

PERVASIVE COMPUTING
 Projektdokumentation

Fakultät Vermessung, Informatik und Mathematik
der Hochschule für Technik Stuttgart

Team Co2

im Juni 2025

Prüfender Professor: Prof. Dr. Stefan Knauth

Inhaltsverzeichnis

1	Einleitung	4
1.1	Ausgangslage	4
1.2	Möglichkeiten	4
2	Projektplanung	5
2.1	Idee	5
2.2	Organisation	5
2.2.1	Kommunikation	5
2.3	Zeiterfassung	6
3	Entwicklung	7
3.1	Versionsverwaltung	7
3.2	Entwicklungsumgebung	7
3.2.1	Code-Editor	7
3.3	Programmiersprachen, Paketverwaltung, Containerisierung	8
4	Hardware	9
4.1	Herausforderungen bei der Inbetriebnahme	9
4.2	Anpassungen an Firm/Hardware	9
5	Systemarchitektur	10
5.1	Datenerfassung und Verarbeitung	10
5.1.1	MQTT Broker	11
5.1.2	Stream Processing	12
5.1.3	Influx DB	15
5.1.4	Nutzung der InfluxDB UI	17
5.2	Backend Rest-API	19
5.2.1	Wahl des Backend-Frameworks	19
5.2.2	Django und Django-Ninja	19
5.2.3	Aufbau und Funktionsumfang der Backend-API	20
5.3	Testdaten-Generierung	20
5.4	Frontend	21
5.4.1	Vue.js	21
5.4.2	Komponenten	22
5.4.3	Views	23
5.4.4	Router	24
5.4.5	Auth-Store	25
5.4.6	Frontend Dokumentation	25

6	Projektstruktur	27
6.1	Übersicht der Ordner	27
6.2	Setup-Anleitung der Projektumgebung	28
6.2.1	Repository-Übersicht	28
6.2.2	Projekt starten	28
6.2.3	InfluxDB-Anbindung konfigurieren	28
6.2.4	Testdaten generieren (Entwicklung)	29
7	Fazit	30
7.1	Zielerreichung	30
7.2	Herausforderungen	30
7.3	Ausblick	30
8	Quellen	31

1 Einleitung

1.1 Ausgangslage

Während der Corona-Pandemie wurden an der Hochschule für Technik Stuttgart sogenannte CO₂-Ampeln eingeführt, Mikrocontroller mit einem CO₂-Sensor, Wlan-Anbindung und LEDs für die Darstellung der CO₂-Konzentration. Diese Ampeln messen den CO₂-Gehalt der Raumluft und stellen ihn ähnlich einer Ampel dar. So gibt es auf der Ampel LEDs, welche im Kreis angeordnet sind und sich im Uhrzeigersinn anschalten, wenn der CO₂-Gehalt steigt. Außerdem wechseln die LEDs der Ampel die Farben von grün über gelb zu rot und fangen beim Erreichen der letzten LED das blinken an. Die Ampeln hatten die Aufgabe, die Studierenden als auch Dozenten auf die erhöhte CO₂ Konzentration aufmerksam zu machen, damit der Raum gelüftet werden konnte. Man versuchte hiermit die Konzentration von Covid-Viren in der Raumluft zu reduzieren und so einen Vorlesungsablauf auch während der Pandemie zu ermöglichen.

Nach dem Ende der Pandemie wurden diese Ampeln weiter als Indikator für das Lüften verwendet, da eine erhöhte CO₂-Konzentration die Aufnahmekapazität der Studierenden beeinträchtigt.

1.2 Möglichkeiten

Ursprünglich konnten auf der Ampel bis zu 80 Messdaten zwischengespeichert werden, allerdings wurde mit den Daten im Speicher nichts weiter gemacht, sie wurden wieder überschrieben. Zusätzlich zur CO₂-Konzentration bietet die Sensorik der Ampel noch die Möglichkeit, ebenfalls die Luftfeuchtigkeit, als auch die Temperatur im Raum zu erfassen. Infolge der Verwaisung des ursprünglichen Ampelprojekts, durch das Ausscheiden des ursprünglichen Betreuers, stellte sich die Frage, wie man weiter mit diesem Projekt verfahren soll.

2 Projektplanung

Nach der Gruppenfindung starteten wir mit der Projektfindung und entschieden uns schnell für das CO₂-Ampel Projekt.

2.1 Idee

Die Projektidee bestand darin, das bestehende Projekt der CO₂-Ampel weiterzuentwickeln. In der bisherigen Implementierung war es möglich, über einen öffentlichen MQTT-Broker (<https://test.mosquitto.org/>) auf einzelne CO₂-Ampeln zuzugreifen. Dies setzte jedoch voraus, dass man die jeweilige MAC-Adresse der Ampel kannte, da diese zur Bildung des jeweiligen MQTT-Topics verwendet wurde. Die Anzeige der Sensordaten war somit nur für jeweils eine Ampel möglich, und ein übergreifender Überblick über mehrere Messpunkte gleichzeitig war nicht vorgesehen. Außerdem wurden die Daten nur für einen bestimmten Zeitraum lokal auf der CO₂-Ampel gespeichert und waren somit nicht wirklich persistent. Ziel war es daher, diese Einschränkung zu überwinden und eine skalierbare Lösung zur Aggregation und Visualisierung mehrerer Sensoren zu schaffen.

Es sollte also ein dedizierter Service aufgebaut werden. Dieser Service sammelt und bündelt zentral die Messdaten aller CO₂-Ampeln, unabhängig von den jeweiligen MAC-Adressen. Dies ermöglicht eine einheitliche Datenhaltung sowie eine systematische Auswertung. Ergänzend dazu stellt eine Webanwendung die aggregierten Werte benutzerfreundlich dar und bietet Funktionen zur Visualisierung.

2.2 Organisation

Die Organisation des Projekts erfolgte überwiegend informell. Eine klassische Aufgabenverteilung über Ticketsysteme, Kanban-Boards oder dedizierte Projektmanagement-Tools wurde nicht eingesetzt. Stattdessen stützte sich die Gruppenarbeit auf eine enge Abstimmung in Echtzeit sowie kurze Kommunikationswege.

2.2.1 Kommunikation

Die interne Kommunikation erfolgte über mehrere Kanäle:

- **WhatsApp:** Für spontane Absprachen und die tägliche Koordination.
- **Zoom Workplace:** Für virtuelle Meetings und pair-programming.
- **Projektstunden (Donnerstags):** Für wöchentliche Abprache, gemeinsame Entwicklung und Diskussionen.

Die Aufgabenverteilung wurde größtenteils mündlich oder über Textnachrichten abgestimmt. Auch der Projektfortschritt wurde auf diese Weise regelmäßig abgeglichen. Trotz

des Verzichts auf formale Tools konnte so ein kontinuierlicher Arbeitsfluss gewährleistet werden.

2.3 Zeiterfassung

Zur Erfassung der individuellen Arbeitszeiten kam die Zeiterfassungssoftware **Clockify** zum Einsatz. Die Teammitglieder haben hier ihre aufgewendeten Stunden dokumentiert, was eine transparente Übersicht über den zeitlichen Aufwand ermöglichte. Die erfassten Zeiten dienten insbesondere der internen Planung sowie der späteren Auswertung des Projektverlaufs.

3 Entwicklung

In diesem Kapitel werden die Werkzeuge, die uns das Arbeiten, als auch das Zusammenarbeiten, an der selben Code-Base ermöglicht haben erklärt.

3.1 Versionsverwaltung

Zur Verwaltung unseres Codes nutzten wir das Versionsverwaltungstool Git. Da Git ein plattformunabhängiges Programm ist und die Versionsverwaltung ohne Plattform nur lokal erfolgt, benötigten wir noch eine geeignete Plattform um unsere Versionen „abzulegen“. Hier fiel die Entscheidung auf das eigens von der HfT Stuttgart bereitgestellte Gitlab. Ein weiterer Grund für diese Entscheidung war auch der Umstand, dass unser Projekt ein bestehendes ergänzen sollte. Das zu erweiternde Projekt war ebenfalls auf dieser Plattform abgelegt, hierdurch war es uns möglich dieses Projekt zu „forken“. Bei einem Fork wird der aktuelle Projektstand in ein neues Projekt, beziehungsweise eine neue Ablage kopiert. In dieser neu entstandenen Ablage konnten wir dann eine Version entwickeln, welche unserer Zielsetzung dienlich war.

Auf der Gitlab-Plattform der HfT Stuttgart ist es möglich Nutzer in Gruppen zusammenzufassen. Diesen Umstand machten wir uns im Verlauf dieses Projekts zu nutze und erstellten uns eine Gruppe, in der wir weitere Repositories (Ablagen) erstellten. Dies geschah aus der Notwendigkeit heraus, unser Projekt in mehrere Software-Komponenten zu unterteilen um die Übersichtlichkeit zu erhöhen und eine Modularität zu gewährleisten.

3.2 Entwicklungsumgebung

Auf Grunde der weiten Verbreitung von Linux als Betriebssystem von Servern, entschieden wir uns dazu, unser Projekt ebenfalls in einer Linux-Umgebung zu entwickeln. Hierzu verwendeten wir die LTS Version 24.04 von Ubuntu, innerhalb des Windows Subsystem for Linux. Das führte zu einer nahezu nativen Linux Entwicklungserfahrung mit dem Vorteil die bekannten Windows Tools, wie beispielsweise Powerpoint, trotzdem nutzen zu können.

3.2.1 Code-Editor

Unser hauptsächlich verwendeter Code-Editor war VS Code. Für diesen Editor entschieden wir uns aufgrund seiner einfachen Remote-Verbindung auf WSL und dem großen Angebot an Erweiterungsmöglichkeiten. Durch dieses Angebot ermöglicht VS Code ein simultanes Entwickeln in mehreren Programmiersprachen. Ein weiterer Vorteil ist das „Lazy Loading“ dieser Erweiterungen, bei welchem immer nur die gerade für diese Sprache benötigten Addons geladen werden.

3.3 Programmiersprachen, Paketverwaltung, Containerisierung

C++ Im hardwarenahen Teil unseres Projekts kam die Programmiersprache C++ zum Einsatz. Hier unterstützte auch die VS Code Erweiterung PlatformIO beim Aufspielen neuer Firmwareversionen auf den Mikrocontroller.

Python In weiten Teilen unseres Projekts findet außerdem die Sprache Python Verwendung. Durch den Umstand, dass die Einbindung von weiteren Paketen in Python meist in einer virtuellen Umgebung (.venv) passiert, waren wir gezwungen uns um eine Paketverwaltung zu kümmern. Hierzu verwendeten wir den Paketmanager UV, welcher von Astral entwickelt wurde und komplett in Rust geschrieben ist, weshalb er sehr schnell ist.

Weiter bietet dieser Paketmanager den Vorteil, dass er keine lokale Python Installation benötigt, sondern diese ebenfalls in der virtuellen Umgebung einbinden kann. All unsere Unterprojekte, welche die Sprache Python verwenden, enthalten eine sogenannte `pyproject.toml` Datei. In dieser Datei werden alle Abhängigkeiten und ihre Versionen in der Python-Umgebung definiert.

Um eine einheitliche Formatierung innerhalb der Unterprojekte, die Python nutzen, zu gewährleisten verwendeten wir den Ruff-Linter, welcher ebenfalls von Astral entwickelt wurde. Auch die Eigenschaften des Linters werden in der `pyproject.toml` festgelegt, damit beispielsweise jeder Programmierer die selbe Zeilenlänge (80 Characters) nutzt.

Er erleichterte es uns, den Code immer im selben Format zuhalten, weil wir ihn normalerweise immer vor einem Commit durch einen Konsolenbefehl ausführten.

JavaScript Eine weitere Sprache, die während der Entwicklung dieses Projekts eingesetzt wurde, ist JavaScript. Hierzu installierten wir node.js als Laufzeitumgebung innerhalb der WSL. Weiter verwendeten wir nvm als JavaScript Paketmanager, da dieser mit node.js installiert wurde. Die Pakete, welche eingebunden werden sollen, werden hier in der `package.json` definiert und müssen vor der Verwendung durch npm installiert werden.

Weiter nutzten wir auch noch einige Frameworks und Programme, die aber im Laufe dieser Dokumentation erklärt werden.

Docker Das Softwarekonstrukt unseres Projekts besteht aus mehreren Komponenten, die modular Zusammenwirken. Um dieses Zusammenspiel zu erleichtern nutzen wir mehrere Dockerfiles und eine `docker-compose` Datei. Der Vorteil der hier durch entsteht, ist die schon genannte Modularität. Bei einer Weiterverwendung durch andere, besteht somit die Möglichkeit, eben nur bestimmte Teile dieses Projekts zu nutzen, da diese unabhängig voneinander funktionieren.

4 Hardware

Die Hardware der CO₂-Ampel basiert im Wesentlichen auf einem Mikrocontroller (ESP8266 oder ESP32), einem CO₂-Sensor (Sensirion SCD30) sowie einem LED-Ring (NeoPixel WS2812) zur Visualisierung der Messwerte. Das Projekt bietet damit eine leicht zugängliche Möglichkeit, den CO₂-Gehalt, die Feuchtigkeit und die Temperatur der Luft zu messen und anzuzeigen. Für unser Projekt gab es keine Notwendigkeit, die bereits existierende Hardware anzupassen.

4.1 Herausforderungen bei der Inbetriebnahme

Im Rahmen unseres Weiterentwicklungsprojekts kam es insbesondere bei dem Versuch, die ESP-Boards mit unserer abgeänderten Firmware zu flashen, zu einigen Schwierigkeiten. Nicht alle ESP8266-Boards konnten sofort geflasht werden. Die Ursache waren fehlende Treiber. Für die Serielle Schnittstelle zwischen Mikrocontroller und PC hat der Mikrocontroller einen USB-zu-Seriell-Chipsatz. Bei ESP8266-Boards ist dieser Chipsatz meist ein *CH340* oder *CH341*. Im Gegensatz zu etwas teureren Chipsätzen wie dem *CP2102* benötigen die *CH* Chipsätze auf manchen Systemen eine manuelle Treiber-Installation. Der genannte *CP2102* erfährt weniger Treiber Probleme da er einen integrierten Treiber mit sich bringt. Er wird meist in ESP32-Boards eingebaut.

Ein weiterer Aspekt war die korrekte Definition des verwendeten Boards und der Umgebung („environment“) in der Firmware. Falsch gewählte Konfigurationen im `platformio.ini`-File oder eine fehlerhafte Auswahl in der Arduino IDE führten teilweise dazu, dass der Upload Prozess nicht erfolgreich abgeschlossen werden konnte. Eine saubere Abgrenzung der Entwicklungsumgebungen für ESP8266 und ESP32, sowie eine klare Definition aller Board-spezifischen Parameter, war daher essenziell.

4.2 Anpassungen an Firm/Hardware

Unser Projekt sah es nicht vor die Hardware der CO₂-Ampel anzupassen, diese ist schließlich im ursprünglichen Projekt ausgereift und flächendeckend in der HFT deployed. Auch in der Firmware wurden nur minimale Änderungen vorgenommen. Das Json Format in dem die CO₂-Ampeln ihre MQTT Nachrichten an den Broker senden haben wir etwas angepasst.

5 Systemarchitektur

5.1 Datenerfassung und Verarbeitung

Die Datenerfassung unseres Systems erfolgt über Sensoren, die auf einem Mikrocontroller (ESP8266) betrieben werden. Diese messen regelmäßig die **CO-Konzentration**, **Temperatur** und **Luftfeuchtigkeit**. Die gesammelten Daten werden periodisch über das **MQTT-Protokoll** an einen Server übermittelt, auf dem ein MQTT-Broker läuft.

Damit dies korrekt funktioniert, muss die Datei `config.h` in der Firmware auf dem Mikrocontroller entsprechend konfiguriert werden. Folgende Parameter sind darin festgelegt:

- `WIFI_SSID`, `WIFI_PASSWORD` (WLAN-Zugangsdaten)
- `MQTT_SERVER`, `MQTT_PORT`, `MQTT_TOPIC_PREFIX` (MQTT-Einstellungen)

Diese Konfiguration ermöglicht eine hohe Flexibilität, da die Firmware so vorbereitet ist, dass sie je nach Umgebung verschiedene Übertragungswege nutzen kann. Für unser Projekt wurden folgende Dienste aktiviert bzw. deaktiviert:

- `AMPEL_WIFI`: aktiviert
- `AMPEL_MQTT`: aktiviert
- `LoRaWAN`: deaktiviert

Das Nachrichtenformat wurde zu Beginn des Projekts angepasst. Ursprünglich enthielten die MQTT-Nachrichten nur Zeitstempel und Messwerte. Die MAC-Adresse war lediglich im Topic enthalten und nicht eindeutig. Das neue Format enthält die MAC-Adresse im Payload und sieht wie folgt aus:

Code-Ausschnitt 5.1: Payload

```
1 {  
2   "metadata": {  
3     "mac-address": "AC:0B:FB:DA:9A:DB",  
4     "time": "2025-04-10 11:33:21+01"  
5   },  
6   "co2": 528,  
7   "temp": 26.0,  
8   "rh": 24.2  
9 }
```

5.1.1 MQTT Broker

Der MQTT-Broker stellt das zentrale Kommunikationselement im Projekt dar. Er ermöglicht das sogenannte Publish/Subscribe-Modell.

Wir haben uns für den Open-Source-Broker Eclipse Mosquitto entschieden, der leichtgewichtig, stabil und kompatibel mit MQTT 3.1, 3.1.1 und 5.0 ist. Mosquitto eignet sich besonders für ressourcenschonende IoT-Projekte und lässt sich gut in Containerumgebungen integrieren.

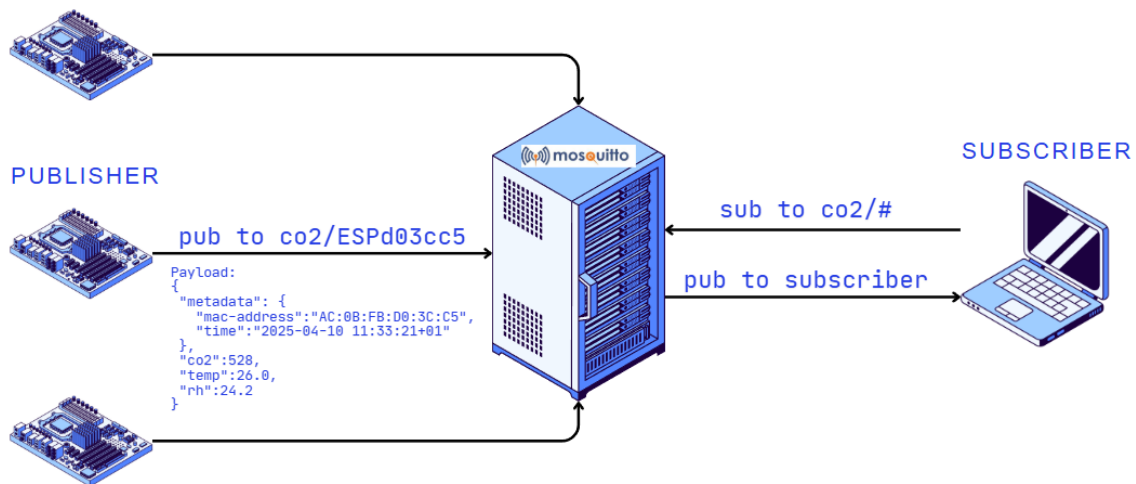


Abbildung 5.1: Schematische Darstellung der MQTT-Kommunikation im CO₂-Messsystem

Die obige Abbildung visualisiert die Kommunikation zwischen mehreren Sensorgeräten (*Publisher*), einem zentralen MQTT-Broker (*Mosquitto*) und einem Datenverarbeitungssystem (*Subscriber*).

Publisher (linke Seite)

- Mehrere Mikrocontroller (z. B. ESP8266) erfassen regelmäßig Umweltdaten wie **CO₂-Wert**, **Temperatur** und **Luftfeuchtigkeit**.
- Diese Messdaten werden im **JSON-Format** an den MQTT-Broker gesendet (**publish**).
- Das verwendete Topic lautet z. B. `co2/ESPd03cc5`.
- Die vollständige MAC-Adresse ist im Payload enthalten und ermöglicht eine eindeutige Gerätezuordnung.

MQTT-Broker (Mitte)

- Der zentrale MQTT-Broker (z. B. **Mosquitto**) empfängt die Nachrichten der Publisher.
- Er verwaltet alle Topics (z. B. `co2/ESPd03cc5`) und leitet die Nachrichten an alle Subscriber weiter.

Subscriber (rechte Seite)

- Das Verarbeitungssystem (z. B. ein **Unser Stream-Processing-Service**) abonniert das Topic `co2/#`.
- Der Wildcard `#` sorgt dafür, dass alle Nachrichten unterhalb von `co2/` empfangen werden, unabhängig vom Gerät.

Konfiguration: Die Konfiguration erfolgt über die Datei `mosquitto.conf`, in der unter anderem folgende Parameter gesetzt wurden:

Code-Ausschnitt 5.2: `mosquitto.conf`

```
1 listener 1883
2 listener 8883
3 listener 9001
4 protocol websockets
5
6 persistence true
7 persistence_file mosquitto.db
8 persistence_location /mosquitto/data/
9
10 allow_anonymous true
```

- **listener:** Definiert die Ports, auf denen der Broker Verbindungen akzeptiert.
- **protocol websockets:** Aktiviert WebSocket-Support für Web-Anwendungen.
- **persistence:** Stellt sicher, dass Nachrichten auch nach Neustarts erhalten bleiben.
- **allow__anonymous:** Erlaubt Verbindungen ohne Authentifizierung, für unsere Testumgebung ausreichend.

5.1.2 Stream Processing

Um zu verstehen, was die Aufgaben unseres Stream Processing ist, wäre es nicht schlecht zu verstehen was ein Stream ist. Ein „Stream“ bezeichnet einen kontinuierlichen Fluss von Datenpunkten, die nicht wie in einer statischen Datenbank vollständig vorliegen, sondern laufend eintreffen. Typische Eigenschaften eines Streams:

- Daten werden kontinuierlich übertragen
- Potenziell unendlich
- Zeit spielt eine zentrale Rolle („Wann wurde der Datenpunkt empfangen?“)

Idee und Ziel unseres Stream Processings

Das Ziel des Stream Processings ist es, die eintreffenden Sensordaten sofort nach Eingang zu verarbeiten, anzureichern und in die Timeseriesdatabase (InfluxDB) zu schreiben. Die Verarbeitung geschieht dabei auf der edge, wie unser ganzes Projekt. Der Vorteil: Geringe Latenz, direkte Kontrolle und keine zusätzliche Zwischenschicht.

Architektur:

Die Python-Komponente subscribed auf das MQTT-Topic `co2/#`.

Jede eintreffende Nachricht wird geparkt (JSON → Python-Dict).

Die MAC-Adresse wird mithilfe der Datei `mac_to_room.json` automatisch einem Raum zugeordnet.

Die Daten werden in die InfluxDB geschrieben, inklusive Zeitstempel und Tags für spätere Abfragen.

Technische Umsetzung

Die Verarbeitung der eingehenden Datenströme erfolgt im Python-Modul `MQTTClientHandler`. Dieses Modul ist verantwortlich für:

- die Verbindung zum MQTT-Broker,
- das Abonnieren des Datenstroms (Topics),
- das Verarbeiten der empfangenen Nachrichten und
- das Schreiben der bereinigten und angereicherten Daten in die InfluxDB.

Aufbau und Ablauf

Beim Start der Anwendung wird eine Instanz von `MQTTClientHandler` erzeugt. Die Initialisierung erfolgt in der Methode `__init__`:

Code-Ausschnitt 5.3: Initialisierung des MQTT-Clients

```
1 self.client = mqtt.Client()
2 self.client.on_connect = self.on_connect
3 self.client.on_message = self.on_message
```

Damit wird sichergestellt, dass beim Verbindungsaufbau die Methode `on_connect` und beim Empfang von Nachrichten die Methode `on_message` aufgerufen wird.

on_connect: Verbindung zum MQTT-Broker

Die Methode `on_connect` wird aufgerufen, sobald eine Verbindung zum Broker besteht:

Code-Ausschnitt 5.4: Herstellen der MQTT-Verbindung

```
1 def on_connect(self, client, userdata, flags, rc):
2     if rc == 0:
3         self.logger.info("MQTT-Verbindung erfolgreich.")
4         client.subscribe(self.topic)
```

Ein erfolgreicher Verbindungsaufbau (Return Code 0) führt dazu, dass das definierte Topic (z.B. `co2/#`) abonniert wird. Dadurch können alle relevanten Sensordaten empfangen werden.

on_message: Verarbeitung eingehender Nachrichten

Diese Methode ist der Kern des Stream-Processings. Bei jeder eingehenden Nachricht wird folgendes ausgeführt:

Code-Ausschnitt 5.5: Verarbeitung der MQTT-Nachricht

```
1 payload = json.loads(msg.payload)
2 metadata = payload.get("metadata", {})
3 mac = metadata.get("mac-address")
```

Die Nachricht wird als JSON dekodiert. Danach wird überprüft, ob die MAC-Adresse vorhanden ist und ob sie im Mapping (mac_to_room.json) bekannt ist.

Fallunterscheidung: Bekannte vs. unbekannte MAC-Adressen

Falls die MAC-Adresse unbekannt ist, wird sie automatisch dem Mapping hinzugefügt:

Code-Ausschnitt 5.6: Prüfung auf bekannte MAC-Adressen

```
1 if mac not in self.mac_to_room:
2     self._handle_unknown_mac(mac)
3     return
```

Code-Ausschnitt 5.7: Ergänzen unbekannter MAC-Adressen ins Mapping

```
1 self.mac_to_room[mac] = ""
2 jsonhandler.write_json(self.mac_to_room, self.MAPPING_FILE_NAME)
```

Dies verhindert, dass Daten unbekannter Geräte in die InfluxDB gelangen und bereitet gleichzeitig das Mapping für spätere Konfiguration vor.

_write_to_influxdb: Schreiben in die Datenbank

Ist die MAC-Adresse bekannt, wird der Datensatz in die InfluxDB geschrieben:

Code-Ausschnitt 5.8: Schreiben eines Datenpunkts in die InfluxDB

```
1 self.influx_writer.write_point(
2     measurement=self.influx_writer.MEASUREMENT_NAME,
3     tags={
4         self.influx_writer.TAG_ROOM: self.mac_to_room[metadata["mac-address"]],
5         self.influx_writer.TAG_MAC: metadata["mac-address"],
6     },
7     fields={
8         self.influx_writer.FIELD_CO2: payload["co2"],
9         self.influx_writer.FIELD_TEMP: payload["temp"],
10        self.influx_writer.FIELD_HUMIDITY: payload["rh"],
11    },
12    timestamp=metadata["time"],
13 )
```

Zuordnung der InfluxDB-Felder: Die genaue Struktur und Bedeutung der verwendeten Felder und Tags wird im nächsten Kapitel zum InfluxDB-Datenmodell erläutert.

Logging und Fehlerbehandlung

Alle wichtigen Schritte sind mit Logging-Anweisungen versehen:

Code-Ausschnitt 5.9: Fehlerbehandlung bei ungültigen JSON-Nachrichten

```
1 except json.JSONDecodeError as e:  
2     self.logger.error(f"Fehler beim Parsen der Nachricht: {e}")
```

Dies erleichtert Debugging und Monitoring. Fehlerhafte oder unvollständige Daten werden abgefangen, ohne den Ablauf zu unterbrechen.

Vorteile der Modulstruktur

- **Modularität:** Hilfsmethoden wie `_handle_unknown_mac` und `_write_to_influxdb` sorgen für wartbaren Code.
- **Robustheit:** Ungültige Daten führen nicht zum Absturz.
- **Erweiterbarkeit:** Weitere Validierungen sind leicht integrierbar.
- **Testbarkeit:** Methoden sind einzeln testbar.

5.1.3 Influx DB

Für unser Projekt war früh klar, dass wir mit sogenannten **Time Series Daten** arbeiten, also zeitlich indizierten Messwerten, wie sie typischerweise bei Sensoren auftreten (CO₂-Konzentration, Temperatur, Luftfeuchtigkeit). Für diesen speziellen Anwendungsfall bieten klassische relationale Datenbanken (wie MySQL oder PostgreSQL) nur begrenzte Unterstützung in Bezug auf Performance und Ausdrucksstärke.

Deshalb fiel unsere Wahl auf **InfluxDB**, eine speziell für Zeitreihendaten entwickelte Datenbank. Unsere Hauptgründe im Überblick:

- Zeitreihen-optimiert: schnell bei zeitbasierten Abfragen
- Open Source: vollständig kostenlos nutzbar
- Docker-freundlich: schnell aufgesetzt und portabel
- Eigene Web-Oberfläche (GUI): mit integriertem Dashboarding, Abfrageeditor und Benutzerverwaltung

Datenmodell von InfluxDB

InfluxDB verwendet ein eigenes Datenmodell, das sich vom relationalen Modell unterscheidet, jedoch eine ähnliche logische Struktur aufweist:

Tabelle 5.1: Vergleich InfluxDB und relationale Datenbank

InfluxDB-Konzept	Bedeutung	Vergleich Relationale DB
Bucket	Sammlung thematisch passender Daten (optional mit definierter Aufbewahrungszeit)	Datenbank
Measurement	Gruppe von Messwerten gleichen Typs	Tabelle
Sample	Einzelner Messpunkt mit Zeitstempel, Feldern und Tags	Zeile

Beispiel in unserem Projekt

- **Bucket:** `co2-test`
- **Measurement:** `sensor_data`
- **Tag:** MAC-Adresse, Raumnummer
- **Field:** CO₂, Temperatur, Luftfeuchtigkeit
- **Sample:** Ein Datenpunkt mit CO₂, Temperatur, Luftfeuchtigkeit, MAC-Adresse, Raumname und Zeitstempel

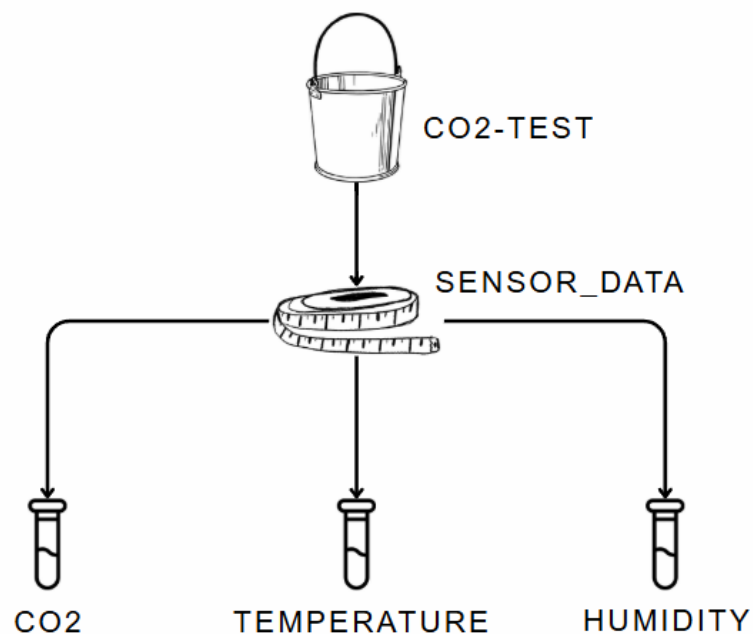


Abbildung 5.2: Schema der InfluxDB-Datenstruktur

Das Line Protocol in InfluxDB

Das **Line Protocol** ist das zentrale Datenformat, mit dem Datenpunkte in InfluxDB geschrieben werden. Es handelt sich um eine einfache, textbasierte Notation, die speziell für hohe Schreibgeschwindigkeiten optimiert wurde. Jeder Datenpunkt wird als eine einzige Textzeile dargestellt.

Allgemeine Struktur

```
measurement,tag1=value1,tag2=value2 field1=value1,field2=value2 timestamp
```

Line Protocol in unserem Projekt

In unserem Projekt sieht ein typischer Datensatz im Line Protocol folgendermaßen aus:

```
sensor_data,mac=AC:OB:FB:DA:9A:DB,room=1/210 co2=528,temperature=26.0,  
humidity=24.2 1685794100000000000
```

5.1.4 Nutzung der InfluxDB UI

Da unsere Anwendung modular aufgebaut ist, besteht neben dem Vue.js-Frontend auch die Möglichkeit, direkt die grafische Benutzeroberfläche (UI) von InfluxDB zu verwenden. Diese eignet sich besonders für Entwickler, Tester und alle, die Daten interaktiv untersuchen oder schnelle Analysen durchführen möchten, ganz ohne eigene Visualisierungslogik.

Zugriff auf die UI

Die InfluxDB UI ist über den Browser zugänglich, sobald der InfluxDB-Container läuft. Standardmäßig ist sie erreichbar unter:

```
http://localhost:8086
```

Zugangsdaten und Konfiguration

Die Zugangsdaten für die Benutzeroberfläche werden über Umgebungsvariablen und Secret-Dateien in der `docker-compose`-Konfiguration gesetzt. Diese befinden sich im Verzeichnis `services/influxdb/`.

- **Benutzername:** admin
- **Passwort:** password

Hinweis: Beim Ändern dieser Dateien ist darauf zu achten, dass keine überflüssigen Leerzeichen oder Zeilenumbrüche enthalten sind. Diese könnten von InfluxDB fälschlich eingelesen werden und zu Anmeldeproblemen führen.

Features der InfluxDB UI

1. Abfrage-Editor (Query Editor)

Im *Data Explorer* können Flux-Abfragen visuell oder manuell erstellt werden, z. B. um:

- alle Werte eines bestimmten Sensors anzuzeigen,
- Mittelwerte über Zeiträume zu berechnen,
- nach Raum oder MAC-Adresse zu filtern.

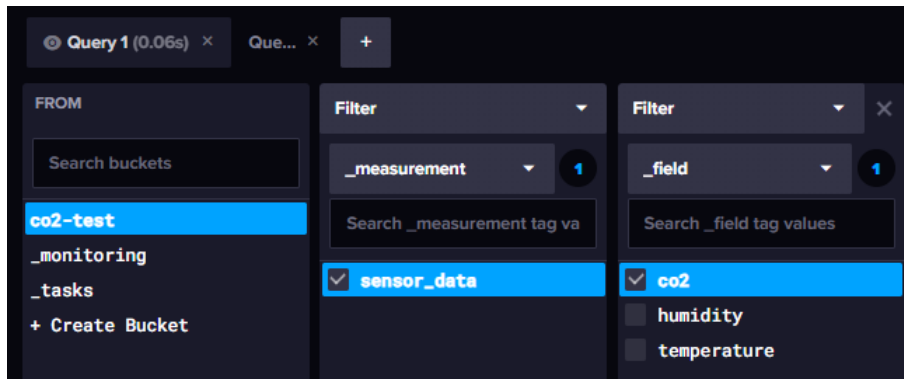


Abbildung 5.3: Query Builder Beispiel

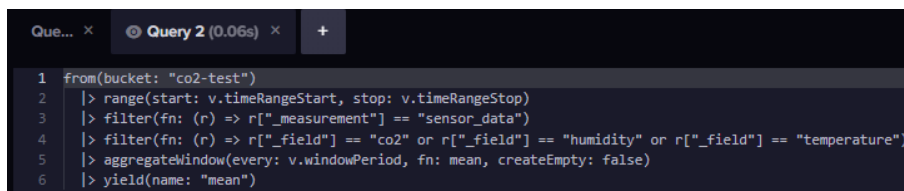


Abbildung 5.4: Script Editor Beispiel

2. CSV-Export

Abfrageergebnisse lassen sich bequem als CSV-Dateien exportieren und z. B. in Excel oder Python weiterverarbeiten.

3. Notebooks

Mit dem Notebook-Feature lassen sich benutzerdefinierte Dashboards direkt in der InfluxDB UI erstellen. Diese kombinieren mehrere Flux-Abfragen, Diagramme und Auswertungen auf einer Seite und eignen sich hervorragend für Monitoring, Fehleranalyse oder explorative Datenanalyse.

Ein Notebook kann beispielsweise alle CO₂-Werte eines bestimmten Raumes grafisch darstellen, zeitlich aggregieren (z. B. Durchschnitt pro Stunde) und mit Temperatur und Luftfeuchtigkeit vergleichen.

Die erstellten Notebooks lassen sich regelmäßig öffnen, exportieren oder sogar automatisieren, z. B. zur Berichterstellung oder zur Weiterverarbeitung in anderen Tools.

4. Dashboarding und Visualisierung

Die UI unterstützt verschiedene Visualisierungsformen, darunter Liniendiagramme, Balkendiagramme und Tabellenansichten.

5. Alerting (optional)

Über die UI können auch sogenannte Alerts definiert werden, z.B. bei Überschreitung eines CO₂-Grenzwerts. In unserem Projekt wurde dieses Feature nicht aktiv verwendet, es kann jedoch für zukünftige Erweiterungen relevant werden.

5.2 Backend Rest-API

Da wir in der Darstellung der aufgezeichneten Daten uneingeschränkte Freiheit haben wollten, entschieden wir uns dazu hierfür eine eigene Softwarelösung zu bauen. Weiter reizte uns die Herausforderung und der hierdurch entstehende Lernerfolg an diesem Unterfangen.

Um das darstellende Frontend mit den Daten aus der Influx DB zu versorgen, entschieden wir uns dazu eine Rest-API zu nutzen. Ein weiterer Vorteil dieses Vorgehens war der Umstand, dass hierdurch die Anfragen an die Datenbank, fest in der Datenbank eigenen Sprache „Influx Query Language“, im Methodenkörper der definierten Endpunkte eingebettet werden konnten.

Hierdurch war es uns möglich unsere gewünschten Datenbankanfragen konstant mit reproduzierbarem Ergebnis abzufragen. Weiter ermöglichte uns der Nutzen dieser Endpunkte das Definieren des Rückgabeformats dieser Daten, als auch ihre Aufbereitung.

5.2.1 Wahl des Backend-Frameworks

Im Zuge unserer Projektplanungsphase ist uns aufgefallen, dass uns die Verwendung der Sprache Python die Erstellung einiger Softwarekomponenten erheblich erleichtert, da mit wenigen Zeilen Code viel Funktionalität erreicht werden kann. Deshalb fiel die Wahl des Backend-Frameworks auf Django, weil es das Framework mit der größten Nutzerbasis innerhalb des Python-Kosmos ist.

5.2.2 Django und Django-Ninja

Wir starteten die Entwicklung des Backends mit dem Framework Django, wechselten, beziehungsweise erweiterten dieses dann im Laufe der Entwicklung mit Django-Ninja. Die Entscheidung zum Wechsel erfolgte auf Grund folgender Themenpunkte:

- All unsere Kommunikation mit der API erfolgt im JSON-Format, welches nativ von Django-Ninja unterstützt wird. Durch Schemas ist es möglich die Daten der Anfrage zu validieren als auch das Format der Antwort vorzugeben.
- Weiter bietet Ninja die native Einbindung und Umsetzung des OpenAPI Standards, somit werden die Endpunkte, welche die Rest-API zu Verfügung stellt automatisch dokumentiert in „Swagger UI“ (siehe Abbildung 5.5).
- Durch den Aufruf der Swagger UI URL (<http://127.0.0.1:8000/api/docs>) im Browser, ist es ebenfalls möglich die Endpunkte der API zu testen.

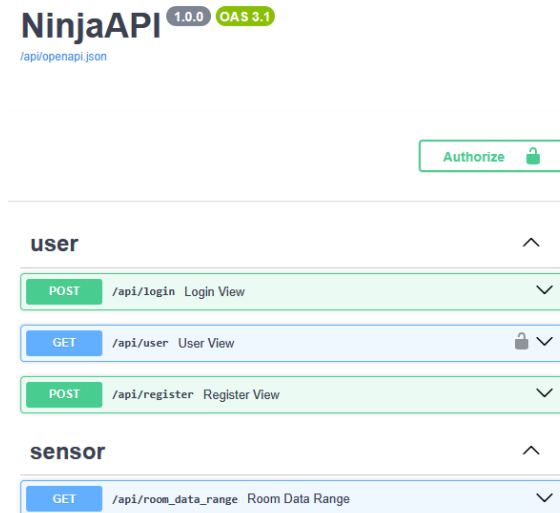


Abbildung 5.5: Dokumentation der API über Swagger

Code-Ausschnitt 5.10: Login Endpunkt

```

1 @user_router.post("/login", response=TokenOut)
2 def login_view(request, data: LoginSchema):
3     user = authenticate(username=data.username, password=data.password)
4     if user is None:
5         raise HttpError(401, "Invalid credentials")
6     token = create_jwt_token(user.id)
7     return {"token": token}

```

5.2.3 Aufbau und Funktionsumfang der Backend-API

Ein Nutzer kann sich bei der API mit einem Nutzernamen und einem Passwort registrieren. Der Name und der Passwort-Hash werden in der SQLite Datenbank von Django gespeichert. Es gibt zwei Router, die URL-Endpunkte bereitstellen. Einmal `user.py` und einmal `sensor.py`. Der User-Router stellt alle Endpunkte, die eine Nutzerinteraktion ermöglichen, beispielsweise die Login-Funktion zur Verfügung (siehe Code-Ausschnitt 5.10) als auch die Nutzerdatenabfrage und die Registrierung. Eine Logout Funktion wird im Backend nicht bereitgestellt, da wir uns für eine Authentifizierung über JWT-Token entschieden haben und dieser automatisch im Backend nach der eingestellten Zeit gelöscht wird, weshalb sich ein Nutzer dann wieder einloggen muss. Die Einstellungen hierzu findet man im Pfad: `backend/app/core/settings.py`. Alle anderen Funktionen, die Daten der CO2-Ampeln in der InfluxDB abfragen sind im Sensor-Router abgelegt. Hierdurch erreichen wir eine Modularität und könnten diese Endpunkte theoretisch auch ohne Authentifizierung nutzbar machen.

5.3 Testdaten-Generierung

Um diverse Funktionen unserer Software zu erproben bedarf es an Datenpunkten. Da nicht jedes Gruppenmitglied eine CO2-Ampel oder den Testserver (Laptop mit Linux) zuhause

hatte und somit keinen „Datengenerator“ zur Hand hatte, schrieben wir uns ein Python-Skript, das eine CO2-Ampel authentisch nachahmen konnte.

Im Ordner `data_generation` ist dieses Skript als separates Python Projekt abgelegt. Es bindet ebenfalls einen MQTT-Client ein und sendet zufällige Datenpunkte, nach dem Muster unseres „echten“ Mikrocontrollers. Die übermittelten Werte sind zwar zufällig generiert, aber realen Werten nachempfunden, dass heißt der Zufallsgenerator wurde auf eine Spanne begrenzt, welche auch in der realen Welt auftreten kann.

Wenn das Skript gestartet wird, erstellt es einen MQTT-Client, der sich mit unserem eigentlichen Broker verbindet. Während der Entwicklungsphase lief unsere Anwendung lokal auf dem jeweiligen Rechner des Gruppenmitglieds somit war der Broker unter der Adresse `http://localhost:1883` zu erreichen. Nachdem sich das Skript mit dem lokalen Broker verbunden hat, abonniert es noch das sogenannte Topic, auf welchem die Daten veröffentlicht werden sollen. Anschließend beginnt es innerhalb einer `while-True` Schleife die generierten Daten an den Broker zu senden. Um die Kadenz der gesendeten Daten zu beeinflussen, wurde eine Sleep-Time eingebaut. Es werden die MAC-Adressen aus unserem `mac_to_room.json` Mapping verwendet, um ein Testen der Darstellung genau dieser Sensoren im Frontend zu ermöglichen. Die Parameter mit welchen das Skript gestartet wird, sind in folgendem Code-Ausschnitt 5.11 zu sehen.

Code-Ausschnitt 5.11: Parameter der Testdatengenerierung

```

1 if __name__ == "__main__":
2     broker = "localhost"
3     port = 1883
4     topic = "co2/x"
5     delay = 2.0
6
7     # Change this path as needed:
8     macs_json_path = (
9         Path(__file__).parent
10        / ".."
11        / "stream_processing"
12        / "mac_to_room.json"
13    )
14
15    publish_loop(broker, port, topic, macs_json_path, delay)

```

5.4 Frontend

Unser Ziel mit dem Frontend war es, die vom Microchip-Controller erfassten Daten schnell, verständlich und effizient darzustellen. Die Informationen sollten dabei strukturiert für jedes Gebäude (Bau) und jeden Raum zugänglich sein. Zum Schutz der sensiblen Daten wurde ein Login-System integriert, das den Zugriff auf autorisierte Nutzer beschränkt.

5.4.1 Vue.js

Das Framework, für das wir uns entschieden haben, ist Vue.js. Der Hauptgrund dafür war, dass wir es in der Vorlesung Internetprogrammierung behandeln und dadurch direkt im Projekt anwenden konnten. Außerdem ist Vue einfach aufgebaut, komponentenbasiert und

eignet sich gut, um schnell übersichtliche und reaktive Benutzeroberflächen zu entwickeln. Unser Projekt folgt dem typischen Vue.js Aufbau: Es gibt Komponenten, Views, einen Router und einen kleinen Auth-Store mit Pinia.

5.4.2 Komponenten

Die Komponenten sind für einzelne, wiederverwendbare Elemente da Beispiele dafür sind:

- BauCard.vue zeigt ein Gebäude an
- RoomCard.vue einen Raum
- Chart.vue ist für die Diagramme zuständig
- NavBar.vue ist, wie der Name schon sagt, die Navigation oben

Aus diesen Komponenten bauen wir die größeren Views zusammen. Die ChartCard.vue Komponente ist dafür zuständig, eine kleine Kachel mit einem Messwert anzuzeigen, zum Beispiel Temperatur, Luftfeuchtigkeit oder CO₂-Wert in einem Raum. Sie ist in drei typische Teile gegliedert:

Template (HTML) Im Template-Teil wird das Layout beschrieben. Es gibt ein div mit CSS-Klassen, das die Werte übersichtlich anzeigt:

Code-Ausschnitt 5.12: template beispiel

```
1 <template>
2   <div class="chart-card">
3     <h3>{{ title }}</h3>
4     <Line :data="chartData" :options="chartOptions" />
5   </div>
6 </template>
```

Script (JS) Im Script-Teil wird das Liniendiagramm mithilfe von `vue-chartjs` konfiguriert. Die Farbe der Linie wird dabei nicht intern festgelegt, sondern über die Prop `borderColor` von außen übergeben, zum Beispiel durch die `ChartView`-Komponente. Auf diese Weise kann für jeden Messwert (z. B. CO₂, Temperatur, etc.) eine passende Farbe zugewiesen werden.

Code-Ausschnitt 5.13: Script beispiel

```
1 <script setup>
2 ...
3 const chartData = {
4   labels: props.labels,
5   datasets: [
6     {
7       label: props.title,
8       data: props.data,
9       fill: false,
10      borderColor: props.borderColor, % <-- Farbe von au en   begeben
11      tension: 0.1,
12    },
```

```

13 ],
14 }
15 ...
16 </script>

```

Style (CSS) Hier wird der „Style“ also wie die Komponente aussieht festgelegt:

Code-Ausschnitt 5.14: Style beispiel

```

1 <style scoped>
2 .card {
3   border: 1px solid #ccc;
4   padding: 1rem;
5   border-radius: 0.5rem;
6   text-align: center;
7 }
8 </style>

```

5.4.3 Views

Die Views sind die Seiten der App:

- LoginView.vue und RegisterView.vue: Login und Registrierung mit einfachem Formular
- HomeView.vue: Startseite nach dem Login, zeigt alle Gebäude
- BuildingsView.vue: zeigt alle Räume in einem bestimmten Gebäude
- ChartView.vue: Dort werden dann die Raumdaten die der Controller liefert als Diagramm gezeigt
- RoomNavView.vue: ist eine extra Seite zur Navigation durch Gebäude und Räume

ChartView.vue In dieser View werden die Messdaten für einen bestimmten Raum dargestellt. Dazu wird beim Laden der Seite ein API-Call an das Backend gemacht, um aktuelle Werte wie Temperatur, Luftfeuchtigkeit und CO₂ abzurufen. Der folgende Codeausschnitt zeigt, wie die Daten geladen und verarbeitet werden:

Code-Ausschnitt 5.15: Api Call beispiel

```

1 <script lang="ts">
2 ...
3 async function loadData() {
4   ...
5   const url = 'http://localhost:8000/api/room_data_range?room=
6               ${encodeURIComponent(room
7               )}&start=${startISO}&stop=${stopISO}'
8
9   const response = await fetch(url)
10  const json = await response.json()
11  ...
12 }
13 </script>

```

Code-Ausschnitt 5.16: Messwerte extrahieren

```

1 labels.value = entries.map(([timestamp]) =>
2   new Date(timestamp).toLocaleString(...)
3 )
4 co2Data.value = entries.map([, values]: any) => Number(values.co2)
5 temperatureData.value = entries.map([, values]: any)
6   => Number(values.temperature)
7 humidityData.value = entries.map([, values]: any)
8   => Number(values.humidity)

```

Übergabe an ChartCard.vue Die aufbereiteten Messdaten (Werte und Zeitstempel) werden im Template über Props an drei **ChartCard**-Komponenten übergeben. Jede Karte stellt dabei ein eigenes Diagramm dar.

Code-Ausschnitt 5.17: Daten übergabe an ChartCard

```

1 <ChartCard
2   title=" C O   Verlauf "
3   :labels="labels"
4   :data="co2Data"
5   borderColor="#ff6384"
6 />
7
8 <ChartCard
9   title="Temperatur Verlauf "
10  :labels="labels"
11  :data="temperatureData"
12  borderColor="#36a2eb"
13 />
14
15 <ChartCard
16   title="Luftfeuchtigkeit Verlauf "
17   :labels="labels"
18   :data="humidityData"
19   borderColor="#4bc0c0"
20 />

```

5.4.4 Router

Der Router (index.ts) verbindet die Views mit den URLs. Also zum Beispiel, wenn man auf `/rooms/1%2F210` geht, zeigt er die Messdaten vom Raum mit ID 1/210 an. Auch beim Login oder Logout wird man automatisch zur richtigen Seite weitergeleitet. Das läuft über den Router.

5.4.5 Auth-Store



Abbildung 5.6: Gebäude und Räume

Die Anmeldung läuft über einen kleinen Auth-Store mit Pinia.

Pinia ist das offizielle State-Management-System für Vue 3 und deutlich einfacher und moderner als sein Vorgänger Vuex.

- speichert, ob der Nutzer eingeloggt ist
- schickt Login-/Logout-Requests an das Backend
- holt den aktuellen Benutzer
- merkt sich alles im *localStorage*, damit beim Reload nichts verloren geht

5.4.6 Frontend Dokumentation

In Abbildung 5.6 ist zu erkennen, dass für jedes Gebäude einzelne Räume dargestellt werden, die interaktiv auswählbar sind. wenn man auf einer dieser Räume klickt wird eine neue Seite geöffnet auf der man dann die Daten von dem Raum sieht. (Siehe Abbildung 5.7)

Daten für Raum: 1/108

Start: 06/20/2025, 09:42 PM

Stop: 06/27/2025, 09:42 PM

Zeitraum aktualisieren

Zeige die letzte Woche

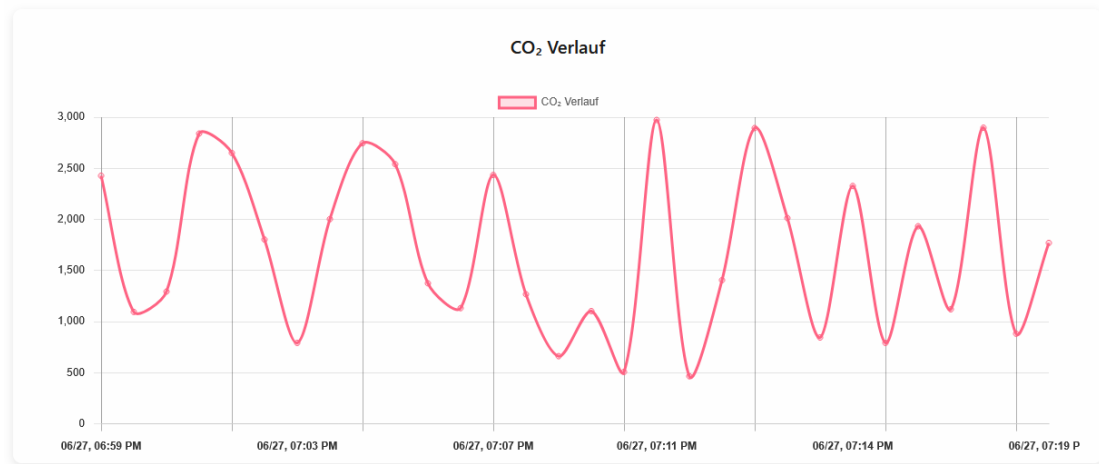


Abbildung 5.7: Daten

6 Projektstruktur

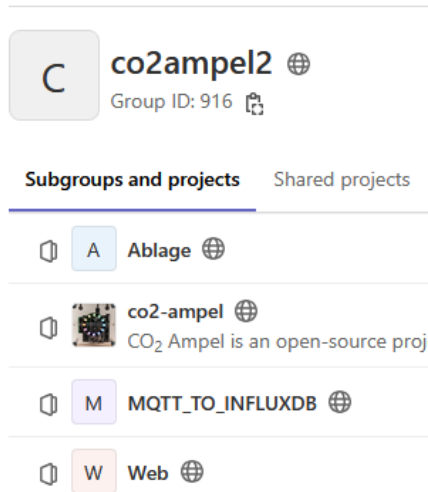


Abbildung 6.1: Gruppen-Repository

Wie bereits im Kapitel Entwicklung beschrieben haben wir eine Gruppe im Gitlab der HfT Stuttgart eingerichtet, in welcher wir die verschiedenen Unterprojekte abgelegt haben. In diesem Kapitel wollen wir nochmal auf die gesamte Projektstruktur eingehen und erklären.

Zu Beginn unsres Projektes gab es nur das „co2-ampel2“ Repository, welches eine Fork das ursprünglichen Ampel-Projektes ist. In diesem Repository ist die Anpassung der Mikrocontroller-Software durchgeführt worden. Im „MQTT_TO_INFLUXDB“ Repository wurde die Entwicklung des Stream-Processing begonnen, aber der Inhalt dieses Repositories wurde später in das „web“ Repository eingegliedert und dort weiterentwickelt. Somit sind die Repositories mit den Namen „MQTT_TO_INFLUXDB“ und „Ablage“ veraltet und mittlerweile nicht mehr in Gebrauch.

6.1 Übersicht der Ordner

Der hardwarenahe Teil ist im „co2-ampel“ Repository untergebracht und der ergänzende und gänzlich neue Softwarebaustein im „web“ Repository. Deshalb ist es angebracht ein kurze Übersicht zum web-Repository zu geben und die Softwarebausteine (Ordnerstruktur) in ihm zu erklären.

- **Backend** enthält das Django Backend und all seine Konfigurationsdateien.
- **Frontend** enthält das Vue Frontend und all seine Konfigurationsdateien.
- **data_generation** enthält das Python-Skript zur Testdatengenerierung.
- **stream_processing** enthält ein Python-Modul zur Übertragung von MQTT-Daten in die InfluxDB.
- **Services** enthält Konfigurationsdateien für den InfluxDB-Container als auch für den MQTT-Broker und dessen Datenbank.
- **utils** enthält Hilfsmodule, die von den anderen Projekten genutzt werden.

6.2 Setup-Anleitung der Projektumgebung

6.2.1 Repository-Übersicht

- **Frontend und Backend:** <https://transfer.hft-stuttgart.de/gitlab/co2ampel2/web>
- **Firmware:** <https://transfer.hft-stuttgart.de/gitlab/co2ampel2/co-2-ampel>

Hinweis: In der Datei `config.h` müssen die WLAN-Zugangsdaten sowie die Adresse der jeweiligen CO₂-Ampel korrekt eingetragen werden.

6.2.2 Projekt starten

Zum Starten der Webanwendung (Frontend + Backend) mit Docker:

Code-Ausschnitt 6.1: Docker-Projektstart

```
1 docker-compose up --build
```

6.2.3 InfluxDB-Anbindung konfigurieren

Da die InfluxDB zu Beginn nicht korrekt mit dem Django-Ninja-Backend verbunden ist, müssen API-Calls authentifiziert werden. Dazu sind folgende Schritte notwendig:

1. InfluxDB im Browser öffnen: <http://localhost:8086/signin>
2. Anmeldung:
 - Benutzername: `admin`
 - Passwort: `password`
3. Nach dem Login auf **Python** klicken und dann auf **Get Token**.
4. Den generierten Token kopieren und in der Datei `.env.docker` im Verzeichnis `backend` einfügen (ersetzen).
5. Container neu starten:

Code-Ausschnitt 6.2: Docker-Container neu starten

```
1 docker-compose down
2 docker-compose up --build
```

Nach diesen Schritten sind die API-Calls gegen die InfluxDB authentifiziert und funktionsfähig.

6.2.4 Testdaten generieren (Entwicklung)

Zur Entwicklung und zum Testen der Anwendung kann ein Daten-Generator verwendet werden. Die Schritte sind wie folgt:

1. In das Datenverzeichnis wechseln:

```
1 cd data_generation
```

2. Virtuelle Umgebung einrichten:

```
1 uv sync
```

3. Umgebung aktivieren:

```
1 source .venv/bin/activate
```

4. Das Pythonskript starten:

```
1 python main.py
```

Hinweis: Bei der Verwendung von `uv` muss das Tool gegebenenfalls zuerst installiert werden. Eine Anleitung dazu findet sich auf der offiziellen <https://pypi.org/project/uv/>.

7 Fazit

7.1 Zielerreichung

Insgesamt haben wir die wesentlichen Ziele unseres Projekts erreicht. Die CO₂-Ampeln konnten erfolgreich in ein zentrales System eingebunden werden, das die Messdaten über einen eigenen MQTT-Broker empfängt, über ein Stream-Processing-Modul verarbeitet und in einer InfluxDB speichert. Zusätzlich wurde eine Webanwendung entwickelt, die die erfassten Daten darstellt und den Zugriff auf mehrere Sensoren gleichzeitig ermöglicht.

Die Zusammenarbeit im Team hat insgesamt gut funktioniert. Auch wenn nicht immer alles sofort geklappt hat, konnten wir uns gut abstimmen und gemeinsam Lösungen finden. Die Kommunikation über verschiedene Kanäle war unkompliziert und hilfreich, besonders bei der Entwicklung und beim Testen der Software.

Der persönliche Lernerfolg war groß und hat sich bereits in anderen Projekten als hilfreich erwiesen.

7.2 Herausforderungen

Trotz eines frühen Projektstarts haben wir es nicht geschafft, direkt von Anfang an mit voller Geschwindigkeit zu arbeiten. Gerade in der Anfangsphase haben wir viel Zeit in kleinere Details investiert, was uns im späteren Verlauf etwas ausgebremst hat. Manche Probleme wie beispielsweise ein fehlerhaftes Dockerfile haben uns deutlich länger beschäftigt als geplant und den Fortschritt verzögert.

Auch in der Umsetzung kam es immer wieder zu technischen Stolpersteinen, die viel Zeit gekostet haben, obwohl die eigentlichen Aufgaben im Rückblick nicht allzu komplex waren. Hier hätten wir mit mehr Fokus auf das Wesentliche vermutlich effizienter arbeiten können.

Für zukünftige Projekte wäre es daher sinnvoll, von Beginn an klare Prioritäten zu setzen und sich nicht zu lange mit nebensächlichen Problemen aufzuhalten. Dadurch könnten technische Schwierigkeiten schneller überwunden und die Entwicklungszeit besser genutzt werden.

7.3 Ausblick

Für eine zukünftige Weiterentwicklung sehen wir mehrere sinnvolle Ansätze. Insbesondere das Frontend bietet Potenzial zur Verbesserung, etwa durch ein überarbeitetes Design oder sogar den Umstieg auf die integrierte Oberfläche der InfluxDB, um die Visualisierung direkt dort abzubilden. Außerdem wären automatisierte Deployments sinnvoll, um den Betrieb langfristig sicherzustellen.

8 Quellen

Hochschule für Technik Stuttgart: *CO2-Ampel Firmware*.

<https://transfer.hft-stuttgart.de/gitlab/co2ampel/ampel-firmware>

Code-Ausschnitte

5.1	Payload	10
5.2	mosquitto.conf	12
5.3	Initialisierung des MQTT-Clients	13
5.4	Herstellen der MQTT-Verbindung	13
5.5	Verarbeitung der MQTT-Nachricht	14
5.6	Prüfung auf bekannte MAC-Adressen	14
5.7	Ergänzen unbekannter MAC-Adressen ins Mapping	14
5.8	Schreiben eines Datenpunkts in die InfluxDB	14
5.9	Fehlerbehandlung bei ungültigen JSON-Nachrichten	15
5.10	Login Endpunkt	20
5.11	Parameter der Testdatengenerierung	21
5.12	template beispiel	22
5.13	Script beispiel	22
5.14	Style beispiel	23
5.15	Api Call beispiel	23
5.16	Messwerte extrahieren	24
5.17	Daten übergabe an ChartCard	24
6.1	Docker-Projektstart	28
6.2	Docker-Container neu starten	28

Abbildungsverzeichnis

5.1	Schematische Darstellung der MQTT-Kommunikation im CO ₂ -Messsystem	11
5.2	Schema der InfluxDB-Datenstruktur	16
5.3	Query Builder Beispiel	18
5.4	Script Editor Beispiel	18
5.5	Dokumenation der API über Swagger	20
5.6	Gebäude und Räume	25
5.7	Daten	26
6.1	Gruppen-Repository	27