

Aufbau mit ESP8266

Diesen Beitrag gibt es auch als [PDF-Dokument](#).

Dass ESP32 und ESP8266, selbständig e-Mails versenden können, habe ich kürzlich in einem Beitrag gezeigt. Dafür ist ein Mail-Konto bei einem Provider, wie gmail, erforderlich, welcher eine entsprechende Schnittstelle zur Verfügung stellt. In diesem und drei weiteren Posts stelle ich, im Zusammenwirken mit verschiedenen Schaltungen vier weitere Möglichkeiten des Nachrichtenversands vor. Ich beginne heute mit **IFTTT**. Das ist das Akronym für **If This Then That**. Hinter diesem Namen steht ein Web-Portal, das diverse Dienste zur Verfügung stellt, unter anderem, den Versand von e-Mails, getriggert durch einen Post von einem ESP32 oder ESP8266. Dazu brauchen wir einen Account bei IFTTT. Pro Konto kann man zwei Anwendungen kostenlos erstellen. Wie das funktioniert, beschreibe ich in dieser Folge aus der Reihe

MicroPython auf dem ESP32 und ESP8266

heute

IFTTT-Nachrichten von ESP32 und ESP8266

Die heutige Schaltung stellt einen Personenzähler dar, der über Ultraschall Leute zählen kann, die einen Saal oder ein Zimmer betreten. Den Sensor, einen HC-SR04, habe ich so an der Tür angebracht, dass sich bis zur passierenden Person ein

Abstand von ca. 30 cm ergibt. Das nächste Hindernis sollte dann einen Abstand von einem Meter oder mehr haben. Die Grenzwerte lassen sich im Programm an die bestehenden örtlichen Verhältnisse natürlich anpassen. Der Zählvorgang passiert, wenn die Person den Schallkegel verlässt. Eine Ein Hysterese, also ein Ausschlussbereich von möglichen Entfernungen, stellt sicher, dass Mehrfachzählungen unterbunden werden. Der Controller wird getriggert, wenn eine Entfernung von weniger als 30 cm gemessen wird. Erst wenn der Schallweg dann wieder mehr als einen Meter beträgt, wird der Zähler um eins erhöht.

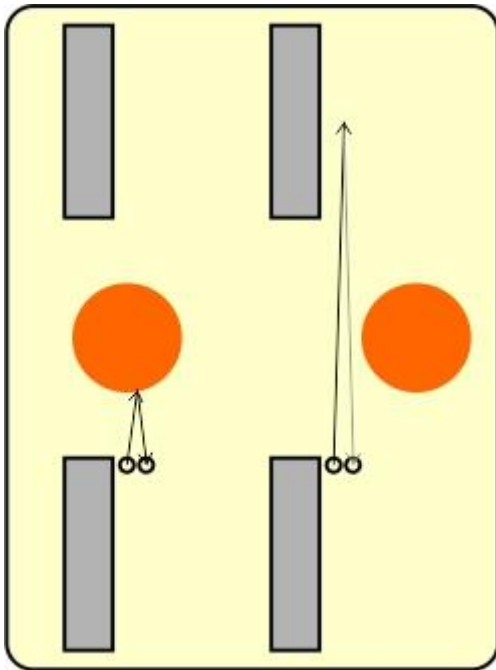


Abbildung 1: Anbringen des Sensors

Durch die Vorgabe eines Zählwert-Limits, kann gesteuert werden, wann der Controller die IFTTT-Anwendung triggern soll. Kurze Zeit später trifft die Mail dann auf dem angegebenen Konto ein.

Hardware

Jeder der angeführten Controller-Typen ist grundsätzlich einsetzbar, zumindest in diesem Beitrag. Bei Verwendung eines BME280 scheidet der ESP8266 allerdings aus, wegen Speicherplatzmangels. Deshalb habe ich hier einen SHT21 für die Temperaturmessung eingesetzt. Ein weiterer Grund für diese Entscheidung ist, dass der Baustein, wie auch das Display, über den I2C-Bus ansteuerbar ist.

Um den Zustand der Schaltung jederzeit auch direkt vor Ort einsehen zu können, habe ich dem ESP ein kleines Display spendiert. Über die Flash-Taste ist ein geordneter Abbruch des Programms möglich, falls zum Beispiel Aktoren sicher ausgeschaltet werden müssen, wie hier die LED. Dem ESP8266 D1 mini muss man dazu ein extra Tastenmodul spendieren, weil es selbst keine Flash-Taste hat.

1	D1 Mini NodeMcu mit ESP8266-12F WLAN Modul oder D1 Mini V3 NodeMCU mit ESP8266-12F oder NodeMCU Lua Amica Modul V2 ESP8266 ESP-12F WIFI oder NodeMCU Lua Lolin V3 Module ESP8266 ESP-12F WIFI oder ESP32 Dev Kit C unverlötet oder ESP32 Dev Kit C V4 unverlötet oder ESP32 NodeMCU Module WLAN WiFi Development Board mit CP2102 oder NodeMCU-ESP-32S-Kit oder ESP32 Lolin LOLIN32 WiFi Bluetooth Dev Kit
1	0,91 Zoll OLED I2C Display 128 x 32 Pixel
1	LED , zum Beispiel rot
1	Widerstand 330 Ω
1	Widerstand 1,0 k Ω
1	Widerstand 2,2 k Ω
2	MB-102 Breadboard Steckbrett mit 830 Kontakten
1	KY-004 Taster Modul
diverse	Jumper Wire Kabel 3 x 40 STK. je 20 cm M2M/ F2M / F2F evtl. auch 65Stk. Jumper Wire Kabel Steckbrücken für Breadboard
optional	Logic Analyzer

Damit neben dem Controller noch Steckplätze für die Kabel frei sind, habe ich zwei Breadboards, mit einer Stromschiene dazwischen, zusammengesteckt.

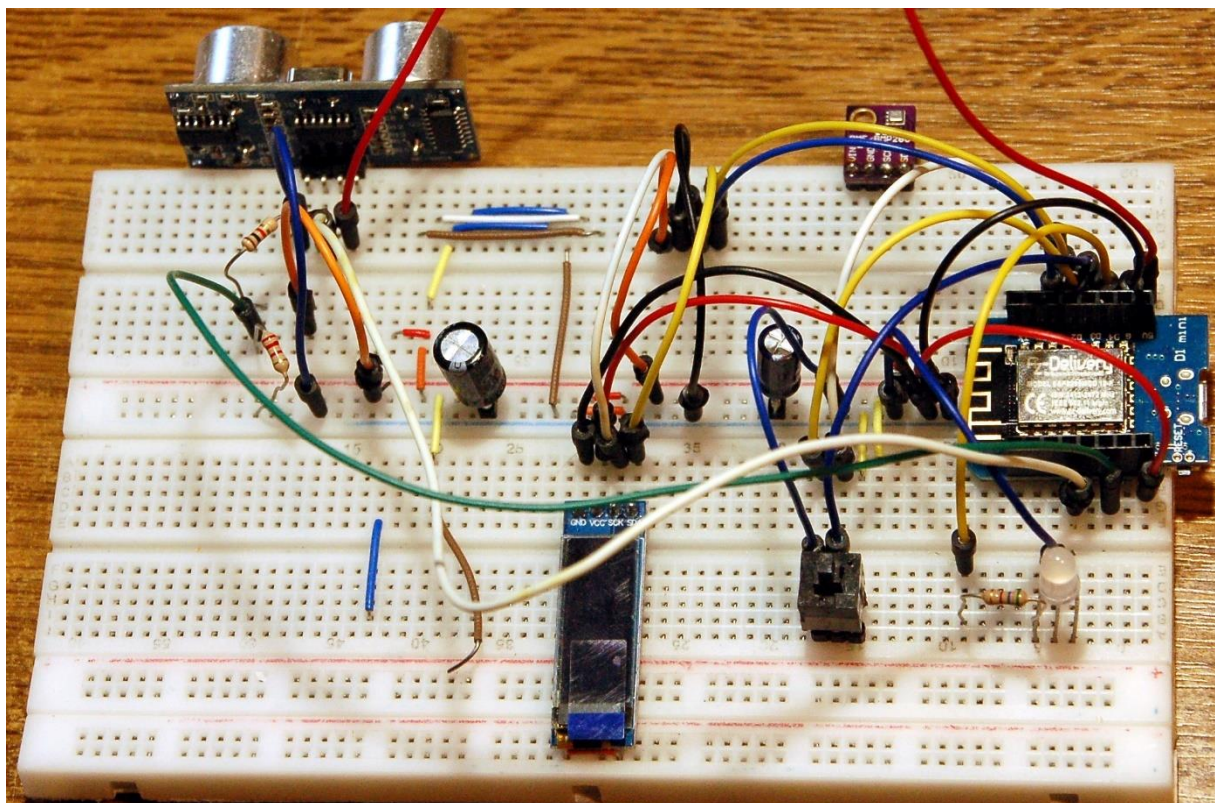


Abbildung 2: Entfernungsmesser mit Ultraschall - ESP8266

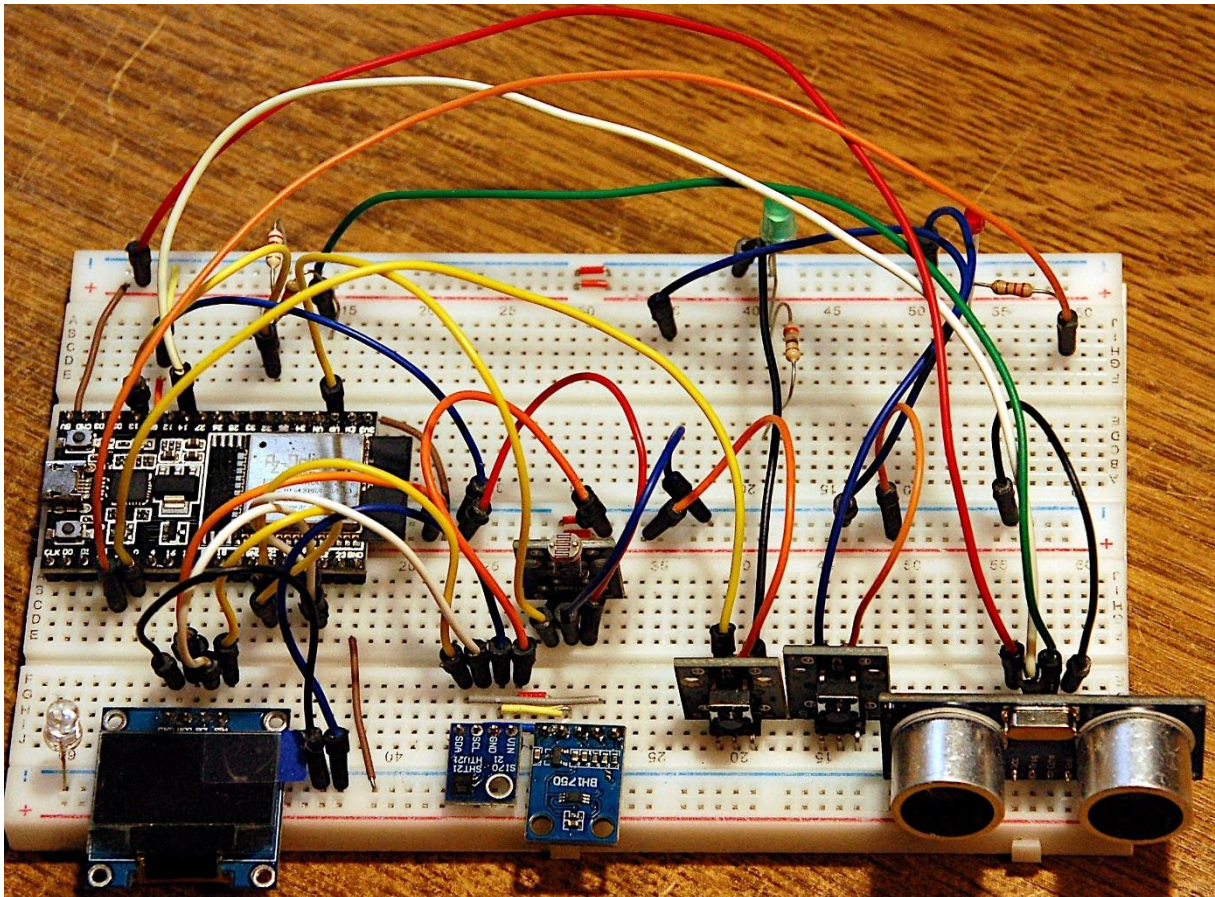


Abbildung 3: Entfernungsmesser mit Ultraschall - ESP32

Abschließend zur Hardware kommen jetzt noch die Schaltbilder für ESP32 und ESP8266.

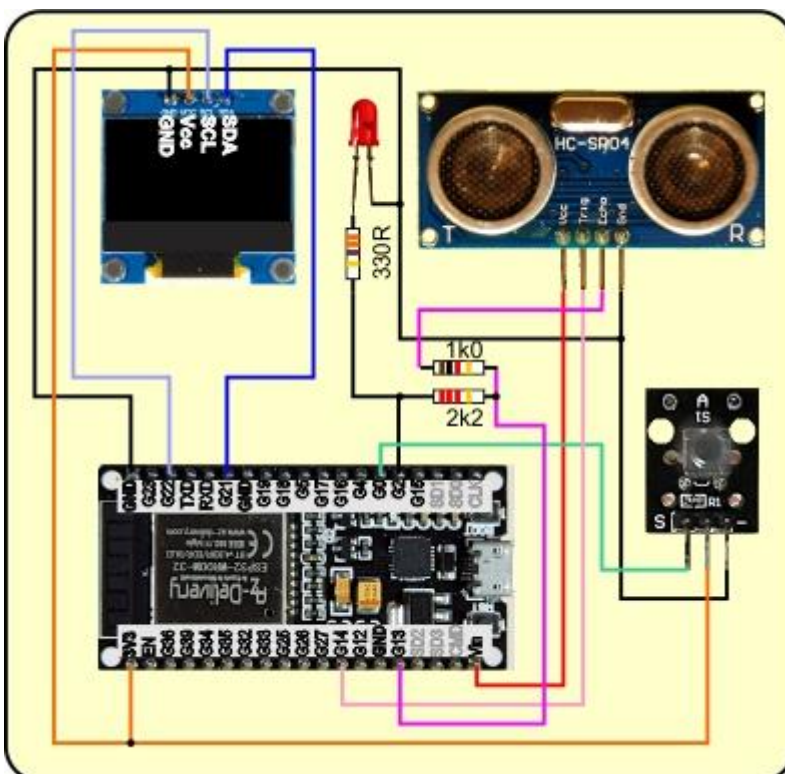


Abbildung 4: Schaltung für ESP32

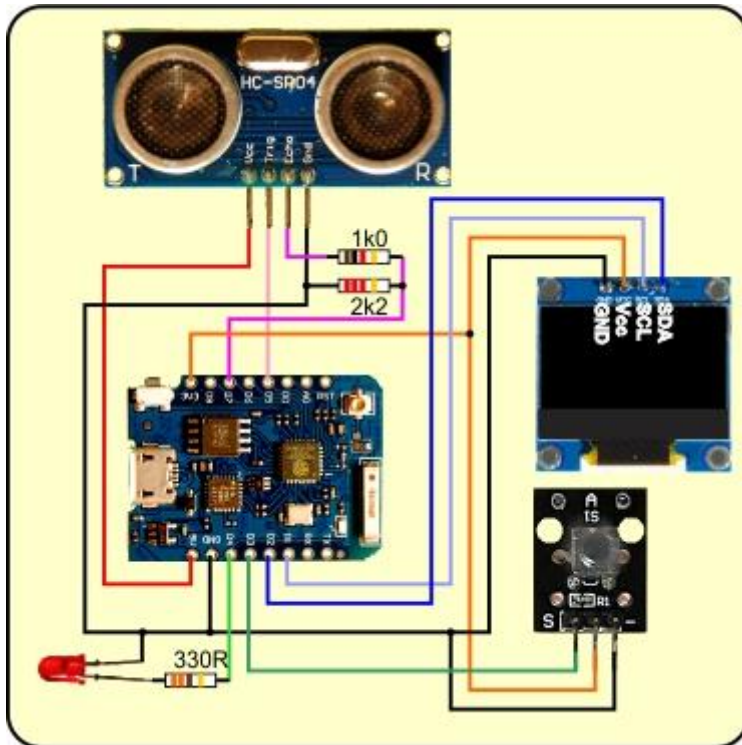


Abbildung 5: Schaltung für ESP8266

Die Software

Fürs Flashen und die Programmierung des ESP32:

[Thonny](#) oder
[µPyCraft](#)

Verwendete Firmware für den ESP32:

[v1.19.1 \(2022-06-18\) .bin](#)

Verwendete Firmware für den ESP8266:

[v1.19.1 \(2022-06-18\) .bin](#)

Die MicroPython-Programme zum Projekt:

[ssd1306.py](#) Hardwaretreiber für das OLED-Display

[oled.py](#) API für das OLED-Display

[sht21.py](#) Treiber für das SHT21-Modul

[urequests.py](#) Treibermodul für den HTTP-Betrieb

[timeout.py](#) Softwaretimer-Modul

[ifttt.py](#) Demoprogramm für den e-Mailversand

MicroPython - Sprache - Module und Programme

Zur Installation von Thonny finden Sie hier eine [ausführliche Anleitung \(english version\)](#). Darin gibt es auch eine Beschreibung, wie die [Micropython-Firmware](#) (Stand 18.06.2022) auf den ESP-Chip [gebrannt](#) wird.

MicroPython ist eine Interpretersprache. Der Hauptunterschied zur Arduino-IDE, wo Sie stets und ausschließlich ganze Programme flashen, ist der, dass Sie die MicroPython-Firmware nur einmal zu Beginn auf den ESP32 flashen müssen, damit der Controller MicroPython-Anweisungen versteht. Sie können dazu Thonny, µPyCraft oder esptool.py benutzen. Für Thonny habe ich den Vorgang [hier](#) beschrieben.

Sobald die Firmware geflasht ist, können Sie sich zwanglos mit Ihrem Controller im Zwiegespräch unterhalten, einzelne Befehle testen und sofort die Antwort sehen, ohne vorher ein ganzes Programm kompilieren und übertragen zu müssen. Genau das stört mich nämlich an der Arduino-IDE. Man spart einfach enorm Zeit, wenn man einfache Tests der Syntax und der Hardware bis hin zum Ausprobieren und Verfeinern von Funktionen und ganzen Programmteilen über die Kommandozeile vorab prüfen kann, bevor man ein Programm daraus strickt. Zu diesem Zweck erstelle ich auch gerne immer wieder kleine Testprogramme. Als eine Art Makro fassen sie wiederkehrende Befehle zusammen. Aus solchen Programmfragmenten entwickeln sich dann mitunter ganze Anwendungen.

Autostart

Soll das Programm autonom mit dem Einschalten des Controllers starten, kopieren Sie den Programmtext in eine neu angelegte Blankodatei. Speichern Sie diese Datei unter boot.py im Workspace ab und laden Sie sie zum ESP-Chip hoch. Beim nächsten Reset oder Einschalten startet das Programm automatisch.

Programme testen

Manuell werden Programme aus dem aktuellen Editorfenster in der Thonny-IDE über die Taste F5 gestartet. Das geht schneller als der Mausklick auf den Startbutton, oder über das Menü **Run**. Lediglich die im Programm verwendeten Module müssen sich im Flash des ESP32 befinden.

Zwischendurch doch mal wieder Arduino-IDE?

Sollten Sie den Controller später wieder zusammen mit der Arduino-IDE verwenden wollen, flashen Sie das Programm einfach in gewohnter Weise. Allerdings hat der ESP32/ESP8266 dann vergessen, dass er jemals MicroPython gesprochen hat. Umgekehrt kann jeder Espressif-Chip, der ein kompiliertes Programm aus der Arduino-IDE oder die AT-Firmware oder LUA oder ... enthält, problemlos mit der MicroPython-Firmware versehen werden. Der Vorgang ist immer so, wie [hier](#) beschrieben.

Einrichten eines IFTTT-Accounts

Folgen Sie dem Link zu ifttt.com und klicken Sie auf **Get started** rechts oben.

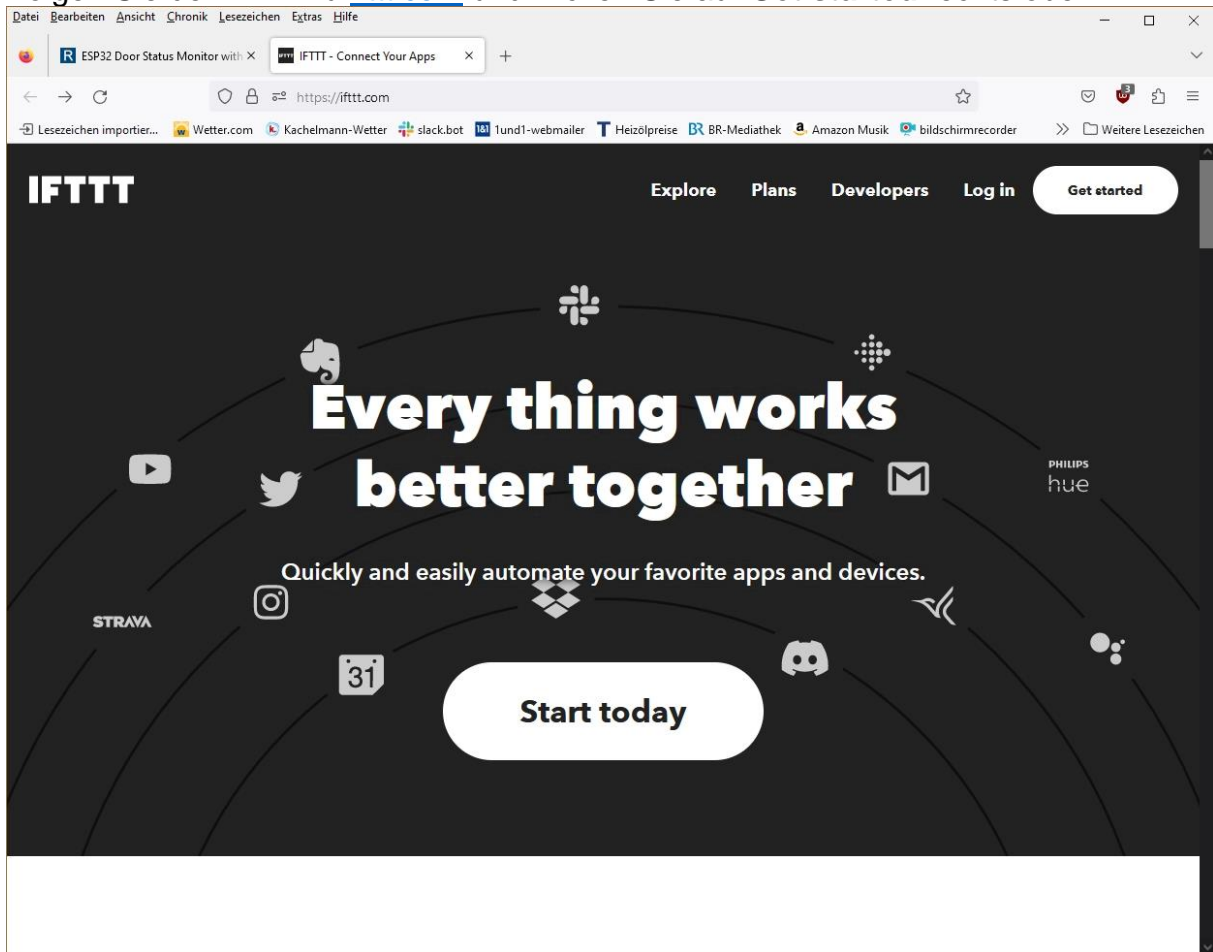


Abbildung 6: IFTTT-Startseite

Ich möchte mich nicht über ein Google-Konto anmelden sondern mit meiner e-Mailadresse – **sign up**.

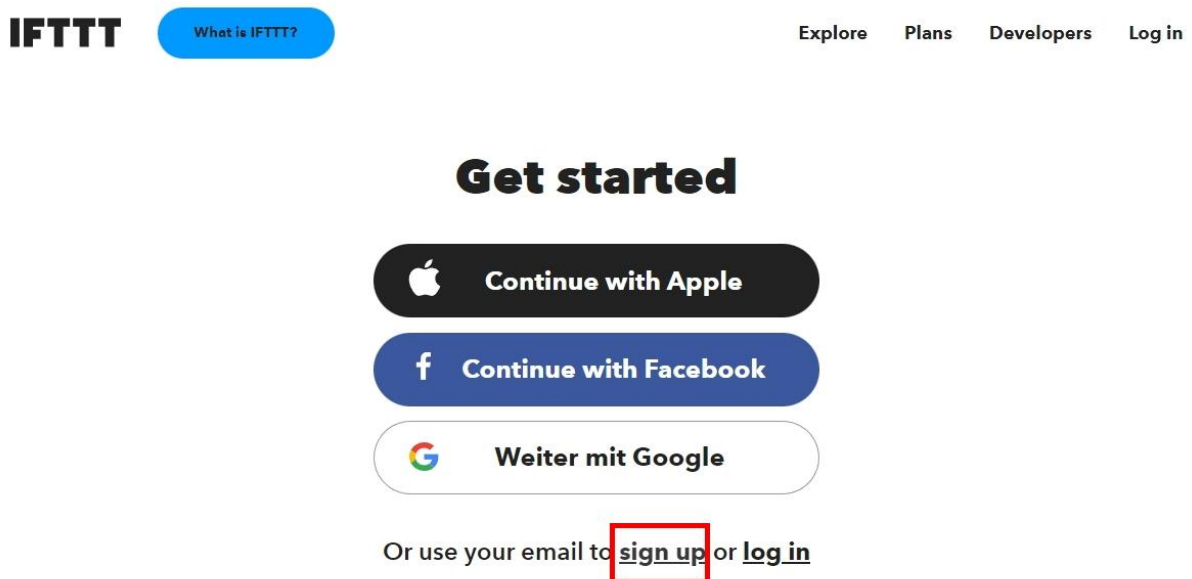


Abbildung 7: Sign up - Anmelden oder Registrieren

Registrieren mit Mailadresse und einem Passwort, es sollte nicht das sein, mit dem Sie auf dem Mailserver einloggen. Aufpassen! Das Passwort wird nur einmal eingegeben und nicht verifiziert!

IFTTT

[Learn about IFTTT AI](#)

[Explore](#)

[Plans](#)

[Developers](#)

[Log in](#)

Sign up


Get started


Abbildung 8: Registrieren


[< Back](#)

Let's start!

What mobile device(s) do you currently use?
This is important and helps us find the best Applets for you.

1 ☒  **Android**

 **iPhone**

 **Neither**

Continue

2

Abbildung 9: optional - Handy-App herunterladen

Mindestens eines der Felder muss man anklicken, sonst geht's nicht weiter.

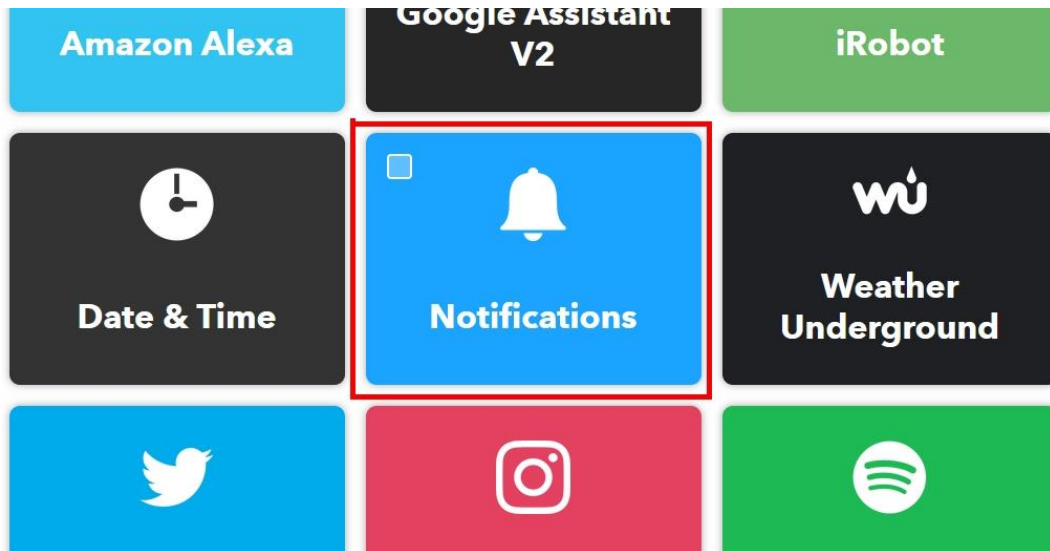


Abbildung 10: _5Mindestens ein Feld muss markiert werden

Wir brauchen für unseren Zweck keinen Vollzugriff, deshalb – **Not now**.



Start your trial

Subscribe to get full access to popular Applets and features. Make your favorite things work better together.

Continue

Not now

Abbildung 11: Wir wollen keinen Pro-Account

Jetzt ganz nach unten scrollen – **Get startet**

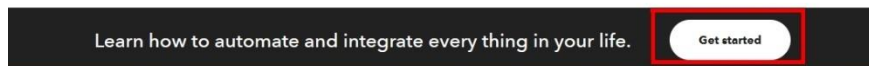
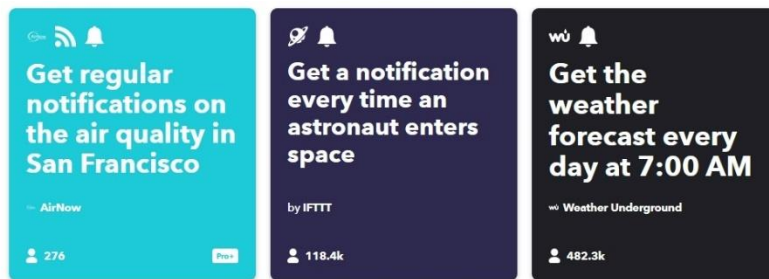


Abbildung 12: Get started - Legen wir los

Anlegen einer Anwendung

In unserem Account legen wir jetzt ein Applet an. Starten Sie auf der Hauptseite mit **Create**.

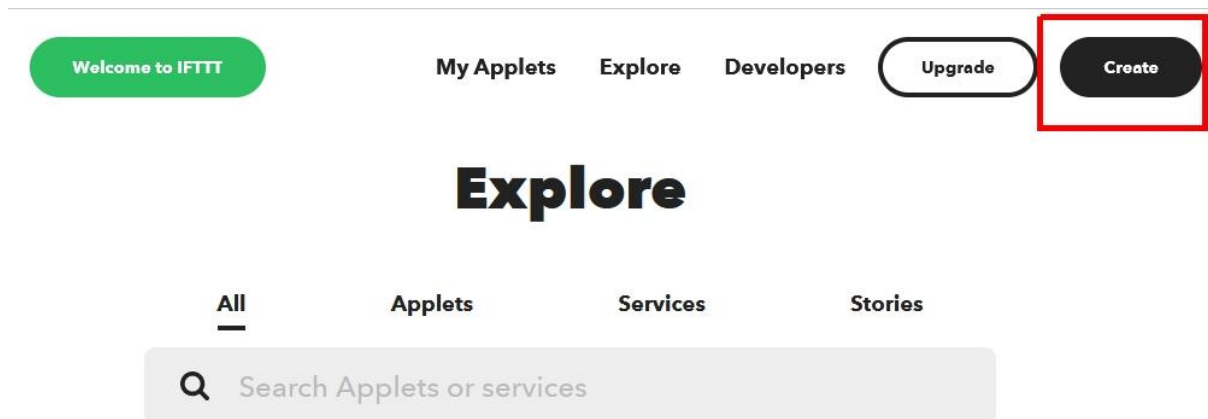


Abbildung 13: Wir starten mit Create

Zuerst muss ein Trigger, also ein Auslöser, definiert werden. Klick auf **Add**.

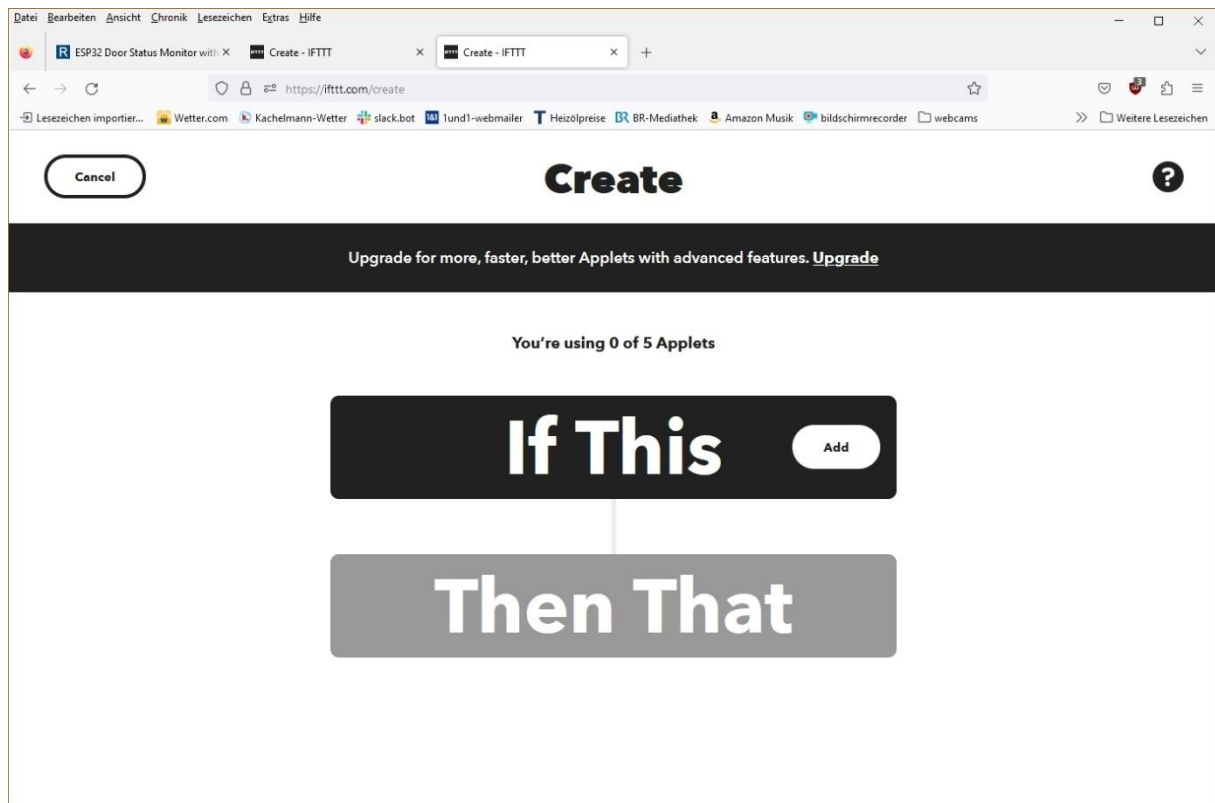


Abbildung 14: Wir erzeugen einen Trigger

Geben Sie **webhook** in das Suchfeld ein und klicken Sie dann auf das Symbol **Webhooks**.

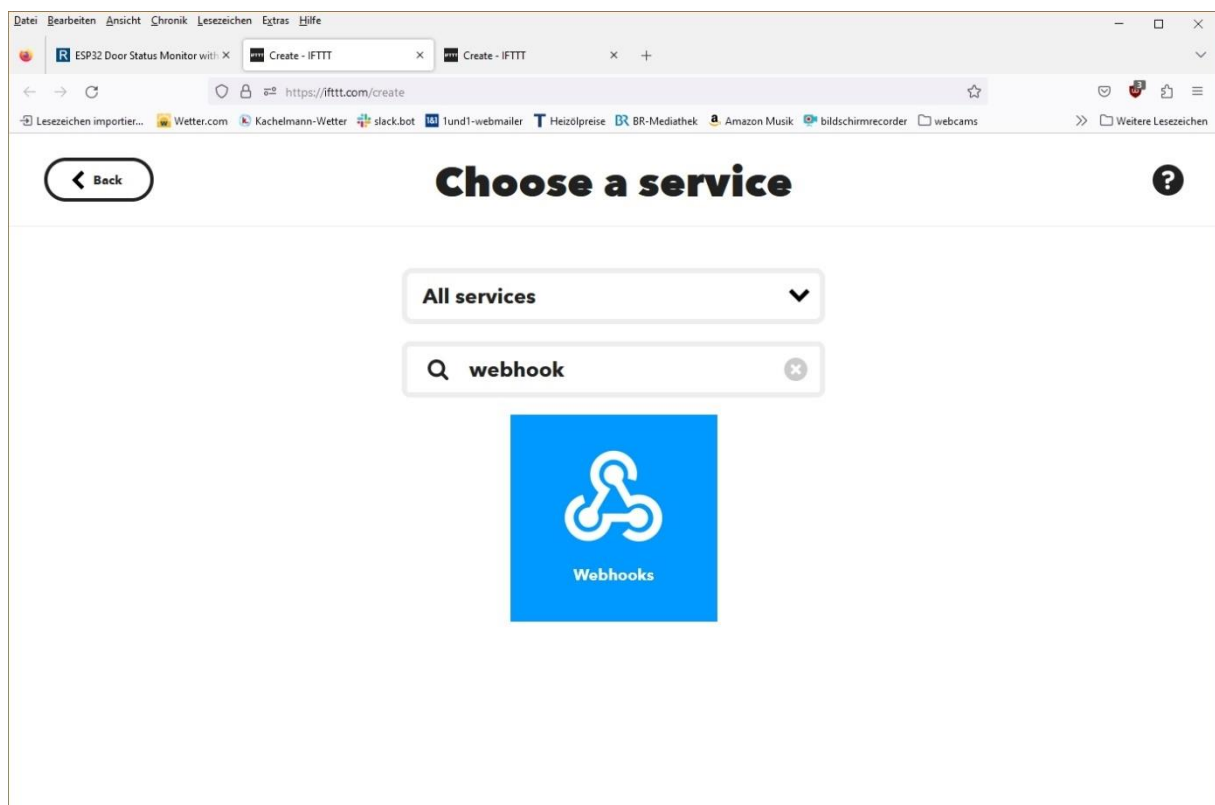


Abbildung 15: Ein Webhook wird benötigt

Wir werden eine Web-Anfrage senden, checken Sie das mittlere Feld.

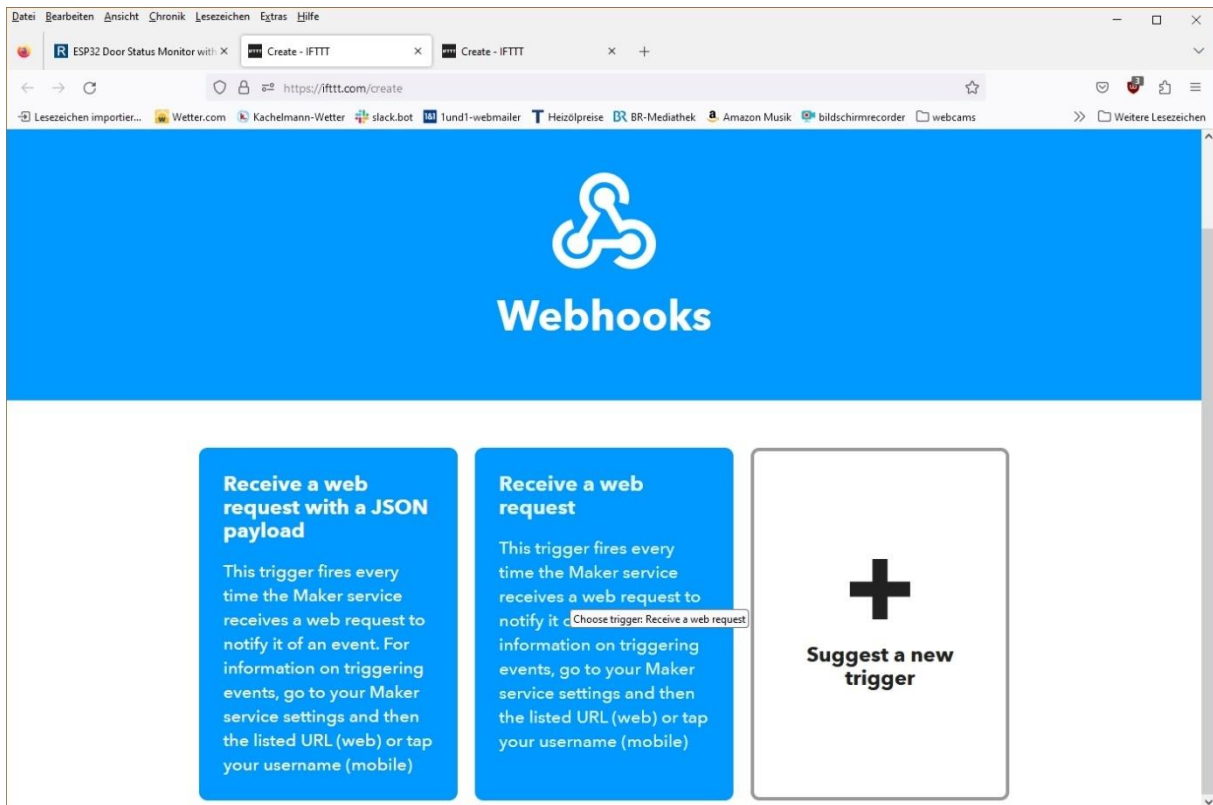


Abbildung 16: Receive a webrequest

Geben Sie einen Namen für die App ein und klicken Sie auf **Create Trigger**.

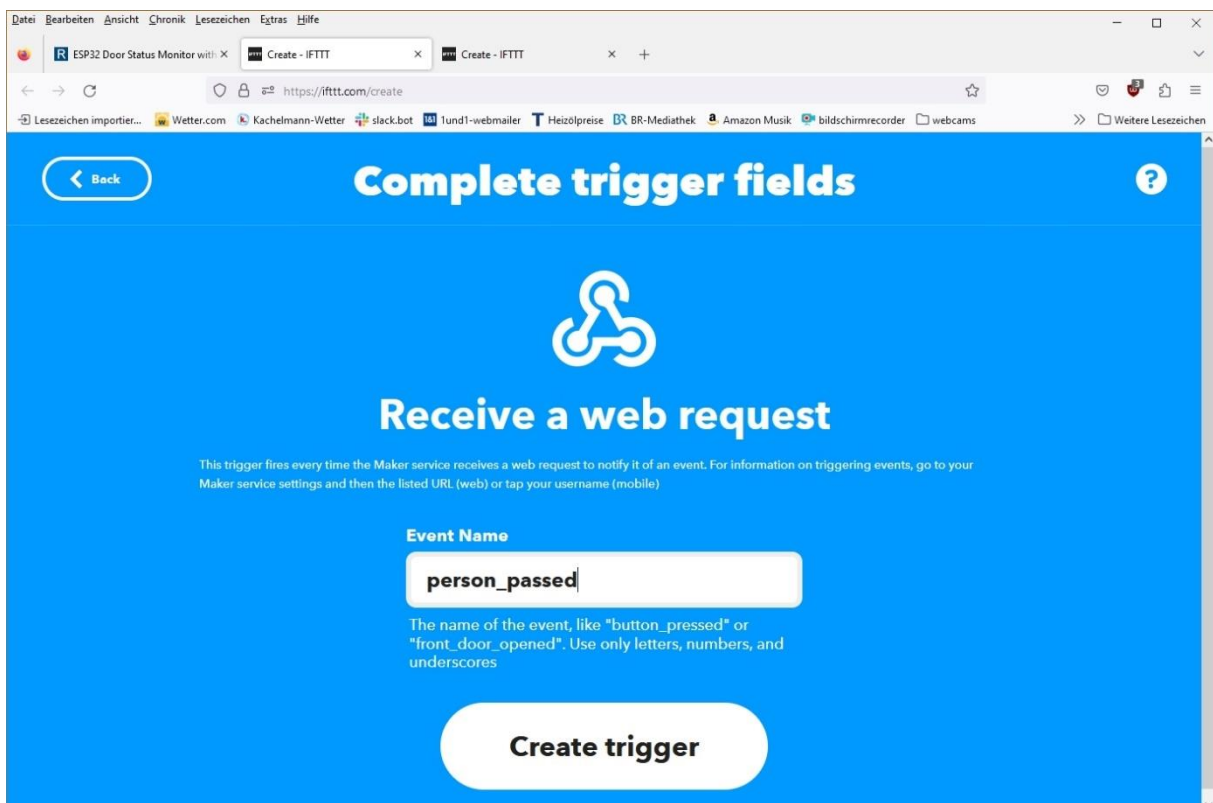


Abbildung 17: Trigger generieren

Damit ist die Erzeugung des Webhooks abgeschlossen. Klicken Sie jetzt auf **Then** und schreiben Sie **email** in das Suchfeld. Dann klicken Sie auf das linke Symbol - **Email**.

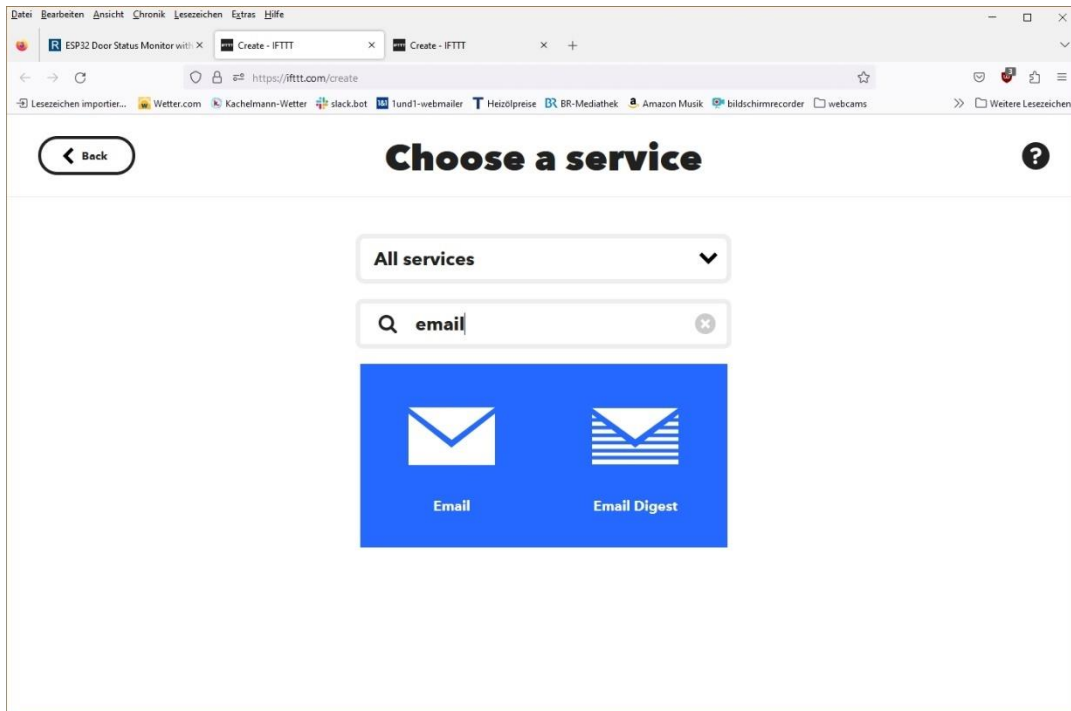


Abbildung 18: Einen Dienst auswählen

Verbinden Sie jetzt den Service mit dem Trigger – **Connect**.

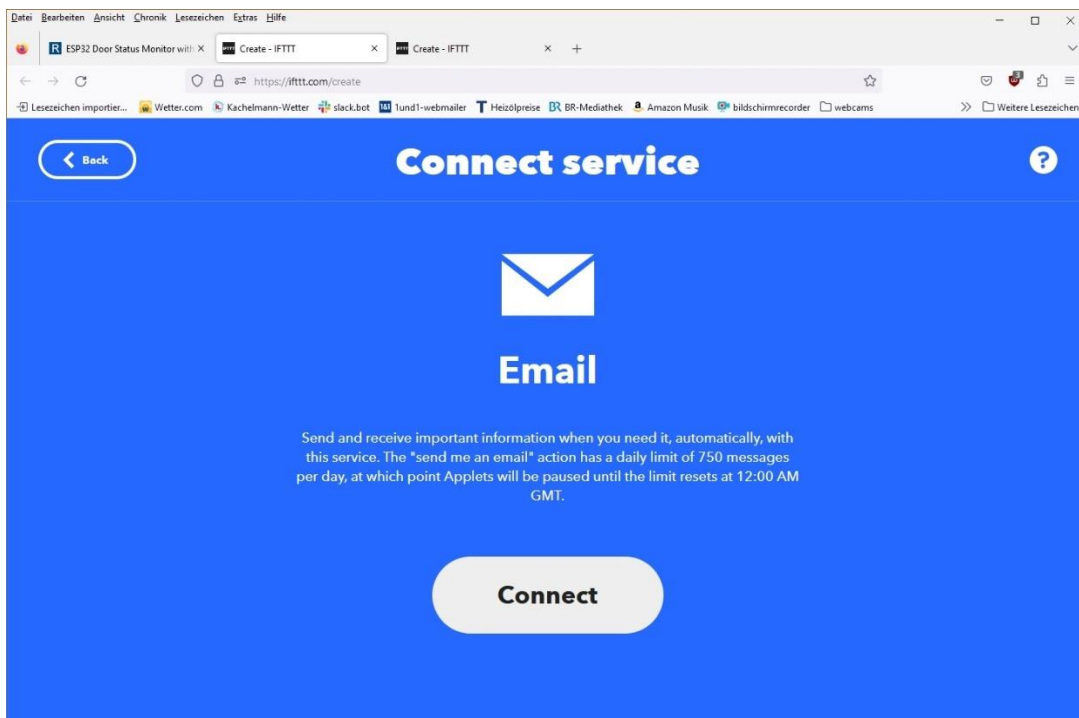


Abbildung 19: Service verbinden

Als nächstes wird die Empfängeradresse der Mails eingetragen. An dieses Konto verschickt IFTTT gleich eine Mail mit einer Pin, die ins nächste Feld eingetragen werden muss.

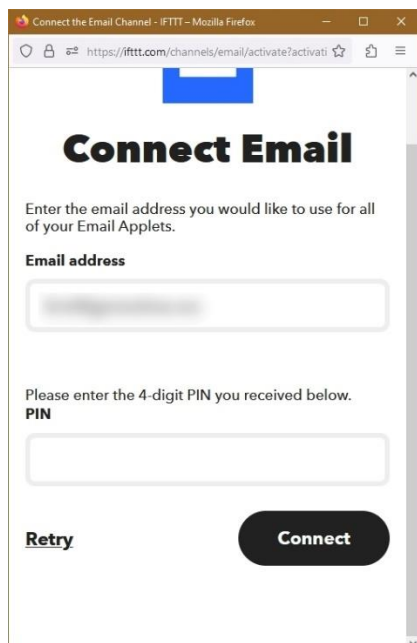
The screenshot shows a web browser window with the URL 'https://ifttt.com/channels/email/activate?activati'. The page title is 'Connect the Email Channel - IFTTT - Mozilla Firefox'. The main heading is 'Connect Email'. Below it, the text says 'Enter the email address you would like to use for all of your Email Applets.' There is a text input field labeled 'Email address'. Below that, the text says 'Please enter the 4-digit PIN you received below.' There is a text input field labeled 'PIN'. At the bottom left is a link 'Retry' and at the bottom right is a black button labeled 'Connect'.

Abbildung 20: Mailempfänger angeben

Schauen Sie im Mail-Konto nach, dort sollte eine Mail von IFTTT angekommen sein. Übertragen Sie die Pin in das Formular und klicken Sie auf **Connect**.

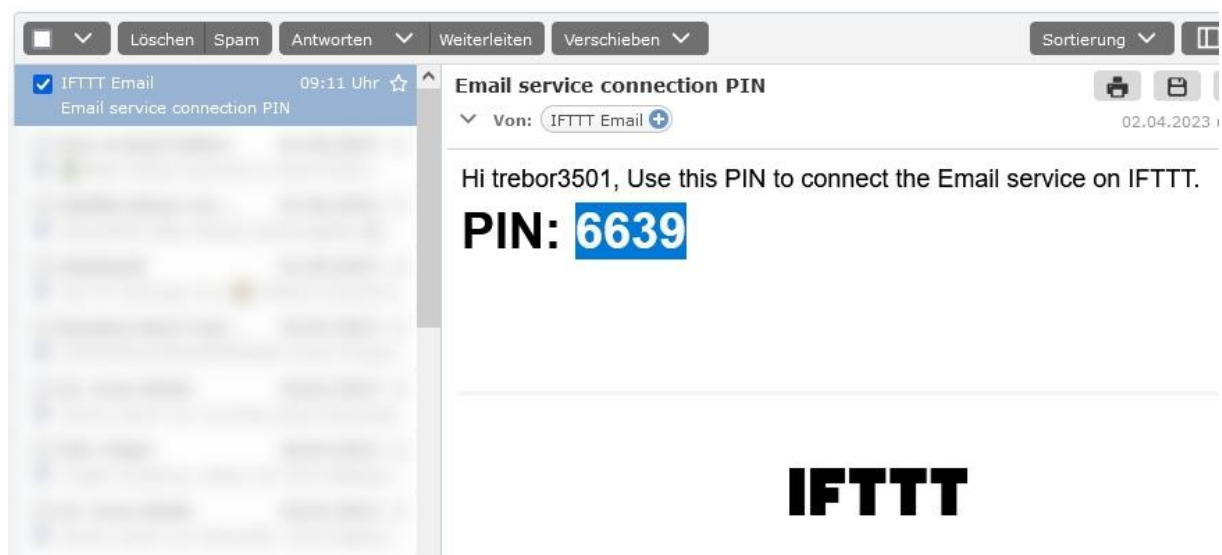


Abbildung 21: Postfach öffnen und Pin übertragen

Nun werden der Betreff und der Text der Mail editiert – **Create action**.

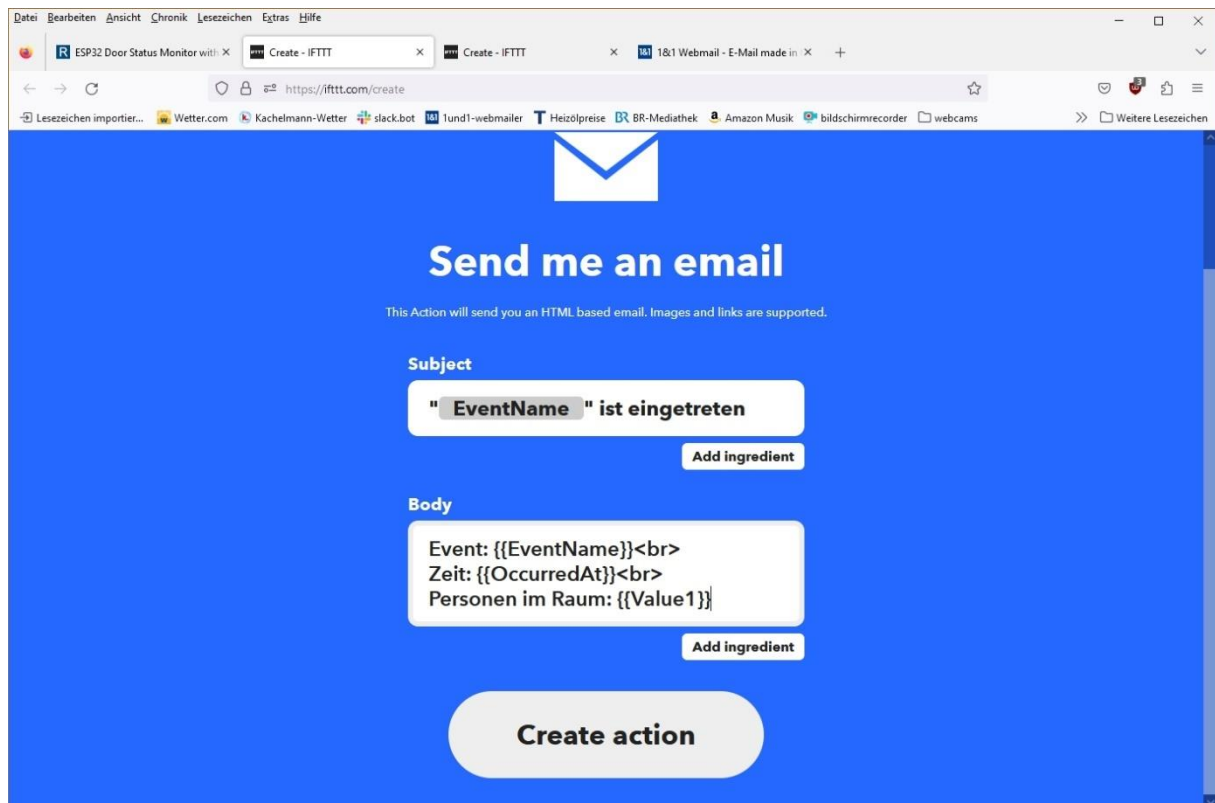


Abbildung 22: Inhalt der Mail editieren

Mit **Continue** gelangen Sie zur nächsten Seite mit der Übersicht der App.

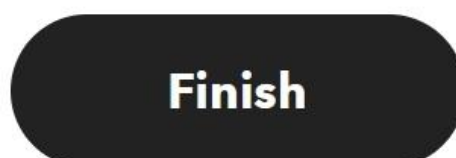
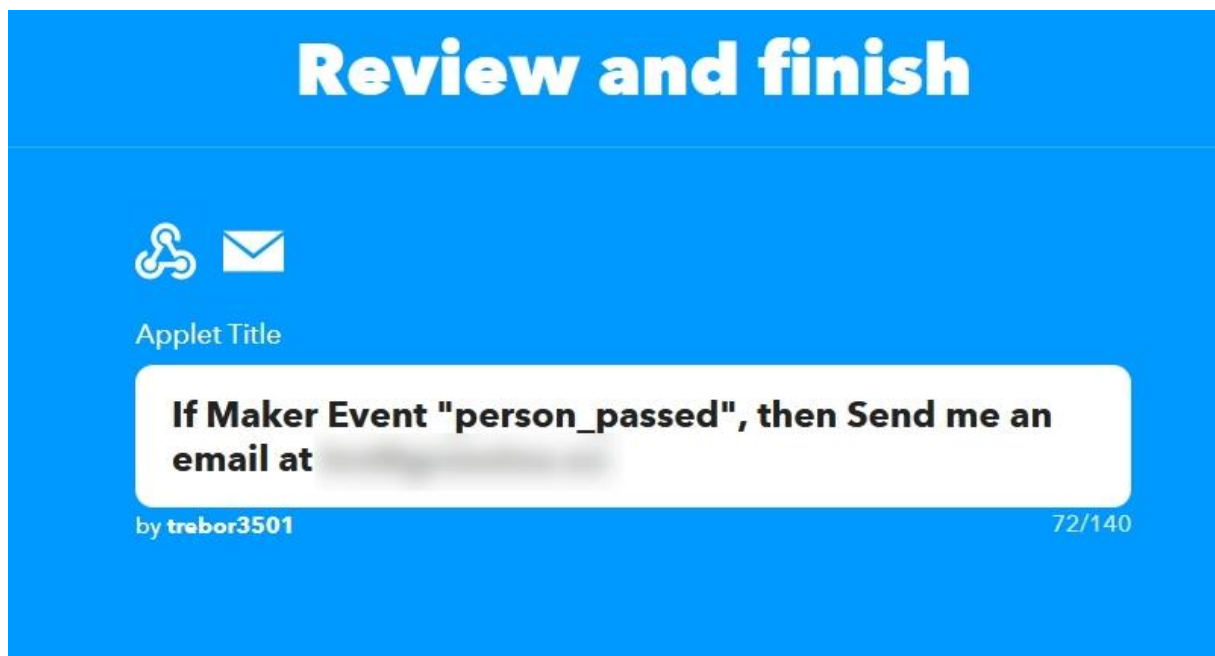


Abbildung 23: Zusammenfassung

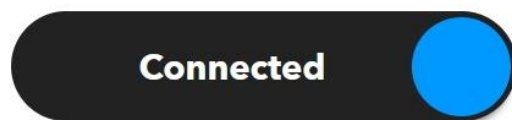
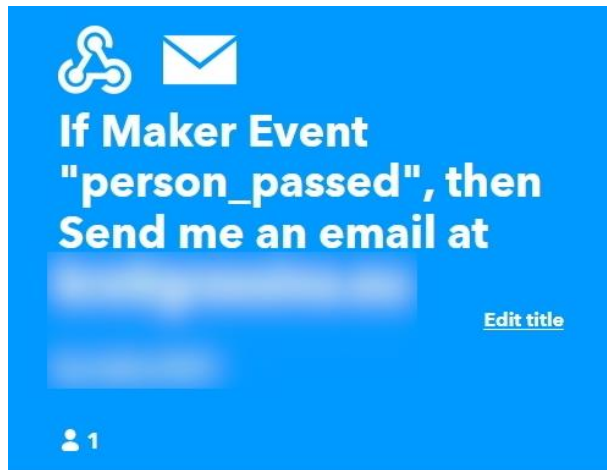


Abbildung 24: Bestätigung

Gehen Sie jetzt auf [maker_webhooks](#) – Documentation.

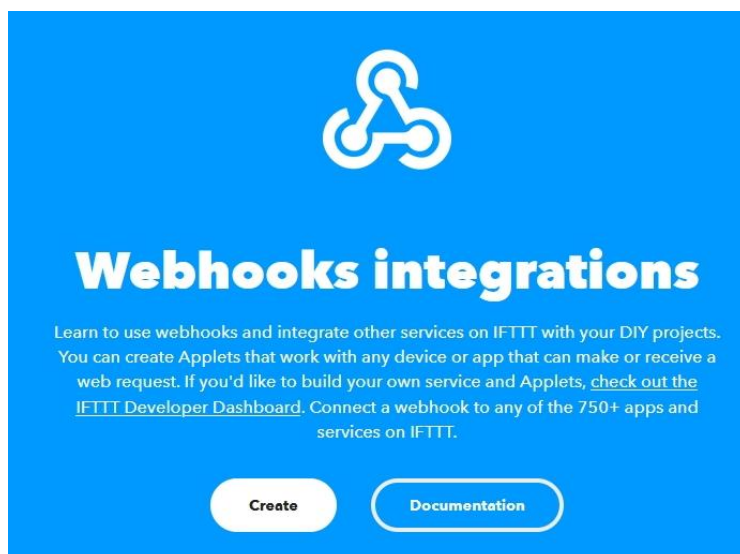


Abbildung 25: Zur Dokumentation

Auf dieser Seite finden Sie den 22-stelligen API-Key, den Sie sich notieren oder besser kopieren sollten, denn sie brauchen ihn später. Füllen Sie im Formular die Felder aus und klicken Sie auf **Test it**, um eine Testmail zu versenden.



Your key is: **cvU6**

[Back to service](#)

To trigger an Event with an arbitrary JSON payload

Make a POST or GET web request to:

```
https://maker.ifttt.com/trigger/{event}/json/with/key/cvU6XnVQCSfUVy9oyVXesc
```

* *Note the extra /json path element in this trigger.*

With any JSON body. For example:

```
{ "this" : [ { "is": { "some": [ "test", "data" ] } } ] }
```

You can also try it with `curl` from a command line.

```
curl -X POST -H "Content-Type: application/json" -d '{"this":[{"is":{"some":["test","data"]}}]}' https://maker.ifttt.com/trigger/{event}/json/with/key/cvU6XnVQCSfUVy9oyVXesc
```

Please read [our FAQ](#) on using Webhooks for more info.

Test It

Abbildung 26: Test-Mail versenden

Haben Sie die Mail bekommen?

Das Programm

Die Anschlüsse auf dem ESP32 sind so gewählt, dass die Nummern auch für den ESP8266 gelten. Somit arbeitet das Programm für beide Controllerfamilien ohne Änderungen. Es beginnt mit einer Übersetzungstabelle für den ESP8266. Diese Familie ärgert die User gerne mit ständigen Neustarts. Deshalb sollte nach dem Flashen der Firmware als erstes **webrepl** getötet werden.

```
# ifttt.py
#
# Pintranslator fuer ESP8266-Boards
# LUA-Pins      D0 D1 D2 D3 D4 D5 D6 D7 D8
# ESP8266 Pins  16  5  4  0  2 14 12 13 15
#                SC SD                T
#
# Nach dem Flashen der Firmware auf dem ESP8266:
# import webrepl_setup
# > d fuer disable
# Dann RST; Neustart!
```

Dann erledigen wir das Importgeschäft. Von **machine** holen wir die Klassen **Pin** und **SoftI2C**, von **time** die Funktion **sleep**. Timeout ist ein Eigenbau-Modul, das drei nichtblockierende Softwaretimer enthält. Von den Funktionen wird eine Funktion zurückgegeben, eine sogenannte [Closure](#). Indem ich mit dem `*` alles von dem Modul in den globalen Namensraum einbinde, kann ich später auf die Funktionen so

zugreifen, als ob sie direkt im Programm deklariert worden wären, also ohne das sonst übliche Objekt-[Prefix](#).

Die Klasse **SHT21** bedient das SHT21-Platinchen, die Klasse **OLED** stellt ein API für das Display zur Verfügung. Sie benötigt die Klasse **SSD1306**, weshalb die Datei **ssd1306.py** auch in den Flash des Controllers hochgeladen werden muss. Weil beim ESP8266 das Modul **urequests** nicht im Kernel enthalten ist, muss auch dieses hochgeladen werden. Der ESP32-Kernel enthält das Modul bereits. Es dient zum Senden und Empfangen von HTML-Paketen. Für die Anbindung an das lokale Netzwerk brauchen wir **network**. **sys** liefert uns Möglichkeiten, den Controllertyp abzufragen und das Programm geordnet zu verlassen.

```
from machine import Pin, SoftI2C
from time import sleep
from timeout import *
from sht21 import SHT21
from oled import OLED
import urequests as requests
import network
import sys
```

Im nächsten Abschnitt setzen wir die URL für den HTTP-Request zusammen und geben die [Credentials](#) für den Router-Zugriff an.

```
iftttKey="here goes your key"
iftttApp="person_passed"
iftttUrl='http://maker.ifttt.com/trigger/'+\
        iftttApp+'/with/key/'+iftttKey

mySid = 'EMPIRE_OF_ANTS'; myPass = "nightingale"
```

Über die Variable **sys.platform** ermitteln wir den Controllertyp, stellen dementsprechend die GPIO-Pins für den I2C-Bus ein und erzeugen ein Bus-Objekt.

```
if sys.platform == "esp8266":
    i2c=SoftI2C(scl=Pin(5),sda=Pin(4))
elif sys.platform == "esp32":
    i2c=SoftI2C(scl=Pin(22),sda=Pin(21))
else:
    raise RuntimeError("Unknown Port")
```

Das I2C-Bus-Objekt reichen wir an die Konstruktoren des OLED-Displays und des SHT21 weiter.

```
d=OLED(i2c,heightw=32) # 128x32-Pixel-Display
d.clearAll()
d.writeAt("PERSON COUNTER",0,0)

sht=SHT21(i2c)
```

Dann deklarieren wir das Tastenobjekt, den GPIO-Ausgang für die LED, den Triggerausgang für den HC-SR04 und den Eingang für das Laufzeitsignal vom Ultraschallsensor. Der Ultraschallgeber wird durch eine fallende Flanke an GPIO14 getriggert. Er sendet dann nach ca. 250ms einen 40kHz-Burst mit 200ms Dauer aus. Danach geht der Echo-Ausgang unmittelbar auf HIGH-Pegel. Wird dann ein Echo empfangen, fällt der Ausgangspegel auf LOW. Der Controller muss die Laufzeit ermitteln und durch zwei teilen, weil der Weg ja zweimal durchlaufen wird.

```
taste=Pin(0,Pin.IN,Pin.PULL_UP)
red=Pin(2,Pin.OUT,value=0)
trigger=Pin(14,Pin.OUT, value=1)
echo=Pin(13,Pin.IN)
limit=10
temperatur=20
```

Nachdem der Zähler 10 erreicht hat, wird eine Mail getriggert. Vorab deklarieren wir schon einmal die Variable **temperatur**, weil die Schallgeschwindigkeit Temperaturabhängig ist, muss sie berücksichtigt werden. Der SHT21 sagt sie uns später.

Das [Dictionary](#) **connectstatus** übersetzt die numerischen Werte für den Verbindungsstatus in Klartext.

```
connectStatus = {
    1000: "STAT_IDLE",
    1001: "STAT_CONNECTING",
    1010: "STAT_GOT_IP",
    202:  "STAT_WRONG_PASSWORD",
    201:  "NO AP FOUND",
    5:    "UNKNOWN",
    0: "STAT_IDLE",
    1: "STAT_CONNECTING",
    5: "STAT_GOT_IP",
    2: "STAT_WRONG_PASSWORD",
    3: "NO AP FOUND",
    4: "STAT_CONNECT_FAIL",
}
```

Mit der Funktion **hexMac()** erfahren wir die MAC-Adresse des Station-Interfaces des Controllers im Klartext. Diese muss im Router eingetragen werden, sonst verweigert dieser dem Controller den Zugang. Meistens geschieht der Eintrag über das Menü WLAN – Sicherheit. Das genaue Vorgehen verrät das Handbuch des Routers.

```
# ***** Funktionen definieren *****
def hexMac(byteMac):
    """
    Die Funktion hexMAC nimmt die MAC-Adresse im Bytecode
    entgegen und bildet daraus einen String fuer die Rueckgabe
    """
    macString = ""
```

```

for i in range(0,len(byteMac)):      # Fuer alle Bytewerte
    macString += hex(byteMac[i])[2:]  # ab Position 2 bis Ende
    if i <len(byteMac)-1 :            # Trennzeichen
        macString += "-"
return macString

```

Wir tasten das Bytes-Objekt, das uns der Funktionsaufruf **nic.config('mac')** liefert, Zeichen für Zeichen ab, machen daraus eine Hexadezimalzahl und bauen daraus den String auf, den die Funktion zurückgibt.

```

>>> nic.config('mac')
b'\xf0\x08\xd1\xd1m\x14'

```

```

>>> hex(m[0])
'0xf0'

```

```

>>> hex(m[0]) [2:]
'0xf0'

```

```

>>> hex(m[0])[2:]
'f0'

```

Die steigende Flanke an **echo** triggert einen Interrupt, also eine Programmunterbrechung. Das Programm kann gerade irgendwo sein, wenn der Pegel an GPIO13 steigt. Jetzt muss jedenfalls erst einmal die Systemzeit in Microsekunden genommen werden. Weil eine ISR (aka Interrupt Service Routine) keinen Wert zurückgeben kann, wie eine normale Funktion, muss die Zeitmarke über eine globale Variable, hier **start**, ans das Hauptprogramm gemeldet werden. Das ist nur möglich, wenn die Variable in der Funktion als **global** deklariert wird.

Dann setzen wir für den nächsten Durchlauf den Triggerausgang wieder auf HIGH. und ändern den Handler so ab, dass der Eingang auf die fallende Flanke des Echo-impulses reagieren kann. Der Handler ist die Routine, die im Falle der Unterbrechungsanforderung ausgeführt werden soll. Zuletzt wird noch die LED eingeschaltet. Das reicht, mehr sollte man hier nicht reinpacken, ISRs sollen möglichst kurz gehalten werden.

```

def startZeit(pin):
    global start
    start=ticks_us()
    trigger.on()
    echo.irq(handler=stoppZeit,trigger=Pin.IRQ_FALLING)
    red.on()

```

Der nächste Interrupt, ausgelöst durch die fallende Flanke des Echo-Impulses, wird durch **stoppZeit()** bedient. Die global deklarierten Variablen **ende** und **fertig** liefern die gewünschten Informationen ans Hauptprogramm zurück. Die IRQ-Behandlung wird deaktiviert und die LED ausgeschaltet.


```
def stoppZeit(pin):
    global ende, fertig
    ende=ticks_us()
    fertig=True
    echo.irq(handler=None, trigger=Pin.IRQ_RISING)
    red.off()
```

Die Funktion **entfernung()** berechnet aus dem übergebenen Argument in **runtime** die Entfernung mit Hilfe der Schallgeschwindigkeit. Dabei wird berücksichtigt, dass die Schallgeschwindigkeit mit der Temperatur ansteigt und zwar um 0,6m/s. Sofern die Laufzeit weniger als 15ms (entspricht 2,56m @20°C) beträgt, ist der Messwert weitgehend zuverlässig. Bei mehr als 15ms Laufzeit wird 999 als Fehlercode zurückgegeben.

```
def entfernung(runtime):
    if runtime <= 15:
        return int((331.5 + 0.6*temperatur)*runtime/2)
    else:
        return 999
```

Mit dem Aufruf der Funktion **messen()** wird ein Messvorgang getriggert. Wieder werden Variablen als global bekanntgemacht. **fertig** setzen wir auf **False** und stellen für die Bedienung der Unterbrechungsanforderung (IRQ = Interrupt Request) als ISR **startZeit()** ein. Mit **trigger.off()** geht der Ausgang auf LOW, und der HC-SR05 beginnt seinen Zyklus. Der Timer **expired()** wird auf 200ms gestellt. Er bricht den Messvorgang ab, wenn der HC-SR04 kein Echosignal empfängt. Dass ein Messvorgang gestartet wurde, sagen wir der Hauptschleife mit der Variablen **triggered**.

In der Funktion **senden()** senden wir eine Nachricht an IFTTT und nehmen die Antwort des Servers entgegen. Die Nachricht mit der URL wird mit der Methode POST abgeschickt. Mit im Gepäck befindet sich das Dictionary {'value1': str(cnt)}, das mit dem Parameter json überreicht wird. In gleicher Weise wird der Headertyp übergeben. Über den json-Code werden Sie in einer späteren Folge noch etwas erfahren.

```
def senden(cnt):
    resp = requests.post(iftttUrl,
                        json={'value1': str(cnt)},
                        headers={'Content-Type':
'application/json'})
    if resp.status_code == 200:
        print("Transfer OK")
    else:
        print("Transfer-Fehler")
    resp.close()
    gc.collect()
```

requests.post() gibt ein response-Objekt zurück. Im Attribut **status_code** befindet sich der Rückgabe-Code des Servers, 200 bedeutet fehlerfrei angekommen.
resp.close() schließt die Unterhaltung und **gc.collect()** räumt den Speicher auf.

Der Controller stellt jetzt die Verbindung mit dem Router her. Dazu wird erst einmal das Accesspoint-Interface ausgeschaltet, wenn es denn überhaupt an war. Das ist beim ESP8266 manchmal nötig. Dann schalten wir das Station-Interface ein, der Controller arbeitet ja als Client. Die Schaltsekunde danach ist wichtig und vermeidet interne WLAN-Fehler.

```
# ***** Bootsequenz *****  
#  
nic=network.WLAN(network.AP_IF)  
nic.active(False)  
  
nic = network.WLAN(network.STA_IF) # erzeugt WiFi-Objekt nic  
nic.active(True)                  # nic einschalten  
sleep(1)  
  
MAC = nic.config('mac') # binaere MAC-Adresse abrufen und  
myMac=hexMac(MAC)      # in eine Hexziffernfolge umgewandelt  
print("STATION MAC: \t"+myMac+"\n") # ausgeben
```

Wir lesen die MAC-Adresse aus, lassen sie in Klartext umformen und im Terminal ausgeben.

```
if not nic.isconnected():  
    nic.connect(mySid, myPass)  
    print("Status: ", nic.isconnected())  
    d.writeAt("WLAN connecting",0,1)  
    points="....."  
    n=1  
    while nic.status() != network.STAT_GOT_IP:  
        print(".",end='')  
        d.writeAt(points[0:n],0,2)  
        n+=1  
        sleep(1)
```

Wenn die Schnittstelle noch keine Verbindung zum Router hat, stellen wir mit **connect()** eine her. Dabei werden SSID und Passwort übertragen. Der Status wird abgefragt und ausgegeben. Solange wir vom [DHCP](#)-Server auf dem Router noch keine IP-Adresse bekommen haben, wird im Sekundenabstand ein Punkt ausgegeben. Das sollte nicht länger als 4 bis 5 Sekunden dauern.

Dann fragen wir den Status ab und lassen uns die Verbindungsdaten, IP-Adresse, Netzwerkmaske und Gateway, mitteilen.

```
print("\nStatus: ",connectStatus[nic.status()])
```

```

d.clearAll()
STAconf = nic.ifconfig()
print("STA-IP:\t\t", STAconf[0], "\nSTA-NETMASK:\t", STAconf[1], \
      "\nSTA-GATEWAY:\t", STAconf[2] , sep=' ')
d.writeAt(STAconf[0],0,0)
d.writeAt(STAconf[1],0,1)
d.writeAt(STAconf[2],0,2)
sleep(3)
d.clearAll()
d.writeAt("AMBIANT DATA",2,0)

```

Ein paar Startwerte werden gesetzt, bevor es in die Hauptschleife geht.

```

triggered=False
ende=start=0
alt=neu = 0
personen=0
expired=TimeOutMs(0)
nextScan=TimeOutMs(20)
messen()
count=0

```

In der Hauptschleife warten wir auf das Eintreten verschiedener Ereignisse. Wurde eine Messung getriggert, aber kein Echo festgestellt, dann muss die Messung als fehlerhaft abgebrochen werden. Wir setzen **runtime**, **ende** und **start** auf 0, schalten die IRQ-Behandlung zuerst definitiv aus und dann auf Start. Die Triggerleitung legen wir auf HIGH und setzen **fertig** auf **False**.

```

if triggered and expired():
    runtime=ende=start=0
    echo.irq(handler=None,trigger=Pin.IRQ_RISING)
    echo.irq(handler=startZeit,trigger=Pin.IRQ_RISING)
    trigger.on()
    fertig=False

```

Wenn eine Messung fertig ist, muss festgestellt werden, ob der Entfernungswert im richtigen Bereich liegt. Zuerst setzen wir aber **fertig** auf **False** und berechnen dann die Laufzeit in Millisekunden. Diesen Wert geben wir an die Funktion **entfernung()** weiter, die uns die Distanz zum Messobjekt liefert. In der Testphase können wir die Werte im Display ausgeben lassen.

```

if fertig:
    fertig=False
    runtime = (ende - start)/1000
    distance=entfernung(runtime)
#    print("runtime",runtime, "distance",distance)
    if distance < 300: # Person im Erfassungskegel
        neu=1
    elif distance > 700:

```

```
neu=0
triggered=False
```

Ist die Distanz kleiner als 30cm, dann befindet sich eine Person im Schallkegel, verlässt sie diesen, setzen wir ab 70cm die Variable **neu** auf 0.

Nur wenn die Person den Schallkegel verlassen hat, zählen wir um eins weiter. Ist **limit** erreicht, triggern wir eine e-Mail von IFTTT und setzen **limit** um 10 höher. Stets lassen wir uns die Personenanzahl und das aktuelle Limit im Terminal und am Display ausgeben.

```
if alt==1 and neu==0:
    count+=1
    if count == limit:
        senden(count)
        limit+=10
#    print(count, limit,)
d.clearFT(0,1,15,2,False)
d.writeAt("Personen:{}".format(count),0,1,False)
d.writeAt("Limit: {}".format(limit),0,2)
```

Danach setzen wir grundsätzlich alt auf neu.

Mit dem Timer **nextScan()** können wir die zeitliche Abfolge von Messvorgängen steuern. Das Messintervall muss länger als 20ms sein. Wir lesen die Rohtemperaturwerte vom SHT21 ein und lassen diese in die aktuelle Celsius-Temperatur umrechnen. Danach triggern wir eine neue Messung und starten den Timer neu.

```
if nextScan():
    sht.readTemperatureRaw()
    temperatur=sht.calcTemperature()
    messen()
    nextScan=TimeOutMs(100)
```

Das letzte Ereignis, das eintreten kann, ist eine gedrückte Abbruchtaste. In diesem Fall schalten wir die LED aus und beenden das Programm.

Im Produktionsbetrieb muss der Controller autonom ohne PC und USB-Kabel starten. Damit er das kann, speichern Sie nach beendeter Testphase das Programm [ifttt.py](#) als **main.py** direkt im Flash des Controllers ab. Das erreichen Sie am schnellsten mit **Strg + Shift +S**, im Dialogfenster wählen Sie **MicroPython device**.

Ausblick

Mit IFTTT können Sie nur reine Text-Nachrichten bekommen. In der nächsten Folge zeige ich Ihnen, wie man eine grafische Aufbereitung von Messwerten bekommen kann. Wir werden einen BME280 für Messung von Temperatur, Luftdruck und relativer Luftfeuchte benutzen sowie einen BH1750 zur Ermittlung der Helligkeit.

In einer weiteren Folge werden wir interaktiv. Über **Telegram** steuern wir einen ESP32 und fragen zum einen wieder dieselben Sensoren ab, zum anderen schalten wir eine LED über den Bot.

Wer statt **Telegram** lieber **Whatsapp** benutzen möchte, kann sich Messwerte und Zustände auch auf diesem Weg zustellen lassen. Wie das geht, erkläre ich in der letzten Folge zum Thema Messaging.

Ganz nebenbei erfahren Sie auch etwas über die Ähnlichkeit zwischen MicroPythons Dictionarys und dem JSON-Code sowie über Einsatz von Callback-Routinen.

Neugierig geworden? Dann bleiben Sie dran!