**Report Assignment 4**

Group Members

Reindl Hannes 01532129

Sammer Manuel 11903022

Platzer Andreas 11771913

**Task 1) Temperature Sensing**

Since we missed the deadline for the 3rd assignment, we used the tutorial posted in slack to implement the temperature sensing.

Link: <https://st-page.de/2018/01/20/tutorial-raspberry-pi-temperaturmessung-mit-ds18b20/>

Following the tutorial and the figure, the temperature sensor (DS18B20) was connected to pi depicted below:



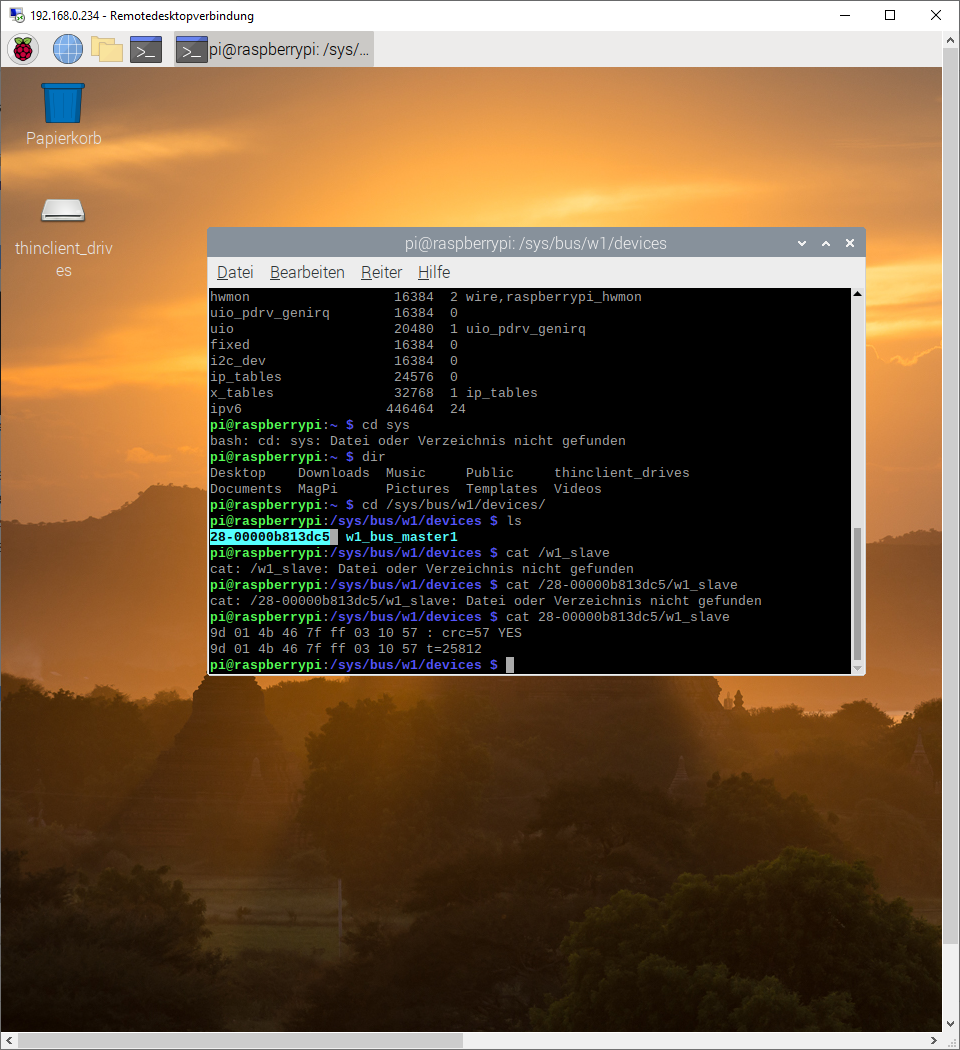
Ground

Data

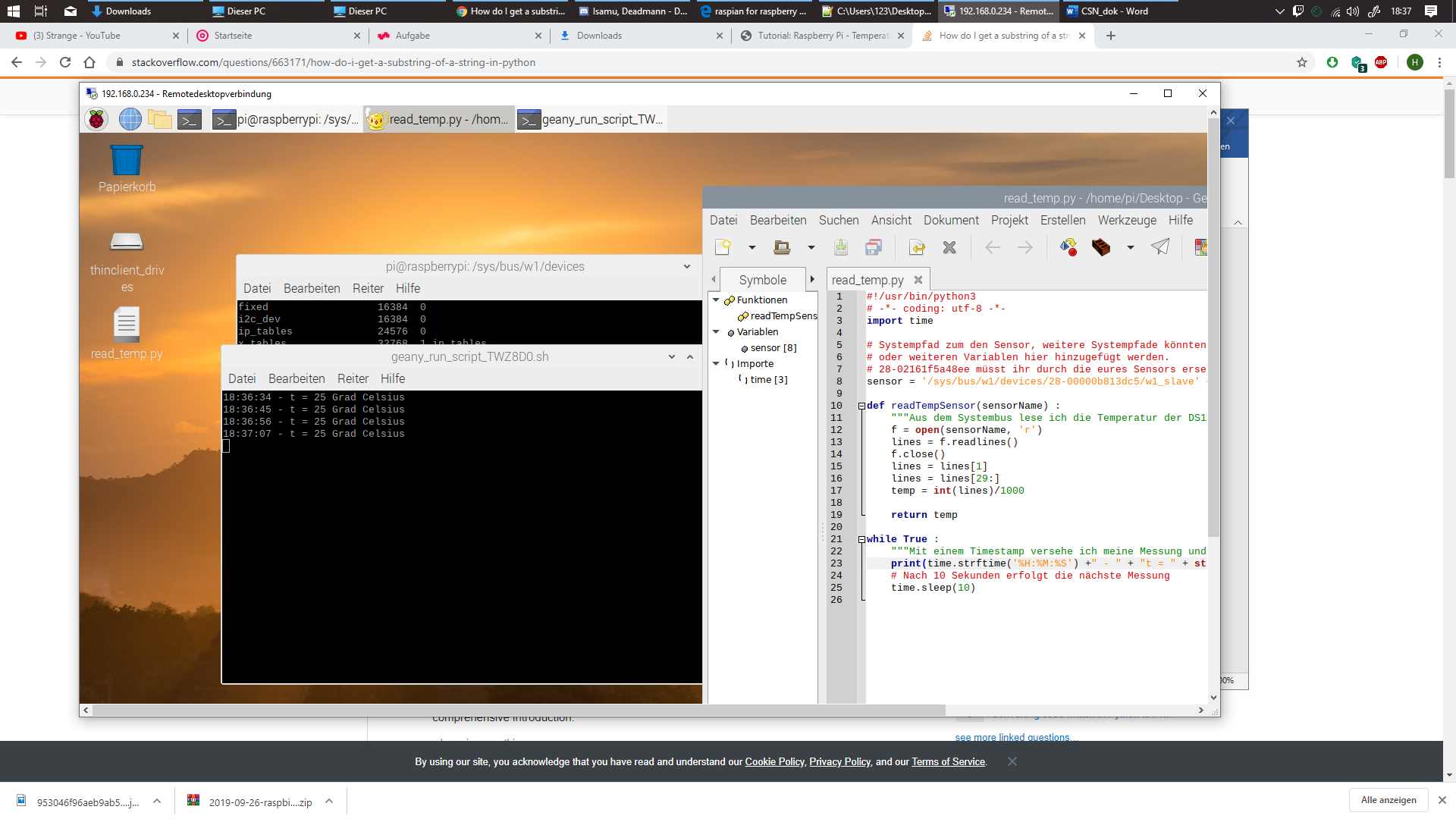
Vcc (3.3v)

After connecting the temperature sensor, the value can be read from terminal. First, we need to know the device ID. Opening the folder in the terminal via “cd /sys/bus/w1/devices/” the connected devices can be seen. Our device has the ID: 28-00000b813dc5

The command “cat 28-00000b813dc5/w1\_slave” read from the temperature sensor, which can be seen in the next picture:



Note that the temperature value is written down as a multiple of 1000. The current temperature is t = 25.812°C.

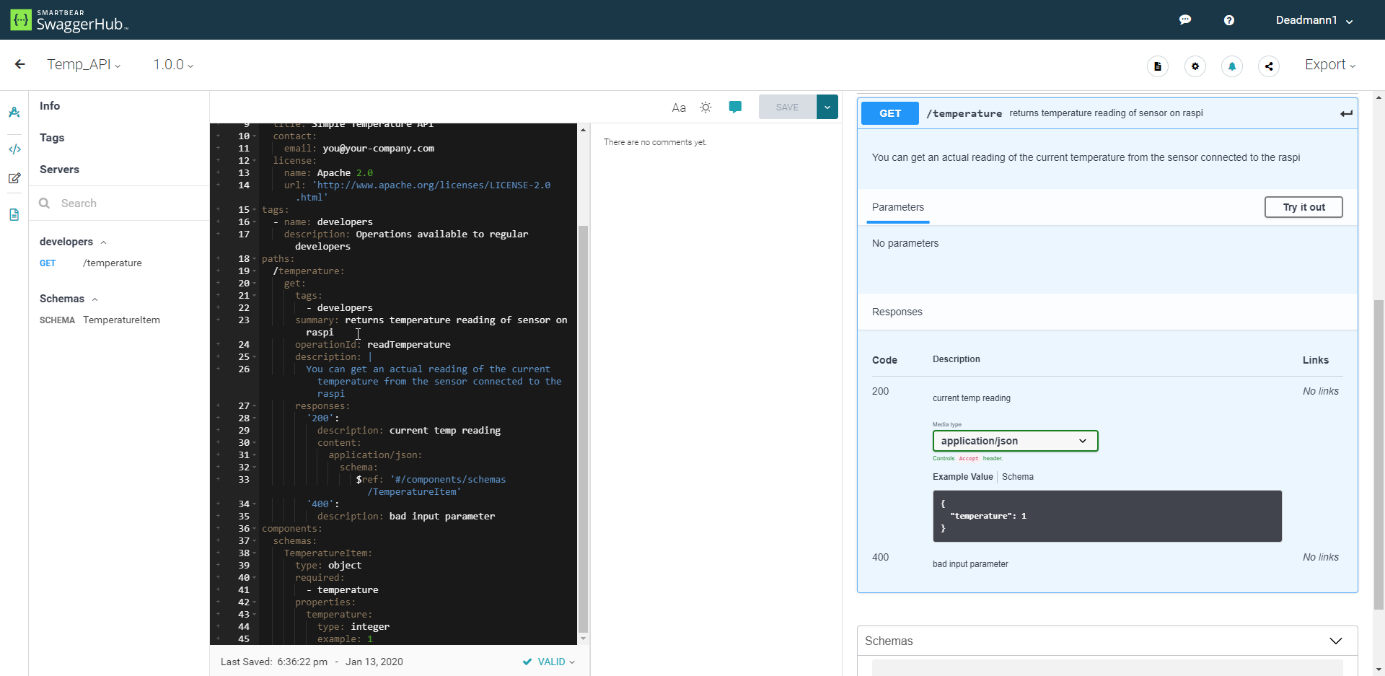
Now to automate the process of reading the temperature, a python script was written, which simple reads via the 1-wire-protocol and slices the given string to only contain the temperature value. The value gets polled every second.

Code file: Task\_1/read\_temp\_python.py

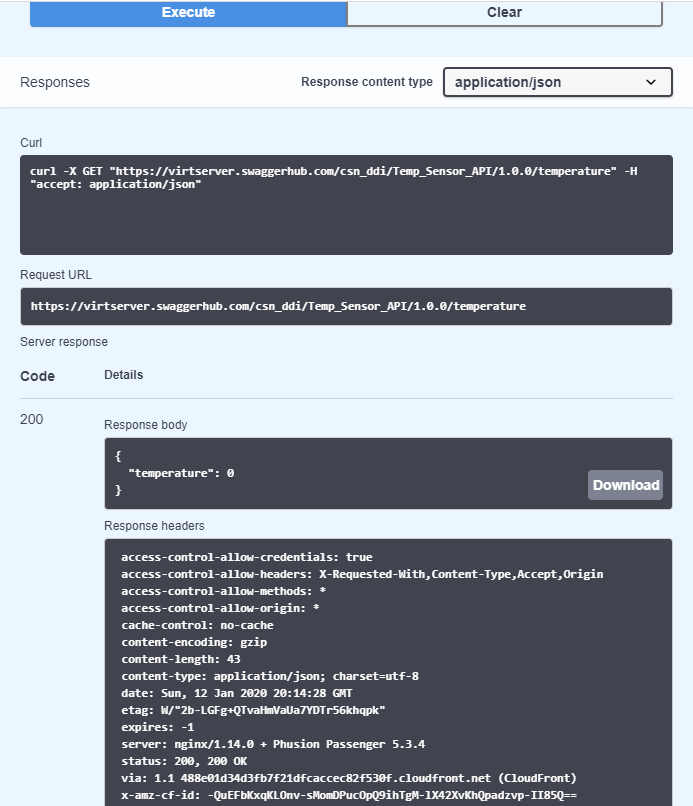
For testing purposes, the temperature was written to the terminal.

**Task 2) Creating an API**

In this task a simple web server should be created which allows to request the temperature via the HTTP protocol.

Going to the recommended website <https://swagger.io/>, and simply changing a given template to fit our purposes the API was created.

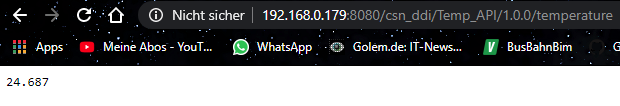
We used swagger with Open API 3.00 and used the given template “simple template”. Afterwards we deleted the things which were not needed and rewrote a model to our current temperature model. Then we added the path “temperature” which returns an object of the type temperature model.

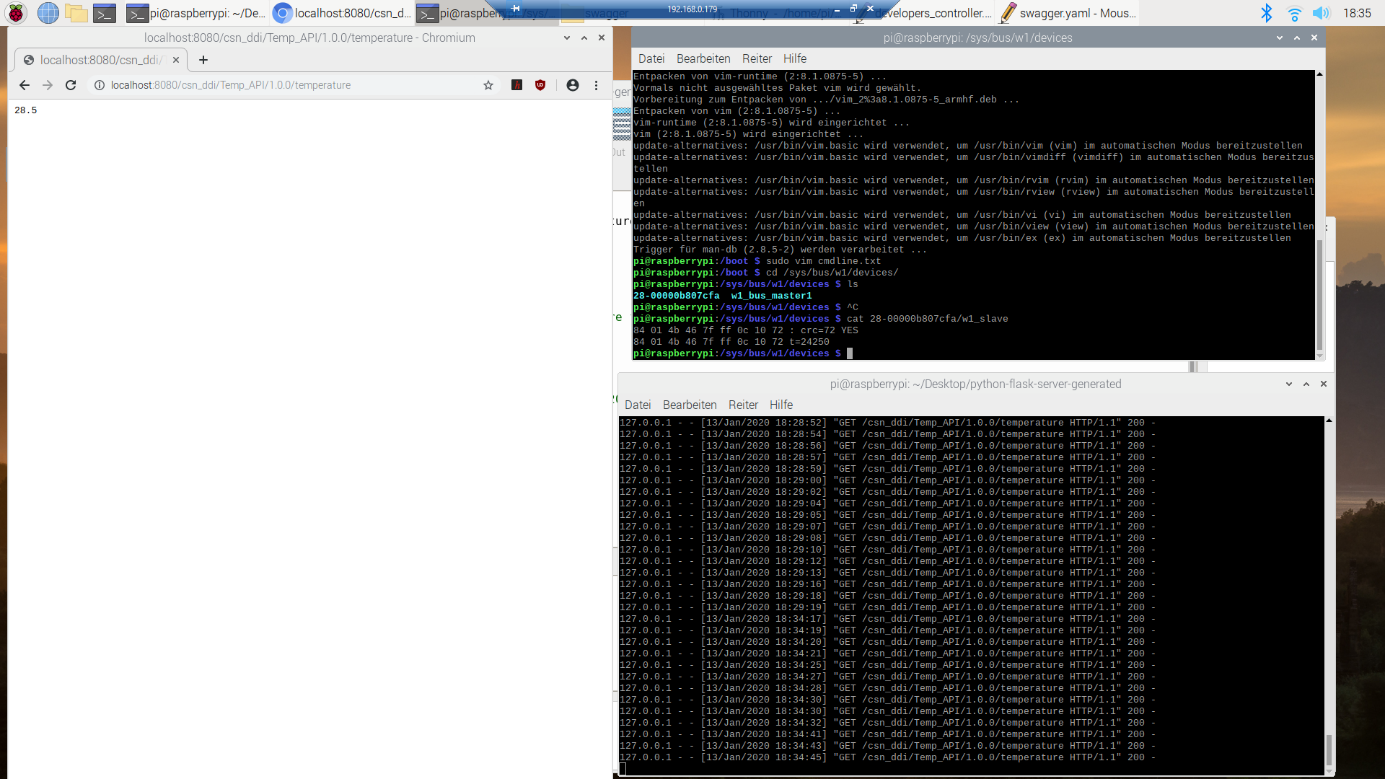
Using the websites build in simulation function, the API was tested.

The Export button allowed us to download a server stub for python-flask.

From now on, for testing purposes, we used a RPi 3, which was way faster than the RPi 1 and allowed complex tasks (such as moving a window) in much less time. Since this no Bare-Metal programming anymore and the same operating system is used by the two RPi’s, it should not make any difference.

Here we the HTTP Response in the browser. The server can also be seen which is currently working on the request.





This server was only tested in LAN, because port forwarding was not set up for public use.

**Task 3) CoAP Protocol**

First the plugin Cu4Cr was installed to enable google chrome sending coap packets and allowed us to debug the code. Link: <https://github.com/mkovatsc/Copper4Cr>

For the CoAP implementation a GitHub repository was used named aiocoap.   
Link: <https://github.com/chrysn/aiocoap>

Following the guided tour, the aiocoap package was downloaded onto the RPi using

pip3 install --upgrade "aiocoap[all]"

and starting the server from the root folder by

./server.py

The server.py file already has some defined resources, which can be found via the CoAP .well-known/core. Writing this command line into the terminal

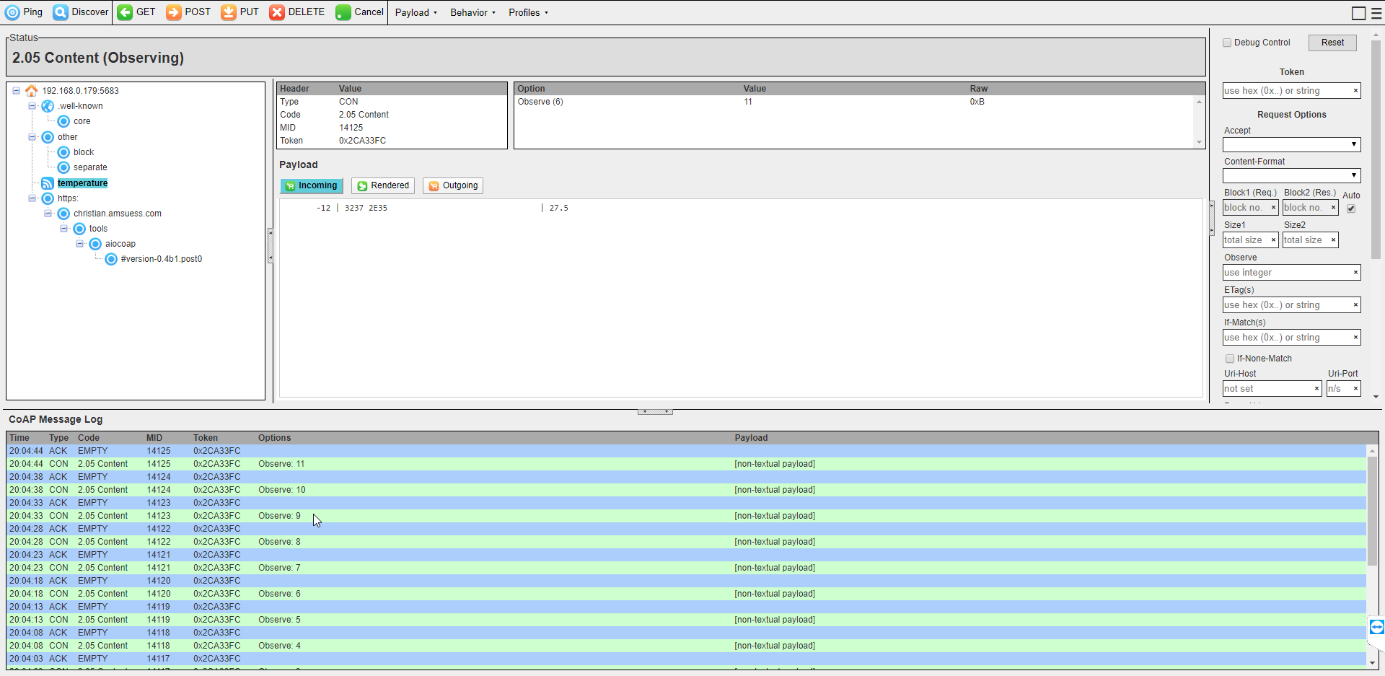
./aiocoap-client coap://localhost/.well-known/core

Shows the following resources:

</time>; obs, </.well-known/core>; ct=40, </other/separate>, </other/block>

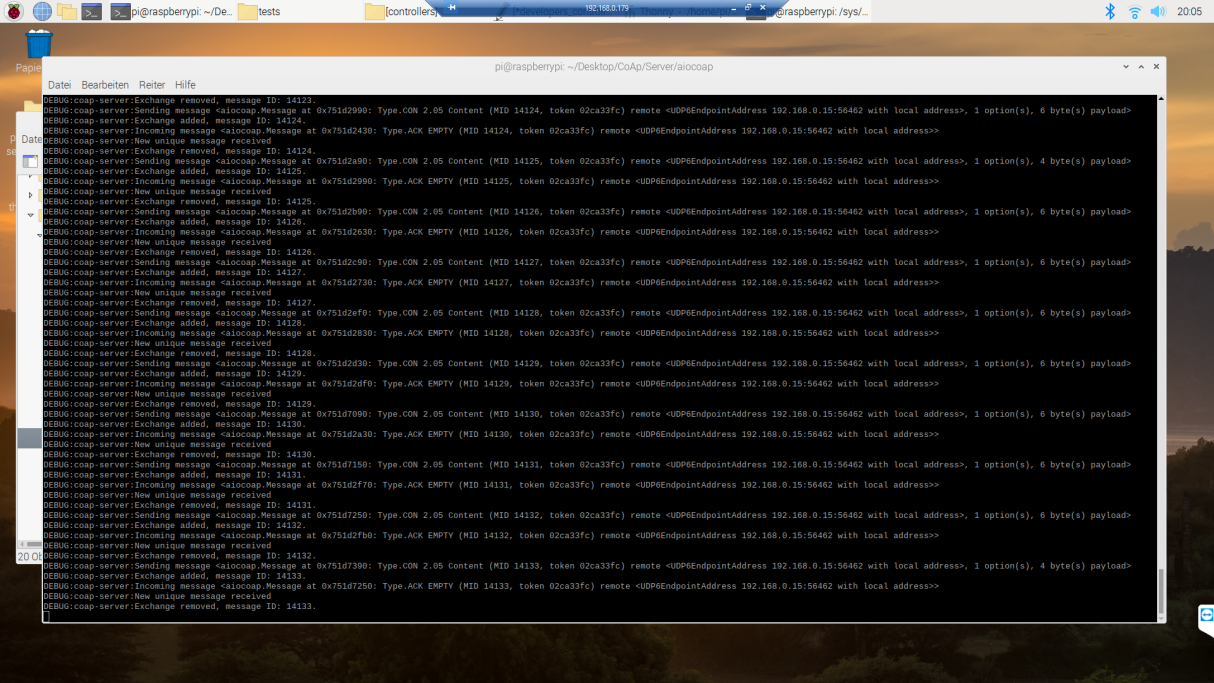
The resource time and the resource other, which has 2 sub resources. The assignment explicitly states to make sure the server can use the OBSERVER method and implement.

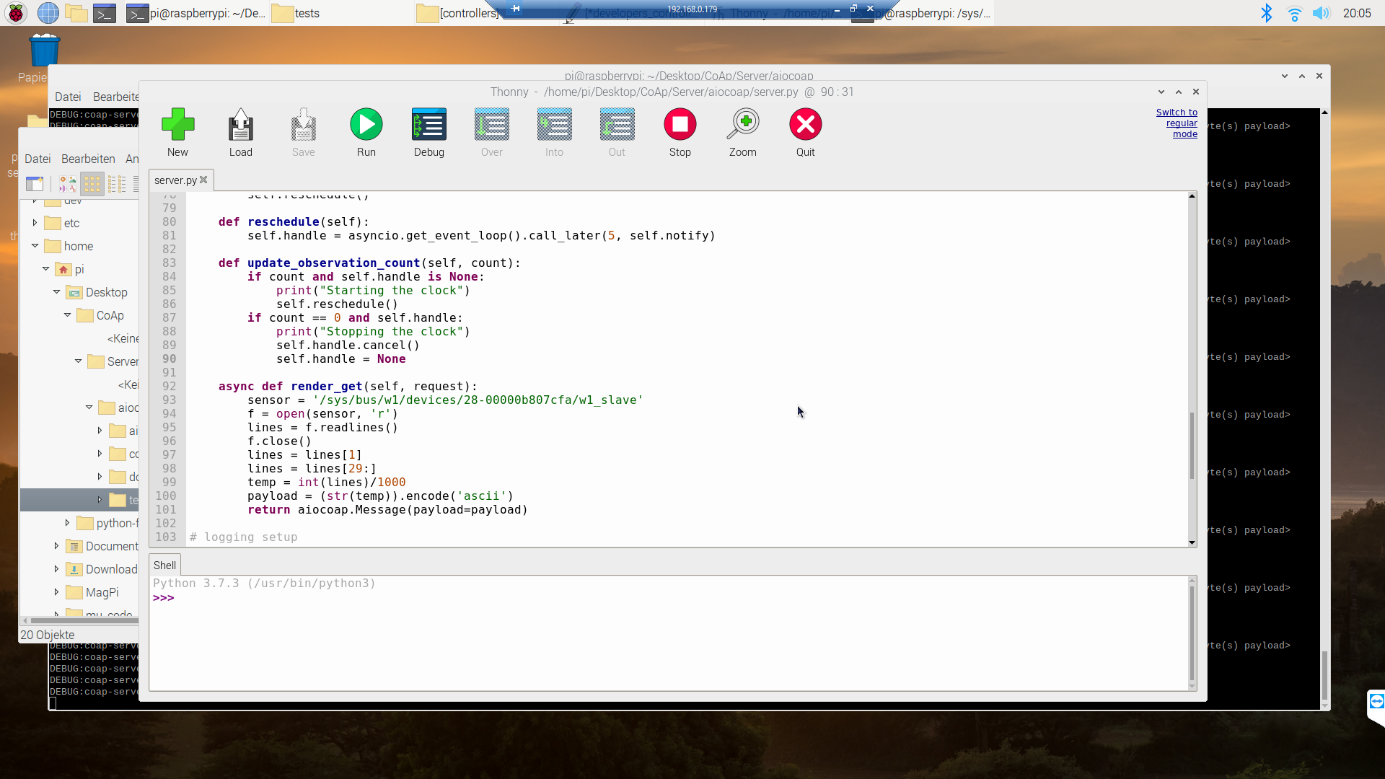
Within the file “server.py” the resources can be defined. Rewriting the time resource, which was defined as an observable resource, and copying the already written program to read the temperature from Task 1, this task was close to being fulfilled. Simply starting the server again and accessing it through the Cu4Cr plugin for Google Chrome we saw this:



Temperature

Via Oberserver Method

And the messages displayed on the terminal which started the server.

  
This is the function which computes the response, changed to deliver our temperature sensor value.

Benefits of CoAP:

Constrained Application Protocol is a lightweight protocol which allows reliable data transfer (CON, ACK) using UDP. It also has significantly less overhead than HTTP which allows efficient transfer for small packet sizes. Using the tutorial GitHub, it was really easy to setup, but there is way less information and examples easily accessible. The CoAP protocol was not implemented naturally into our browser yet, so additional software is required to test and debug it.

**Task 4) MQTT**

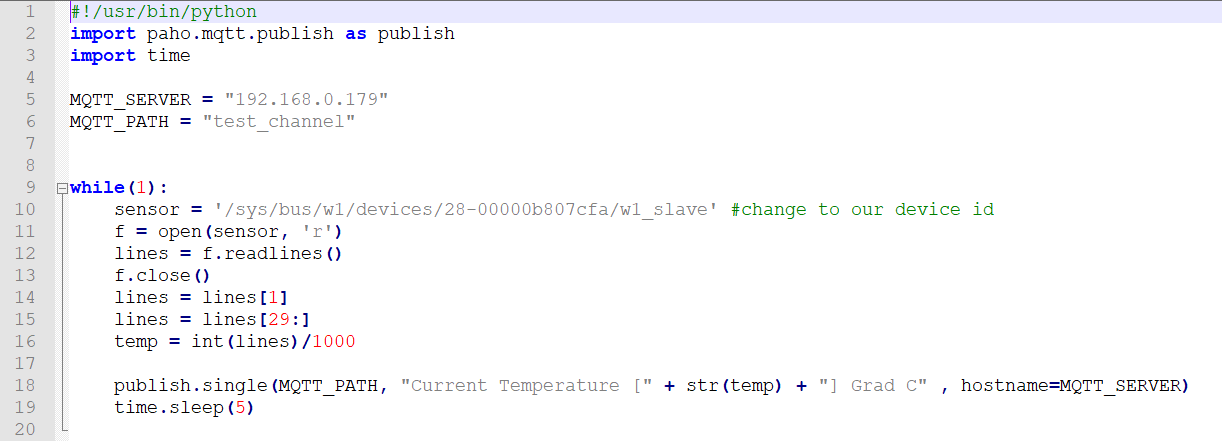
For the last task a messaging protocol out of {MQTT, AMQP and XMPP} should be implemented. After finding a useful tutorial to MQTT which looked tailored to our problem, we simply choose MQTT. The required ecosystem is only python interpreter and mosquitto which is described below.

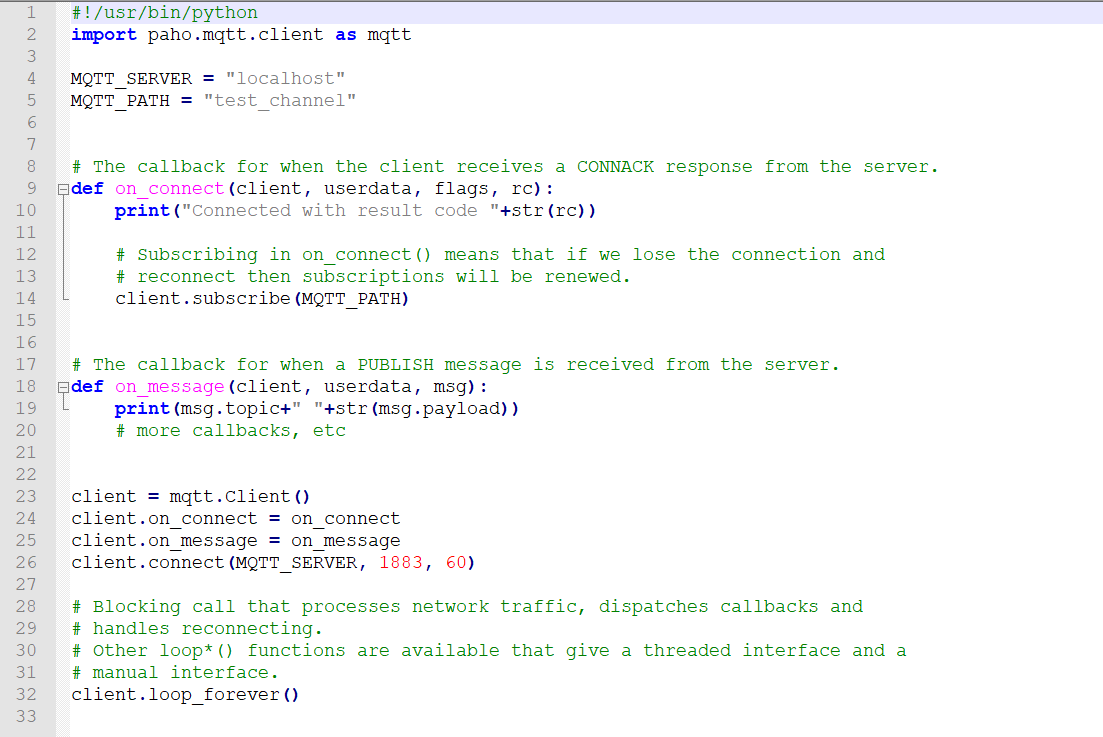
Link: <https://tutorials-raspberrypi.com/raspberry-pi-mqtt-broker-client-wireless-communication/>

First, we installed mosquitto, which is a MQTT library. After installing it an MQTT server is already running and listens to the MQTT Port number.

sudo apt-get install -y mosquitto mosquitto-clients

Then we programmed, a publisher and subscriber clients. The publisher publishes data to the broker, which informs the clients which subscribed to the channel and forwards it.

The publish and subscriber programs in python:



Both of the clients were programmed quite simple, because they are only used to demonstrate the usage of MQTT with the RPi and temperature sensor. The publisher publishes the temperature sensors data every 5 seconds.