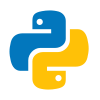
******

Ipek, Atilla & Roser, Kevin

02.06.2025| GEWERBLICH-TECHNISCHE SCHULE OFFENBURG, E1FI1 2024/2025, HUG CHRISTOPHER

Internet der Dinge

MIT WEMOS D1 MINI ESP8266

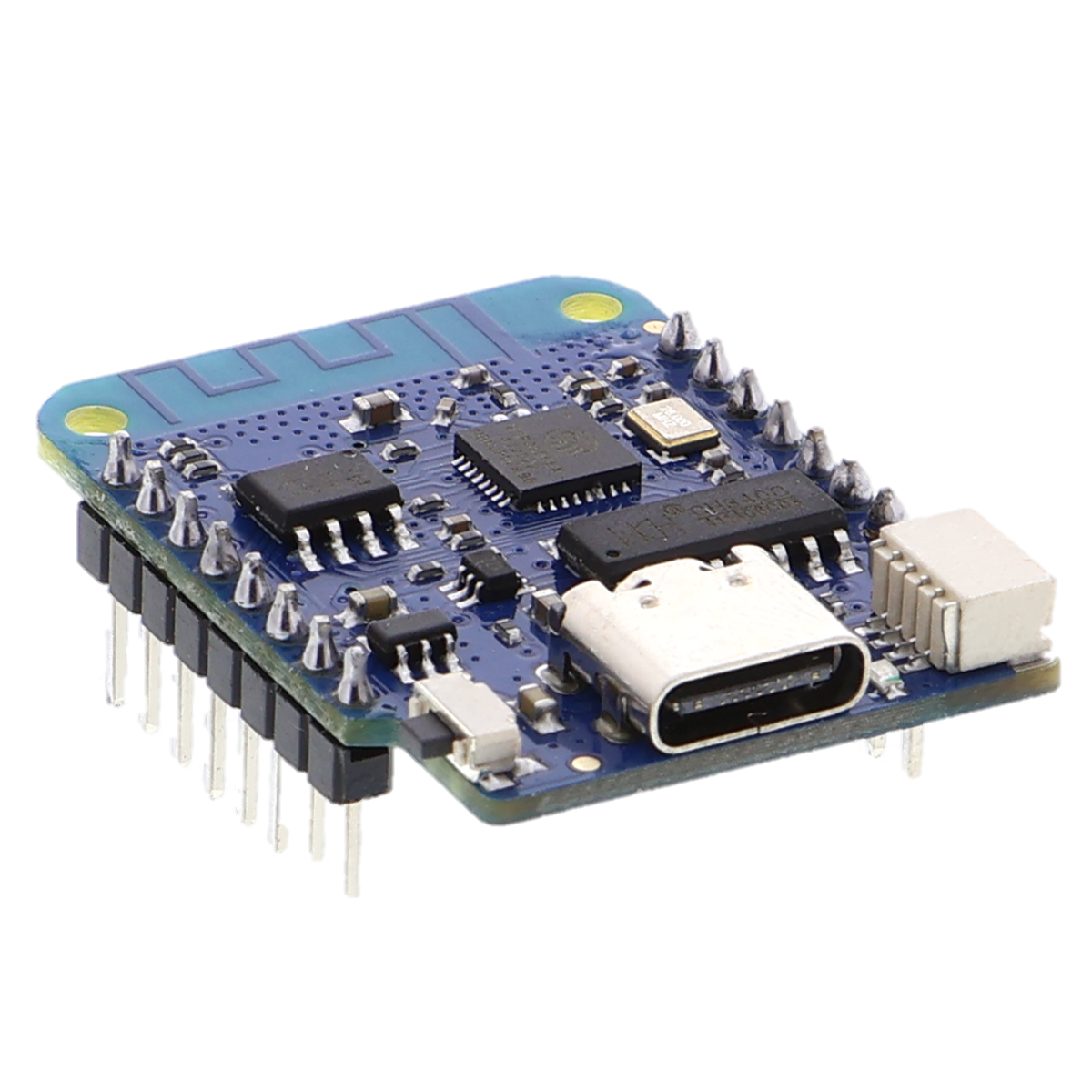
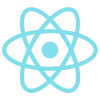
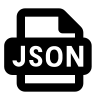
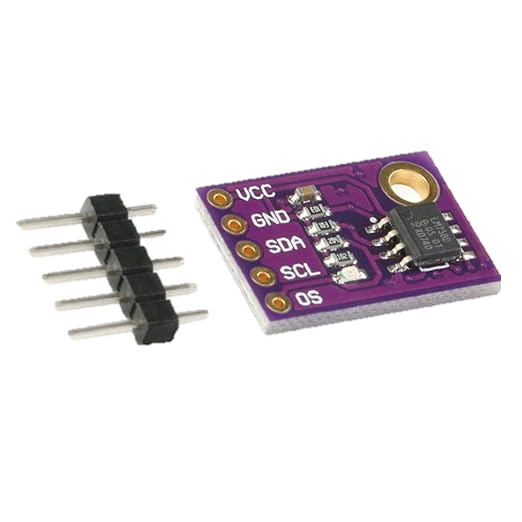
****

Abbildung 1

Abbildung 2

Inhaltsverzeichnis

[I. Abbildungsverzeichnis 0](#_Toc200096867)

[II. Abkürzungsverzeichnis 0](#_Toc200096868)

[III. Literaturverzeichnis 0](#_Toc200096869)

[1 Einleitung 1](#_Toc200096870)

[1.1 Motivation/Problemstellung 1](#_Toc200096871)

[1.2 Zielgruppe 1](#_Toc200096872)

[2 Projektbeschreibung 2](#_Toc200096873)

[2.1 Projektidee & Ziele 2](#_Toc200096874)

[2.2 Technische Spezifikationen 2](#_Toc200096875)

[2.3 Hardwarearchitektur 3](#_Toc200096876)

[2.4 Softwarearchitektur 3](#_Toc200096877)

[2.4.1 IDE und verwendete Libaries 3](#_Toc200096878)

[2.4.2 Backend 4](#_Toc200096879)

[2.4.3 Frontend 7](#_Toc200096880)

[2.5 Verwendete Web-Technologien 8](#_Toc200096881)

[2.5.1 HTML (HyperText Markup Language) 8](#_Toc200096882)

[2.5.2 JSON (JavaScript Object Notation) 8](#_Toc200096883)

[2.5.3 XML (eXtensible Markup Language) 8](#_Toc200096884)

[3 Schluss 9](#_Toc200096885)

[3.1 Zusammenfassung des Projekts 9](#_Toc200096886)

[3.2 Mögliche Erweiterungsideen: 10](#_Toc200096887)

[3.3 Reflexion (Was lief gut? Was war schwierig?) 10](#_Toc200096888)

[4 Kommunikationsplan 11](#_Toc200096889)

[A. Anhang 12](#_Toc200096890)

[A.1 Flussdiagramm des Datamanagers 13](#_Toc200096891)

[A.2 Blockschaltbild des ganzen Systems 14](#_Toc200096892)

[A.3 Struktogramm von Temperatursensorauswertung 14](#_Toc200096894)

[A.4 Flussdiagramm von Main-Klasse 15](#_Toc200096895)

[A.5 Klassendiagramm des ganzen Systems 16](#_Toc200096896)

[A.6 Konfigurationsdatei *Config.py* 17](#_Toc200096897)

[A.7 Minimalistische HTML-Struktur des ESP 17](#_Toc200096898)

1. Abbildungsverzeichnis

[Abbildung 2 1](#_Toc199768734)

https://www.amazon.de/HiLetgo-Temperatursensor-Interface-Precision-Temperature/dp/B082M65D8P

[Abbildung 1 1](#_Toc199768735)

https://funduinoshop.com/elektronische-module/sonstige/mikrocontroller/wemos-d1-mini-v4.0-esp8266-mikrocontroller-lolin-d1-mini-v4.0-kompatibel

1. Abkürzungsverzeichnis

**Frontend** - grafische Benutzeroberfläche (GUI), mit der Ihre Benutzer direkt interagieren können.

**Backend** - bezeichnet in der Regel den funktionalen Teil eines digitalen Produkts wie beispielsweise Webseiten oder Apps und bezieht sich meist auf Client Server.

**REST-API** - Eine REST-API (Representational State Transfer Application Programming Interface) ist eine Art von API, die den REST-Architekturstil befolgt

**IDE** - "Integrated Development Environment" (Integrierte Entwicklungsumgebung).

**Watchdog** - bezeichnet eine Funktion zur Ausfallerkennung eines [Systems](https://de.wikipedia.org/wiki/System).

**React-SPA** – React- Single Page Application ist eine Webanwendung, die auf einem einzigen HTML-Dokument basiert, das beim ersten Laden der Seite komplett vom Server geladen wird.

**CDN** - Content Delivery Network

**UI** - interaktive Benutzer Oberfläche (User Interface)

1. Literaturverzeichnis

*CJMCU LM75 Temperatursensor*. (kein Datum). Von https://www.analog.com/media/en/technical-documentation/data-sheets/lm75.pdf abgerufen

*React Google Charts*. (kein Datum). Von https://react-google-charts.com/ abgerufen

*React-js*. (kein Datum). Von https://reactjs.org/ abgerufen

*REST api*. (kein Datum). Von https://restfulapi.net/ abgerufen

*Thonny*. (kein Datum). Von https://thonny.org/ abgerufen

*w3schools HTML*. (kein Datum). Von https://www.w3schools.com/html/html\_intro.asp abgerufen

*w3schools JSON*. (kein Datum). Von https://www.w3schools.com/js/js\_json.asp abgerufen

*w3schools XML*. (kein Datum). Von https://www.w3schools.com/xml/xml\_whatis.asp abgerufen

*Wemos D1 Mini*. (kein Datum). Von https://www.wemos.cc/en/latest/d1/d1\_mini.html abgerufen

# Einleitung

## Motivation/Problemstellung

Warum ist IoT wichtig? Warum der Wemos D1 Mini?

## Zielgruppe

Schüler:innen, Lehrer, mit der Erweiterung jeder der im Office arbeitet

# Projektbeschreibung

## Projektidee & Ziele

Ziel unseres Projekts ist es, Temperaturdaten in zwei unterschiedlichen Dartstellungsformen Bereitzustellen und zu speichern:

* ***Kurzzeitansicht***: die letzten 50 Sekunden, in 5 Sekunden abständen
* ***Langzeitansicht***: die letzten 5 Tage, in 1 Stunde abständen

Die Daten werden einen HTTPS-Server zur Verfügung gestellt, der direkt auf dem Wemos D1 Mini gehostet wird.

Die Kurzzeitansicht präsentiert die Temperaturwerte in tabellarischer, kompakter Form und die Langzeitansicht stellt die Daten grafisch darstellt, wodurch sie so eine übersichtliche und verständliche Auswertung der Temperaturentwicklung über einen längeren Zeitraum ermöglicht.

Der Zugriff der Daten erfolgt **plattformunabhängig** über das lokale Netzwerk **per Web-Interface.**

Zusätzlich ist das Projekt robust mit der **Watchdog**-Überwachung und skalierbar durch die gewählte Softwarearchitektur mit dem modularen Aufbau.

Die Gesamtstruktur des Systems ist in *Blockschaltbild 1* dargestellt.

## Technische Spezifikationen

* **Mikrocontroller**: ESP8266 (Wemos D1 Mini)
* **Temperatursensor**: TMP75 (I2C-Interface), +- 0.5° Genauigkeit
* **Programmiersprache**: MicroPython
* **Webserver**: Integrierter HTTP-Server
* **Frontend**: React-basierte Single-Page-Application
* **Datenformat**: JSON und XML für API-Endpoints

## Hardwarearchitektur

Uns wurde als zentrale Steuereinheit der **Wemos D1 mini** zur Verfügung gestellt. Er ist ein kompakter Mikrocontroller auf Basis des ESP8266EX-Chips mit integriertem **WLAN**. Das Board verfügt über **4MB Flash-Speicher** sowie einen **32 Bit-Prozessor** und bietet eine Vielzahl an I/O Pins für externe Komponenten.[[1]](#footnote-1)

Im unserem Projekt nutzen wir die Serial-Data- (**SDA**) und Serial-Clock-Leitung (**SCL**) des I2C Busses, um den **CJMCU-75 Temperatursensor** auszulesen und die aktuelle Umgebungstemperatur zu erfassen.[[2]](#footnote-2)

Als Beispiels Aktor dient die OnBoard-LED des Wemos D1 Mini, welche über den GPIO-Pin 2 angesteuert werden kann.

## Softwarearchitektur

Die Anwendung folgt einer strikten Trennung zwischen Frontend und Backend:

* **Frontend**: React-SPA gehostet auf GitHub Pages CDN
* **Backend**: ESP8266 mit minimaler HTML-Struktur und REST-API

### IDE und verwendete Libaries

Für die Entwicklung des **Backend** unseres Projekts wurde die **Thonny IDE** verwendet, da sie sich besonders gut für MicroPython eignet.[[3]](#footnote-3)  
**Visual Studio Code** zum Einsatz kam für das **Frontend** zum Einsatz, um die JSX-Dateien und die Gestaltung der Weboberfläche komfortabel zu gestalten.

Das Backend des Projektes nutzt ausschließlich Micro-Python-kompatible Bibliotheken:

* „***network, socket, time, gc***“ – System und Netzwerkfunktionen
* eigenentwickelte Module: „***SoftwareWatchdog, LEDController, DataManager, WebHandler, WiFiManager, ESP8266WebServer***“

Zur Visualisierung der Daten haben wir uns entschieden, die Bibliothek ***„react-google-charts“*** zu nutzen.[[4]](#footnote-4) Mit ihr realisieren wir einfache und flexible Diagramme, mit denen der Client interagieren kann.

### Backend

Die Entwicklung des Programms erfolgte entlang eines klaren **objektorientierten Konzepts** unter Beachtung des **SRP**-Prinzips (Single Responsibility Principle). Die Klassen sind so strukturiert, dass **jede eine klare abgegrenzte Verantwortung** innerhalb des Systems übernimmt.

Die Übersicht der einzelnen Klassen sind in *Klassendiagramm 1* dargestellt.

* Die TemperatureStation-Klasse (*main.py*) ist die zentrale **Koordinationseinheit** des gesamten Systems und somit zuständig für:
* **Systeminitialisierung**
* **Lebenszyklus-verwaltung**
* **Hauptschleife**

Der Ablauf dieser zentralen Einheit ist in Flussdiagramm 2 veranschaulicht.

* Die DataManager-Klasse (*DataManager.py*) ist verantwortlich für die gesamte Datenverwaltung:
* **Datenpersistierung**
* **Datenerfassung**
* **Ringpuffer-Logik**

Der grundlegende Prozessablauf ist in *Flussdiagramm 1* dargestellt.

* Die WebHanlder-Klasse (*WebHandler.py*) verarbeitet **HTTP-Anfragen** und stellt sie in verschiedenen Datenformaten bereit:

|  |  |  |  |
| --- | --- | --- | --- |
| Endpoint | Method | Response | Zweck |
| / | GET | HTML | Web-Interface |
| /api/kurzzeit | GET | JSON | JSON-Daten der letzten 10 Messwerte (5s Intervall) |
| /api/langzeit | GET | JSON | JSON-Daten der letzten 120 Messwerte (1h Intervall) |
| /api/xml | GET | XML | XML-Format der Kurzzeit-Daten |

* API-Endpoints
* Response-Format (Beispiel Response):

*// JSON Response*

[

["2024-01-15 14:30:25", 23.5],

["2024-01-15 14:30:30", 23.7]

]

*// XML Response*

<temperaturen>

<eintrag>

<zeitpunkt>2024-01-15 14:30:25</zeitpunkt>

<wert>23.5</wert>

</eintrag>

</temperaturen>

Sobald eine HTTP-Anfrage an das Backend gestellt wird, aktiviert sich die OnBoard LED. Sie dient dabei als visuelles Signal und zeigt an, dass eine aktive Kommunikation mit dem System erfolgt.

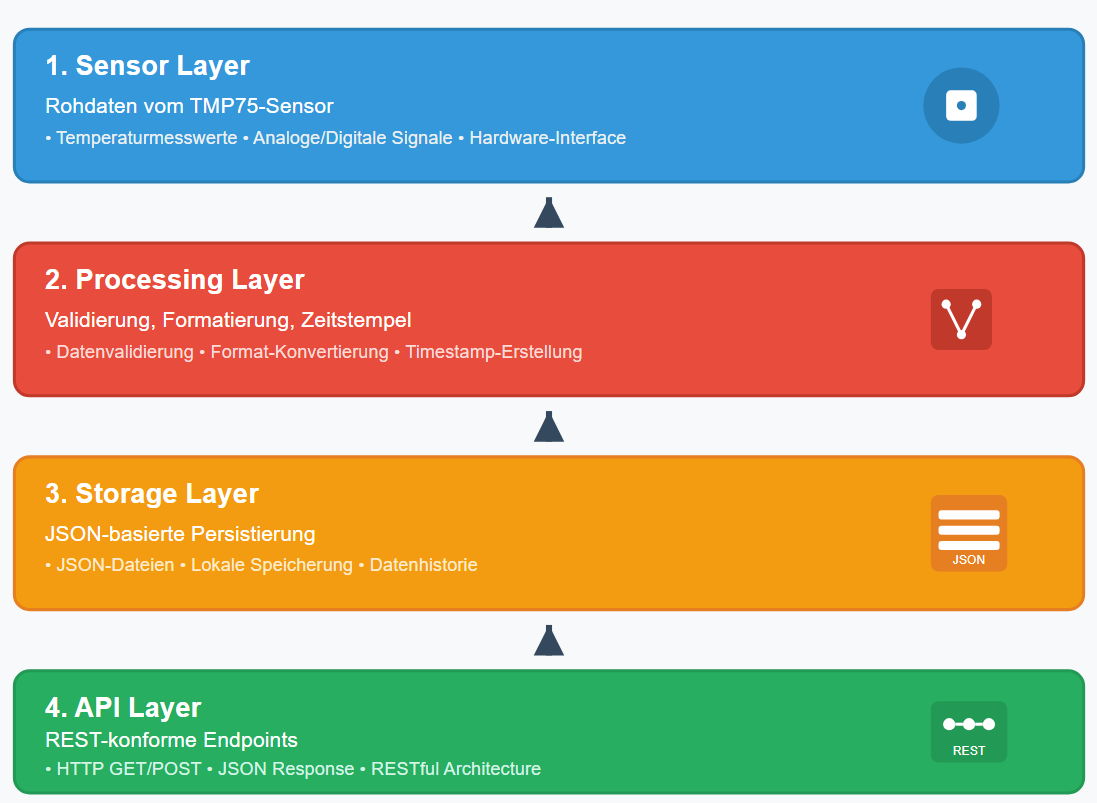
* Für die Verwaltung der Netzwerkverbindung mit Verbindungsüberwachung ist die WifiManager-Klasse (*WifiManager.py*) zuständig. Zusätzlich stellt sie sicher, dass bei einer erfolgreichen Verbindung die aktuelle IP-Adresse des ESP automatisch in einer *launchsettings.json*-Datei gespeichert wird. Dies ermöglicht eine **automatische Kopplung von Backend und Frontend** über das lokale Netzwerk.
* Durch die SoftwareWatchdog-Klasse (*SoftwareWatchdog.py*) ist eine Überwachungsfunktion zur **Systemstabilisierung** implementiert:
* Überwacht regelmäßige "Lebenszeichen" des Systems
* Automatischer Neustart bei Systemhängern
* Konfigurierbares Timeout (Standard: 20 Sekunden)
* Mit der LEDController-Klasse (*LED-Controller.py*) lassen sich einfach neue LEDs modular hinzufügen.
* Das Hilfs-Modul *TMP75.py* ist für die Temperatursensor-Integration da und liest mithilfe der **I2C-Kommunikation** an der Adresse 0x48 den Temperatur Wert aus. Die Funktionsweise dieses Moduls ist in *Struktogramm 1* dargestellt.
* Die *Config.py* ist für die jegliche *Konfiguration* des Systems da.

#### Datenarchitektur

Technisch werden die beiden Datenansichten **Langansicht** und **Kurzansicht** so realisiert, dass es zwei **JSON-Dateien** gibt, in denen die jeweiligen Messwerte persistiert werden. Die Entscheidung **wann** neue Messwerte in die jeweilige Datei geschrieben werden, erfolgt durch die Methode:

* *should\_measure()*

Sie prüft, ob seit der letzten Speicherung genügen Zeit vergangen ist, basierend auf zwei separat konfigurierbaren Zeitintervallen in der *Config.py* - Datei.

Bei Programmstart werden die vorhandenen JSON-Dateien eingelesen und die darin enthaltenen Messwerte in Listen geladen. Diese Listen fungieren als Datenspeicher während der Laufzeit des Programmes. Bei jedem neuen Messwert im *Speichervorgang* wird überprüft ob die maximal Listen Größe erreicht wurde und falls wird der **älteste Eintrag** jeweils entfernt und dann der **neuste Messwert hinzugefügt**. Ist die maximale Größe der Liste noch nicht erreicht, wird der neue Messwert angehängt. Hierdurch entsteht eine einfache, aber effektive **Ringpuffer-logik,** welche eine kontinuierliche Datenaktualisierung ermöglicht die Speicherressourcen schonend für den Mikrocontrollers ist.

Daten

### Frontend

Das Frontend basiert auf einer **React-SPA**, die über GitHub Pages **CDN** geladen wird. Dieses Vorgehen wurde gewählt, um den begrenzten Ressourcen des WEMOS D1 Mini ESP8266 entgegenzusteuern und gleichzeitig eine stabile Systemarchitektur gewährleitet ist:

* **Speicherplatzeffizienz**: Da der Speicherplatz auf dem ESP ohnehin schon begrenzt ist, ist nur eine *minimalistische HTML-Struktur* auf dem ESP.
* **RAM-Optimierung**: Das vollständige Laden und Ausführen der JavaScript-Datei direkt auf dem ESP hätte zu Speicherüberläufen (Out-of-Memory), weshalb die Auslagerung nötig war für unser Vorhaben.
* **Systemstabilität**: Durch diese Auslagerung des Frontend erreichen wir, dass der ESP nur eine statische HTML-Datei und API-Endpunkte bereitstellen muss. Das reduziert die Komplexität und erhöht die Laufzeitstabilität.

Nachteil dieser Auslagerung ist, der ESP benötigt eine **Anbindung an das Internet**, bzw. an das GitHub Pages CDN, ohne solchen Funktioniert die Weboberfläche nicht.

Die eigentliche Anwendungslogik wird dann **clientseitig** ausgeführt und kommuniziert über **REST-APIs** mit dem ESP.

#### Funktionsweise des Frontends

Die Anwendung besteht im Wesentlichen aus einer React-Komponente, sie übernimmt die wesentlichen Funktionen:

* Interaktive *Visualisierung von Temperaturdaten* in einem Liniendiagramm mit Hilfe von der Bibliothek „***react-google-charts“.***
* Kontinuierlicher Datenabruf aktueller und Langzeit Messwerte vom ESP über die beiden REST-API-Endpoints (***/api/langzeit*** *und* ***/api/kurzzeit***), im 5 Sekunden Takt.

Technische Funktionsweise: *Flussdiagramm 3*

## Verwendete Web-Technologien

### HTML (HyperText Markup Language)

Erklär hier kurz die sachen und dann die Rolle dazu in unserem Projekt (Wird noch gemacht)

### JSON (JavaScript Object Notation)

Erklär hier kurz die sachen und dann die Rolle dazu in unserem Projekt (Wird noch gemacht)

### XML (eXtensible Markup Language)

Erklär hier kurz die sachen und dann die Rolle dazu in unserem Projekt (Wird noch gemacht)

### JavaScript-Framework React

React ist ein JavaScript-Framework das von Facebook entwickelt wurde und zur Erstellung von **UIs** verwendet wird. React basiert auf dem Konzept von wiederverwendbaren Komponenten und ermöglicht dynamische **SPAs**. [[5]](#footnote-5)

Grundlegende Konzepte von React:

* **Komponenten-basierte Architektur:** UI wird in kleine wiederverwendbare Bausteine aufgeteilt.
* **Virtual DOM:** Effiziente Aktualisierung der UI durch virtuelles Document Object Model, (tatsächliche veränderte Elemente werden aktualisiert).
* **Deklarative Programmierung:** gewünschte Ergebnis, ohne die einzelnen Schritte zur Umsetzung explizit.
* **State-Management:** Verwaltung von Anwendungszuständen über „Hooks“ wie *useState oder useEffect*.

# Schluss

## Zusammenfassung des Projekts

## Mögliche Erweiterungen

* HTTP-Client in Python
* Client-PC steuert über Temperatur-Abfrage (M) Ventilator. Beispielsweise im Sommer wenn es zu heiß wird, geht der Ventilator von alleine an am Arbeitsplatz

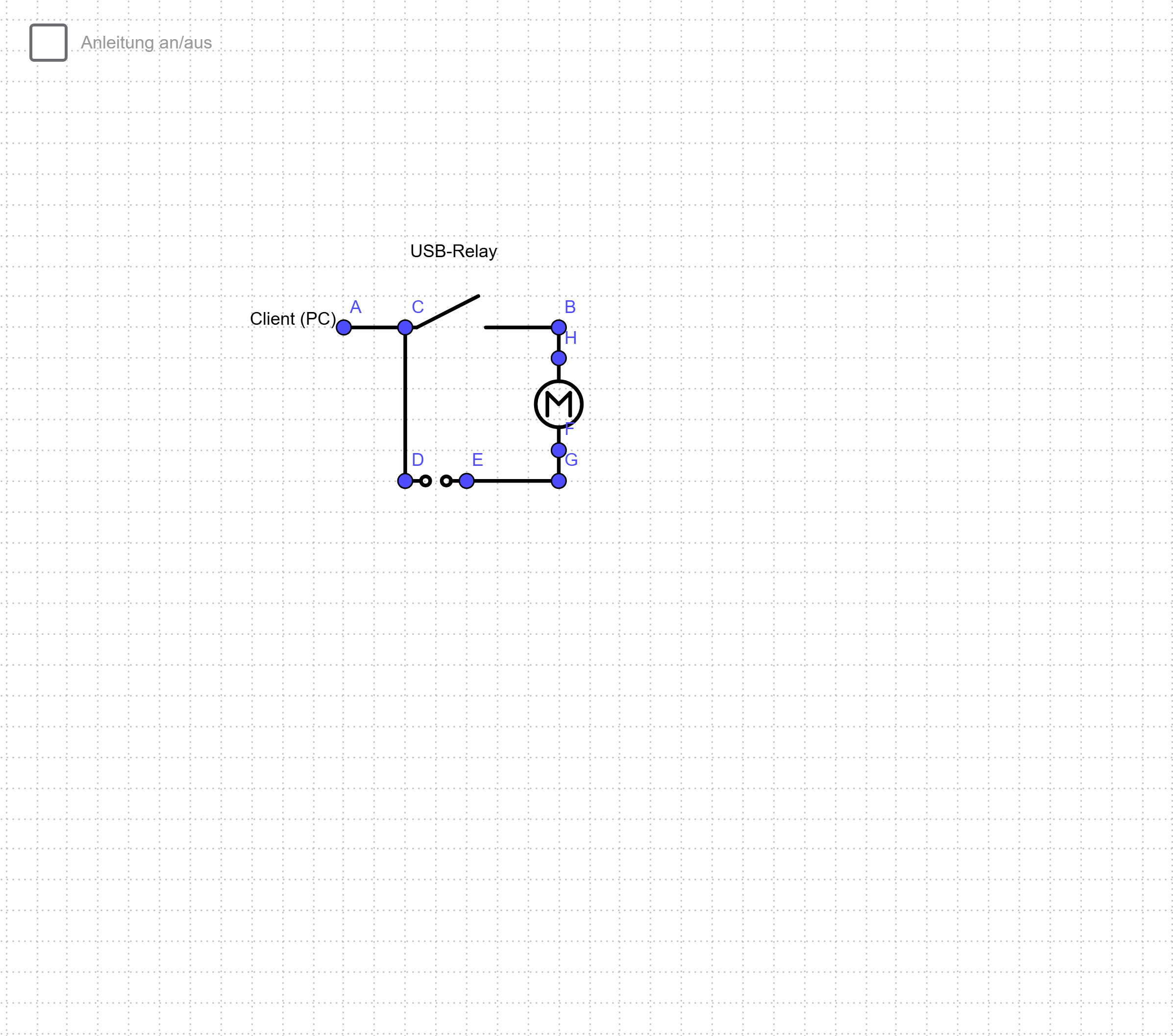
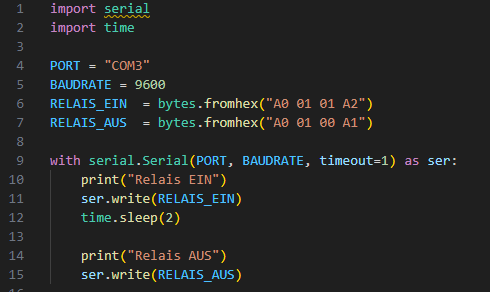


Abbildung 3

über eine serielle Schnittstelle, in Python:



## Reflexion (Was lief gut? Was war schwierig?)

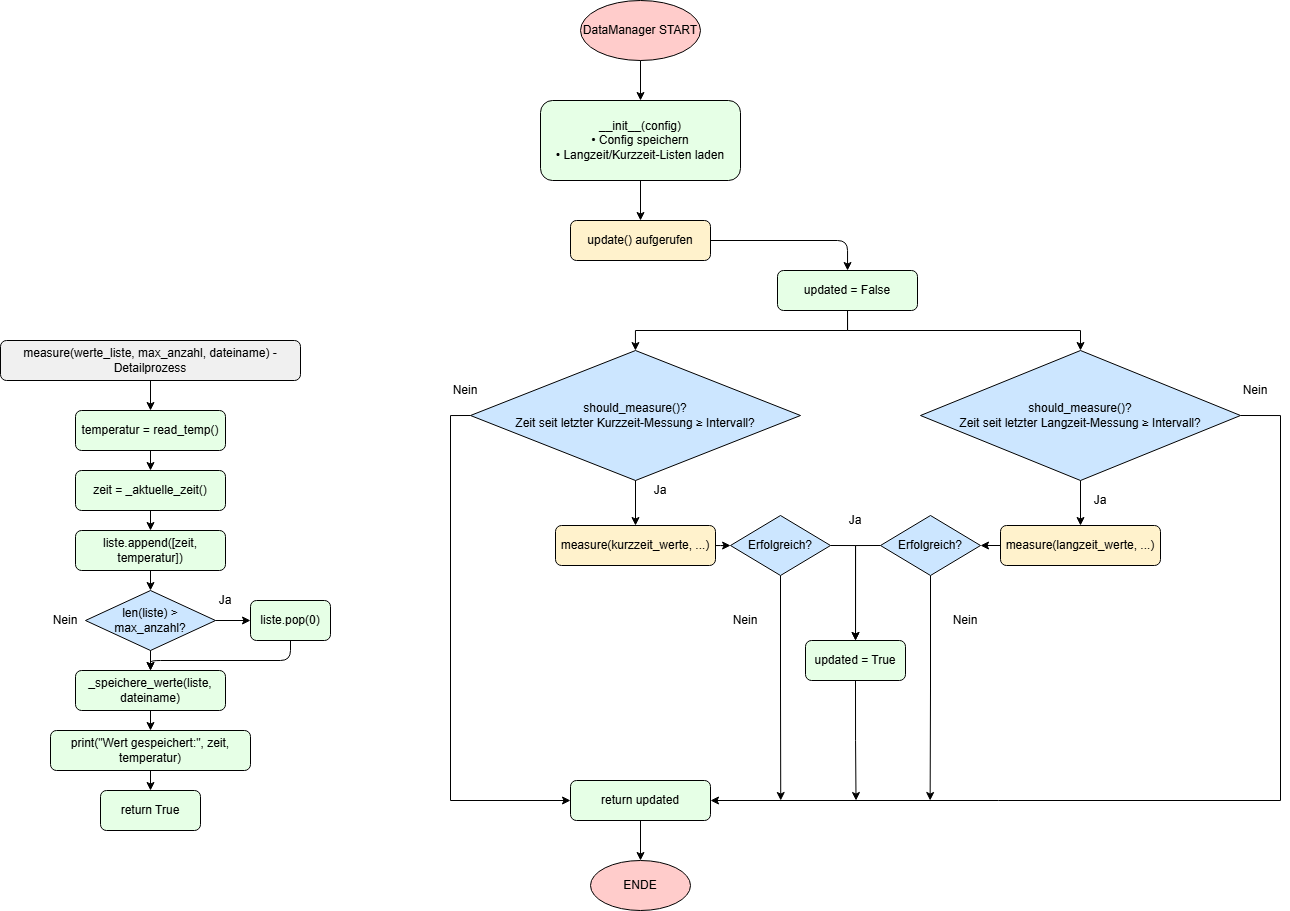
* Wackelkontakt der käbel

# Kommunikationsplan

* Teammitglieder + E-Mail
* Aufgabenverteilung
* Genutzte Kommunikationskanäle

1. Anhang

* Komplette Quellcodes (als Auszug oder Anhang)
* Datenblätter (z. B. PIR, HC-SR04)
* Schaltpläne
* Diagramme (z. B. Blockschaltbild)
* Optional: Link zum Projektdemo-Video
  1. Flussdiagramm des Datamanagers

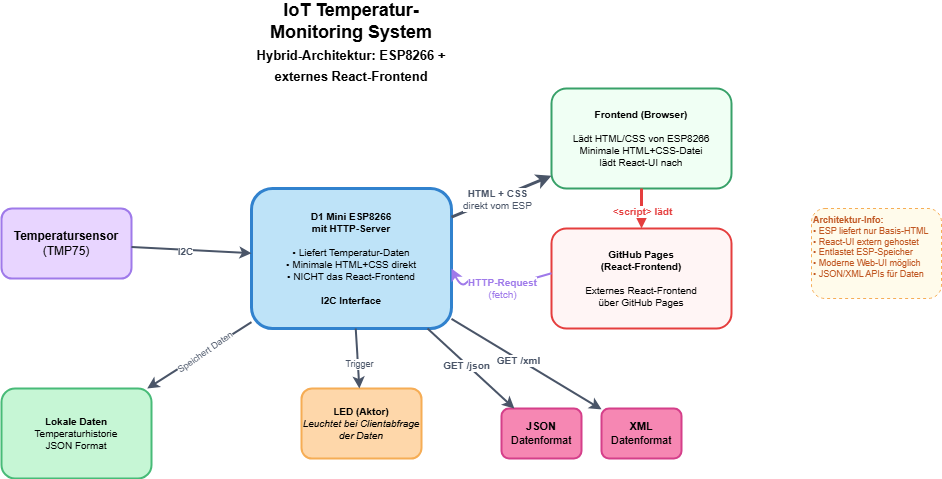


Speichervorgang 1

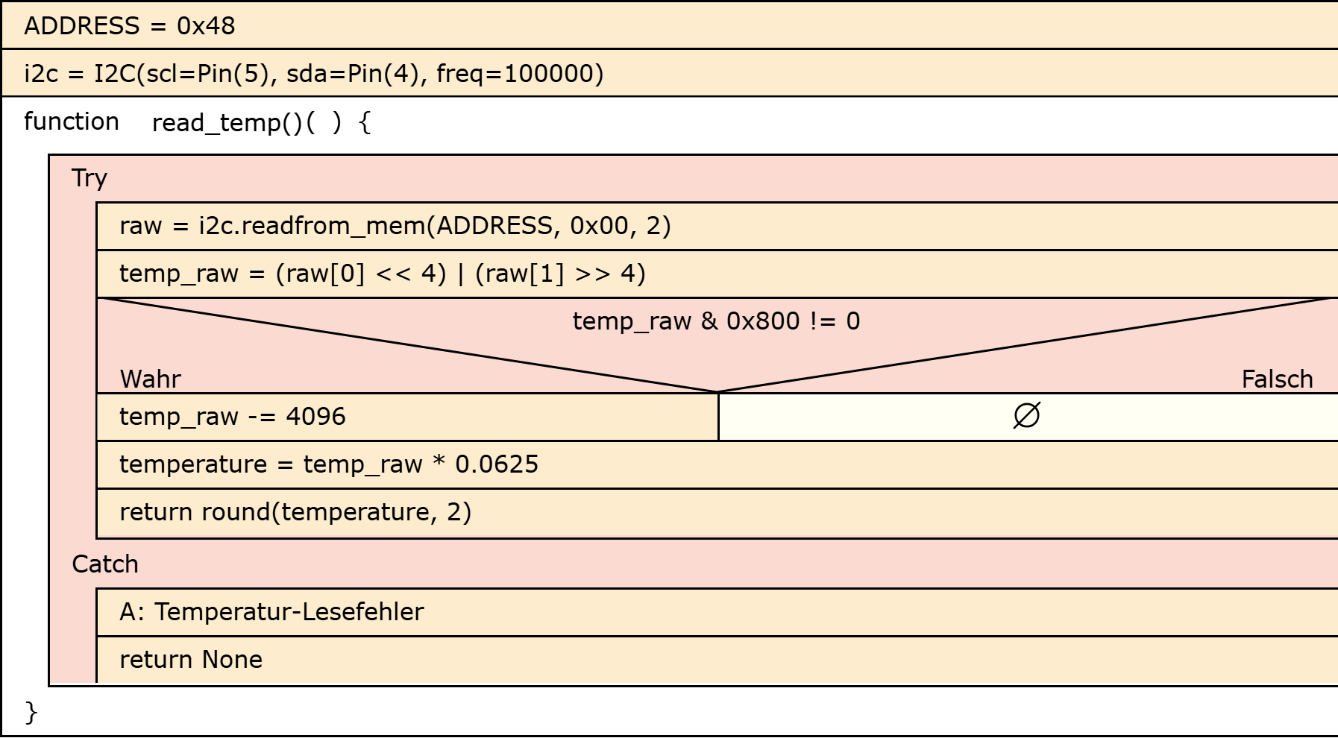
Flussdiagramm 1

* 1. Blockschaltbild des ganzen Systems

Blockschaltbild 1



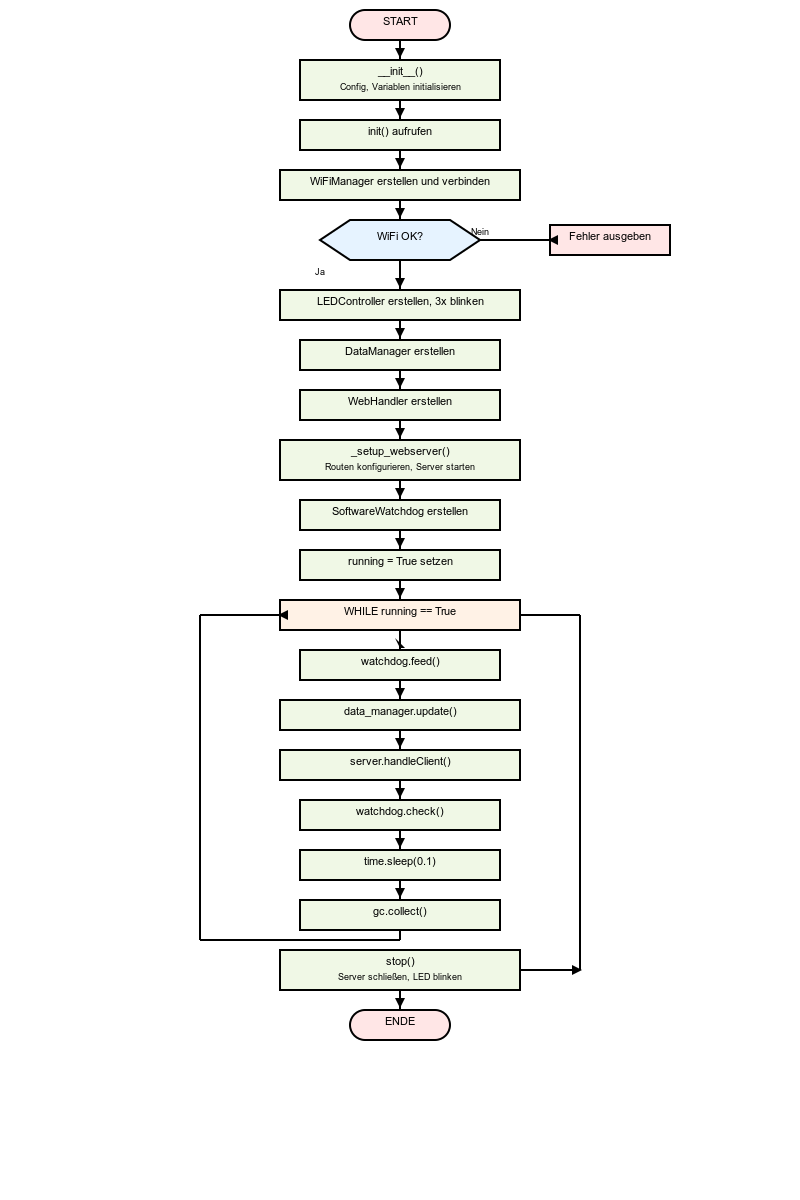
* 1. Struktogramm von Temperatursensorauswertung



Struktogramm 1

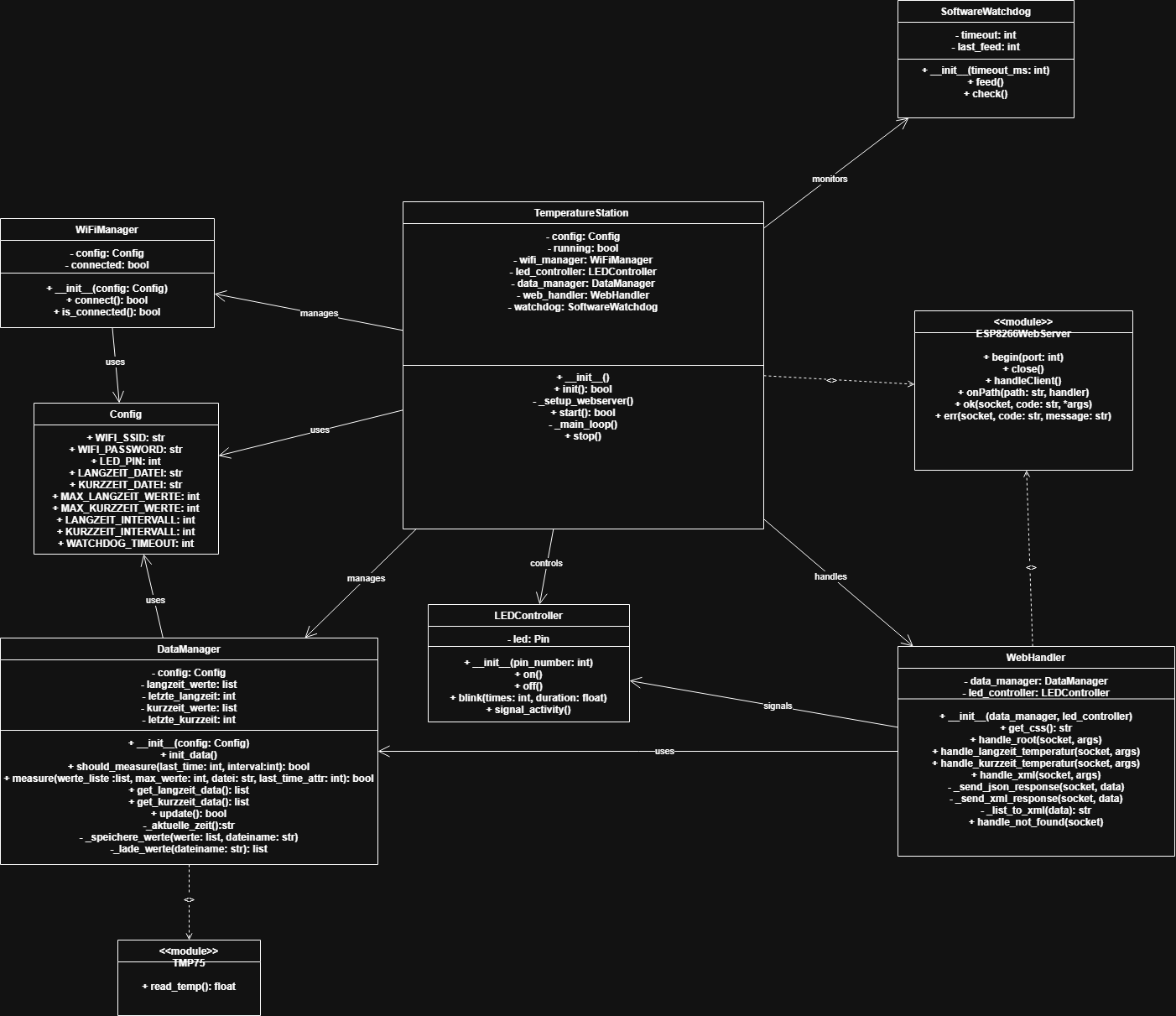
* 1. Flussdiagramm von Main-Klasse

Flussdiagramm 2



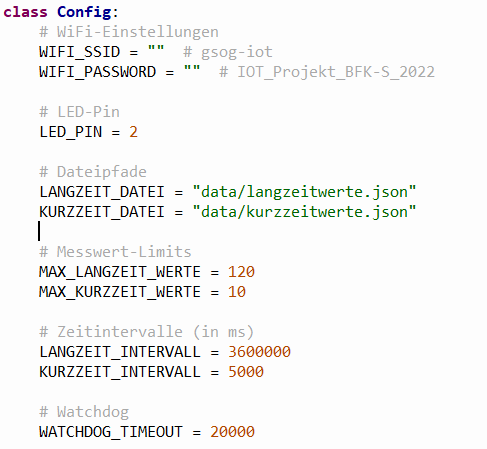
* 1. Klassendiagramm des ganzen Systems

Klassendiagramm 1



* 1. Konfigurationsdatei *Config.py*

Konfiguration 1

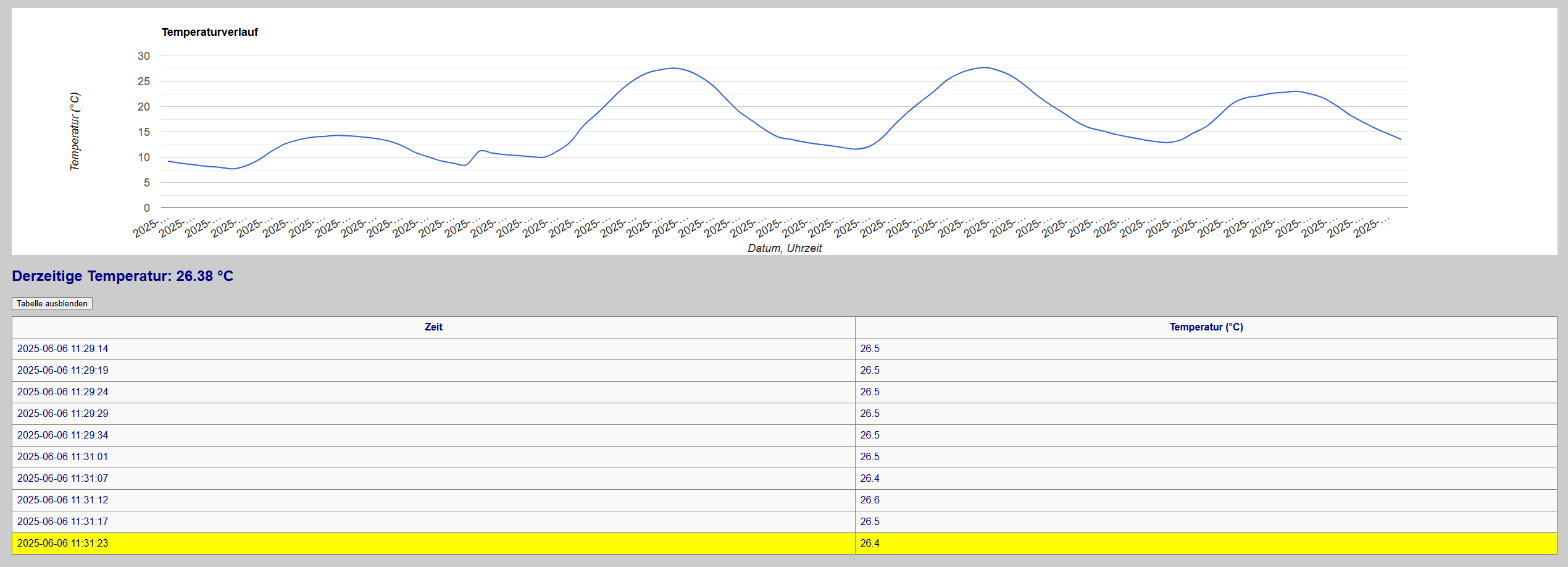


* 1. Minimalistische HTML-Struktur des ESP

minimalistische HTML-Struktur 1

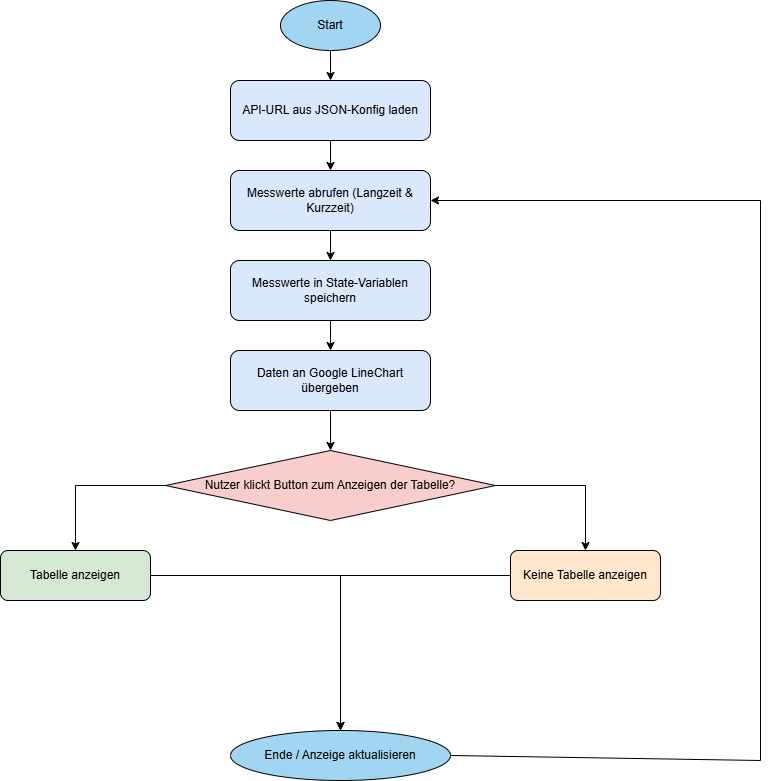


* 1. Visualisierung von Temperaturdaten



Visualisierung von Temperaturdaten 1

* 1. Funktionsweise Frontend



Flussdiagramm 3

1. (Wemos D1 Mini) [↑](#footnote-ref-1)
2. (CJMCU LM75 Temperatursensor) [↑](#footnote-ref-2)
3. (Thonny) [↑](#footnote-ref-3)
4. (React Google Charts) [↑](#footnote-ref-4)
5. (React-js) [↑](#footnote-ref-5)