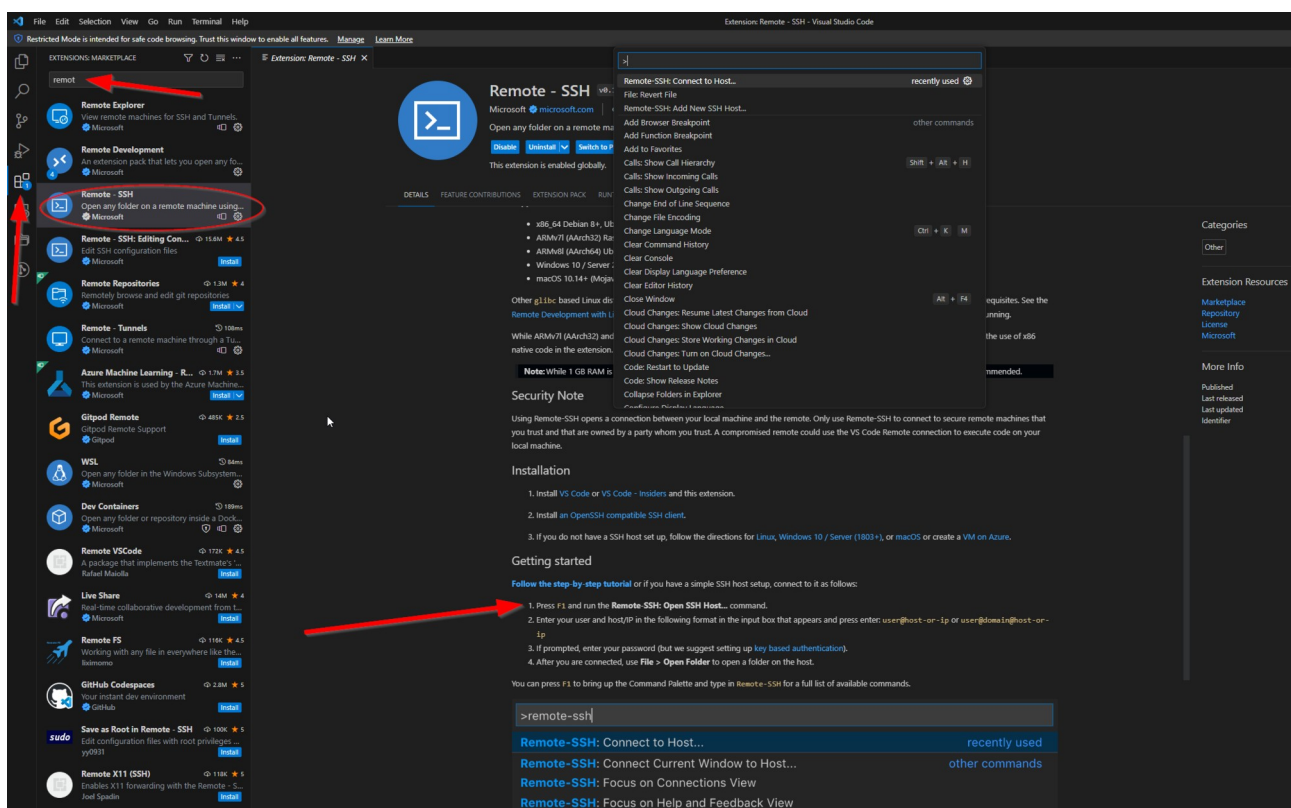
Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Thu Sep 21 21:02:12 2023 from 192.168.178.30
christoph@raspismarhome:~ $
```

Man kann aber auch die IP Adresse des Pi nutzen: ssh [christoph@192.168.1.57](#) zum Beispiel. Wenn

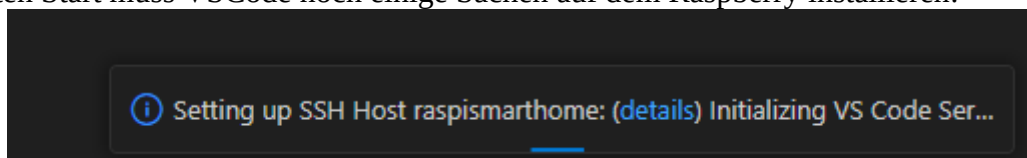
man dann den grünen Text vom eingeloggten Benutzer sieht, passt die SSH Verbindung.

Nach diesem ersten Test der Erreichbarkeit des Raspis wollen wir nun die weitere Einrichtung von essBATT mithilfe von VSCode ermöglichen. Dazu starten wir VSCode zuerst. Dann den roten Pfeilen/Markierungen auf dem Screenshot folgen: Erst den Extensions Tab ganz links auswählen. Nach der „Remote SSH Extension“ suchen und installieren (da ist irgendwo ein install Button). Bei der Gelegenheit installieren wir noch weitere wichtige Extensions: „C/C+“, „Python Extension Pack“ und „JSON“

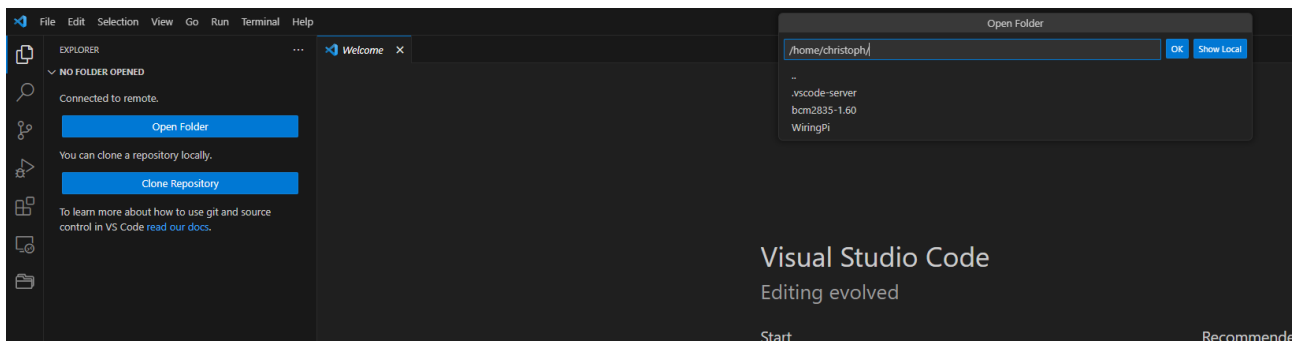
Wenn alle Extensions installiert sind auf die „Remote SSH“ Extension klicken um die Beschreibung zu erhalten. Dort steht dann auch wie man sich zum Raspberry verbindet unter „Getting started“. Dieser Anleitung folgen (F1 drücken, dann dort auch „benutzername@hostname“ eintippen,...).



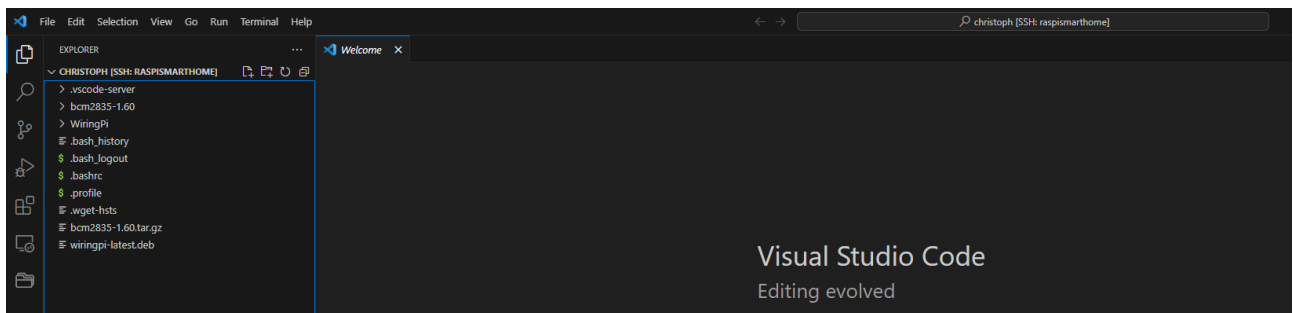
Beim ersten Start muss VSCode noch einige Sachen auf dem Raspberry installieren:



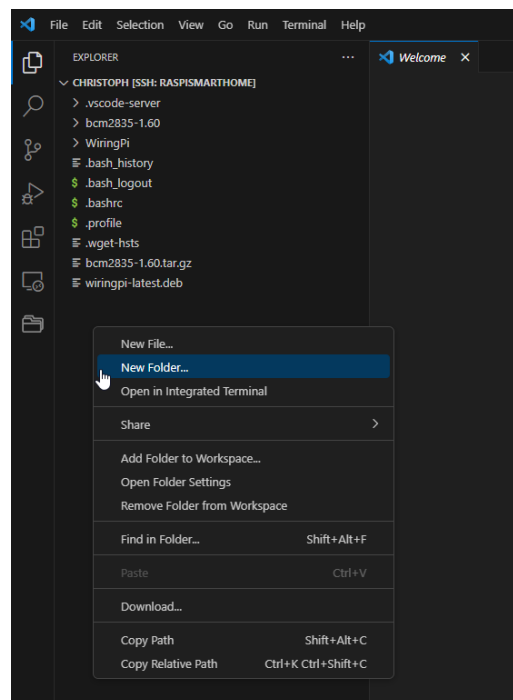
Wenn die Initialisierung abgeschlossen ist wird man gefragt welchen Ordner man öffnen möchte. Dafür nimmt man dann den home Ordner des Benutzers. Bei mir so:



Ich hatte im Vorfeld schon die CAN Bibliothek installiert, weswegen bei mir schon ein paar Dateien vorhanden sind:



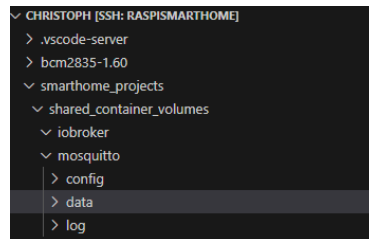
Für die ganzen Smarthome Komponenten den Ordner „*smarthome\_projects*“ anlegen. Dazu Rechtsklick in den Explorer Bereich und dann „*New Folder*“ auswählen:



## Einrichten aller Komponenten mit Docker Compose

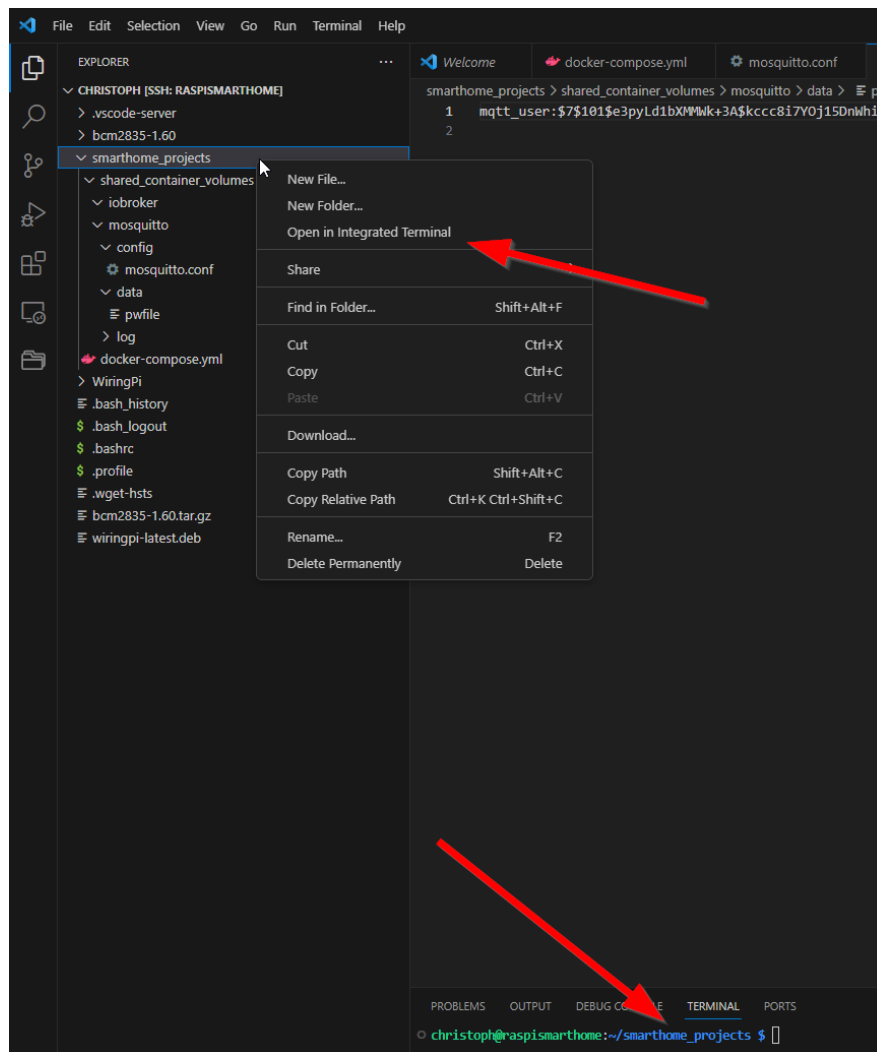
Das hier vorgestellte Victron ESS / Smarthome System besteht aus einigen Komponenten: bis zu drei Pythonskripte für essBATT, ein MQTT Server und ein Smarthomeprogramm. Diese

Komponenten einzeln einzurichten wär viel händische Arbeit. Um uns das Leben etwas zu erleichtern nutzen wir Docker Container und zusätzlich Docker Compose. Das hat den Vorteil, dass man sehr unkompliziert Programme installieren und konfigurieren kann. Und zwar ziemlich viele und alles auf einen Schlag mit den „Compose“ Dateien. Bevor wir aber dazu kommen legen wir schonmal im vorausseilendem Gehorsam eine Unterordnerstruktur in unserem „smarthome\_projects“ Ordner an. Dazu Rechtsklick auf den Ordnernamen im VSCode Explorer und dann „new folder“ auswählen und diesen dann „shared\_container\_volumes“ nennen. In diesem dann die beiden Unterordner „iobroker“ und „mosquitto“ anlegen. Und als Unterordner von „mosquitto“ noch die drei Ordner „data“, „log“ und „config“ anlegen. So siehts dann aus:



Im nächsten Schritt müssen wir Docker auf dem Raspberry installieren. Dazu halten wir uns an folgende Anleitung: <https://docs.docker.com/engine/install/raspberry-pi-os/>

Man kann den ganzen Satz Kommandos einfach von der Seite kopieren (Strg+C) und in die Kommandozeile auf dem Raspberry einfügen (das geht einfach mit RECHTSKlick). Die Kommandozeile wird so geöffnet:



Stand Januar 2024 sehen die Befehle zum Installieren dort so aus:

## Install using the apt repository

Before you install Docker Engine for the first time on a new host machine, you need to set up the Docker `apt` repository. Afterward, you can install and update Docker from the repository.

1. Set up Docker's `apt` repository.

```
# Add Docker's official GPG key:
sudo apt-get update
sudo apt-get install ca-certificates curl
sudo install -m 0755 -d /etc/apt/keyrings
sudo curl -fsSL https://download.docker.com/linux/raspbian/gpg -o /etc/apt/keyrings/docker.asc
sudo chmod a+r /etc/apt/keyrings/docker.asc

# Set up Docker's APT repository:
echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.asc] https://
  $(. /etc/os-release && echo "$VERSION_CODENAME") stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt-get update
```

Der erste Satz Befehle dient dazu den internen Updateserver auf Stand zu bringen:

```
christoph@raspismarthome:~/smarthome_projects $ sudo apt-get update
sudo apt-get install ca-certificates curl gnupg
sudo install -m 0755 -d /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/raspbian/gpg | sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg
sudo chmod a+r /etc/apt/keyrings/docker.gpg
Get:1 http://rasbian.raspberrypi.org/raspbian bullseye InRelease [15.0 kB]
Get:2 http://archive.raspberrypi.org/debian bullseye InRelease [23.6 kB]
Get:3 http://archive.raspberrypi.org/debian bullseye/main armhf Packages [314 kB]
Fetched 353 kB in 1s (469 kB/s)
Reading package lists... Done
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
ca-certificates is already the newest version (20210119).
curl is already the newest version (7.74.0-1.3+deb11u7).
gnupg is already the newest version (2.2.27-2+deb11u2).
0 upgraded, 0 newly installed, 0 to remove and 33 not upgraded.
christoph@raspismarthome:~/smarthome_projects $ echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg] https://download.docker.com/linux/raspbian \
  $(. /etc/os-release && echo "$VERSION_CODENAME") stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt-get update
Hit:1 http://archive.raspberrypi.org/debian bullseye InRelease
Hit:2 http://rasbian.raspberrypi.org/raspbian bullseye InRelease
Get:3 https://download.docker.com/linux/raspbian bullseye InRelease [26.7 kB]
Get:4 https://download.docker.com/linux/raspbian bullseye/stable armhf Packages [26.2 kB]
Fetched 52.8 kB in 1s (70.8 kB/s)
Reading package lists... Done
christoph@raspismarthome:~/smarthome_projects $
```

Der zweite Satz Befehle installiert dann Docker:

```
christoph@raspismarthome:~/smarthome_projects $ sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following additional packages will be installed:
  apparmor dbus-user-session docker-ce-rootless-extras libltdl7 libsllp0 slirp4netns
Suggested packages:
  apparmor-profiles-extra apparmor-utils cgroupfs-mount | cgroup-lite
The following NEW packages will be installed:
  apparmor containerd.io dbus-user-session docker-buildx-plugin docker-ce docker-ce-cli docker-ce-rootless-extras docker-compose-plugin libltdl7 libsllp0 slirp4netns
0 upgraded, 11 newly installed, 0 to remove and 33 not upgraded.
Need to get 92.4 MB of archives.
After this operation, 343 MB of additional disk space will be used.
Do you want to continue? [Y/n] y
Get:1 https://download.docker.com/linux/raspbian bullseye/stable armhf containerd.io armhf 1.6.24-1 [21.0 MB]
Get:2 http://mirror.netcologne.de/raspbian/raspbian bullseye/main armhf apparmor armhf 2.13.6-10 [532 kB]
Get:3 http://rasbian.raspberrypi.org/raspbian bullseye/main armhf dbus-user-session armhf 1.12.24-0+deb11u1 [99.7 kB]
Get:4 http://mirror.netcologne.de/raspbian/raspbian bullseye/main armhf libltdl7 armhf 2.4.6-15 [388 kB]
Get:5 http://rasbian.raspberrypi.org/raspbian bullseye/main armhf libsllp0 armhf 4.4.0-1+deb11u2 [50.2 kB]
Get:6 http://mirror.netcologne.de/raspbian/raspbian bullseye/main armhf slirp4netns armhf 1.0.1-2 [29.0 kB]
Get:7 https://download.docker.com/linux/raspbian bullseye/stable armhf docker-buildx-plugin armhf 0.11.2-1-raspbian.11~bullseye [25.6 MB]
Get:8 https://download.docker.com/linux/raspbian bullseye/stable armhf docker-ce-cli armhf 5:24.0.6-1-raspbian.11~bullseye [12.1 MB]
Get:9 https://download.docker.com/linux/raspbian bullseye/stable armhf docker-ce armhf 5:24.0.6-1-raspbian.11~bullseye [14.2 MB]
Get:10 https://download.docker.com/linux/raspbian bullseye/stable armhf docker-ce-rootless-extras armhf 5:24.0.6-1-raspbian.11~bullseye [8,111 kB]
Get:11 https://download.docker.com/linux/raspbian bullseye/stable armhf docker-compose-plugin armhf 2.21.0-1-raspbian.11~bullseye [10.3 MB]
Fetched 92.4 MB in 8s (11.2 MB/s)
Preconfiguring packages ...
Selecting previously unselected package apparmor.
(Reading database ... 45845 files and directories currently installed.)
Preparing to unpack .../00-apparmor_2.13.6-10_armhf.deb ...
Unpacking apparmor (2.13.6-10) ...
Selecting previously unselected package containerd.io.
```

Wie in der Anleitung vorgeschlagen würde ich das Ganze nochmal testen:

```
christoph@raspismarthome:~/smarthome_projects $ sudo docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
c4018b8bf438: Pull complete
Digest: sha256:4f53e2564790c8e7856ec08e384732aa38dc43c52f02952483e3f003afbf23db
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
   (arm32v7)
3. The Docker daemon created a new container from that image which runs the
   executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it
   to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
$ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
https://hub.docker.com/

For more examples and ideas, visit:
https://docs.docker.com/get-started/

christoph@raspismarthome:~/smarthome_projects $
```

Jetzt muss noch Docker Compose installiert werden:

```
sudo apt install docker-compose
```

und danach alles für den automatischen Neustart konfigurieren:

```
sudo systemctl enable docker
```

## Klonen der essBATT Git Repositories

Zuerst müssen wir sicherstellen das Git auf dem Raspi installiert ist:

```
sudo apt-get install git
```

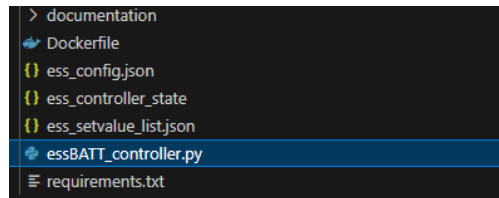
Jetzt klonen wir das essBATT Controller Repository. Dazu öffnen wir, wie weiter oben schon gezeigt, das Terminal im „*smarthome\_projects*“ Ordner.

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
christoph@raspismarthome:~/smarthome_projects $
```

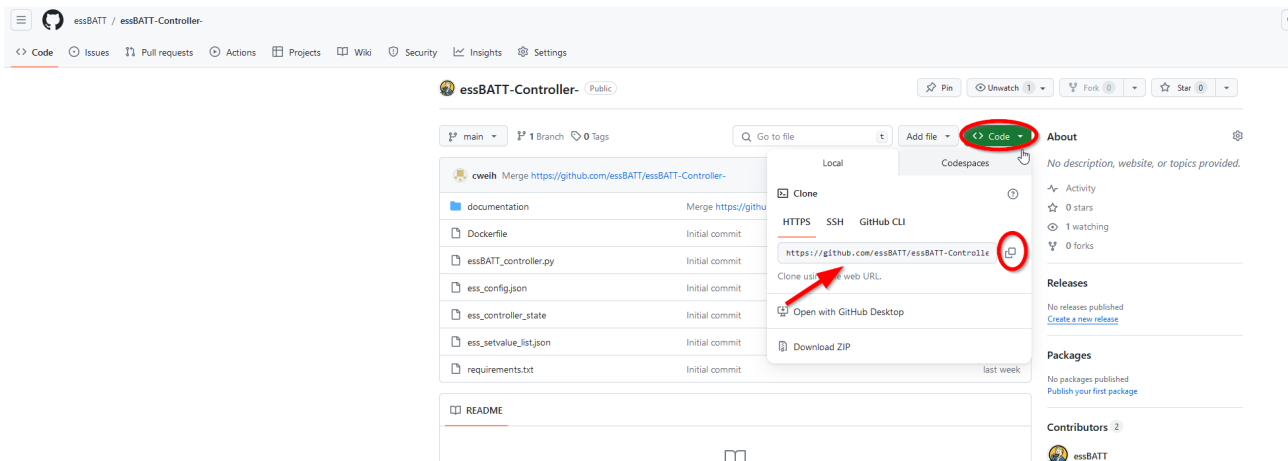
Dort geben wir folgenden Befehl ein:

```
sudo git clone https://github.com/essBATT/essBATT-Controller-.git
```

Die essBATT Dateien sollten sich jetzt im „*/essBATT-Controller-*“ Unterordner befinden (evtl. dauert es einige Zeit bis die Dateien angezeigt werden):



**Kleiner Git Exkurs:** Den Link zum git Repository vom essBATT Controller findet man wie im Bild gezeigt auf der github Seite. Mit „git clone URL“ kann man sich generell den Code von Github lokal auf den Rechner holen:



**Kleiner Exkurs zum „navigieren“ im Terminal:** Bei manchen Befehlen (wie z.B. dem git clone Befehl von oben) ist es wichtig, das man sie aus der richtigen Ordnerebene aufruft. Über VSCode remote kann man, wie gezeigt, schnell über einen Rechtsklick auf einen Ordner und „open in integrated terminal“ ein Terminal auf der benötigten Ordnerebene öffnen. Ist man nur im Terminal unterwegs navigiert man mit Textbefehlen. Dort ist ein wichtiger Befehl „ls“ (für „list“), der die Unterordner und Dateien anzeigt:

```
christoph@raspismarthome:~/smarthome_projects $ ls
can0_bridge can0_bridge.log docker-compose.yml ess_controller.log ess_controller.log.1 shared_container_volumes victron_ess_home_control
christoph@raspismarthome:~/smarthome_projects $ cd shar
```

Eine Ordnerebene wechselt man mit „cd“ (change directory). Fängt man dann an den Namen des Unterordners einzutippen in den man wechseln möchte, kann man die TAB Taste für Autovervollständigung nutzen. Die Ordner Ebenen nach oben erreicht man mit „cd ..“ und ins HOME Verzeichnis mit „cd ~“:

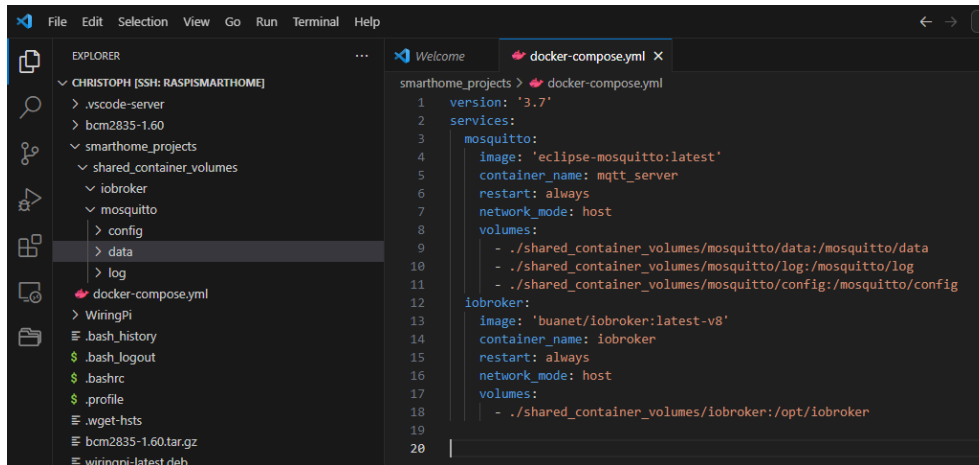
```
christoph@raspismarthome:~/smarthome_projects $ cd shared_container_volumes/
christoph@raspismarthome:~/smarthome_projects/shared_container_volumes $ cd .
christoph@raspismarthome:~/smarthome_projects/shared_container_volumes $ cd ..
christoph@raspismarthome:~/smarthome_projects $ cd ~
christoph@raspismarthome:~ $
```

## Docker Compose file

Bevor wir jetzt final alles auf einmal starten können, müssen wir das Docker Compose File an die



richtige Stelle schieben. Und zwar befindet sich diese Datei im „misc“ Ordner, den wir gerade per „git clone“ runtergeladen haben. Zieh von dort die Datei „docker-compose.yml“ per Drag&Drop im VSCode Explorer in den „smarthome\_projects“ Ordner (also von „./smarthome\_projects/essBATT-Controller/misc“ nach „./smarthome\_projects“. Nach Doppelklick auf die „docker-compose.yml“ Datei sieht das Ganze in etwa so aus:



The screenshot shows the VS Code interface with the Explorer sidebar on the left and the Editor on the right. The Explorer sidebar shows the file structure of the project, with the 'docker-compose.yml' file selected under the 'smarthome\_projects' directory. The Editor shows the content of the 'docker-compose.yml' file, which is a YAML configuration for Docker Compose. The configuration defines two services: 'mosquitto' and 'iobroker'. The 'mosquitto' service uses the 'eclipse-mosquitto:latest' image and has a 'container\_name' of 'mqtt\_server'. It is configured to restart 'always' and is connected to the 'host' network. It also has two volumes: 'data' and 'log', both mapped to the 'shared\_container\_volumes' directory. The 'iobroker' service uses the 'buanet/iobroker:latest-v8' image and has a 'container\_name' of 'iobroker'. It is also configured to restart 'always' and is connected to the 'host' network. It has one volume: 'opt', mapped to the 'shared\_container\_volumes/iobroker' directory.

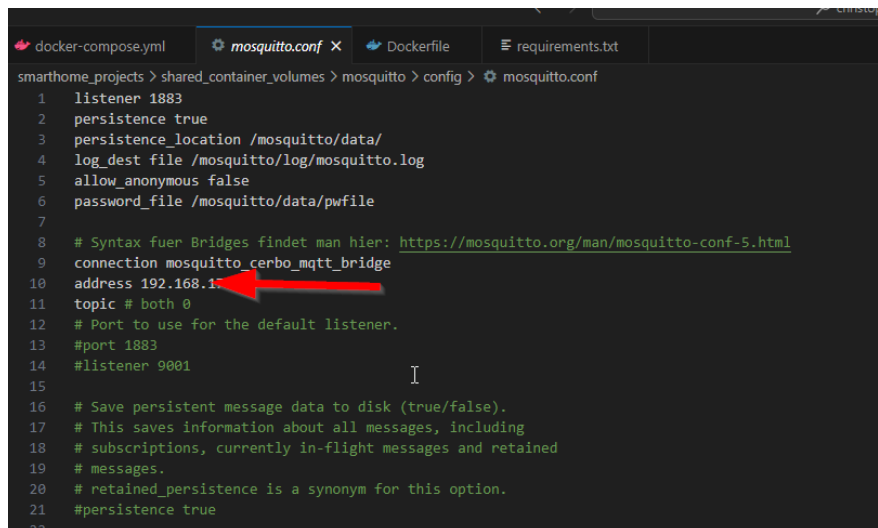
```
1 version: '3.7'
2 services:
3   mosquitto:
4     image: 'eclipse-mosquitto:latest'
5     container_name: mqtt_server
6     restart: always
7     network_mode: host
8     volumes:
9       - ./shared_container_volumes/mosquitto/data:/mosquitto/data
10      - ./shared_container_volumes/mosquitto/log:/mosquitto/log
11      - ./shared_container_volumes/mosquitto/config:/mosquitto/config
12   iobroker:
13     image: 'buanet/iobroker:latest-v8'
14     container_name: iobroker
15     restart: always
16     network_mode: host
17     volumes:
18       - ./shared_container_volumes/iobroker:/opt/iobroker
19
20
```

In der docker-compose Datei müssen keine Änderungen vorgenommen werden – es sei denn man hat schon einen MQTT Server und iobroker bzw. irgendein anderes Smarthomesystem. In dem Fall einfach die entsprechenden Services die man nicht benötigt rauslöschen.

Vielleicht noch zur Erklärung zu den Zeilen mit „timezone“: die dienen dazu, das die Systemzeit in jedem Docker Container der Systemzeit des Raspis entspricht. Das ist defaultmäßig nicht der Fall und man wundert sich beim zeitgesteuerten Laden später, das um 01:00Uhr Timbuktu Zeit geladen wurde.

## MQTT Server Konfiguration

Unser MQTT Server Mosquitto wird später automatisch über Docker Compose installiert. Damit er aber ordnungsgemäß funktioniert müssen noch einige Schritte durchgeführt werden. Zuerst kopieren wir die im Unterkapitel Klonen der essBatt Git Repositories geklonte Datei „mosquitto.conf“ aus dem „misc“ Ordner in den vorhin angelegten „mosquitto/config“ Ordner und öffnen diese dann.



```
1 listener 1883
2 persistence true
3 persistence_location /mosquitto/data/
4 log_dest file /mosquitto/log/mosquitto.log
5 allow_anonymous false
6 password_file /mosquitto/data/pwfile
7
8 # Syntax fuer Bridges findet man hier: https://mosquitto.org/man/mosquitto-conf-5.html
9 connection mosquitto_cerbo_mqtt_bridge
10 address 192.168.1.
11 topic # both 0
12 # Port to use for the default listener.
13 #port 1883
14 #listener 9001
15
16 # Save persistent message data to disk (true/false).
17 # This saves information about all messages, including
18 # subscriptions, currently in-flight messages and retained
19 # messages.
20 # retained_persistence is a synonym for this option.
21 #persistence true
22
```

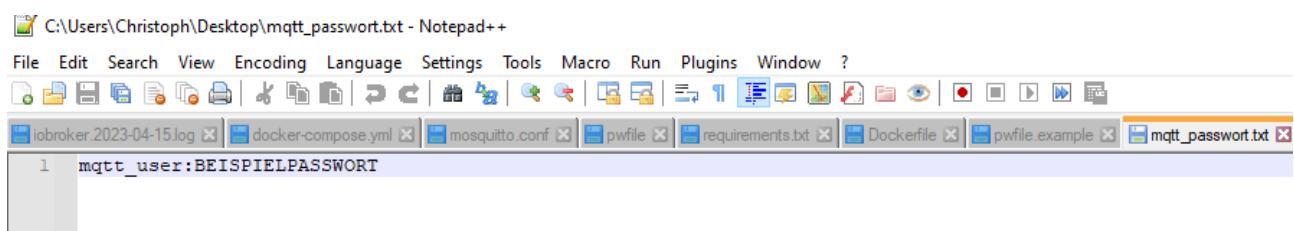
Dort werden verschiedene Sachen konfiguriert von denen wir uns die meisten jetzt nicht genauer anschauen müssen. In der ersten Zeile muss der Port auf dem Mosquitto lauscht angegeben werden – für diese Anleitung lassen wir den Wert 1883 aber unverändert.

An der Stelle des roten Pfeils muss die IP Adresse eures Cerbos/VenusOS eingetragen werden. Im Wesentlichen wird in diesen drei Zeilen eine Brücke gebaut, die die Nachrichten des Cerbo internen MQTT Servers an unseren eigenen MQTT Server weiterleitet.

In der Zeile 6 sieht man, das wir eine Passwortdatei verlangen. Diese müssen wir jetzt als nächstes generieren. Dafür müssen wir an dieser Stelle einen Umweg gehen und erst noch Mosquitto für Windows runterladen und installieren:

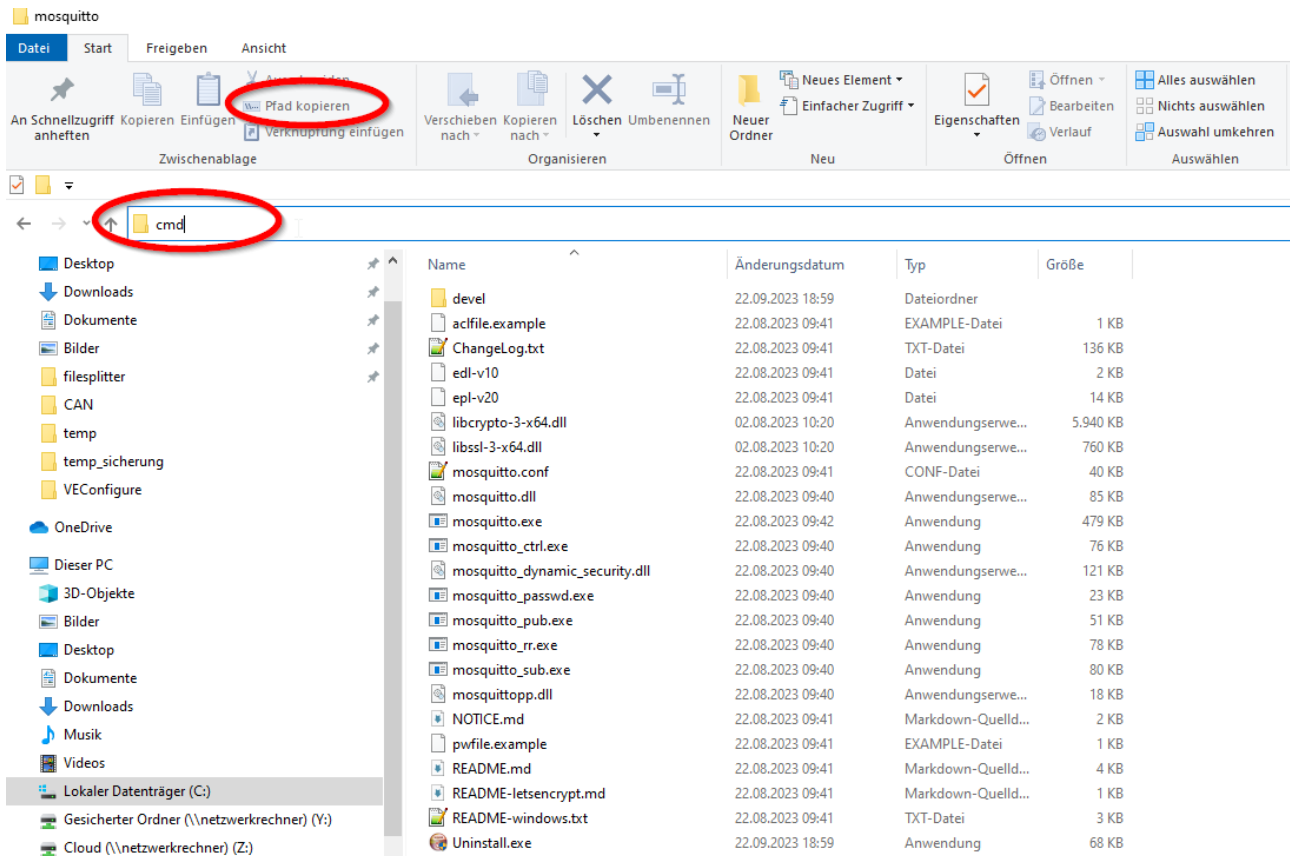
<https://mosquitto.org/download/>

Danach legen wir z.B. auf dem Desktop eine Textdatei an, die die gewünschten Logindaten enthält. Bei mir heißt die Datei „mqtt\_passwort.txt“ und das Passwort muss so formatiert sein:



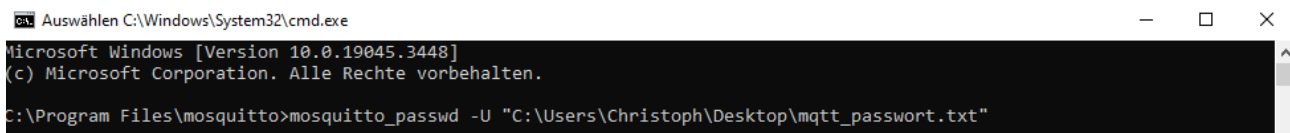
```
1 mqtt_user:BEISPIELPASSWORT
```

**WICHTIG:** Speichert euch euren Nutzernamen (könnt ihr wählen wie ihr wollt – bei mir hier „mqtt\_user“) und das Passwort irgendwo sicher weg. Wann immer irgend ein Tool MQTT Botschaften senden oder empfangen will werden diese Daten benötigt. Ich speicher sowas alles in KeePass. Jetzt speichert und schließt die Datei. Danach geht ihr in den mosquitto Installationsordner in windows und öffnet dort die Kommandozeile indem ihr in die Pfadzeile „cmd“ eintippt und dann „Enter“ drückt:



Dann in die Kommandozeile folgenden Befehl eingeben und mit Enter bestätigen:

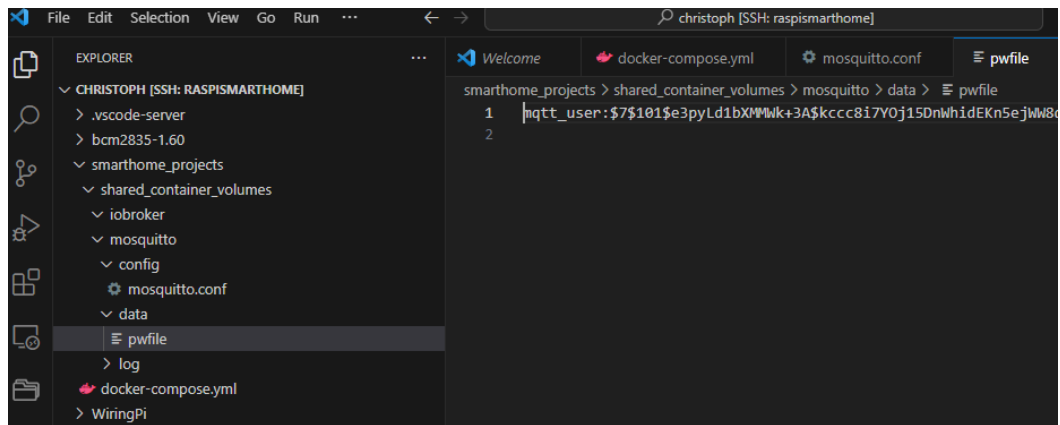
`mosquitto_passwd -U „PFADZUEURERPASSWORTDATEI“`



Wenn ihr die Passwortdatei jetzt öffnet werdet ihr feststellen, das euer altes Passwort jetzt komisch aussieht. Dann ist alles korrekt!

Diese veränderte Passwortdatei benennt ihr jetzt um in „*pwfile*“ und ENTFERNT die Endung .txt.

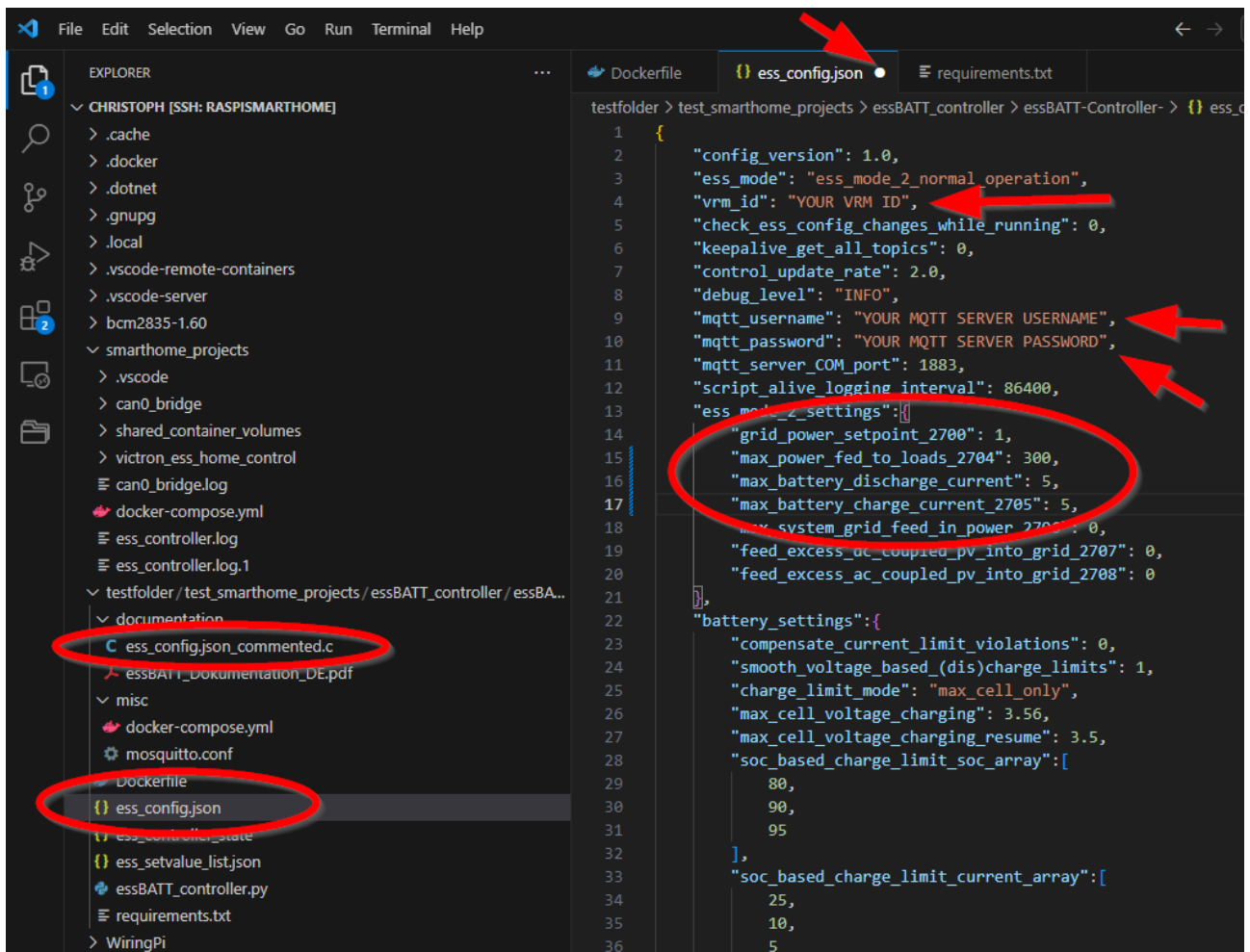
Diese Passwortdatei zieht ihr jetzt in den „*data*“ Ordner im VSCode. Wenn man jetzt nochmal in die „*mosquitto.conf*“ Datei schaut sieht man, das wir der Zeile 6 genüge getan haben und eine „*pwfile*“ Datei in dem richtigen Ordner haben. So sieht es dann aus:



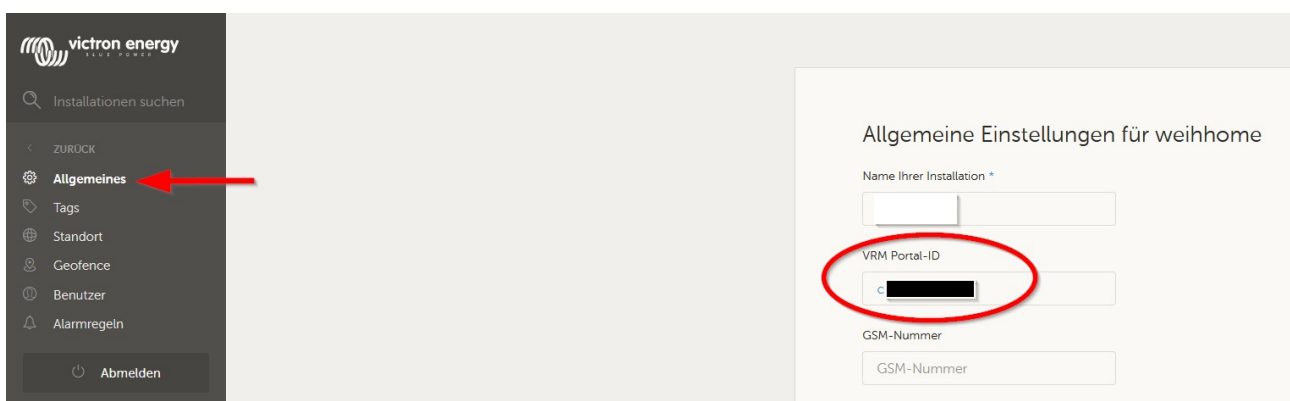
## Konfiguration des essBATT Controllers

Da wir im letzten Schritt alle Komponenten auf einmal starten müssen wir jetzt noch den essBATT Controller konfigurieren. **ACHTUNG:** in dieser Konfiguration könnt ihr Einstellungen vornehmen die euer System beschädigen könnten. Eure Komponenten sind zwar noch über die Einstellungen in euren Multis und eurem BMS geschützt, aber Schäden können nicht ausgeschlossen werden. Lest euch in jedem Fall das Kapitel Konfiguration vom essBATT Controller genau durch und versteht die Einstellungen. Wenn das für euch ein Buch mit sieben Siegeln ist, ist essBATT vermutlich (noch) nichts für euch.

Die Settings in der default Konfiguration passen ganz gut zu einem LiFePo4 Akku und sind aus Sicherheitsgründen auf 5A Lade- und Entladestrom und 300W Multileistung begrenzt. Für einen ersten Start macht es vielleicht Sinn nicht die volle Systemleistung einzustellen.



Die Datei in der alle Einstellungen für den essBATT Controller vorgenommen werden heißt „ess\_config.json“. Die öffnet ihr und gebt eure VRM ID, euren MQTT Username und euer MQTT Passwort ein. Die VRM ID findet ihr in eurem VRM Portal unter „Einstellungen → Allgemeines“.



Sowohl VRM ID als auch Username und Passwort müssen innerhalb von Anführungszeichen stehen. Zahlenwerte werden in JSON generell ohne Anführungszeichen eingegeben. Im „documentation“ Ordner findet ihr eine kommentierte „ess\_config.json“ Datei, die eine kurze Beschreibung der einzelnen Parametern gibt. Wenn ihr die zu eurem System passenden Einstellungen in dieser Datei vorgenommen habt müsst ihr die Datei speichern (z.B. mit Strg + s).

Ihr erkennt eine ungespeicherte Dateiänderung an dem weißen Punkt neben dem Dateinamen in VSCode (siehe roter Pfeil im Bild).

## Konfiguration des Cerbo / VenusOS

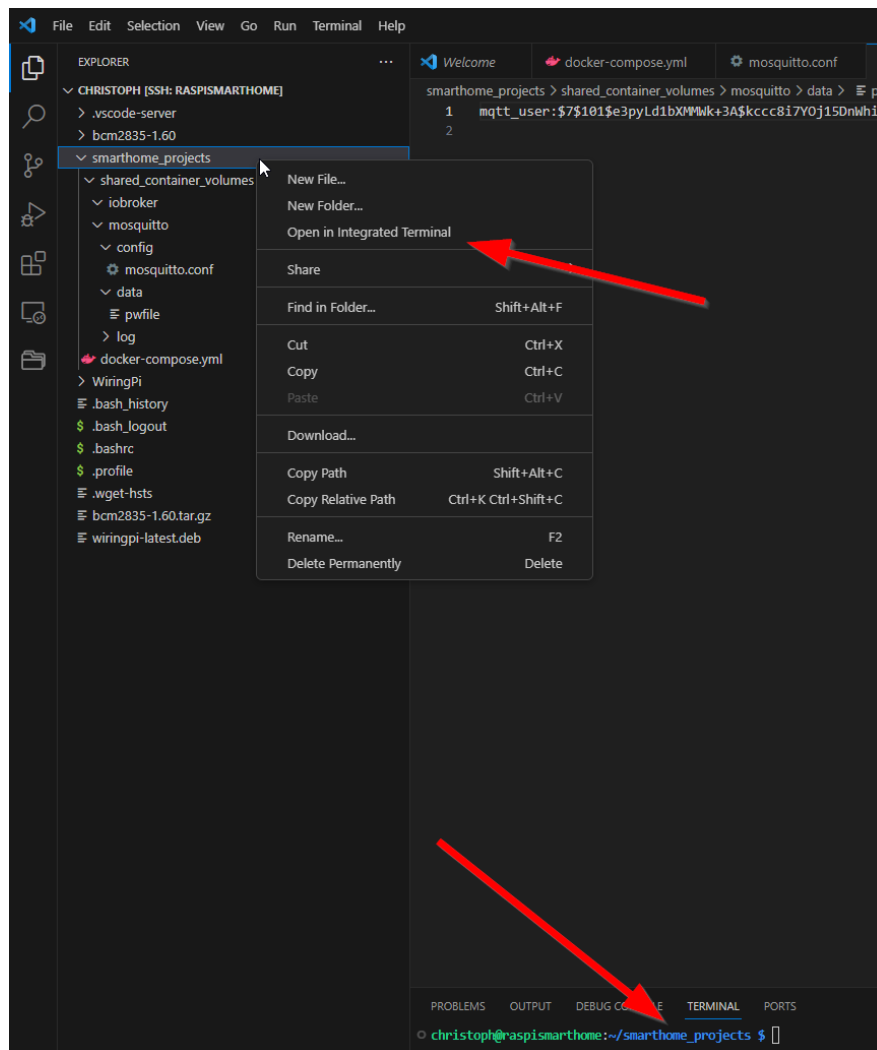
Damit der essBATT Controller gleich mit dem Victron System kommunizieren kann müssen auch dort noch Einstellungen vorgenommen werden. Es muss sichergestellt sein das der ESS Mode richtig konfiguriert ist und das die MQTT Schnittstelle aktiviert wurde. Näheres im Kapitel Einstellugen im Cerbo / VenusOS.

## Konfiguration der Multis

Zu guter Letzt macht es Sinn die Einstellungen im Multi noch einmal zu überprüfen. Dazu mehr im Kapitel Einstellungen in den Multis.

## Starten der Docker Container

Jetzt sind wir im Großen und Ganzen fertig und können alles installieren und aktivieren. Das Installieren aller Container im „Compose“ File wird wieder über die Kommandozeile ausgeführt. Die lokale Kommandozeile nutzen wir einfach auch in VSCode und öffnen die Kommandozeile gleich im „*smarthome\_projects*“ Ordner, weil dort ja auch unsere „compose“ Datei liegt. Es gibt eine Fehlermeldung wenn die Terminalbefehle nicht aus diesem Pfad ausgeführt werden:



Unten im Terminal geben wir jetzt folgenden Befehl ein:

```
sudo docker-compose up -d
```

```
christoph@raspismarhome:~/smarthome_projects $ sudo docker-compose up -d
Pulling mosquitto (eclipse-mosquitto:latest)...
latest: Pulling from library/eclipse-mosquitto
af09961d4a43: Pull complete
42a0caf2b06a: Pull complete
00869a7064da: Pull complete
Digest: sha256:6e957b3dffe57afa895bed1e311db6f786eb93de70519df2dc81ef77ad6a21d3
Status: Downloaded newer image for eclipse-mosquitto:latest
Pulling iobroker (buanet/iobroker:latest-v8)...
latest-v8: Pulling from buanet/iobroker
69e0260f539f: Pull complete
6a5182a48838: Pull complete
4824f3f302dc: Pull complete
dc5a5e496d9c: Pull complete
4f4fb700ef54: Pull complete
Digest: sha256:d5348ef1cfc2a9647ef24bf70c9182eb3f077805998f95ba883ca25eb7133a22
Status: Downloaded newer image for buanet/iobroker:latest-v8
Creating mqtt_server ... done
Creating iobroker ... done
christoph@raspismarhome:~/smarthome_projects $
```

Um zu checken ob alles läuft:

```
sudo docker compose ps
```

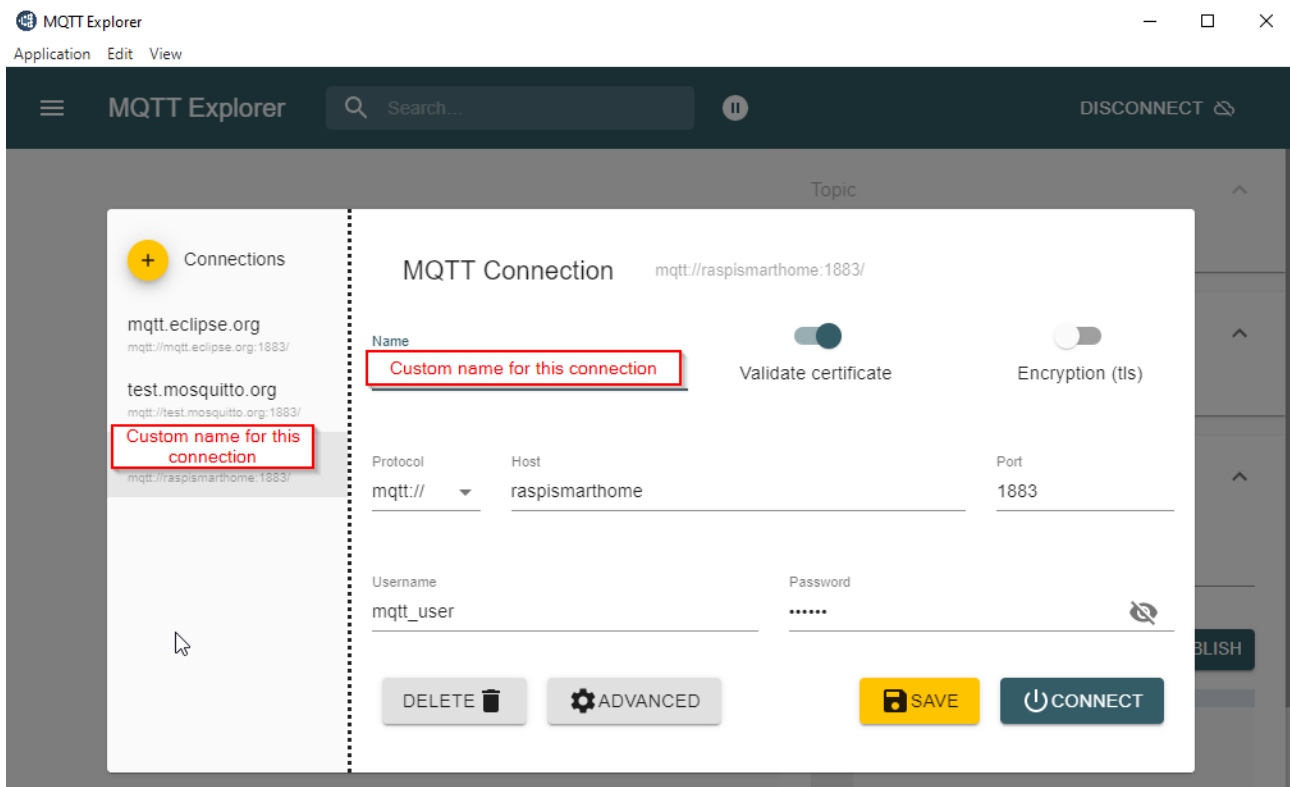
NAME	IMAGE	COMMAND	SERVICE	CREATED	STATUS
can0_bridge_container	saarthome_projects_can0_bridge_service	"python ./can0_bridge.py"	can0_bridge_service	2 months ago	Up 3 weeks
ess_controller_container	saarthome_projects_ess_controller_service	"python ./ess_controller.py"	ess_controller_service	2 weeks ago	Up 2 weeks
iobroker	bianet/iobroker:latest-v8	"bin/bash -c /opt/scripts/iobroker_startup.sh"	iobroker	2 months ago	Up 3 weeks (healthy)
mqtt_server	eclipse-mosquitto:latest	"docker-entrypoint.sh /usr/sbin/mosquitto -c /mosquitto/config/mosquitto.conf"	mosquitto	2 months ago	Up 3 weeks

Ihr solltet jetzt drei Einträge sehen die ggf. etwas anders heißen als hier bei mir. Der can0\_bridge\_container ist korrekterweise bei euch jetzt nicht zu sehen.

## MQTT Server Testing

Jetzt überprüfen wir ob wir uns mit unserem gewählten Passwort mit dem MQTT Server verbinden können und Botschaften sehen. Dafür gibt es ein super Tool: <https://mqtt-explorer.com/>

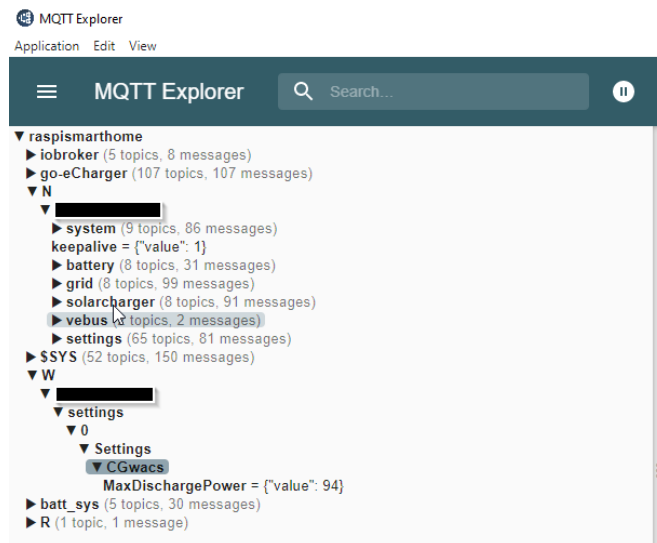
Da kann man einfach nur die .exe Datei als portable Software runterladen und starten. Folgendes musste ich dort eintragen mit meinem MQTT Benutzernamen „mqtt\_user“ und meinem Raspberry Hostnamen „raspismarthome“:



Jetzt solltet ihr die MQTT Botschaften sehen die von den einzelnen Geräten gesendet werden. Insbesondere „N“ und „W“ sind wichtig, da unter diesen „Reitern“ die Victron Botschaften aufgeführt werden. Unter „N“ sieht man die Botschaften die aus dem Victronsystem ausgelesen wurden und unter „W“ die Botschaften die man selbst geschrieben hat. Der essBATT Controller sendet selbstständig die MQTT Keepalive Botschaft, sodass Victron dauerhaft auch Werte raussendet. Fällt die Keepalive Botschaft aus (z.B. weil man den essBATT Controller Docker Container stoppt) hört Victron nach etwa einer Minute auch auf die Botschaften zu senden. In dem Fall werden die Topics „N“ und „W“ gelöscht und tauchen gar nicht mehr in dieser Liste auf. Wenn ihr unter „N“ die Topics „system“, „grid“, „battery“, „settings“ und „vebus“ (ggf. auch Solarcharger

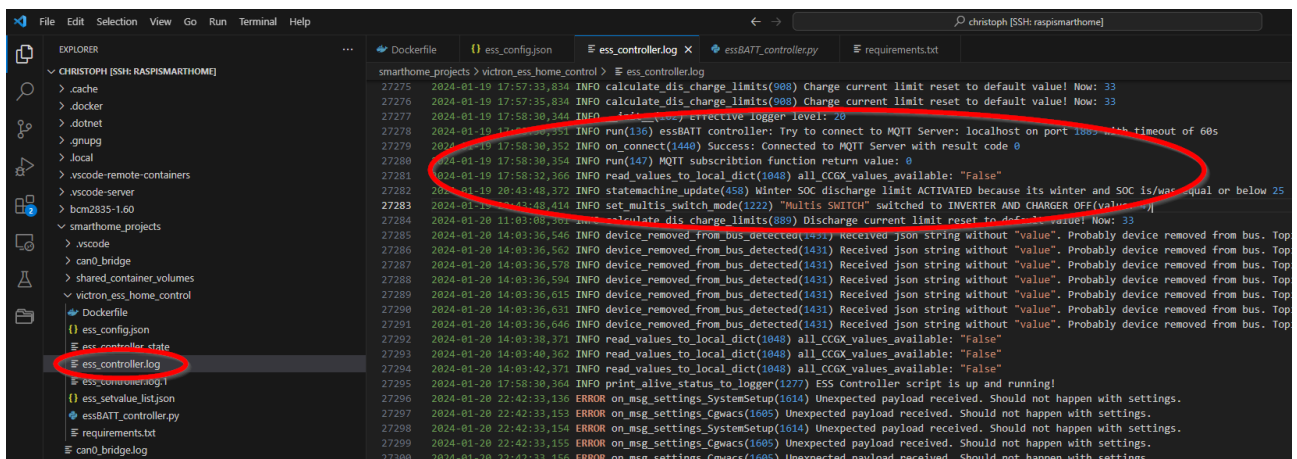


wenn ihr welche verbaut habt und es Tag ist) seht, ist essBATT erfolgreich eingerichtet und arbeitet schon.



## Das essBATT Controller Logfile

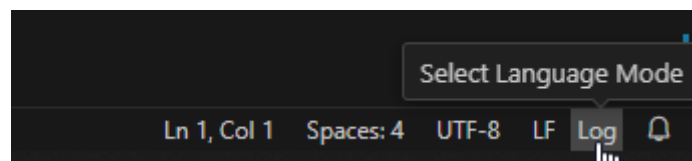
Wenn die vorherigen Schritte alle erfolgreich waren läuft der essBATT Controller jetzt. Die erste Anlaufstelle um das zu überprüfen bzw. mögliche Fehlerzustände zu erkennen ist das Logfile. Dieses hat essBATT automatisch im essBATT-Controller- Ordner angelegt. Reinschauen könnt ihr in die Datei „essBATT\_controller.log“ (im Screenshot heißt sie etwas anders) mit VSCode.



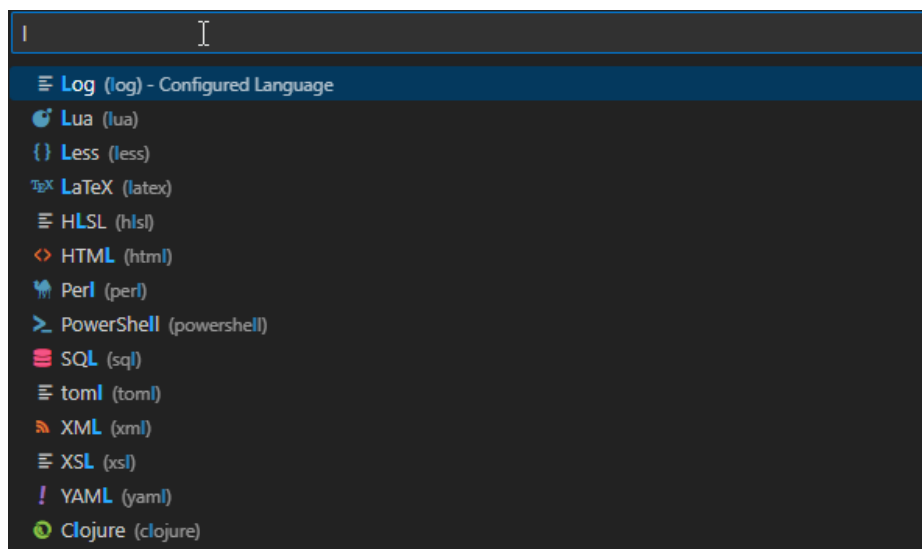
Dort sieht man ob essBATT sich mit MQTT verbinden konnte bzw. ob irgendwelche Fehler aufgetreten sind. Z.b. gibt die Zeile mit **all\_CCGX\_values\_available: "False"** an, das in dem Zyklus nicht alle benötigten Werte von Batterie, Netzzähler oder Victron vorlagen. In solchen Zyklen bleibt essBATT im Wesentlichen untätig. Wenn das nur vereinzelt passiert ist das kein Problem. Zum Beispiel verliert mein Netzzähler mehrmals am Tag die Verbindung für einige Sekunden und dann wird diese Information ausgegeben. Hinweis: Wenn die maximale Größe dieser

Logdatei erreicht ist wird automatisch eine zweite Logdatei (...log.1) angelegt. Wenn auch diese voll ist werden die Logginginformationen wieder in die erste Datei geschrieben. Dabei werden die alten Einträge gelöscht. Im Normalbetrieb sollte man eine Logginghistorie von mehreren Wochen haben. Aktiviert man den „DEBUG“ Modus fürs Logging hat man nur noch eine Historie von ~1 Tag, weil einfach viel mehr Informationen in das Logfile geschrieben werden. Änderungen an der Logdateigröße gehen nur direkt in der Datei „essBATT\_controller.py“.

**Exkurs VSCode:** Wenn ihr mit VSCode in die Logdatei mit der Endung „log.1“ schaut werdet ihr feststellen, das die schönen farblichen Hervorhebungen fehlen und alles als schnöder weißer Text angezeigt wird. Das liegt daran, das VSCode diese Dateiendung nicht automatisch richtig zuordnen konnte. Unten rechts könnt ihr auswählen wie VSCode eine Datei interpretieren/anzeigen soll:



Es öffnet sich ein Menü in dem man nach der richtigen Interpretation suchen kann – in diesem Fall jetzt „log“:



## Nachträgliche Änderung von essBATT Controller Einstellungen

Jetzt wollen wir ggf. einige Änderungen an der Konfiguration vornehmen. Zum Beispiel ärgern wir uns jetzt das wir mit den Defaulteinstellungen in der essBATT Konfiguration nur einen maximalen Ladestrom von 5A zulassen, unsere PV Anlage aber gerne 100A in die Batterie pumpen würde und wir den kostbaren Strom gerade verschenken. Oder wir wollen den „DEBUG“ Modus für das Logfile aktivieren oder Victron auffordern nicht nur die benötigten MQTT Botschaften zu schicken sondern ALLE Werte auf die man per MQTT Zugriff hat. Dazu nehmen wir zuerst die gewünschten Einstellungen in der Konfig Datei vor, wie im letzten Unterkapitel vorgestellt.

```

1 {
2   "config_version": 1.0,
3   "ess_mode": "ess_mode_2_normal_operation",
4   "vrm_id": "YOUR VRM ID",
5   "check_ess_config_changes_while_running": 0,
6   "keepalive_get_all_topics": 1,
7   "control_update_rate": 2.0,
8   "debug_level": "DEBUG",
9   "mqtt_username": "YOUR MQTT SERVER USERNAME",
10  "mqtt_password": "YOUR MQTT SERVER PASSWORD",
11  "mqtt_server_COM_port": 1883,
12  "script_alive_logging_interval": 86400,
13  "ess_mode_2_settings": {
14    "grid_power_setpoint_2700": 1,
15    "max_power_fed_to_loads_2704": 5000,
16    "max_battery_discharge_current": 33,
17    "max_battery_charge_current_2705": 33,
18    "max_system_grid_feed_in_power_2706": 0,
19    "feed_excess_dc_coupled_pv_into_grid_2707": 0,
20    "feed_excess_ac_coupled_pv_into_grid_2708": 0
  }
}

```

Mit „*keepalive\_get\_all\_topics*“ gesetzt auf 1 fordert man Victron auf ALLE Botschaften zu senden. Mit „*debug\_level*“ auf „*DEBUG*“ erhält man detaillierte Ausgaben im Logfile und mit „*max\_battery\_charge\_current\_2705*“ setzt man den erlaubten maximalen Ladestrom für die Batterie. Zum Rumspielen mit Zahlenwerten ist noch der Parameter „*check\_ess\_config\_changes\_while\_running*“ interessant. Setzt man den auf 1 kann man viele Einstellungen (z.B. alle unter „*ess\_mode\_2\_settings*“ oder alle unter „*battery\_settings*“) setzen während das Skript läuft und die Änderungen werden dann sofort (nach Speichern der Konfigurationsdatei) angewendet.

Jetzt wo wir alle gewünschten Einstellungen vorgenommen haben müssen wir den Docker Container in dem der essBATT Controller läuft stoppen und dann neu starten. Dazu öffnen wir wieder, wie bereits mehrfach in dieser Doku gezeigt, ein Terminal im Pfad des „*smarthome\_projects*“ Ordner. Dort geben wir ein

```
sudo docker compose ps
```

NAME	IMAGE	COMMAND	SERVICE	CREATED	STATUS	PORTS
can0_bridge_container	smarthome_projects:can0_bridge_service	"python ./can0_bridge.py"	can0_bridge_service	7 weeks ago	Up 5 days	
ess_controller_container	smarthome_projects:ess_controller_service	"python ./ess_controller.py"	ess_controller_service	7 weeks ago	Up 11 minutes	
iobroker	buonet/iobroker:latest-v8	"/bin/bash -c /opt/scripts/iobroker_startup.sh"	iobroker	7 weeks ago	Up 5 days (healthy)	
mqtt_server	eclipse-mosquitto:latest	"/docker-entrypoint.sh /usr/sbin/mosquitto -c /mosquitto/config/mosquitto.conf"	mosquitto	7 weeks ago	Up 5 days	

Danach suchen wir uns den Namen des Service aus der Spalte „SERVICE“. Bei euch heißt der „*essBATT\_controller\_service*“ (der Name wurde in der *docker-compose.yml* Datei vorgegeben). Nutzt den Namen der bei euch aufgeführt wird um diesen Befehl auszuführen:

```
sudo docker-compose stop essBATT_controller_service
```

Danach startet den essBATT Controller Service direkt wieder – die neue Konfiguration ist ja schon gespeichert und wird bei einem Neustart auch neu eingelesen.

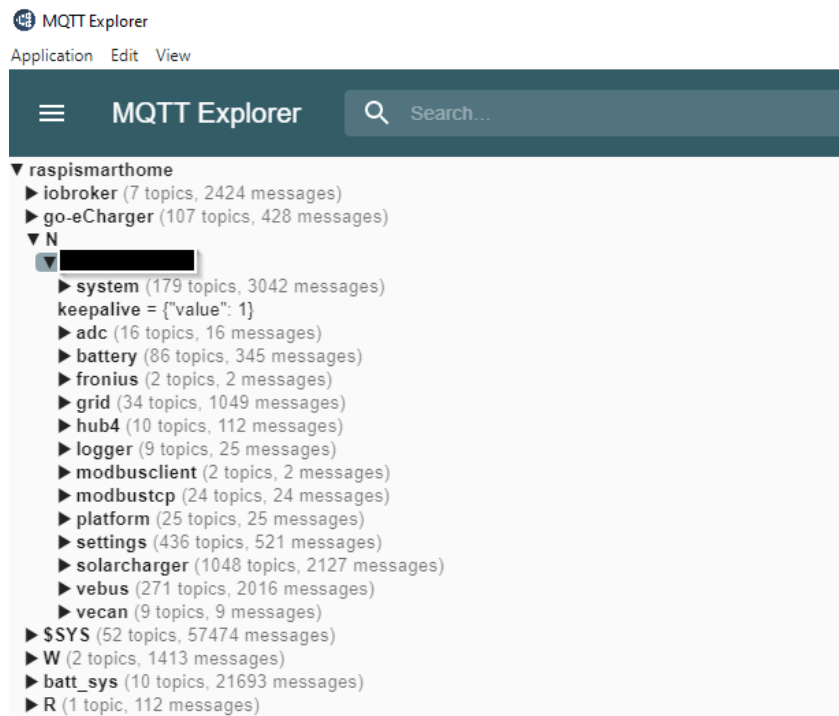
```
sudo docker-compose start essBATT_controller_service
```

Schneller gehts wenn man statt der beiden Befehle hintereinander einfach einmal den „*restart*“

Befehl nutzen:

```
sudo docker-compose restart essBATT_controller_service
```

Ob das Neustarten geklappt hat oder nicht könnt ihr jetzt z.B. im Logfile überprüfen. Wenn ihr Victron aufgefordert habt alle MQTT Topics zu senden könnt ihr das jetzt zusätzlich im MQTT Explorer überprüfen. Da sieht man jetzt erheblich mehr Einträge als vorher (siehe Bild). Hinweis: Bevor ihr jetzt alles Tage/Wochen lang laufen lasst schaltet das Senden aller Nachrichten wieder aus, da andernfalls die Buslast und die Rechenlast auf dem Cerbo unnötig hoch ist, was schwer zu findende Probleme verursachen könnte:



## Konfiguration vom essBATT Controller

Text

## Eigene Weiterentwicklungen und Debugging vom essBATT Controller

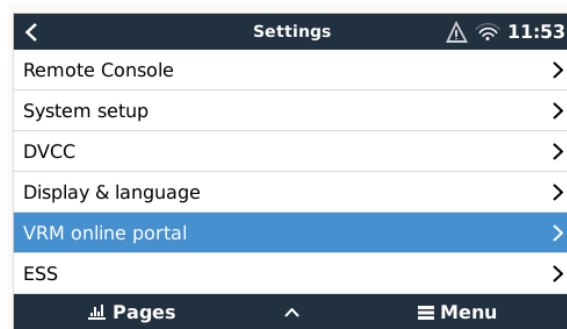
text

## Konfiguration des Victron ESS

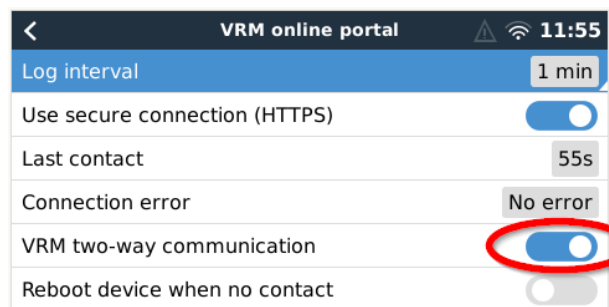
Text

## Einstellungen im Cerbo / VenusOS

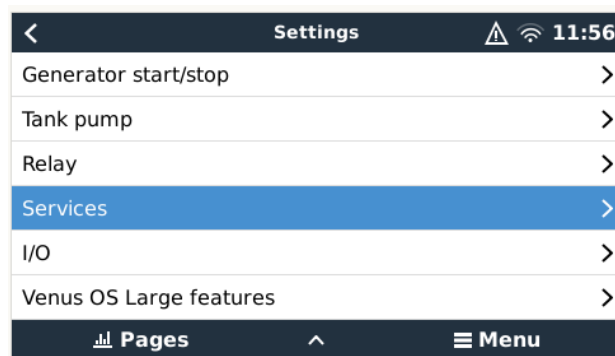
Zuerst überprüfen wir ob alle benötigten Kommunikationspfade freigeschaltet sind. Dazu in der Cerbo / VenusOS Konsole zum Menüpunkt „VRM online portal“ gehen.



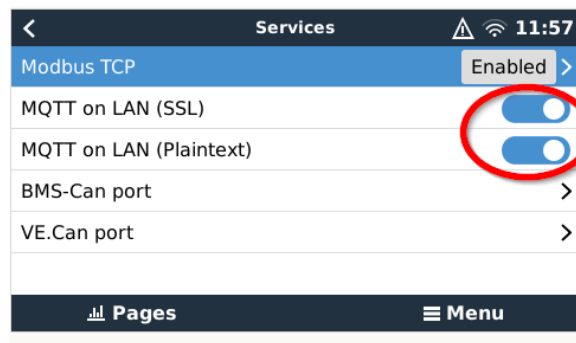
Stelle sicher das die „Zwei Wege Kommunikation“ aktiviert ist.



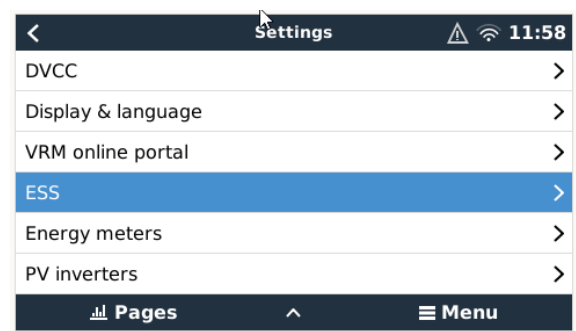
Danach schauen wir unter „Services“.



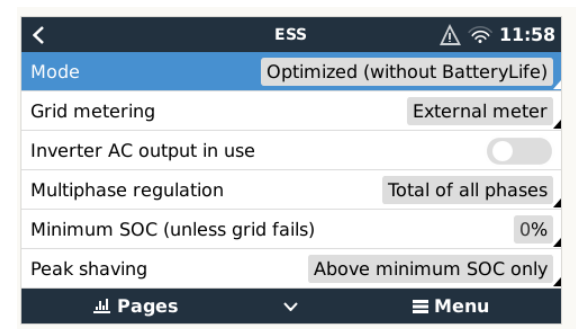
Dort müssen die beiden MQTT Menüpunkte aktiviert sein:



Als letztes Überprüfen wir unsere ESS Einstellung:



So sieht es bei mir aus. Für den essBATT Controller muss dort **NICHT** „external control“ aktiviert werden. Das ist nur für den ESS Mode 3 den essBATT nicht verwendet.



## Einstellungen in den Multis

text

## Konfiguration des (physischen / übergeordnetem) BMS und Balancers

Text

# NEEY Balancer

texxt

## Einrichten von iobroker

Der iobroker läuft schon und man kann ihn erreichen im Browser wenn man eintippt:

<http://raspismarthome:8081>

Die Ersteinrichtung ist selbsterklärend. Man sollte wegen verschiedener Adapter aber tatsächlich auch den richtigen Wohnort eingeben.

Ein wichtiges Thema ist aber die Verschlüsselung. Der iobroker steuert das ganze Haus und sollte nicht zugänglich sein für andere. Hier halte ich mich an diese Anleitung (die auch als PDF abliegt):

<https://smarthome.buanet.de/2021/01/ssl-und-authentifizierung-fuer-den-iobroker-admin/>

Mir ist das als unsicher angezeigt Zertifikat egal – Hauptsache ich bin mit https// verbunden und niemand kann den Datenverkehr mitlesen.

Zur Nutzung von iobroker kann man ganze Bücher füllen. Ich will nur auf den wichtigsten Punkt aufmerksam machen. Der Adapter (der iobroker Name für Plugin) „backup“. Der zieht von deiner ganzen iobroker Konfiguration in regelmäßigen Abständen Sicherungen. Die Sicherungen kann man dann auch über VSCode im iobroker Ordner sehen und da rauskopieren um die sicher abzulegen. Ich habe so viele lokale Skripte und Adapter und auch eine GUI für mein Smartphone gebaut zur Steuerung und Anzeige der MQTT Werte, das ein Verlust der Konfiguration unbezahlbar wäre. Mit „backup“ kann man die vollständige Konfig wiederherstellen. Auch beim Wechsel auf eine neue Major Version von iobroker sollte man den Weg über „backup“ gehen, weil dieses Tool auch in neueren Versionen wieder alles sauber einrichtet.

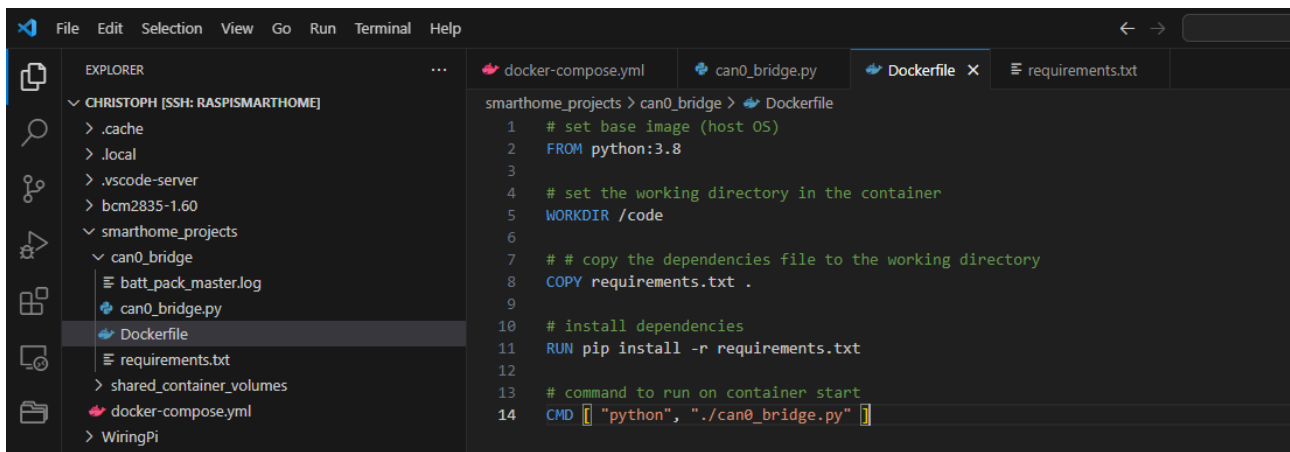
### 5. Einrichten von Python Skripten als Docker Container

Hier am Beispiel von meinem CAN0 Python Skript. Diese Anleitung ist aber auch relevant wenn man z.B. die Regelung vom Cerbo über das Python Skript was ich noch entwickeln muss starten möchte.

Für mein neues Python Skript lege ich einen eigenen Docker Container an. Wenn man nicht alle Skripte in einem Container hat, kann man für jedes Skript andere Python Paket Versionen installieren und ist insgesamt erheblich flexibler. Skripte die sich untereinander aufrufen müssen natürlich in einen Container.

Zuerst habe ich mir in meinem „smarthome\_projects“ Ordner einen Unterordner für den neuen Container mit dem Python Skript angelegt. In diesen Ordner muss das Pythonscript selbst („can0\_bridge.py“), das Dockerfile um den Container zu bauen (da wird nur Python installiert weil wir in dem Container ja auch nur ein einziges Python Skript ausführen wollen), die „requirements.txt“ Datei, in der alle Pakete stehen die in diesem Python zusätzlich installiert werden sollen zusammen mit den FIXEN (wichtig) Versionen. Es ist wichtig die Versionen zu fixieren, damit die Skripte auch in vielen Jahren noch so funktionieren. Wenn man immer die neuesten Pakete installiert, kann sich da so viel geändert haben, dass das Ursprungsskript nicht mehr lauffähig ist.

So sieht dann die neue Ordnerstruktur aus:

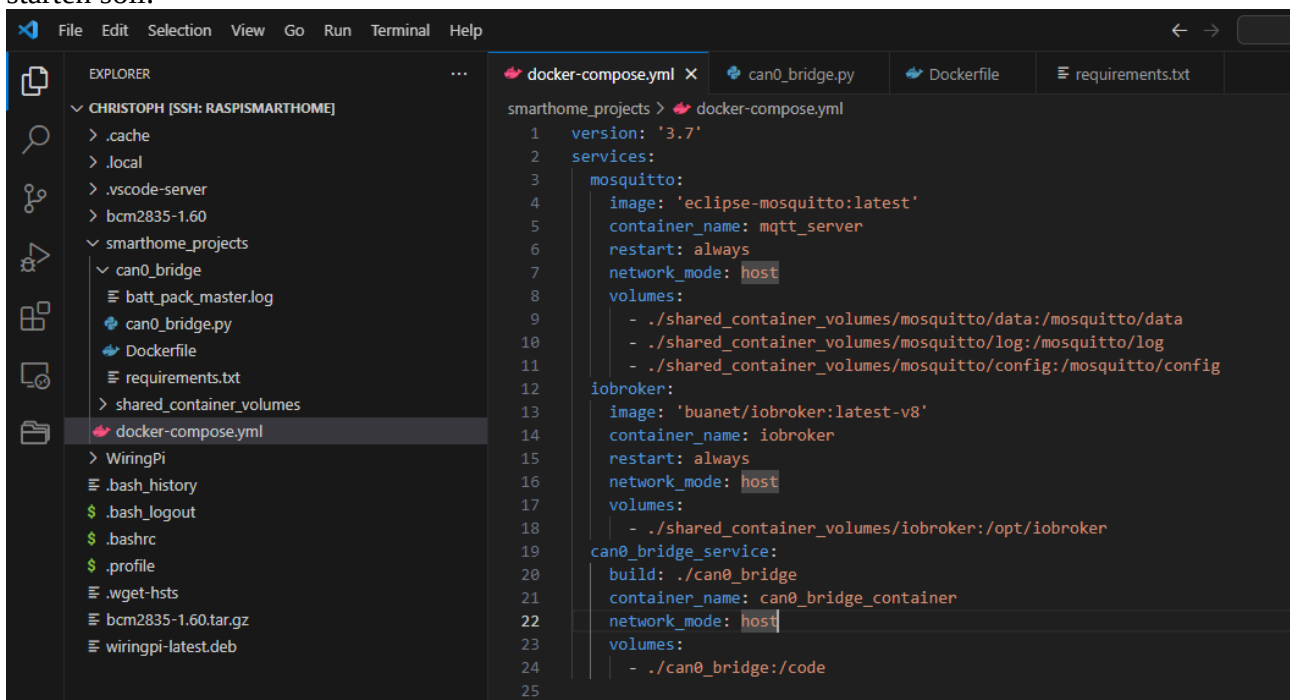


The screenshot shows the VS Code interface with the Explorer on the left and the Dockerfile editor in the center. The Explorer shows the project structure for 'CHRISTOPH [SSH: RASPISMARTHOME]' with folders like '.cache', '.local', '.vscode-server', 'bcm2835-1.60', 'smarthome\_projects', and 'WiringPi'. The 'smarthome\_projects' folder is expanded, showing 'can0\_bridge' with files 'batt\_pack\_master.log', 'can0\_bridge.py', 'Dockerfile', and 'requirements.txt'. The 'Dockerfile' is selected and its content is displayed in the editor. The Dockerfile content is as follows:

```
1 # set base image (host OS)
2 FROM python:3.8
3
4 # set the working directory in the container
5 WORKDIR /code
6
7 # # copy the dependencies file to the working directory
8 COPY requirements.txt .
9
10 # install dependencies
11 RUN pip install -r requirements.txt
12
13 # command to run on container start
14 CMD ["python", "./can0_bridge.py"]
```

Zusätzlich zu der Ordnerstruktur sieht man auch gleich schon den Inhalt vom Dockerfile (liegt zusammen mit dieser Doku ab): es wird ein Python 3.8 installiert, in dem Container ist der Hauptpfad „Code“ angelegt. Dann werden mit „pip install“ alle benötigten Pakete installiert. Zuletzt wird noch der Befehl genannt, was beim Start des Containers erfolgen soll, nämlich die Datei „can0\_bridge.py“ ausführen.

Jetzt muss noch die „docker\_compose.yml“ angepasst werden, die den Container erzeugen und starten soll:



The screenshot shows the VS Code interface with the Explorer on the left and the docker-compose.yml editor in the center. The Explorer shows the same project structure as before, but now 'docker-compose.yml' is selected in the 'smarthome\_projects' folder. The 'docker-compose.yml' content is displayed in the editor. The docker-compose.yml content is as follows:

```
1 version: '3.7'
2 services:
3   mosquitto:
4     image: 'eclipse-mosquitto:latest'
5     container_name: mqtt_server
6     restart: always
7     network_mode: host
8     volumes:
9       - ./shared_container_volumes/mosquitto/data:/mosquitto/data
10      - ./shared_container_volumes/mosquitto/log:/mosquitto/log
11      - ./shared_container_volumes/mosquitto/config:/mosquitto/config
12   iobroker:
13     image: 'buanet/iobroker:latest-v8'
14     container_name: iobroker
15     restart: always
16     network_mode: host
17     volumes:
18       - ./shared_container_volumes/iobroker:/opt/iobroker
19   can0_bridge_service:
20     build: ./can0_bridge
21     container_name: can0_bridge_container
22     network_mode: host
23     volumes:
24       - ./can0_bridge:/code
```

Der untere Teil ist neu dazu gekommen. Ich hab den Service einfach „can0\_bridge\_service“ genannt und den Container „can0\_bridge\_container“. Entscheidend sind die beiden Zeilen „build“ und „volumes“. Bei „build“ gibt man an, wo das Dockerfile liegt, für den ein Container angelegt werden soll. Die „docker-compose.yml“ Datei liegt im Hauptordner „smarthome\_projects“. Das ist der Hauptpfad für Docker Compose. Der Hauptpfad ist in Linux „./“ - also der Punkt. Relativ zu diesem „./“, also dem Hauptpfad, soll jetzt ein Ordner „can0\_bridge“ existieren, indem Docker Compose das Dockerfile erwartet um den „can0\_bridge\_service“ zu bauen. Für weitere Skripte bräuchte man dann entsprechend wieder neue Unterordner mit eigenem „Dockerfile“ und „requirements.txt“.

Mit „volumes: ./can0\_bridge:/code“ sorgt man wieder für eine Verbindung zwischen dem



Dockercontainer und Ordern auf die wir aus dem Raspberry Betriebssystem zugreifen können. Der im Raspberry Betriebssystem vorhandene Unterordner „./can0\_bridge“ (in dem ja unser can0\_bridge.py Skript liegt) wird auf den „code“ Ordner im Container „gemappt“ – also sozusagen „verbunden“. Den „code“ Ordner hatten wir ja schon im Dockerfile als „Workdir“ – also Hauptverzeichnis angegeben. Somit kann man jetzt aus dem neuen „can0\_bridge\_container“ Container auf alle Dateien in dem „./can0\_bridge“ Unterordner zugreifen. Nice to know: ich lege in meinem Skript Logdateien ab, die die gesamte CAN0 Kommunikation und den Skriptablauf dokumentieren/loggen können. Die kann ich im Container einfach in das Hauptverzeichnis schreiben und die tauchen dann natürlich auch automatisch im „gemappten“/„verbundenen“ Ordner ausserhalb des Containers auf.

Jetzt ist alles vorbereitet und wir erzeugen jetzt den Neuen (und die alten, die aber nicht neugestartet werden weil sich bei denen nichts geändert hat) Container mit dem bekannten Befehl: `sudo docker-compose up -d`

Um zu überprüfen ob der neue Container läuft wieder

`sudo docker compose ps`

ausführen. Hier findet man jetzt den Container den man angelegt hat:

NAME	IMAGE	COMMAND	SERVICE	CREATED	STATUS	PORTS
can0_bridge_container	smarthome_projects_can0_bridge_service	python ./can0_bridge.py	can0_bridge_service	3 minutes ago	Up 3 minutes	
buanet	buanet/iobroker:latest-v8	"/bin/bash -c /opt/scripts/iobroker_startup.sh"	iobroker	41 hours ago	Up 29 hours (healthy)	
mqtt_server	eclipse-mosquitto:latest	"/docker-entrypoint.sh /usr/sbin/mosquitto -c /mosquitto/config/mosquitto.conf"	mosquitto	41 hours ago	Up 29 hours	

Wenn der Container nicht läuft ist beim Starten des Python Skripts ein Fehler aufgetreten. Um herauszufinden was schief gelaufen ist kann man folgenderweise vorgehen:

`sudo docker ps --filter "status=exited"`

Mit dem Befehl findet man alle beendeten Container und wann sie beendet wurden. Jetzt müssen wir aber noch wissen mit welcher Fehlermeldung der Container aussteigt. Dafür versuchen wir den Container jetzt neu zu starten mit zusätzlicher Ausgabe aller Fehlermeldungen die beim Ausführen auftreten:

`sudo docker start -i can0_bridge_container`

Damit erhalten wir jetzt z.B. folgende Meldung:

```
christoph@raspismarthome:~/smarthome_projects $ sudo docker start -i can0_bridge_container
Traceback (most recent call last):
  File "./can0_bridge.py", line 12, in <module>
    import numpy as np
ModuleNotFoundError: No module named 'numpy'
```

In Zeile 12 unseres Pythonskripts trat ein Fehler auf. Und zwar scheint Numpy nicht installiert zu sein. Schnell noch Numpy in die requirements.txt aufgenommen (siehe „Was tun wenn man ein neues Paket in der „requirements.txt“ einfügt?“) und schon läuft die Kiste.

**Was tun wenn man ein neues Paket in der „requirements.txt“ einfügt?**

**ALTERNATIVE evtl. besser – kommt als nächstes Unterkapitel**

Wenn man dann wieder Docker compose aufruft, wird einfach das alte Docker Image verwendet und das neue Paket nicht installiert. Um das zu erreichen muss man sich erstmal eine Liste mit allen Images zeigen lassen:

`sudo docker image ls`

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
smarthome_projects_can0_bridge_service	latest	d59b6ecb01e1	27 minutes ago	787MB
buanet/iobroker	latest-v8	c42b06d273ac	2 days ago	1.04GB
eclipse-mosquitto	latest	4c61c357c3df	5 days ago	11.3MB
python	3.8	7d9b624d5c89	4 weeks ago	775MB
hello-world	latest	38d49488e3b0	4 months ago	4.85kB

Dann entfernt man das Image welches man geändert hat. Ich kopier mir dafür die IMAGE ID und

nehme folgenden Befehl (jetzt konkret will ich den bridge\_service neu bauen):

```
sudo docker rmi d59b6ecb01e1 --force
```

```
christoph@raspismarthome:~/smarthome_projects $ sudo docker rmi d59b6ecb01e1 --force
Untagged: smarthome_projects_can0_bridge_service:latest
Deleted: sha256:d59b6ecb01e1f006d627446000eb138ad55bbc522b0c24eb8e7d37e4460ba331
Deleted: sha256:10e638f0bd6fcd84cb82bc01d42ed4881dcc1978e78b78b920517152d3aa2ed7
Deleted: sha256:f0684598d5e4d952f460ca4aec6ecf138f908ecfa7eefe917456c9064c7d44f7
Deleted: sha256:7b13af7ce6c6625e15c1da70de1124a23850c4fde3dd815c35531d0fd0bae351
```

Jetzt kann man wieder mit

```
sudo docker-compose up -d
```

alles starten und das Image sollte neu erstellt werden. Es kommt eine Fehlermeldung dass das erwartete Image nicht vorhanden ist, allerdings ist das ja normal, da wir mit „--force“ das alte Image gekillt haben. Mit „y“ bestätigen wir, dass das neue Image verwendet werden soll.

**Was tun wContainer neu erstellen wenn sich etwas geändert hat (requirements.txt oder .py Datei)**

```
docker compose ps
```

```
christoph@raspismarthome:~/smarthome_projects $ docker compose ps
```

NAME	IMAGE	COMMAND	SERVICE	CREATED	STATUS	PORT
can0_bridge_container	smarthome_projects_can0_bridge_service	"python ./can0_bridge.py"	can0_bridge_service	7 weeks ago	Up 5 days	
ess_controller_container	smarthome_projects_ess_controller_service	"python ./ess_controller.py"	ess_controller_service	7 weeks ago	Up 11 minutes	
iobroker	buonet/iobroker:latest-v8	"/bin/bash -c /opt/scripts/iobroker_startup.sh"	iobroker	7 weeks ago	Up 5 days (healthy)	
mqtt_server	eclipse-mosquitto:latest	"/docker-entrypoint.sh /usr/sbin/mosquitto -c /mosquitto/config/mosquitto.conf"	mosquitto	7 weeks ago	Up 5 days	

danach dann den Service den man neu bauen möchte stoppen:

```
docker-compose stop ess_controller_service
```

```
christoph@raspismarthome:~/smarthome_projects $ docker-compose stop ess_controller_service
Stopping ess_controller_container ... done
```

Dann den alten Service entfernen

```
docker-compose rm -f ess_controller_service
```

```
christoph@raspismarthome:~/smarthome_projects $ docker-compose rm -f ess_controller_service
Going to remove ess_controller_container
Removing ess_controller_container ... done
```

Dann den neuen Service bauen

```
docker-compose build ess_controller_service
```

```
christoph@raspismarthome:~/smarthome_projects $ docker-compose build ess_controller_service
Building ess_controller_service
Sending build context to Docker daemon 2.552MB
Step 1/5 : FROM python:3.8
--> 7d9b624d5c89
Step 2/5 : WORKDIR /code
--> Using cache
--> 98cd8adfe7c6
Step 3/5 : COPY requirements.txt .
--> Using cache
--> 23f418ef420b
Step 4/5 : RUN pip install -r requirements.txt
--> Using cache
--> 6da253d6de4c
Step 5/5 : CMD [ "python", "./ess_controller.py" ]
--> Using cache
--> 8920263f0d01
Successfully built 8920263f0d01
Successfully tagged smarthome_projects_ess_controller_service:latest
```

Und dann neustarten:

```
docker-compose up -d ess_controller_service
```

```
christoph@raspismarthome:~/smarthome_projects $ docker-compose up -d ess_controller_service
Creating ess_controller_container ... done
christoph@raspismarthome:~/smarthome_projects $ docker compose ps
```

## 6. Python Skript entwickeln im Docker Container

[https://code.visualstudio.com/docs/devcontainers/containers#\\_open-a-folder-on-a-remote-ssh-host-in-a-container](https://code.visualstudio.com/docs/devcontainers/containers#_open-a-folder-on-a-remote-ssh-host-in-a-container)

<https://code.visualstudio.com/docs/devcontainers/containers>

Für die Erlaubnis der Verbindung in den Containern diese Anleitung nutzen:

<https://docs.docker.com/engine/install/linux-postinstall/>

Aus der Doku:

1. Create the docker group.

```
$ sudo groupadd docker
```

2. Add your user to the docker group.

```
$ sudo usermod -aG docker $USER
```

3. Log out and log back in so that your group membership is re-evaluated.

You can also run the following command to activate the changes to groups:

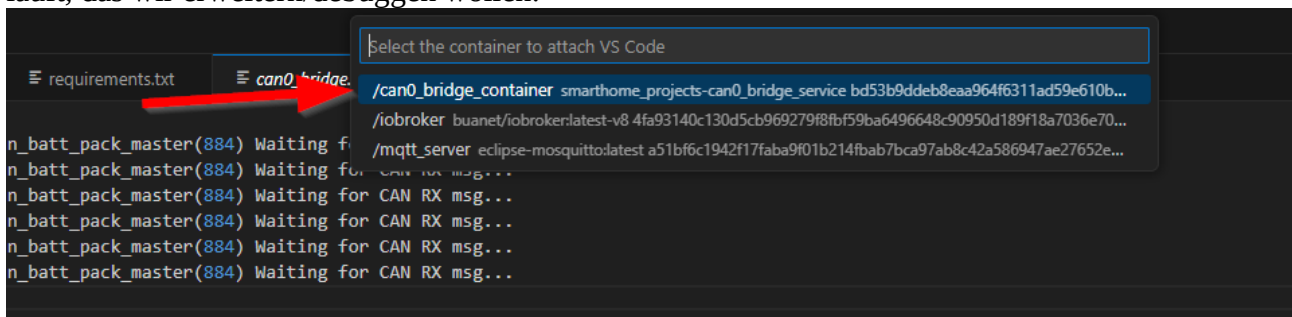
```
$ newgrp docker
```

4. Verify that you can run docker commands without sudo.

```
$ docker run hello-world
```

This command downloads a test image and runs it in a container. When the container runs, it prints a message and exits.

Nachdem man erfolgreich die Dockergruppe erzeugt hat, kann man sich jetzt im VSCode mit dem Container verbinden. Dazu F1 drücken und in der aufpoppenden Zeile „Dev containers: Attach“ eintippen bzw. aus der Liste raussuchen. Dann muss man nochmal sein Passwort eingeben und sieht dann die laufenden Container. Dort wählen wir jetzt den Container aus in dem das Python Skript läuft, das wir erweitern/debuggen wollen:



Es geht ein neues VSCode Fenster auf in dem man wieder sein Passwort eintippen muss. Danach wird im Container alles für die Entwicklung mit VSCode vorbereitet.

### X. Einrichten vom CAN Interface (optional)

Wenn man Zuhause ein günstiges Bussystem verwenden möchte könnte CAN das Mittel der Wahl sein. Super robust, gute Datenraten und erheblich günstiger als KNX Technik. Ich benutz das für

die Steuerung meiner Batterien (Lüfter und Heizung) und um Messdaten zu übertragen. Wenn man sowas machen möchte und das nicht so zuverlässig sein muss, kann man besser WLAN in einem ESP8686 verwenden und die Daten gleich über MQTT rausschicken. Sowas hat man sich super schnell zurecht programmiert und muss keine Kabel ziehen.

2-CH CAN HAT von Waveshare habe ich verwendet um ihn auf den Raspberry zu schnallen. Anleitung zur Einrichtung liegt ab (2-CH CAN HAT - Waveshare Wiki) bzw. findet man online.

Damit der CAN auch dauerhaft verfügbar ist, an die Anleitung aus dem PDF halten was mit abgelegt ist ("Automatically bring up a SocketCAN interface on boot – PragmaticLinux"). Wenn man wie in diesem Fall 2 CAN Interfaces hat muss man wohl (noch nicht getestet) in der 80-can.network Datei statt "can0" einfach "can\*" schreiben. Damit gelten die Settings für alle CAN interfaces. Wie man den unterschiedlichen Interfaces unterschiedliche Settings verpasst ist mir nicht klar – vielleicht braucht man das aber auch nicht.

# Changelog

2024.01.22:  
initiale Version