



WETTERSTATION

Adnan Wase & Toma Sapoundjiev



26. MAI 2025

TGM

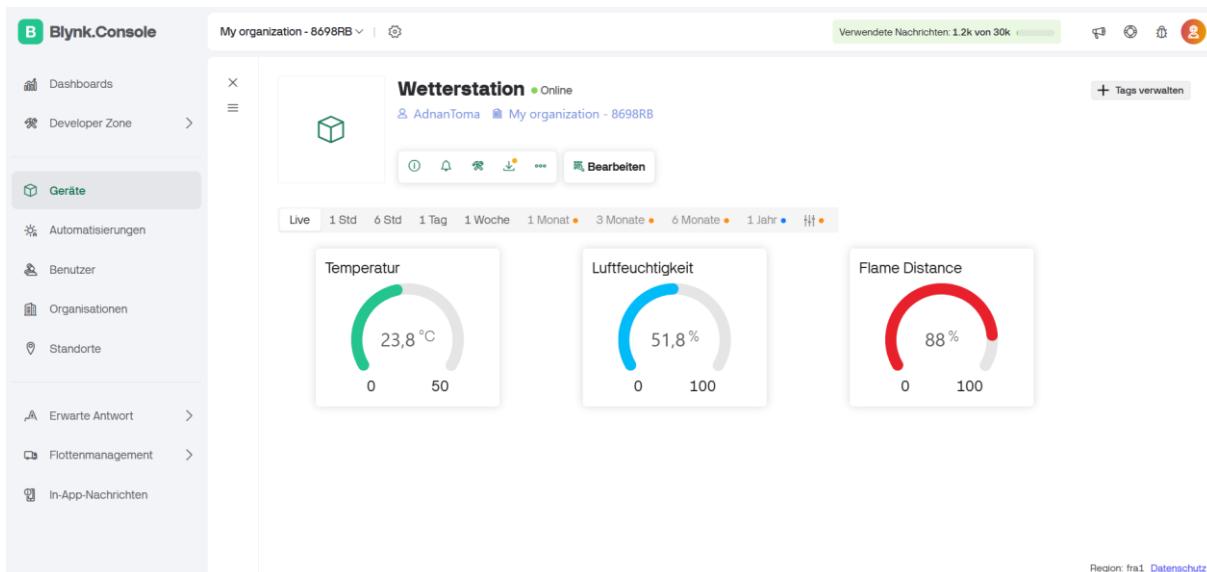
2cHIT

Inhaltsverzeichnis

Einführung	2
Projektbeschreibung	3
Theorie	3
Arbeitsschritt	3
Zusammenfassung.....	3
Literaturverzeichnis.....	4

Einführung

Im Rahmen des Projekts wurde eine einfache Wetterstation umgesetzt, die Temperatur und Luftfeuchtigkeit erfasst. Zudem kann die Wetterstation auch Flammen messen. Die erfassten Messwerte werden sowohl auf einem Display angezeigt als auch über ein Webinterface zur Verfügung gestellt. Zusätzlich werden die Werte auf dem Webinterface auch in einem Graph dargestellt. Der Graph zeigt dann die verschiedenen Messwerte in einem sinnvollen Zeitintervall an. Der Status der Wetterstation wird über die im Microcontroller verbaute LED angezeigt.



Projektbeschreibung

Im Projekt „Wetterstation 2025“ wurde eine umfangreiche Mess- und Visualisierungseinheit mit einem ESP32 C3 DevKit entwickelt. Die Station erfasst in Echtzeit die Temperatur, Luftfeuchtigkeit und den Flammenstatus. Die Daten werden sowohl auf einem OLED-Display angezeigt als auch über ein modernes Webinterface mit Live-Graphen, die Blynk-App und automatisierte Discord-Benachrichtigungen bereitgestellt.

Das System erkennt verschiedene Zustände wie hohe Temperatur oder Verbindungsprobleme und signalisiert diese über eine RGB-Status-LED. Die Benutzeroberfläche ist vollständig webbasiert, live aktualisiert (AJAX) und zeigt neben den aktuellen Werten auch historische Verläufe und Durchschnittswerte.

Zusätzlich wurde eine Discord-Webhook-Funktion implementiert, die alle 4 Minuten automatisch Statusmeldungen verschickt. Die Verbindung wird über WiFiManager konfiguriert, der bei Bedarf einen Access Point startet.

Die Lösung ist modular, cloudfähig, stromsparend und bereit für den Einsatz im Bereich IoT, Schule oder Raumüberwachung.

Theorie

ESP32 C3

Der ESP32 C3 ist das Herzstück der Wetterstation. Dieser Mikrocontroller besitzt integriertes WLAN und eignet sich daher besonders gut für IoT-Projekte. Er hat genug Rechenleistung, um mehrere Aufgaben gleichzeitig zu übernehmen: Daten messen, Webseiten bereitstellen, LEDs steuern und über das Internet kommunizieren. Durch seine kompakte Bauweise passt er in viele Projekte mit wenig Platz.

DHT11 Sensor

Der DHT11 ist ein Sensor, der Temperatur und Luftfeuchtigkeit misst. Er wird über einen einzigen digitalen Pin an den Mikrocontroller angeschlossen und liefert alle zwei Sekunden neue Werte. Zwar ist er nicht der genaueste Sensor, aber für Raumüberwachung reicht er vollkommen aus. Die Ansteuerung erfolgt über eine passende Bibliothek, die das Auslesen der Werte einfach macht.

Flammensensor

Der Flammensensor erkennt infrarotes Licht, das typischerweise von Flammen ausgesendet wird. Je näher eine Flamme am Sensor ist, desto höher ist der ausgelesene Wert. Dieser wird im Projekt in einen Prozentwert umgerechnet, sodass man leicht erkennen kann, wie stark eine Flamme in der Nähe ist. Die Ausgabe erfolgt zusätzlich in Form einer verbalen Beschreibung wie „Keine Flamme“ oder „Sehr nah“.

OLED-Display

Das OLED-Display dient der lokalen Anzeige der aktuellen Messwerte. Es hat eine Auflösung von 128x64 Pixeln und wird über das I2C-Protokoll angesteuert. Die Darstellung erfolgt mit Hilfe der Adafruit-Bibliotheken. Auf dem Display sieht man die aktuelle Temperatur, Luftfeuchtigkeit und den Flammenstatus in Echtzeit. Besonders praktisch ist das Display, wenn kein Internet verfügbar ist.

Webserver

Der ESP32 stellt eine eigene Webseite zur Verfügung, die über das lokale WLAN erreichbar ist. Diese Webseite zeigt nicht nur die aktuellen Messwerte, sondern auch zwei dynamische Diagramme – eines für Temperatur und eines für Luftfeuchtigkeit. Die Webseite lädt sich nicht ständig neu, sondern aktualisiert sich automatisch im Hintergrund. Das geschieht mithilfe von JavaScript und sogenannten Fetch-Anfragen.

Status-LED

Die RGB-LED zeigt an, in welchem Zustand sich die Wetterstation befindet. Leuchtet sie blau, läuft alles normal. Eine orange Farbe bedeutet, dass die Temperatur zu hoch ist. Wenn kein WLAN verbunden ist, blinkt die LED rot. Dadurch kann man auf einen Blick erkennen, ob es ein Problem gibt. Die Ansteuerung erfolgt über die NeoPixel-Bibliothek, die speziell für solche RGB-LEDs gedacht ist.

WLAN-Verbindung

Für die Einrichtung der WLAN-Verbindung wird WiFiManager verwendet. Das ist eine Bibliothek, die es ermöglicht, die WLAN-Zugangsdaten zur Laufzeit einzugeben. Wenn keine gespeicherten Zugangsdaten vorhanden sind oder die Verbindung fehlschlägt, startet der ESP32 einen eigenen Access Point. Man kann sich dann mit diesem verbinden, eine Konfigurationsseite öffnen und das gewünschte WLAN auswählen.

Blynk

Blynk ist eine Plattform, die es erlaubt, Sensordaten auf dem Smartphone anzuzeigen. Die Werte werden über das Internet an einen Server von Blynk gesendet, von wo aus sie in einer App dargestellt werden. Im Projekt werden Temperatur, Luftfeuchtigkeit und der Flammenwert in Prozent an drei virtuelle Pins übermittelt. Die Einrichtung erfolgt mithilfe eines sogenannten Auth-Tokens, den man in der App erstellt.

Discord Webhook

Die Wetterstation sendet regelmäßig Nachrichten an einen Discord-Channel. Diese Nachrichten enthalten aktuelle Daten wie Temperatur, Feuchtigkeit und Flammenstatus. Die Übertragung erfolgt über einen sogenannten Webhook – eine spezielle URL, die Discord bereitstellt. Der ESP32 sendet alle vier Minuten einen HTTP-Request an diese Adresse, der dann als Nachricht im Channel erscheint.

Datenspeicherung und Durchschnittswerte

Alle gemessenen Daten werden in einem Zwischenspeicher auf dem Mikrocontroller abgelegt. Dadurch ist es möglich, Durchschnittswerte über bestimmte Zeiträume zu berechnen – zum Beispiel über die letzten fünf, fünfzehn oder sechzig Minuten. Diese Werte werden im Webinterface übersichtlich angezeigt, damit man schnell erkennen kann, ob sich die Umgebungstemperatur oder -feuchtigkeit stark verändert.

Arbeitsschritte

Aufbau

Zu Beginn unseres Projekts haben wir die komplette Hardware-Schaltung geplant und aufgebaut. Als Steuerzentrale kam ein ESP32 C3 DevKit zum Einsatz. An diesen Mikrocontroller wurden zwei Sensoren angeschlossen: ein DHT11, der Temperatur und Luftfeuchtigkeit misst, sowie ein Flammensensor, der auf Licht- bzw. Flammeneinflüsse reagiert. Beide Sensoren wurden auf dem Breadboard platziert und über digitale Pins mit dem ESP32 verbunden.

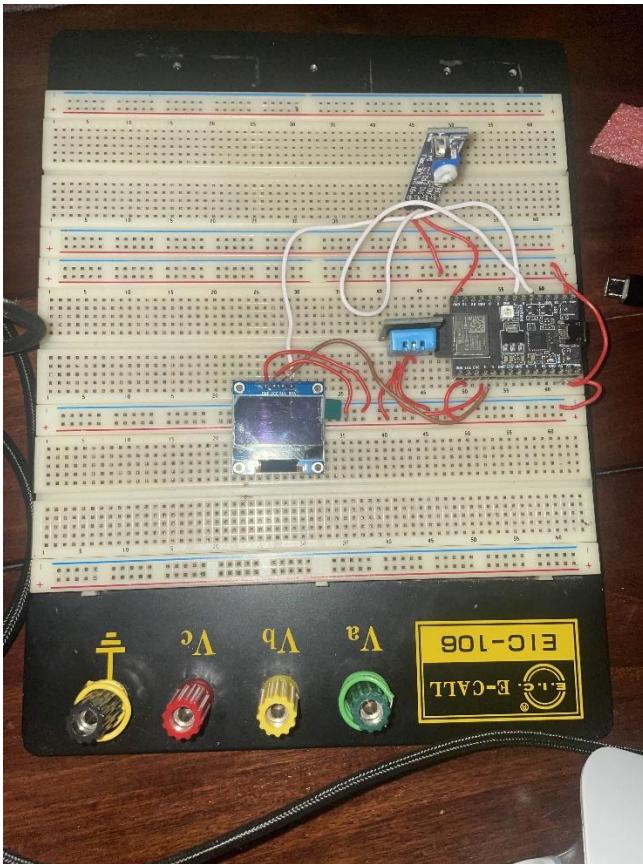
Für die grafische Darstellung der Messdaten haben wir ein OLED-Display vom Typ SSD1306 verwendet. Dieses Display kommuniziert über das I²C-Protokoll. Die dafür notwendigen Leitungen – SDA und SCL – wurden auf GPIO 7 und GPIO 3 gelegt und im Code entsprechend deklariert. Nach erfolgreicher Initialisierung zeigte das Display zuverlässig die aktuellen Sensordaten an.

Zur optischen Rückmeldung des Systemzustands verwendeten wir die eingebaute RGB-LED des ESP32, welche auf GPIO 8 liegt. Diese LED lässt sich mit Hilfe der Adafruit NeoPixel-Bibliothek steuern und wird im Code je nach Zustand unterschiedlich eingefärbt: Blau für normalen Betrieb, Orange bei hoher Temperatur und blinkend Rot bei Verbindungsproblemen.

Die Stromversorgung erfolgte über USB, während der Zugang zum WLAN mit Hilfe von WiFiManager realisiert wurde. Sollte keine bekannte Verbindung bestehen, startet der ESP32 automatisch im Access-Point-Modus und erlaubt die Konfiguration über eine lokale Webseite.

Der Aufbau war kompakt und effizient – alle Bauteile konnten problemlos auf einem Steckbrett untergebracht werden. Nachdem die Schaltung erfolgreich stand, ging es an die Softwareentwicklung. Schritt für Schritt kamen weitere Funktionen hinzu: das Webinterface, die Statusanzeige per LED, die Anbindung an die Blynk-App und schließlich auch die automatisierten Benachrichtigungen über Discord.

Wetterstation



Um den Aufbau zu testen, mussten wir erstmal folgenden Code schreiben:

```
#include "DHT.h"  
#include <Wire.h>  
#include <Adafruit_GFX.h>  
#include <Adafruit_SSD1306.h>  
  
// DHT11 Konfiguration  
#define DHT11_PIN 2  
DHT dht11(DHT11_PIN, DHT11);
```

```
// I2C-Konfiguration  
#define I2C_SDA 7  
#define I2C_SCL 3
```

```
// OLED-Konfiguration
#define SCREEN_WIDTH 128
#define SCREEN_HEIGHT 64
Adafruit_SSD1306 display(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire, -1);

// Flammensensor
const int flameSensorPin = 4;
const int sensorMin = 0;
const int sensorMax = 1024;

void setup() {
    Serial.begin(9600);
    dht11.begin();

    // I2C starten
    Wire.begin(I2C_SDA, I2C_SCL);

    // OLED initialisieren
    if (!display.begin(SSD1306_SWITCHCAPVCC, 0x3C)) {
        Serial.println(F("SSD1306 allocation failed"));
        for ();}
    }

    display.clearDisplay();
    display.setTextSize(1);
    display.setTextColor(SSD1306_WHITE);
    display.setCursor(0, 0);
    display.println("System startet...");
    display.display();
    delay(2000);
```

}

```
void loop() {
    delay(2000); // alle 2 Sekunden neue Messung

    // === DHT11 Messwerte ===
    float humi = dht11.readHumidity();
    float tempC = dht11.readTemperature();
    float tempF = dht11.readTemperature(true);

    // === Flammensensor Messung ===
    int sensorReading = analogRead(flameSensorPin);
    int range = map(sensorReading, sensorMin, sensorMax, 0, 3);

    String flameStatus;
    switch (range) {
        case 0:
            flameStatus = "Keine Flamme";
            break;
        case 1:
            flameStatus = "Entfernte Flamme";
            break;
        case 2:
            flameStatus = "Nahe Flamme";
            break;
        case 3:
            flameStatus = "Sehr nahe oder stark";
            break;
        default:
            flameStatus = "Unbekannt";
            break;
    }
}
```

}

```
// === Serielle Ausgabe ===  
if (isnan(humi) || isnan(tempC) || isnan(tempF)) {  
    Serial.println("Fehler beim Lesen des DHT11!");  
} else {  
    Serial.print("Temp: ");  
    Serial.print(tempC);  
    Serial.print(" C | Feucht: ");  
    Serial.print(humi);  
    Serial.print(" % | Flamme: ");  
    Serial.println(flameStatus);  
}
```

```
// === OLED Anzeige ===  
display.clearDisplay();  
display.setCursor(0, 0);  
  
if (isnan(humi) || isnan(tempC) || isnan(tempF)) {  
    display.println("Sensorfehler!");  
} else {  
    display.print("Temp: ");  
    display.print(tempC);  
    display.println(" C");  
    display.print("Feucht: ");  
    display.print(humi);  
    display.println(" %");  
    display.print("Flamme: ");  
    display.println(flameStatus);  
}  
display.display();
```

}

Erklärung zum Code:

Bibliotheken

```
cpp
KopierenBearbeiten
#include "DHT.h"
#include <Wire.h>
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>
```

Diese Zeilen binden externe Bibliotheken ein:

- **DHT.h**: Für den Zugriff auf den DHT11-Sensor (Temperatur und Luftfeuchtigkeit).
- **Wire.h**: Für die Kommunikation über das I²C-Protokoll (z. B. mit dem Display).
- **Adafruit_GFX** und **Adafruit_SSD1306**: Für die Steuerung des OLED-Displays (Text, Linien usw.).

Sensor- und Display-Konfiguration

```
cpp
KopierenBearbeiten
#define DHT11_PIN 2
DHT dht11(DHT11_PIN, DHT11);
```

Der DHT11-Sensor wird an **Pin 2** angeschlossen und initialisiert.

```
cpp
KopierenBearbeiten
#define I2C_SDA 7
#define I2C_SCL 3
```

Dies sind die manuellen Pin-Zuweisungen für die **I²C-Schnittstelle**, die das Display benötigt.

```
cpp
KopierenBearbeiten
#define SCREEN_WIDTH 128
#define SCREEN_HEIGHT 64
Adafruit_SSD1306 display(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire, -1);
```

Initialisierung des OLED-Displays mit der Standardauflösung **128x64 Pixel**.

```
cpp
KopierenBearbeiten
const int flameSensorPin = 4;
const int sensorMin = 0;
const int sensorMax = 1024;
```

Der **Flammensensor** ist mit **Pin 4** verbunden. `sensorMin` und `sensorMax` dienen zum Skalieren der Messwerte.

setup()

```
cpp
KopierenBearbeiten
void setup() {
    Serial.begin(9600);
    dht11.begin();
    Wire.begin(I2C_SDA, I2C_SCL);
```

Wetterstation

Der serielle Monitor wird aktiviert, der DHT11 gestartet, und die I²C-Kommunikation mit den definierten Pins begonnen.

```
cpp
KopierenBearbeiten
if (!display.begin(SSD1306_SWITCHCAPVCC, 0x3C)) {
    Serial.println(F("SSD1306 allocation failed"));
    for (;;) {
}
```

Das Display wird initialisiert. Falls das fehlschlägt, bleibt das Programm in einer Dauerschleife stehen.

```
cpp
KopierenBearbeiten
display.clearDisplay();
display.setTextSize(1);
display.setTextColor(SSD1306_WHITE);
display.setCursor(0, 0);
display.println("System startet...");
display.display();
delay(2000);
}
```

Der Text „System startet...“ wird auf dem OLED-Display angezeigt. Danach wartet das System 2 Sekunden.

loop()

```
cpp
KopierenBearbeiten
delay(2000);
```

Warte 2 Sekunden zwischen jedem Messzyklus.

```
cpp
KopierenBearbeiten
float humi = dht11.readHumidity();
float tempC = dht11.readTemperature();
float tempF = dht11.readTemperature(true);
```

Temperatur und Luftfeuchtigkeit werden vom DHT11 ausgelesen. Die Temperatur wird sowohl in **Celsius** als auch in **Fahrenheit** gelesen (obwohl Fahrenheit später nicht verwendet wird).

Flammensensor auswerten

```
cpp
KopierenBearbeiten
int sensorReading = analogRead(flameSensorPin);
int range = map(sensorReading, sensorMin, sensorMax, 0, 3);
```

Der analoge Wert des Flammensensors wird in eine Zahl zwischen 0 und 3 umgerechnet.

```
cpp
KopierenBearbeiten
String flameStatus;
switch (range) {
    case 0: flameStatus = "Keine Flamme"; break;
    case 1: flameStatus = "Entfernte Flamme"; break;
    case 2: flameStatus = "Nahe Flamme"; break;
    case 3: flameStatus = "Sehr nahe oder stark"; break;
    default: flameStatus = "Unbekannt"; break;
```

}

Abhängig vom berechneten Bereich (`range`) wird der Flammenstatus als Text gespeichert.

Ausgabe im Serial Monitor

```
cpp
KopierenBearbeiten
if (isnan(humi) || isnan(tempC) || isnan(tempF)) {
    Serial.println("Fehler beim Lesen des DHT11!");
} else {
    Serial.print("Temp: ");
    Serial.print(tempC);
    Serial.print(" C | Feucht: ");
    Serial.print(humi);
    Serial.print(" % | Flamme: ");
    Serial.println(flameStatus);
}
```

Falls der Sensor keine gültigen Werte liefert, wird eine Fehlermeldung ausgegeben. Ansonsten erscheinen die Sensordaten im **seriellen Monitor**.

Anzeige auf dem OLED-Display

```
cpp
KopierenBearbeiten
display.clearDisplay();
display.setCursor(0, 0);
```

Das Display wird gelöscht, der Cursor wird auf den Anfang gesetzt.

```
cpp
KopierenBearbeiten
if (isnan(humi) || isnan(tempC) || isnan(tempF)) {
    display.println("Sensorfehler!");
} else {
    display.print("Temp: ");
    display.print(tempC);
    display.println(" C");

    display.print("Feucht: ");
    display.print(humi);
    display.println(" %");

    display.print("Flamme: ");
    display.println(flameStatus);
}
```

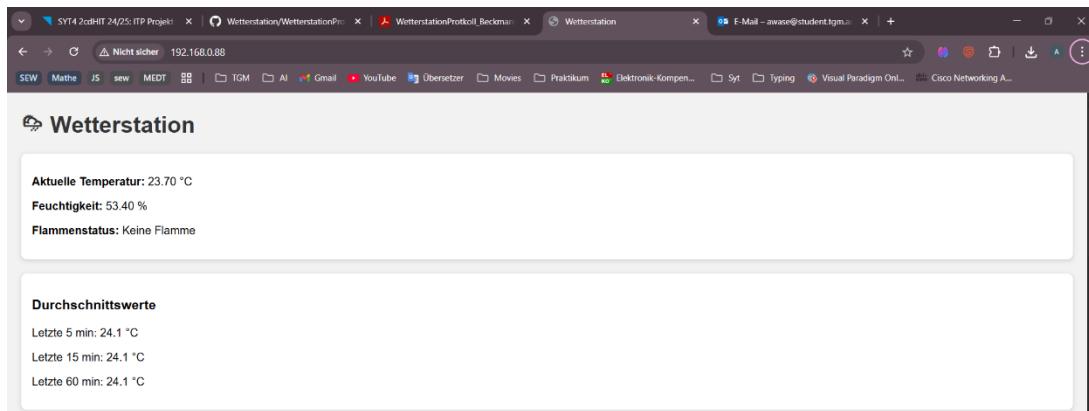
Die gleichen Daten wie im seriellen Monitor werden nun auch auf dem **OLED-Display** angezeigt.

```
cpp
KopierenBearbeiten
display.display();
```

Abschließend werden die geschriebenen Inhalte auf dem Display angezeigt.

Wie das Webinterface funktioniert

Das Webinterface ist eine einfache Webseite, die auf dem ESP32 selbst gehostet wird. Das bedeutet, der ESP32 fungiert nicht nur als Sensor-Controller, sondern gleichzeitig als **kleiner Webserver**. Dadurch kann jedes Gerät im selben WLAN (z. B. Smartphone oder Laptop) über den Browser auf die Live-Daten zugreifen.



Einbindung im Code

Die Einbindung beginnt mit der Zeile:

```
cpp
KopierenBearbeiten
#include <WebServer.h>
```

Damit nutzt du die WebServer-Bibliothek von Arduino, die es dir erlaubt, HTTP-Anfragen zu empfangen und darauf mit HTML-Inhalten zu antworten.

In der `setup()`-Funktion definierst du mit:

```
cpp
KopierenBearbeiten
server.on("/", []() {
    server.send(200, "text/html", webContent);
});
```

dass beim Aufruf der Hauptseite („/“) die HTML-Seite ausgeliefert wird, die über die Funktion `generateWebPage()` im Hintergrund dynamisch erzeugt wird.

Am Ende von `setup()` wird mit

```
cpp
KopierenBearbeiten
server.begin();
```

der Webserver gestartet.

Im `loop()` rufst du

```
cpp
KopierenBearbeiten
server.handleClient();
```

auf, damit der Server jederzeit auf neue Anfragen reagieren kann.

HTML-Generierung im Code

Die Seite wird im Code mit der Funktion `generateWebPage(...)` als ein großer **HTML-String** erstellt. Sie enthält:

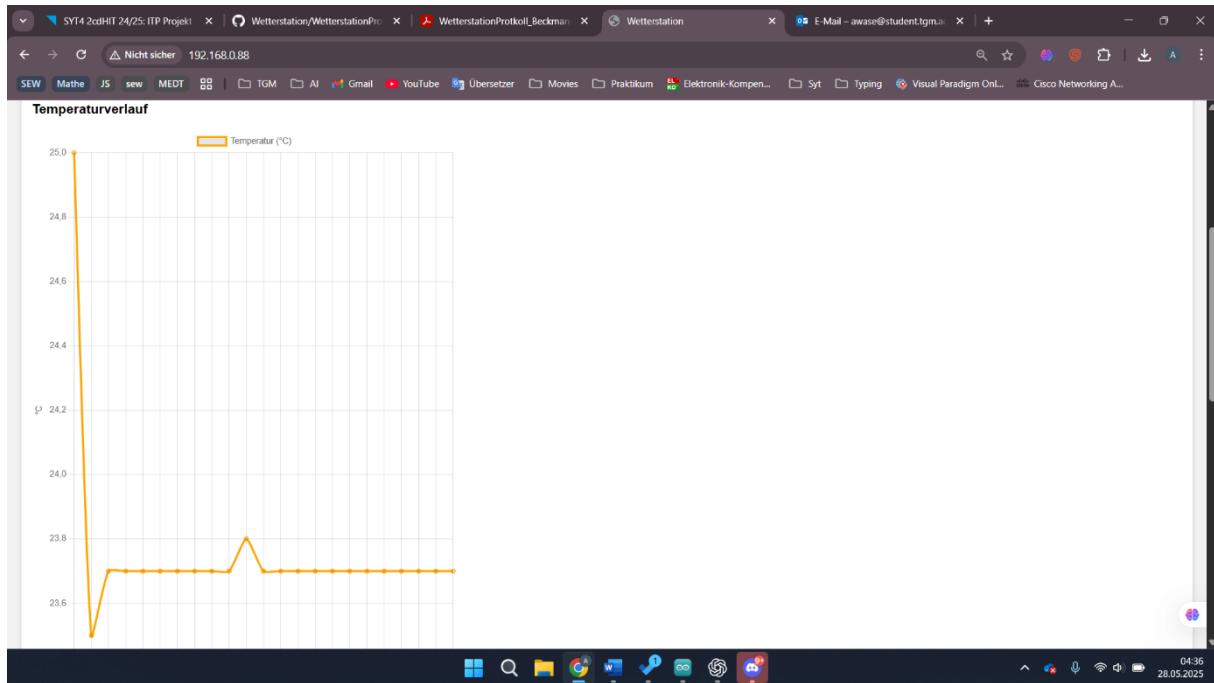
- eine Überschrift (z. B. „Wetterstation“),

Wetterstation

- aktuelle Werte für Temperatur, Luftfeuchtigkeit und Flammenstatus,
- Durchschnittswerte über verschiedene Zeiträume,
- zwei <canvas>-Elemente für die **Grafiken** (Temperaturverlauf und Luftfeuchtigkeitsverlauf),
- und ein eingebettetes JavaScript-Skript mit Chart.js.

Live-Grafiken mit Chart.js

Die Diagramme auf der Webseite werden mit der JavaScript-Bibliothek **Chart.js** dargestellt. Diese wird über das Internet eingebunden:



Wetterstation



```
html
KopierenBearbeiten
<script src='https://cdn.jsdelivr.net/npm/chart.js'></script>
```

Dann wird im Code ein Canvas-Element für die Temperatur und eines für die Luftfeuchtigkeit eingebaut:

```
html
KopierenBearbeiten
<canvas id='chartTemp'></canvas>
<canvas id='chartHum'></canvas>
```

Im Skript-Teil wird aus den gesammelten historischen Daten (gespeichert in `history[]`) eine Datenreihe erstellt. Diese enthält:

- die **x-Achse** (Zeit in Minuten),
- die **y-Achse** (Temperatur- bzw. Feuchtigkeitswerte).

Diese Werte werden direkt in das HTML als JavaScript-Code eingebaut und beim Seitenladen mit Chart.js gerendert. Beispiel:

```
js
KopierenBearbeiten
const chartT = new Chart(ctxT, {
  type: 'line',
  data: {
    labels: ['0', '1', '2', ...],
    datasets: [
      {
        label: 'Temperatur (°C)',
        data: [25.1, 25.3, 25.2, ...],
        borderColor: 'orange',
        fill: false
      }
    ],
    ...
  }
});
```

Dasselbe passiert für die Feuchtigkeitskurve mit einem eigenen Diagramm und blauer Linie.

Automatische Aktualisierung

Damit sich die Seite automatisch aktualisiert, wird alle 10 Sekunden ein JavaScript-Befehl ausgeführt:

```
js
KopierenBearbeiten
setInterval(fetchData, 10000);
```

Die `fetchData()`-Funktion ruft dann neue Daten vom ESP32 ab, indem sie die Seite neu lädt oder über eine `/data`-Route JSON-Daten abfragt (je nach Implementierung). So wird der Inhalt der Graphen laufend aktualisiert, ohne dass der Nutzer manuell neu laden muss.

Status-LED

Um den Zustand der Wetterstation auch visuell anzuzeigen, haben wir die **eingebaute RGB-LED** des ESP32 C3 genutzt. Diese LED befindet sich auf dem Board und ist über **GPIO 8** ansteuerbar. Im Gegensatz zu einer normalen einfarbigen LED kann eine RGB-LED durch Mischen der Farben Rot, Grün und Blau verschiedene Farbtöne darstellen. Damit lässt sich schnell und einfach erkennen, ob das System ordnungsgemäß funktioniert oder ein Problem vorliegt.

Ansteuerung

Zur Steuerung verwenden wir die Bibliothek `Adafruit_NeoPixel`, die speziell für WS2812-kompatible RGB-LEDs entwickelt wurde – wozu auch die eingebaute LED des ESP32 C3 gehört.

Im Code wird die LED so initialisiert:

```
cpp
KopierenBearbeiten
Adafruit_NeoPixel rgbLed(1, statusLedPin, NEO_GRB + NEO_KHZ800);
```

Die Funktion `setLedColor(r, g, b)` wird überall im Programm verwendet, um die Farbe zu ändern. Beispiel: `setLedColor(255, 0, 0)` ergibt reines Rot.

Farbcodes und Bedeutung

Die LED wird in verschiedenen Situationen automatisch eingefärbt:

- **Blau:** Alles funktioniert normal. Die Temperatur ist im Normalbereich und WLAN ist verbunden.
- **Orange (blinkend):** Die Temperatur ist erhöht (über 25 °C). Dies signalisiert eine mögliche Überhitzung.
- **Rot (blinkend):** Keine WLAN-Verbindung. Das System arbeitet lokal, aber hat keine Internetverbindung.

Diese Logik wird durch die Funktion `updateStatusLed(tempC)` im Code umgesetzt. Dort wird geprüft, ob das WLAN verbunden ist und wie hoch die Temperatur ist. Je nach Zustand wird dann zyklisch (alle paar Hundert Millisekunden) die Farbe der LED geändert.

Ein Beispieldausschnitt:

```
cpp
KopierenBearbeiten
if (!onlineModus) {
    // Kein WLAN: LED blinkt rot
    ...
} else if (temp > 25.0) {
    // Hohe Temperatur: LED blinkt orange
    ...
} else {
    // Alles OK: LED leuchtet durchgehend blau
    ...
}
```

Wetterstation

}

Die LED blinkt dabei nicht automatisch – wir steuern das manuell mit `millis()` und einem `lastLedToggle`-Zeitstempel. So verhindern wir, dass der ESP32 blockiert wird.

Zweck und Vorteil

Die RGB-LED dient als schnelle visuelle Rückmeldung:

- Man muss das Webinterface nicht aufrufen, um zu wissen, ob alles in Ordnung ist.
- Besonders bei Netzwerkproblemen oder überhitzten Komponenten ist eine LED-Anzeige hilfreich.
- Sie eignet sich auch für den Betrieb ohne Bildschirm oder ohne USB-Verbindung zum PC.

Blynk-Anbindung

Ein zentraler Bestandteil des Projekts ist die **Blynk-Integration**. Damit wird es möglich, die Wetterstation **über das Internet mit dem Smartphone zu überwachen** – selbst dann, wenn man sich nicht im selben WLAN wie der ESP32 befindet. Blynk bietet eine mobile App und einen Cloud-Server, über den man Werte übertragen und visuell darstellen kann.

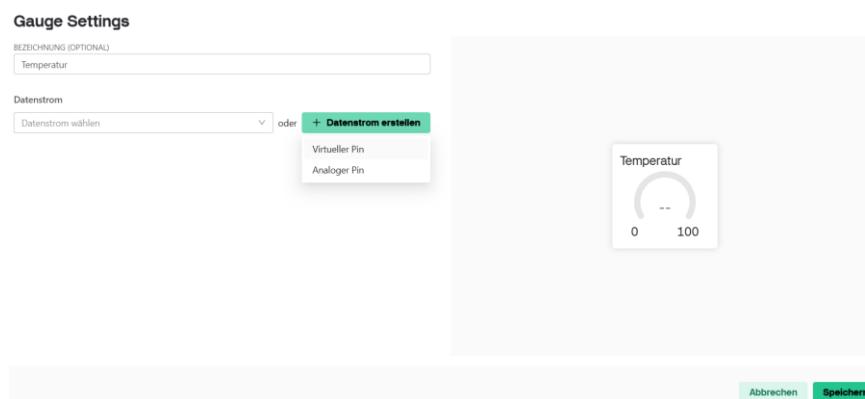
Was ist Blynk?

Blynk ist eine IoT-Plattform, mit der sich Sensoren, Mikrocontroller und Smart-Geräte einfach per App steuern und überwachen lassen. In deinem Fall dient Blynk dazu, die drei wichtigsten Messwerte deiner Wetterstation in Echtzeit anzuzeigen:

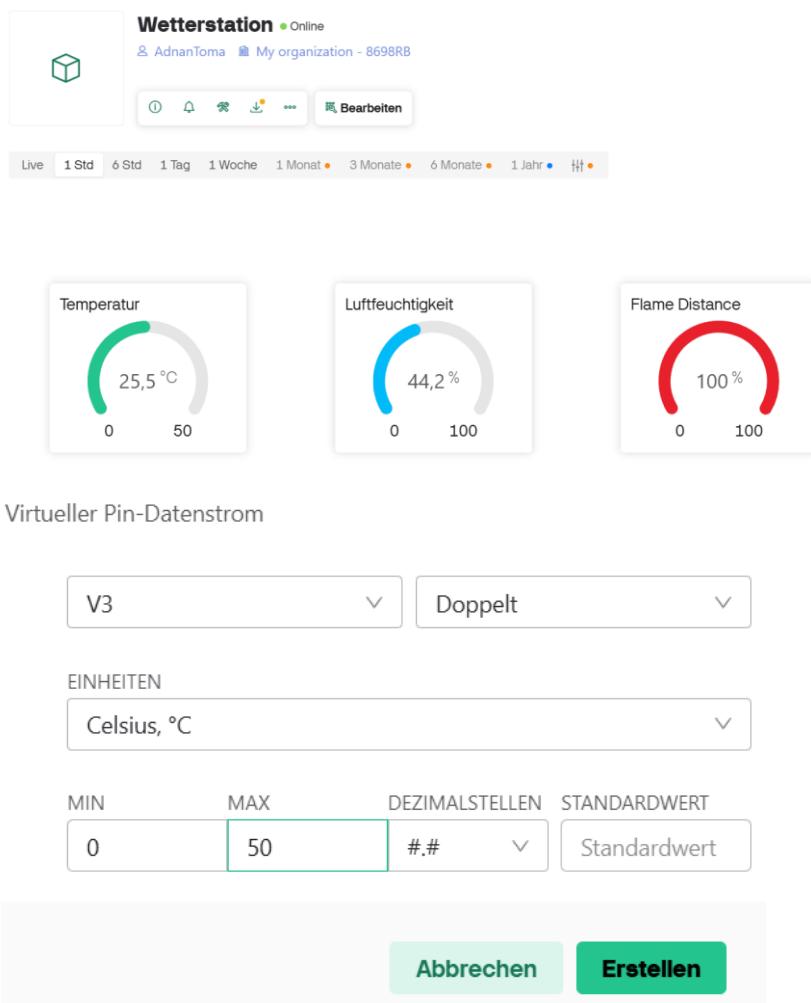
- Temperatur
- Luftfeuchtigkeit
- Flammenstatus (in Prozent)

Technische Umsetzung

Bevor wir etwas im Code ändern, müssen wir einige Einstellungen auf einrichten.



Wetterstation



Im Code wird die Blynk-Bibliothek eingebunden über:

```
cpp
KopierenBearbeiten
#include <BlynkSimpleEsp32.h>
```

Am Anfang werden folgende Makros gesetzt, damit sich dein ESP32 korrekt mit deinem Blynk-Projekt verbinden kann:

```
cpp
KopierenBearbeiten
#define BLYNK_TEMPLATE_ID "TMPL4XF52oeRd"
#define BLYNK_TEMPLATE_NAME "Wetterstation"
#define BLYNK_AUTH_TOKEN "TIsqI9EpyK9PMAupg-IHQQ67sxeN08M1"
```

Diese drei Angaben bekommst du direkt aus der Blynk-App beim Erstellen deines Projekts.

Die Verbindung wird im `setup()` aufgebaut:

```
cpp
KopierenBearbeiten
Blynk.config(BLYNK_AUTH_TOKEN);
Blynk.connect();
```

Damit ist dein Mikrocontroller mit der Blynk-Cloud verbunden und kann Daten austauschen.

Datenübertragung

In der `loop()`-Funktion werden die Sensordaten regelmäßig (alle 2 Sekunden) mit `Blynk.virtualWrite()` an die App geschickt:

Wetterstation

```
cpp
KopierenBearbeiten
Blynk.virtualWrite(V0, tempC);           // Temperatur
Blynk.virtualWrite(V1, humi);             // Luftfeuchtigkeit
Blynk.virtualWrite(V2, flamePercent);    // Flammenintensität (0-100 %)
```

Die virtuellen Pins V0, V1 und V2 müssen in der Blynk-App mit **Gauge**, **Label** oder **Graph-Widgets** verknüpft sein.

Warum ist Blynk nützlich?

Die Integration von Blynk macht deine Wetterstation mobil und **cloudfähig**. Du kannst überall auf der Welt deine Sensorwerte live überwachen, solange dein ESP32 mit dem Internet verbunden ist.

Besonders praktisch:

- Keine zusätzliche Server-Infrastruktur nötig.
- Intuitive App zur Visualisierung.
- Unterstützung für Diagramme, Buttons und Benachrichtigungen.

Discord-Webhook

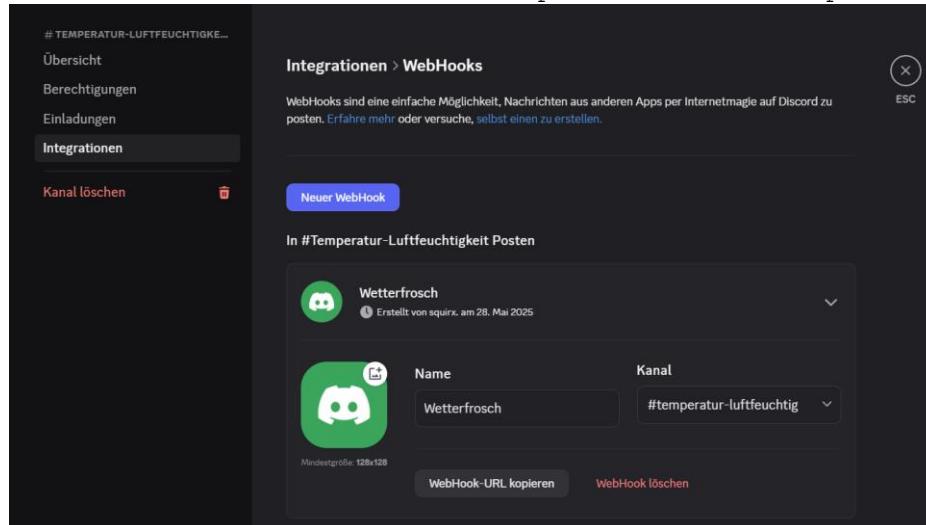
Ein weiteres Highlight der Wetterstation ist die automatische Benachrichtigung über **Discord**. Dabei werden die aktuellen Sensordaten regelmäßig an einen vorher eingerichteten **Discord-Server** gesendet, wo sie in einem bestimmten Kanal als Nachricht erscheinen.

Was ist ein Discord-Webhook?

Ein Webhook ist eine einfache URL, die von einem Dienst (in diesem Fall Discord) bereitgestellt wird, damit andere Programme Daten direkt an einen Channel senden können – ohne dass ein richtiger Bot oder Login nötig ist. Du hast dafür in Discord unter „Integrationen“ einen Webhook erstellt und dessen URL in deinen Code eingebaut.

Beispiel:

```
cpp
KopierenBearbeiten
const char* discordWebhook = "https://discord.com/api/webhooks/...";
```



Wie wird der Webhook angesprochen?

Wetterstation

Im Code hast du die Funktion `sendToDiscord(...)` definiert. Sie verwendet die Bibliothek `HTTPClient`, um eine **HTTP-POST-Anfrage** an die Webhook-URL zu schicken:

```
cpp
KopierenBearbeiten
HTTPClient http;
http.begin(discordWebhook);
http.addHeader("Content-Type", "application/json");
```

Dann wird eine JSON-Nachricht erstellt, die die aktuellen Werte enthält:

```
cpp
KopierenBearbeiten
String payload = "{\"content\": \"\ud83c\udcbb Wetterstation Update\n\ud83c\udcbb Temperatur: " +
+ String(temp) + " \u00b0C\n\ud83c\udcbe Feuchtigkeit: " + String(hum) + " %\n\ud83c\udcbb Flam-
menstatus: " + flame + "\"}";
```

Diese Nachricht wird mit `http.POST(payload)` an Discord gesendet.

Wie oft wird das gesendet?

Du willst nicht jede Sekunde eine Nachricht schicken. Deshalb hast du einen Timer eingebaut, der mithilfe von `millis()` prüft, ob **4 Minuten** vergangen sind:

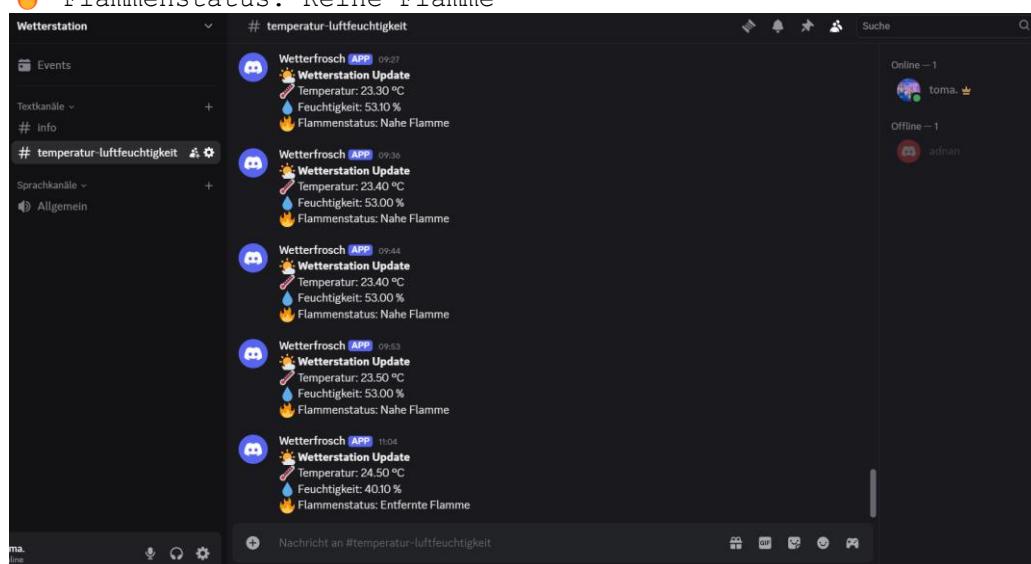
```
cpp
KopierenBearbeiten
if (millis() - lastDiscordSend >= discordInterval) {
    sendToDiscord(tempC, humi, flameStatus);
    lastDiscordSend = millis();
}
```

Diese Logik befindet sich am Ende der `loop()`-Funktion.

Was sieht man auf Discord?

Alle 4 Minuten wird automatisch eine Nachricht in deinem Discord-Channel gepostet, etwa so:

```
yaml
KopierenBearbeiten
\ud83c\udcbb Wetterstation Update
\ud83c\udcbb Temperatur: 23.4 \u00b0C
\ud83c\udcbe Feuchtigkeit: 48.0 %
\ud83c\udcbb Flammenstatus: Keine Flamme
```



So bekommst du regelmäßig Status-Updates direkt auf dein Handy oder am PC, ohne eine App zu öffnen oder im Webinterface nachzusehen.

Warum ist das praktisch?

- Du bekommst regelmäßige Push-Nachrichten zu deinem Projekt.
- Perfekt zur Fernüberwachung oder wenn du das Gerät unbeaufsichtigt laufen lässt.
- Du brauchst keine extra Cloud-Plattform oder Datenbank – nur deinen Discord-Account.

Technische Probleme

Während der Umsetzung des Projekts „Wetterstation“ traten mehrere technische Herausforderungen auf, die sowohl die Genauigkeit der Messdaten als auch die Benutzerfreundlichkeit des Webinterfaces beeinflussten.

Ein zentrales Problem war die Messgenauigkeit des Flammensensors. Ursprünglich war vorgesehen, dass dieser Sensor ausschließlich auf das Vorhandensein einer offenen Flamme reagiert. In der Praxis stellte sich jedoch heraus, dass der Sensor sehr empfindlich gegenüber anderen Lichtquellen ist – insbesondere Sonnenlicht oder künstliches Licht aus LED- oder Halogenlampen. Dadurch kam es mehrfach zu Fehlmessungen, bei denen fälschlicherweise eine Flamme erkannt wurde, obwohl keine vorhanden war. Dies erschwert die Zuverlässigkeit der Anzeige und erforderte manuelle Prüfungen der Messwerte, um die Ursache der Abweichung festzustellen.

Ein weiteres Problem betraf das Webinterface. Ziel war es, die Messdaten automatisch und kontinuierlich im Browser zu aktualisieren, ohne dass die Seite neu geladen werden muss. Zwar wurde eine automatische Aktualisierung per JavaScript implementiert, jedoch funktionierte diese nicht von Anfang an zuverlässig. In mehreren Tests musste die Seite manuell neu geladen werden, um aktuelle Daten angezeigt zu bekommen. Erst durch den Einsatz eines speziellen `/data`-Endpoints und JavaScript-Fetch-Funktionen konnte das Problem behoben werden, sodass die Daten jetzt tatsächlich alle fünf Sekunden aktualisiert werden.

Darüber hinaus fiel auf, dass die Verlaufsgraphen für Temperatur und Luftfeuchtigkeit beim ersten Öffnen der Webseite zunächst leer erschienen. Der Grund dafür war, dass der Graph erst dann Werte anzeigen kann, wenn genügend Daten gesammelt wurden. Da der Verlauf aus mehreren Minuten besteht, kann es – insbesondere beim Neustart oder nach einer längeren Inaktivität – mehrere Minuten dauern, bis die ersten Punkte auf dem Graphen sichtbar werden. Diese Verzögerung war technisch nicht vermeidbar, da der ESP32 die Daten erst im Verlauf der Nutzung sammelt und speichert.

Insgesamt waren diese technischen Probleme lösbar, führten jedoch dazu, dass im Verlauf des Projekts mehrere Optimierungen am Code vorgenommen werden mussten, um die Stabilität und Genauigkeit des Systems sicherzustellen.

Zusammenfassung

Zusammenfassend möchten wir sagen, dass unser Projekt „Wetterstation“ ein voller Erfolg war. Wir haben es geschafft, mithilfe eines ESP32 mehrere Sensoren einzubinden, die Messdaten zuverlässig erfassen und über verschiedene Plattformen ausgeben – darunter ein OLED-Display, ein Webinterface, die Blynk-App und sogar ein Discord-Channel. Trotz kleiner technischer Herausforderungen, wie etwa der

Lichtempfindlichkeit des Flammensensors oder der anfänglichen Probleme mit der Webaktualisierung, konnten wir alle Funktionen erfolgreich umsetzen und stetig verbessern. Das Projekt hat uns nicht nur praktische Programmiererfahrung vermittelt, sondern auch gezeigt, wie vielseitig ein Mikrocontroller in Kombination mit Cloud- und Webdiensten eingesetzt werden kann.

Literaturverzeichnis

arduinogetstarted. (kein Datum). Von <https://arduinogetstarted.com/tutorials/arduino-dht11> abgerufen

blynk. (kein Datum). Von <https://blynk.io/> abgerufen

edgemicrotech. (kein Datum). Von <https://www.edgemicrotech.com/esp32-c3-super-mini-real-time-clock-with-oled-display/> abgerufen

esp32io. (kein Datum). Von <https://esp32io.com/tutorials/esp32-mysql> abgerufen

esp32io. (kein Datum). Von <https://esp32io.com/tutorials/esp32-oled> abgerufen

espboards. (kein Datum). Von <https://www.espboards.dev/blog/send-message-from-esp32-to-discord/> abgerufen

github. (kein Datum). Von <https://github.com/ESP32Async/ESPAsyncWebServer> abgerufen

github-WiFiManager. (kein Datum). Von <https://github.com/tzapu/WiFiManager> abgerufen

randomnerdtutorials. (kein Datum).

randomnerdtutorials. (kein Datum). Von <https://randomnerdtutorials.com/esp32-dht11-dht22-temperature-humidity-sensor-arduino-ide/> abgerufen

theoriecircuit. (kein Datum). Von https://theoretcircuit.com/esp32-projects/interfacing-ws2812b-neopixel-led-strip-with-esp32/#google_vignette abgerufen

w3schools. (kein Datum). Von https://www.w3schools.com/js/js_graphics_chartjs.asp abgerufen