



Ein Batterie Management System für netzparallele
Victron ESS Systeme

Inhaltsverzeichnis

1. Installation von Raspberry OS.....	3
Installation von Raspberry OS.....	3
2. Einrichten von VSCode Remote.....	4
3. Einrichten vom Docker Container für IOBroker und Mosquitto.....	7
4. Starten der Docker Container.....	12
5. Einrichten von iobroker.....	14
6. Python Skript entwickeln im Docker Container.....	18
Changelog.....	20

In dieser Anleitung geht es darum einen Raspberry so einzurichten, das IOBroker, ein MQTT Server und CAN (optional) für Smarthomeanwendungen drauf laufen.

Voraussetzung:

- 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

Installation von Raspberry OS

Installation von Raspberry 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 Sachen einstellen:

- Hostname: Gib einen individuellen Namen – meiner heißt nach seiner Aufgabe „raspismarthome“

- SSH: unbedingt aktivieren – damit richten wir das Ding später ein
- 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 (auch ruhig ein vernünftiges Passwort wählen, weil über die Kiste die gesamte Hausautomatisierung läuft (bzw. laufen kann))

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 PI – einfach mal bei Amazon die Kommentare lesen und dann NICHT bei Amazon bestellen ;-)

2. 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>

Danach VSCode starten.

Dann den roten 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). Wenn die Extension installiert ist draufklicken um die Beschreibung zu erhalten. Dort steht dann auch wie man sich zum Raspberry verbindet unter „Getting started“. Dieser Anleitung folgen – VORHER ABER DEN RASPI STARTEN (der Fehler ist mir natürlich nicht passiert). 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:

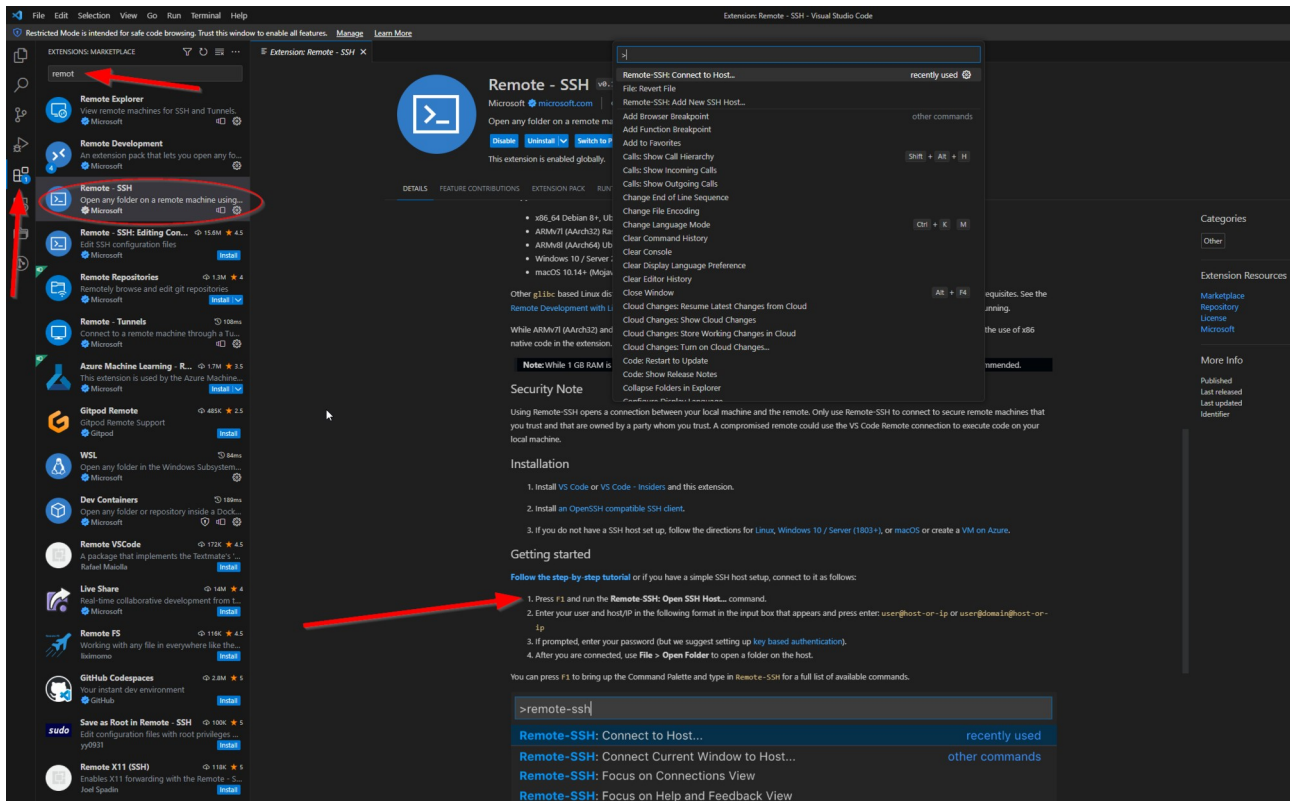


Dort dann „ssh [benutzername@hostname](#)“ eintippen. Bei mir sieht das so aus:

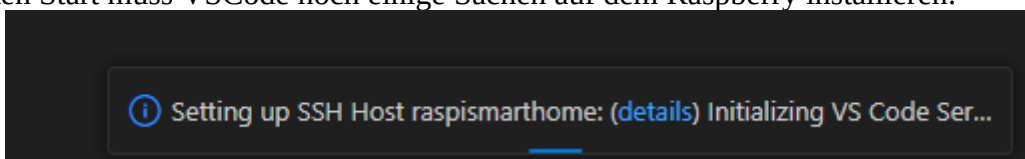
```
christoph@raspismarhome: ~  
Connection to raspismarhome closed.  
C:\Users\Christoph>ssh christoph@raspismarhome  
christoph@raspismarhome's password:  
Linux raspismarhome 6.1.21-v8+ #1642 SMP PREEMPT Mon Apr 3 17:24:16 BST 2023 aarch64  
  
The programs included with the Debian GNU/Linux system 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.

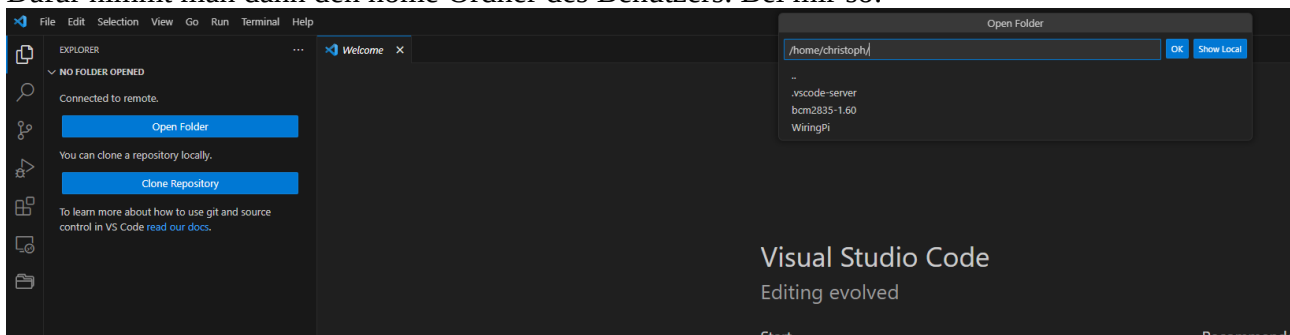
Wenn man dann den grünen Text vom eingeloggten Benutzer sieht, passt das mit der SSH Verbindung. Jetzt einfach das ganze in VSCode wiederholen: F1 drücken und dann dort auch „[benutzername@hostname](#)“ eintippen. Man muss sich dann noch connecten.



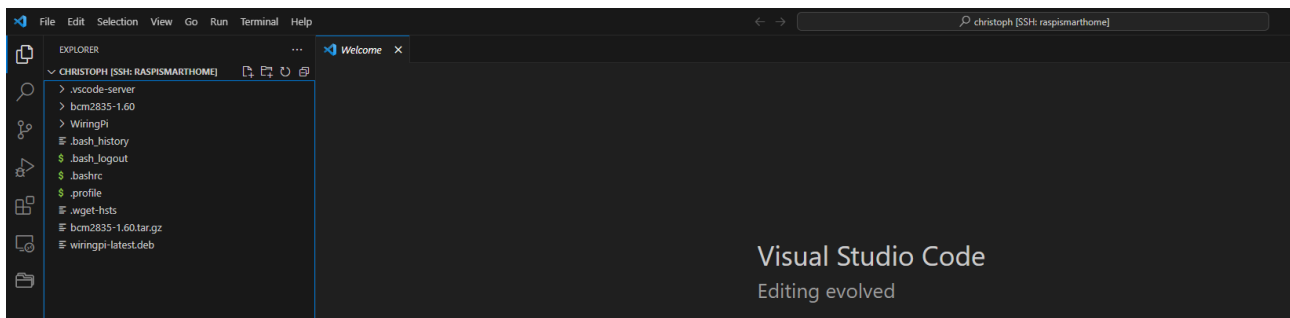
Beim ersten Start muss VSCode noch einige Sachen auf dem Raspberry installieren:



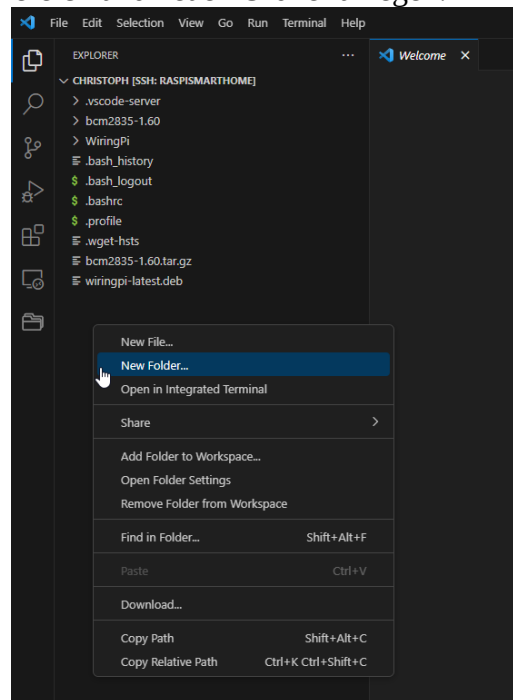
Wenn die Initialisierung abgeschlossen ist wird man gefragt welchen Ordner man öffnen möchte. Dafür nimmt man dann den home Ordner des Benutzers. Bei mir so:



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

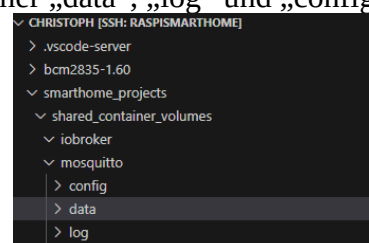


Für mein ganzes Smarthome Zeug habe ich den Unterordner „smarthome_projects“ angelegt. Dazu Rechtsklick in den Explorer Bereich und neuen Ordner anlegen:

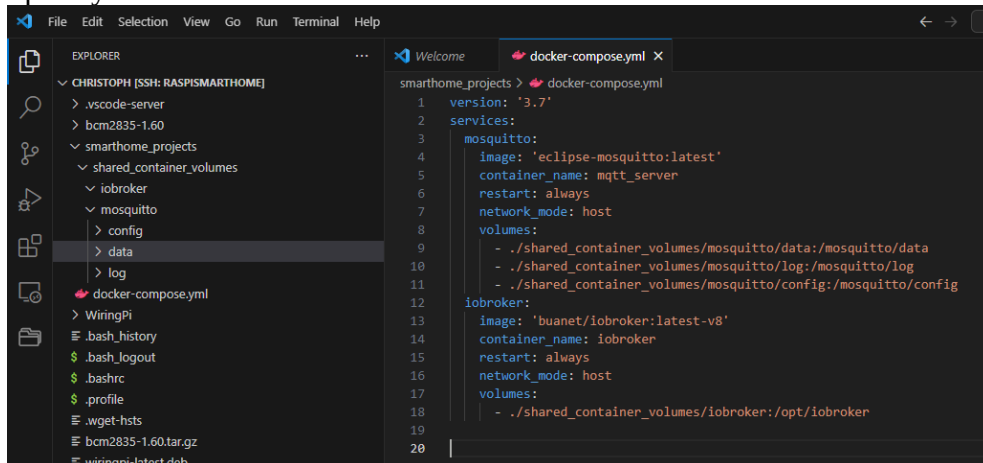


3. Einrichten vom Docker Container für IOBroker und Mosquitto

Die Nutzung von Docker Containern wird den meisten nicht sehr vertraut sein. Die haben aber den Vorteil, das man sehr unkompliziert Programme installieren und konfigurieren kann. Und zwar ziemlich viele und alles auf einen Schlag mit den sogenannten „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 Ordernamen 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:

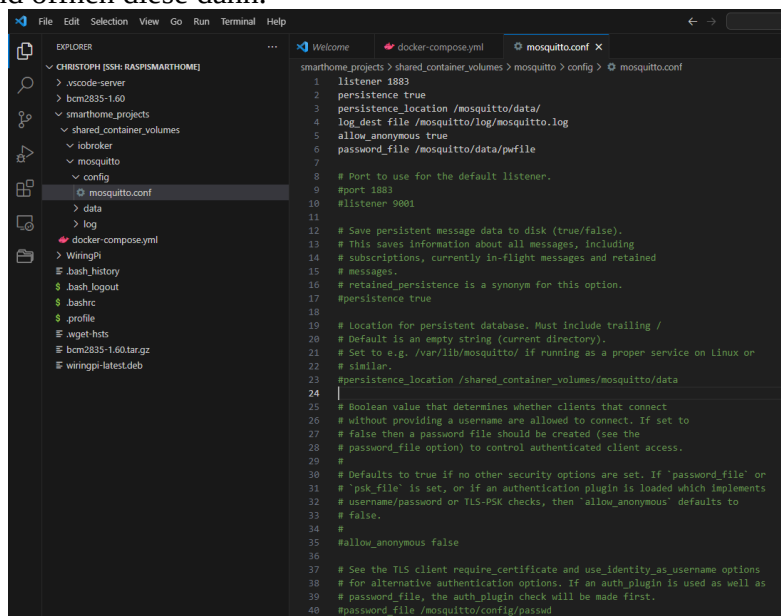


Danach kümmern wir uns um das „Compose“ File. Das liegt dieser Anleitung schon bei und heißt „docker-compose.yml“. Das kann man einfach per „Drag&Drop“ vom Desktop ins VSCode in den Ordner „smarthome_projects“ ziehen und schon landet es auf dem Raspi. Nach Doppelklick auf die „docker-compose.yml“ Datei sieht das Ganze so aus:



```
1 version: '3.7'
2 services:
3   mosquito:
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
```

Unser MQTT Server Mosquitto soll durch Passwortabfrage geschützt werden. Das richten wir als nächstes ein. Zuerst kopieren wir die beiliegende „mosquitto.conf“ Datei in den vorhin angelegten „config“ Ordner und öffnen diese dann.




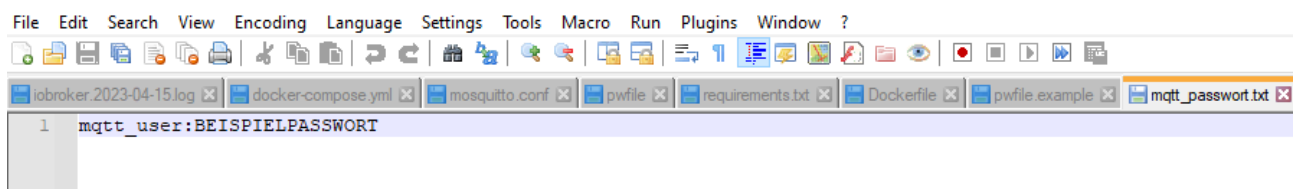
```
1 listener 1883
2 persistence true
3 persistence location /mosquitto/data/
4 log_dest file /mosquitto/log/mosquitto.log
5 allow_anonymous true
6 password_file /mosquitto/data/pwfile
7
8 # Port to use for the default listener.
9 #port 1883
10 #listener 9001
11
12 # Save persistent message data to disk (true/false).
13 # This saves information about all messages, including
14 # subscriptions, currently in-flight messages and retained
15 # messages.
16 # retained persistence is a synonym for this option.
17 #persistence true
18
19 # Location for persistent database. Must include trailing /
20 # Default is an empty string (current directory).
21 # Set to e.g. /var/lib/mosquitto/ if running as a proper service on Linux or
22 # similar.
23 #persistence_location /shared_container_volumes/mosquitto/data
24 |
25 # Boolean value that determines whether clients that connect
26 # without providing a username are allowed to connect. If set to
27 # false then a password file should be created (see the
28 # password_file option) to control authenticated client access.
29 #
30 # Defaults to true if no other security options are set. If 'password_file' or
31 # 'psk_file' is set, or if an authentication plugin is loaded which implements
32 # username/password or TLS-PSK checks, then 'allow_anonymous' defaults to
33 # false.
34 #
35 #allow_anonymous false
36
37 # See the TLS client require_certificate and use_identity_as_username options
38 # for alternative authentication options. If an auth_plugin is used as well as
39 # password_file, the auth_plugin check will be made first.
40 #password_file /mosquitto/config/pwfile
```

Dort werden verschiedene Sachen konfiguriert. In der letzten Zeile sieht man, das wir ein password file verlangen. Dieses müssen wir jetzt als nächstes anlegen. Dafür müssen wir an dieser Stelle einen Umweg gehen und erst noch Mosquitto für Windows runterladen und installieren:

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

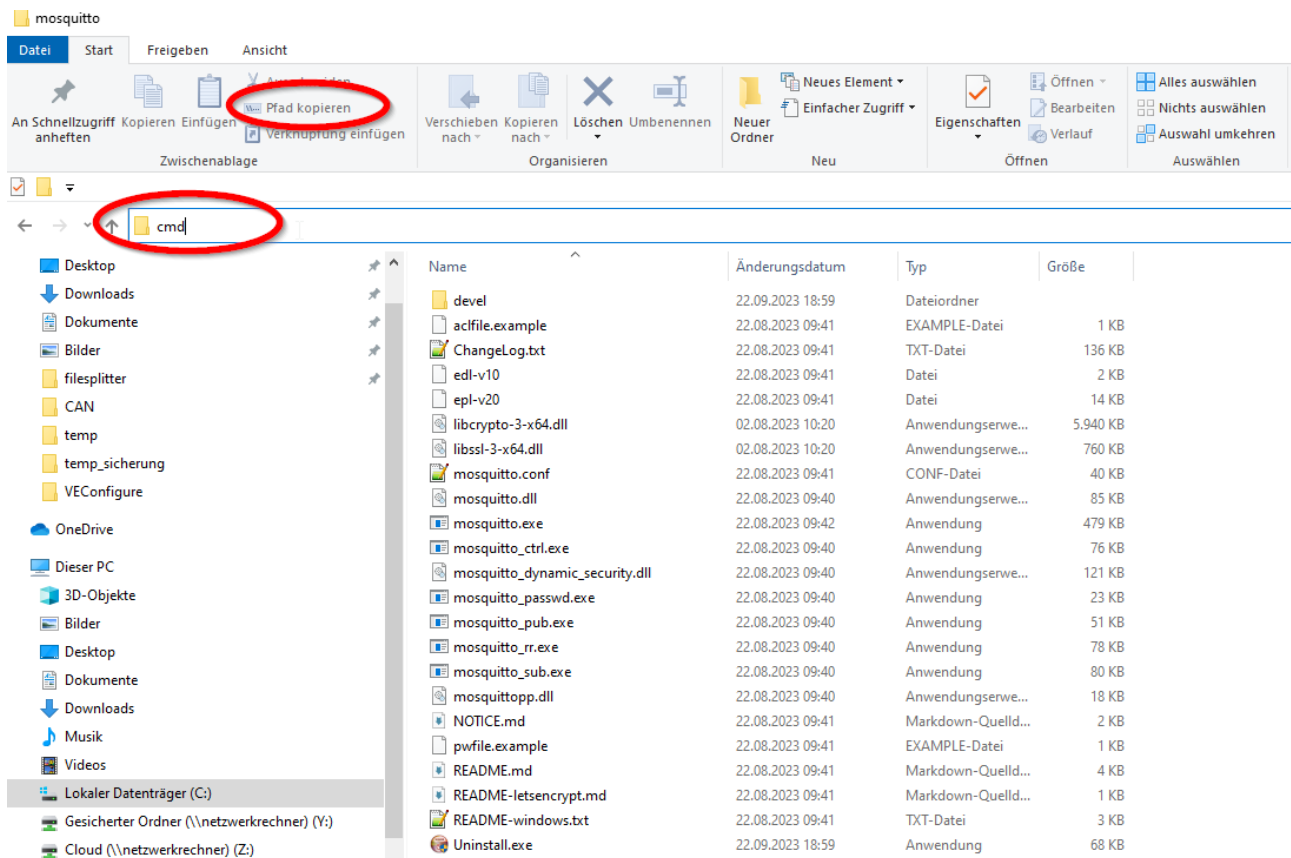
Danach legen wir z.B. auf dem Desktop eine Textdatei an, die die Logindaten enthält. Bei mir heißt die Datei „mqtt_passwort.txt“ und das Passwort muss so formatiert sein:

 C:\Users\Christoph\Desktop\mqtt_passwort.txt - Notepad++

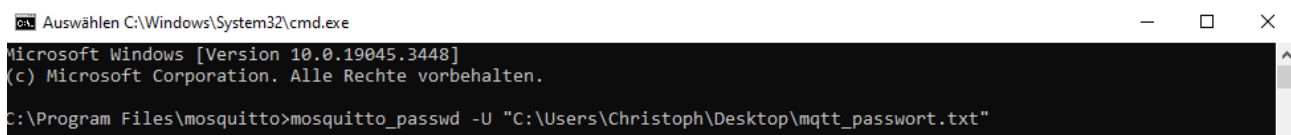


```
1 mqtt_user:BEISPIELPASSWORT
```


WICHTIG: Speichert euch euren Nutzernamen (könnt ihr wählen wie ihr wollt) 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:

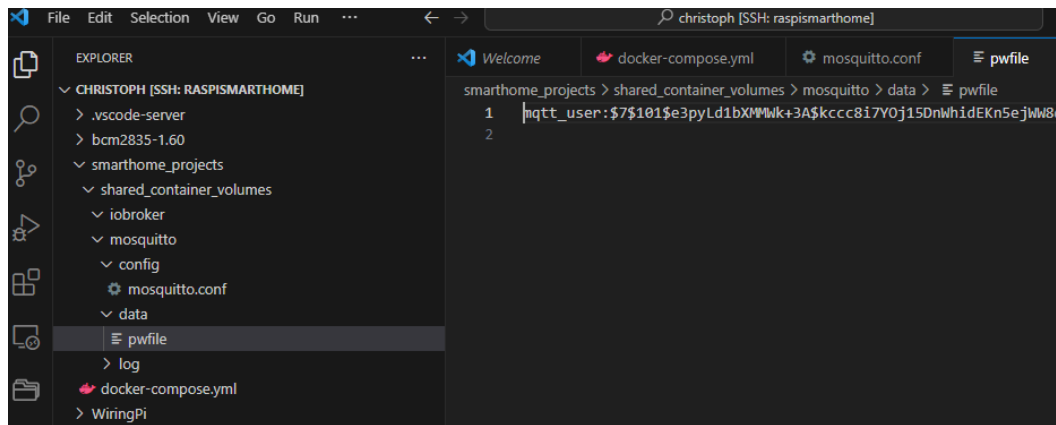


Dann in die Kommandozeile folgenden Befehl eingeben: `mosquitto_passwd -U PFADZUEURERPASSWORTDATEI`

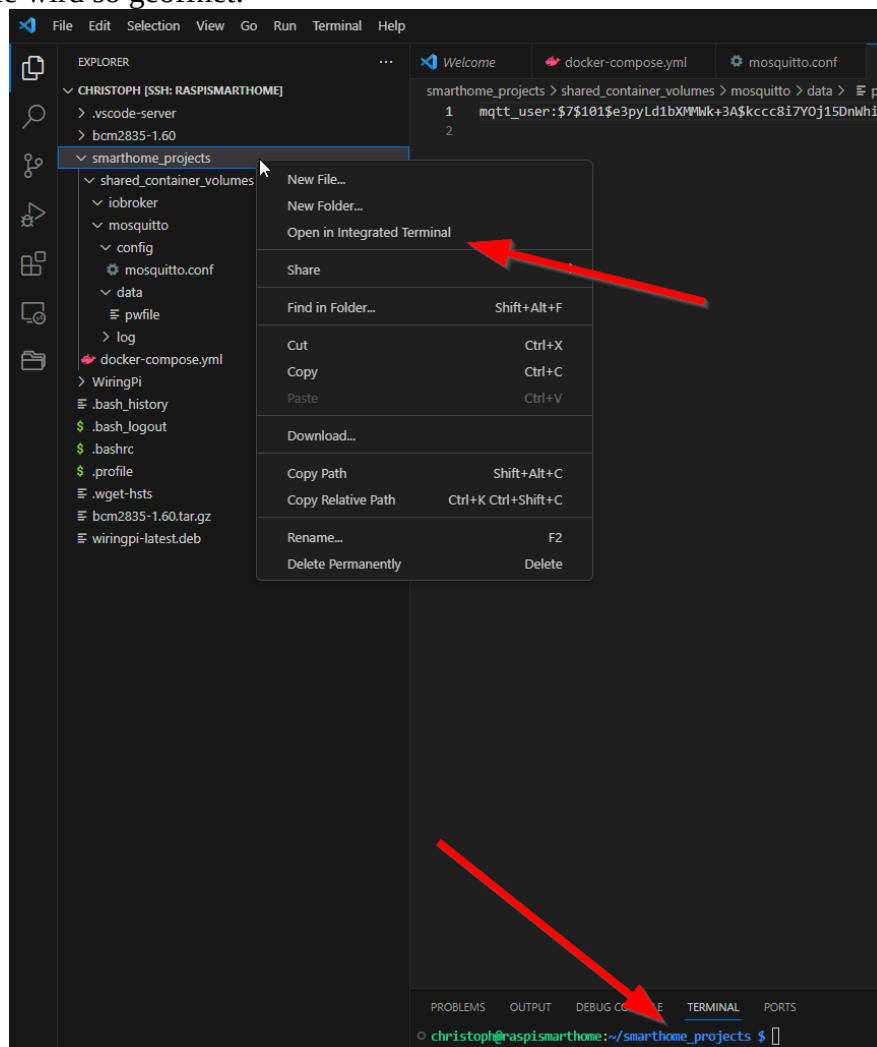


Wenn ihr die Passwortdatei jetzt öffnet werdet ihr feststellen, das euer altes Passwort jetzt komisch aussieht. Dann ist alles korrekt!

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 letzten Zeile genüge getan haben und eine „pwfile“ Datei in dem richtigen Ordner haben. So sieht es dann aus:



Als vorletzten Schritt müssen wir Docker noch 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:



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://raspbian.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" | tee /etc/apt/sources.list.d/docker.list && \
sudo apt-get update
deb [arch=armhf] 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://raspbian.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://raspbian.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://raspbian.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 wie in dieser Anleitung:

<https://bangertech.de/docker-docker-compose-auf-dem-raspberrypi/>

```
sudo apt-get install libffi-dev libssl-dev
sudo apt install python3-dev
sudo apt-get install -y python3 python3-pip
```

Einschub

ACHTUNG: Stand September 2023 hat sich was an den Abhängigkeiten von Docker Compose geändert und es muss noch der Rust Compiler installiert werden. Deswegen noch diesen Befehl zum Installieren von Rust ausführen:

```
curl https://sh.rustup.rs -sSf | sh
```

WICHTIG: Option 2 wählen und dann bei "Default host triple?" ohne Anführungszeichen "arm-unknown-linux-gnueabi" eingeben. Dann bei der Abfrage „Change PATH“ mit „y“ („yes“) antworten. Dann „pip install cryptography“ ausführen was ewig dauert und dann noch (ohne sudo):
pip3 install docker-compose

Hinweis: Ich denke das Problem wird in einigen Monaten behoben sein, deswegen würde ich diesen Einschub nur durchführen wenn die Installation mit einem Fehler abbricht.

#####

Einschub

Wenn die Installation von docker-compose erfolgreich war kann Rust wieder deinstalliert werden.

Gilt nur für den Einschub von oben:

```
rustup self uninstall
```

#####

Danach

```
sudo pip3 install docker-compose
```

HINWEIS: Vielleicht besser diesen Befehl zum Installieren verwenden. Der hat bei mir dann final funktioniert weil meine „ohne sudo“ Lösung dann keine Rechte hatte.:

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

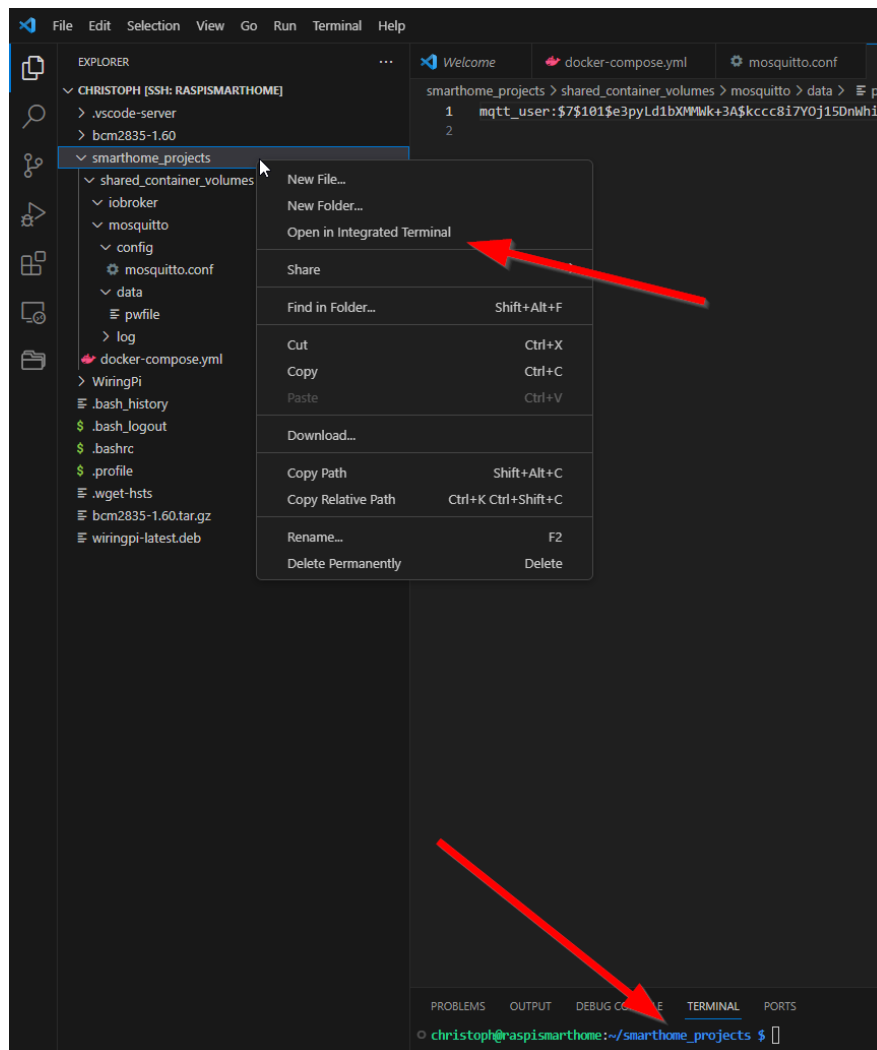
und danach alles für den automatischen Neustart konfigurieren:

```
sudo systemctl enable docker
```

Die Vorbereitung ist damit abgeschlossen. Jetzt gehts richtig los.

4. Starten der Docker Container

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:



Unten im Terminal geben wir jetzt folgenden Befehl ein:
`sudo docker-compose up -d`

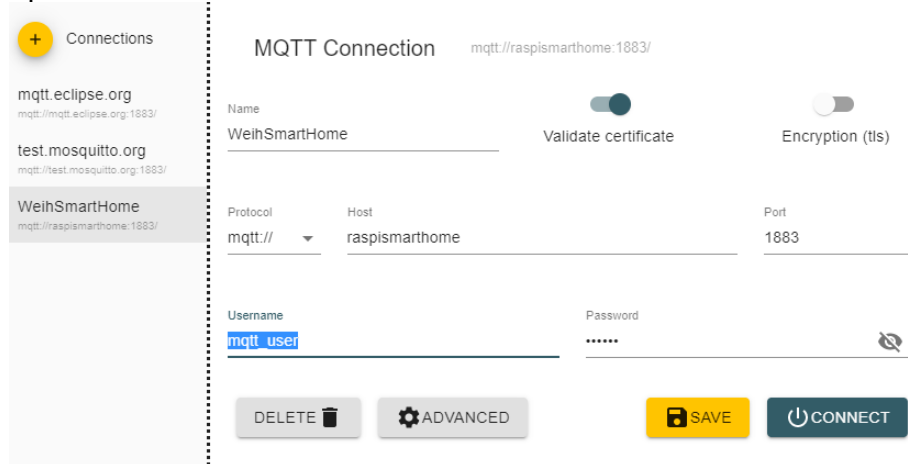
```
christoph@raspismarthome:~/smarthome_projects$ sudo docker-compose up -d
Pulling mosquitto (eclipse-mosquitto:latest)...
latest: Pulling from library/eclipse-mosquitto
af09961d4a43: Pull complete
42a0caf2b06a: Pull complete
00869a7064da: Pull complete
Digest: sha256:6e957b3dffe57afa895bed1e311db6f786eb93de70519df2dc81ef77ad6a21d3
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@raspismarthome:~/smarthome_projects$
```

Um zu checken ob alles läuft:
`sudo docker compose ps`

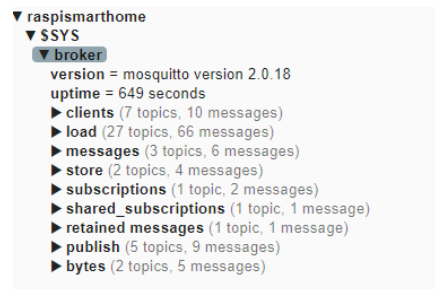
NAME	IMAGE	COMMAND	SERVICE	CREATED	STATUS	PORTS
iobroker	buanet/iobroker:latest-v8	"/bin/bash -c /opt/scripts/iobroker_startup.sh"	iobroker	5 minutes ago	Up 4 minutes (healthy)	
mqtt_server	eclipse-mosquitto:latest	"/docker-entrypoint.sh /usr/sbin/mosquitto -c /mosquitto/config/mosquitto.conf"	mosquitto	5 minutes ago	Up 4 minutes	

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. Das musste ich dort eintragen mit meinem MQTT Benutzernamen „mqtt_user“ und meinem Raspberry Hostnamen „raspismarhome“:



Der MQTT Server nennt sich Broker und meldet einige Standarddaten die man schon sehen kann. Später, wenn weitere Geräte mit dem Server verbunden sind, kann man hier auch deren Botschaften sehen.



5. Einrichten von iobroker

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

<http://raspismarhome: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://smarhome.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.

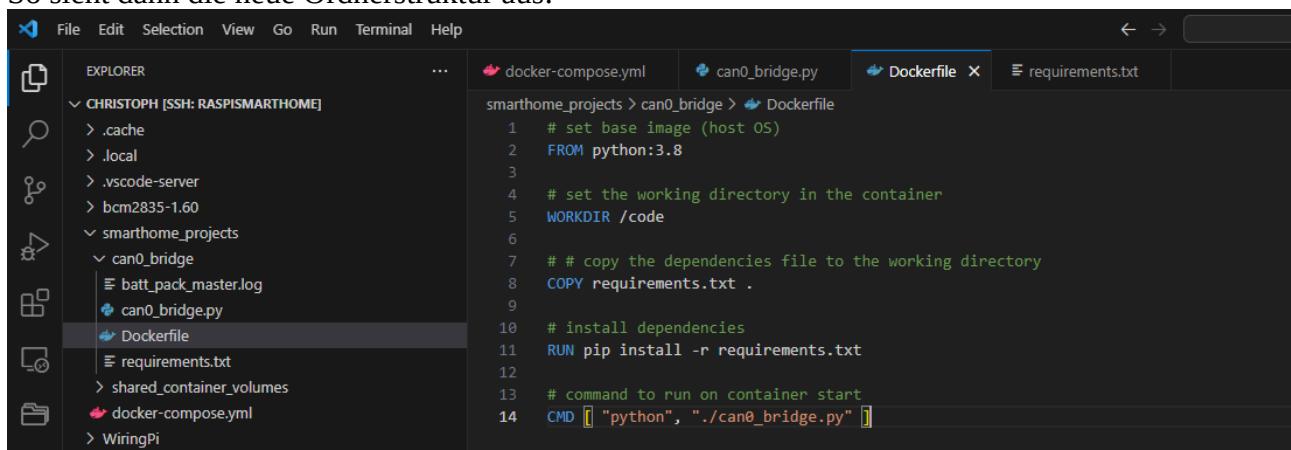
5. Einrichten von Python Skripten als Docker Container

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 soviel geändert haben, dass das Ursprungsskript nicht mehr lauffähig ist.

So sieht dann die neue Ordnerstruktur aus:



The screenshot shows a VS Code editor interface. On the left, the Explorer pane displays a file tree for a project named 'CHRISTOPH [SSH: RASPISMARTHOME]'. The tree structure is as follows:

- CHRISTOPH [SSH: RASPISMARTHOME]
 - .cache
 - .local
 - .vscode-server
 - bcm2835-1.60
 - smarthome_projects
 - can0_bridge
 - batt_pack_master.log
 - can0_bridge.py
 - Dockerfile
 - requirements.txt
 - shared_container_volumes
 - docker-compose.yml
 - WiringPi

The main editor area shows the content of the 'Dockerfile' file, which is located at 'smarthome_projects > can0_bridge > Dockerfile'. The file contains the following code:

```
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:


```

smarthome_projects > docker-compose.yml
1  version: '3.7'
2  services:
3    mosquito:
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
25

```

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 Ordnern 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	CMD	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	
iobroker	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
```

```
christoph@raspismarthome:~/smarthome_projects $ 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:d59b6ecb01e1f006d62744600eb138ad55bbc522b0c24eb8e7d37e4460ba331
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        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             buanet/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>

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:

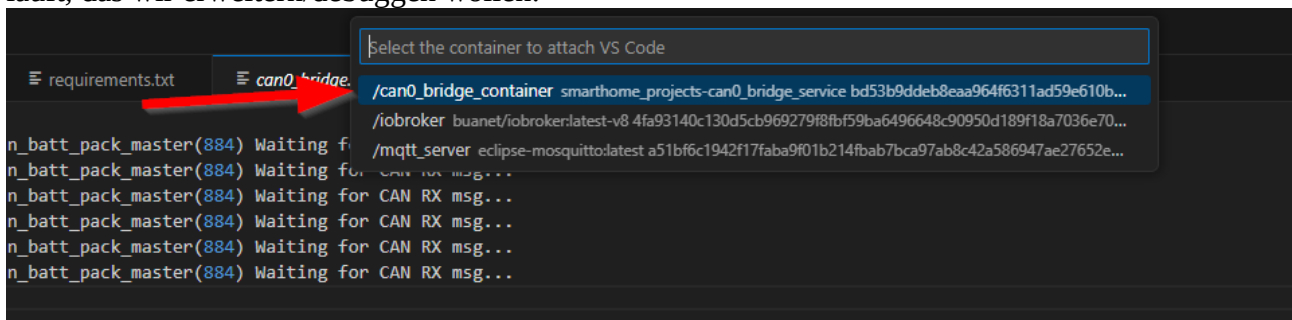
```
$ newgrp docker
```

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 aufpoppersenden 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