Fachhochschule
Dortmund

Bachelorarbeit

Oleksii Baida
Matrikelnummer 7210384

Sicherheits- & Steuerungssytem für das Haus

Bericht

12. Februar 2025

Inhaltsverzeichnis

1 Einleitung

In einer Welt, die zunehmend von vernetzten Geräten und dem Internet der Dinge geprägt ist, wird die Entwicklung effizienter und benutzerfreundlicher Systeme zur Steuerung und Überwachung von Gebäuden immer relevanter. Im Jahr 2024 wurde in Deutschland die Anzahl von über 19 Millionen Haushalten, die ein oder mehrere smarte Geräte besaßen, verzeichnet. Es wird prognostiziert, dass sich diese Zahl innerhalb der nächsten drei Jahre verdoppeln wird [4].

Moderne Steuerungs- und Sicherheitssysteme tragen zur Effizienzsteigerung und Ressourcenschonung bei. Laut Günther Ohland, Vorstandsmitglied des Branchenverbands SSmarthome Initiative Deutschland", ermöglichen diese Systeme eine Reduktion des Heizenergieverbrauchs um 20 bis 30 Prozent [5]. Die Kosten für die smarte Technik rechnen sich in der Regel nach zwei Jahren. Die Systeme übernehmen ein Teil der täglichen Aufgaben, wie das Ein- und Ausschalten des Lichts, die Reglung der Raumtemperatur oder das Aufräumen des Hauses etc. Der Aufgabenbereich der Systeme ist dabei nur nach den Bedürfnissen der Benutzerinnen und Benutzer abgegrenzt.

Im Rahmen meiner Bachelorarbeit wird ein System zur Steuerung und Überwachung des Hauses entwickelt. Das Ziel dieser Arbeit ist die Erstellung einer Schnittstelle, die die Interaktion des Benutzers mit den Geräten in seinem Haushalt ermöglicht und den Benutzer über gefährliche Vorgänge in seinem Haus informiert.

Im Rahmen der Entwicklung dieses Systems wurden die aktuellen Technologien zur Erstellung eines Webinterfaces und zur Kommunikation zwischen den Geräten eingesetzt. **TODO Kurz erklären was in Kapitel 2,3,4,5 ... erklärt wird**

2 Grundlagen & Theorie

In diesem Abschnitt werden die technischen Spezifikationen der verwendeten Komponenten und Technologien detailliert beschrieben.

2.1 Hardware

Das Projekt umfasst verschiedene Hardware-Komponenten, darunter Mikrocontroller, Sensoren und einen Einplatinencomputer.

2.1.1 Arduino Uno

Der Arduino Uno ist ein Mikrocontroller-Board, das sich besonders für die Erfassung und Verarbeitung von Sensordaten eignet. Aufgrund seiner einfachen Programmierbarkeit und vielseitigen Schnittstellen wird es häufig in eingebetteten Systemen sowie in IoT-Anwendungen eingesetzt.

Das Herz des Boards bildet der Mikrocontroller ATmega328P, der über Flash-Speicher und EEPROMElectrically Erasable Programmable Read-Only Memory verfügt. Das EEPROM ermöglicht die nichtflüchtige Speicherung von Daten, so dass diese auch nach dem Ausschalten des Gerätes erhalten bleiben.

Die technischen Spezifikationen des Arduino Uno sind in Tabelle 1 dargestellt.

Modell	Arduino Uno
Mikrocontroller	ATmega328P
Taktfrequenz	16 MHz
Eingangsspannung	12 V
Digital Pins	14
Analoge Eingänge	6
Flash-Speicher	32 KB
SRAM	2 KB
EEPROM	1 KB
Kommunikationsschnittstellen	UART, SPI, I2C
USB-Schnittstelle	USB-B

Tabelle 1: Technische Daten des Arduino Uno

Im Rahmen dieses Projekts wird der Arduino Uno zur Erfassung und Verarbeitung von Sensordaten für die Branderkennung eingesetzt. Darüber hinaus übernimmt er die Steuerung des kontrollierten Zugangs zum Gebäude. Durch die Anbindung an verschiedene Sensoren ermöglicht er eine kontinuierliche Überwachung der Umgebung und kann im Gefahrenfall entsprechende Meldungen auslösen.

2.1.2 ESP8266

Der ESP8266 ist ein kompakter Mikrocontroller mit integriertem WLAN-Modul. Aufgrund seiner kompakten Größe, seines geringen Stromverbrauchs und seiner integrierten WLAN-Schnittstelle eignet er sich besonders für drahtlose Sensornetzwerke und eingebettete Systeme.

Ein zentrales Funktionsmerkmal des ESP8266 ist seine integrierte WLAN-Funktionalität. Das Modul kann direkt mit einem WLAN-Netzwerk verbunden werden oder als Access Point ein WLAN-Netzwerk zur Verfügung stellen. Zusätzlich kann auf dem Mikrocontroller ein einfacher Webserver eingerichtet werden. Damit kann es sowohl als eigenständiger Mikrocontroller als auch als Wi-Fi-Erweiterung für andere Mikrocontroller, z.B. einen Arduino, verwendet werden.

Das Modul basiert auf dem ESP8266 Chip von Espressif Systems. Es verfügt über Flash, SRAM und EEPROM Speicher. Außerdem unterstützt es das UART-Kommunikationsprotokoll.

Die technischen Spezifikationen des ESP8266-01 sind in der Tabelle 2 dargestellt.

Modell	ESP8266-01
Chip	ESP8266EX
Architektur	32-Bit
Taktfrequenz	80 MHz
Eingangsspannung	5 V
Flash-Speicher	1 MB
SRAM	80 KB
Kommunikationsschnittstellen	UART
WLAN-Standard	IEEE 802.11
Sicherheitsmechanismen	WEP, WPA, WPA2

Tabelle 2: Technische Daten des ESP8266

In meinem Projekt wird der ESP8266 als Schnittstelle zwischen dem Arduino und dem zentralen Server verwendet. Er ermöglicht die drahtlose Übertragung von Sensordaten an das Backend und kann auch Steuerbefehle empfangen, um entsprechende Aktionen auszulösen. Durch die direkte WLAN-Anbindung trägt der ESP8266 wesentlich zur flexiblen und ortsunabhängigen Systemsteuerung bei.

2.1.3 ESP32

Das ESP32 ist ein leistungsstarker Mikrocontroller von Espressif Systems. In der Literatur wird das Modul auch als SoC[1] bezeichnet, was bedeutet, dass alle Funktionen des Moduls auf einem Chip integriert sind.

Das ESP32 unterstützt WLAN und Bluetooth. Außerdem verfügt es über Schnittstellen wie I2C, UART, GPIOs, die eine flexible Integration mit Sensoren und anderen Peripheriegeräten ermöglichen.

Die technischen Spezifikationen des ESP32 sind in der Tabelle 3 dargestellt.

Modell	ESP32-WROOM-32
Architektur	32-Bit Dual-Core
Taktfrequenz	Bis zu 240 MHz
Betriebsspannung	3,3 V
Digitale GPIO-Pins	34
Analoge Eingänge	18
Flash-Speicher	4 MB
SRAM	520 KB
Kommunikationsschnittstellen	UART, SPI, I2C, CAN, I2S
USB-Schnittstelle	Mikro-USB
WLAN-Standard	IEEE 802.11 b/g/n
Bluetooth	BLE 4.2, Classic
Energiesparmodi	Deep-Sleep, Light-Sleep

Tabelle 3: Technische Daten des ESP32

[1] Eng. System-on-a-Chip, Deutsch: Ein-Chip-System

M5StickC

Der M5StickC ist eine kompakte Entwicklungsplattform, die auf dem ESP32 basiert. Er verfügt über ein integriertes Display, eine Batterie und zusätzliche integrierte Sensoren.

Der M5StickC verfügt über ein 1,14-Zoll großes LCD-Display, das die visuelle Ausgabe von Sensordaten oder Statusmeldungen ermöglicht. Außerdem enthält er eine integrierte IMU (Inertial Measurement Unit) zur Bewegungs- und Lageerkennung. Eine integrierte Lipo-Batterie ermöglicht den kabellosen Betrieb über einen längeren Zeitraum.

Aufgrund seiner kompakten Größe und der langen Batterielaufzeit eignet er sich besonders für IoT-Systeme.

In diesem Projekt dient der M5StickC als tragbare Steuerungs- und Überwachungseinheit. An das Modul werden Temperatur- und Lichtsensoren angeschlossen. Das Modul wird über WLAN mit dem Server verbunden. Auf dem integrierten Display werden die Daten der Sensoren sowie Verbindungszustände und Meldungen angezeigt.

2.1.4 BME680

Der BME680 ist ein kompakter Umweltsensor von Bosh, der mehrere Funktionen in einem Modul vereint. Das Modul ist mit einem Gassensor, einem Feuchtesensor, einem Luftdrucksensor und einem Temperatursensor ausgestattet. Aufgrund seiner kompakten Größe und hohen Genauigkeit wird das Modul häufig im Bereich der Umweltsensorik eingesetzt.

Mit Hilfe des eingebauten Gassensors und einer vom Hersteller speziell entwickelten Software kann die Luftqualität (IAQ - Eng. Index of Air Quality) gemessen werden. Grundsätzlich handelt es sich um eine Zahl von 0 bis 500, wobei 0 die beste und 500 die schlechteste Luftqualität darstellt. Eine detaillierte Beschreibung des Sensors sowie der Bestimmung der IAQ ist im Datenblatt zum Modul BME680 [9] unter Kapitel 1.2 zu finden. Die Tabelle 4 enthält die wichtigsten technischen Daten des Moduls.

Der BME680 verfügt über I2C- und SPI- Schnittstellen für die Kommunikation mit anderen Geräten.

Modell	BME680
Versorgungsspannung	3,3 V
Schnittstellen	I2C, SPI
Temperaturbereich	-40 bis +85 °C
Luftfeuchtigkeitsbereich	0 – 100 %
Druckbereich	300 – 1100 hPa
Luftqualität	0 - Gute Qualität 500 - schlechte Qualität

Tabelle 4: Technische Daten des BME680

In meinem Projekt wird das Modul BME680 zur Messung von Temperatur und Luftqualität eingesetzt.

2.1.5 VCNL4040

Der VCNL4040 ist ein hochpräzises optisches Modul. Das Modul enthält einen Näherungssensor (Proximity Sensor PR), einen Lichtsensor (Ambient Light Sensor) und eine Infrarot-Leuchtdiode (IRED).

Mit Hilfe des Näherungssensors und der IRED kann das Modul den Abstand zu einem Objekt bis zu einer Entfernung von 200 mm in Echtzeit messen. Der Lichtsensor misst die Helligkeit der Umgebung.

Das VCNL4040 verfügt über eine I2C-Schnittstelle zur Kommunikation mit anderen Geräten.

Eine detaillierte Beschreibung des Moduls sowie die technischen Daten sind dem Datenblatt [10] zu entnehmen. In der Tabelle 5 sind die wichtigsten Daten aufgelistet.

Modell	VCNL4040
Messprinzip	Infrarot-Näherungssensor
Versorgungsspannung	3,3 V
Schnittstellen	I2C
Erfassungsbereich	0 – 200 mm
Maximale Abtastrate	20 Hz

Tabelle 5: Technische Daten des VCNL4040

In diesem Projekt wird das Modul als Lichtsensor zur Bestimmung der Umgebungshelligkeit eingesetzt. Außerdem wird es zur Näherungserkennung verwendet, um berührungslose Steuerungen zu ermöglichen.

2.1.6 Raspberry Pi

2.2 Kommunikationsprotokolle

2.2.1 HTTP

2.2.2 MQTT

2.2.3 UART

2.2.4 I2C

2.3 Software

2.3.1 PlatformIO

2.3.2 Uvicorn

2.3.3 HTML & TailwindCSS

2.3.4 Javascript

2.3.5 WebSocket

2.3.6 Linux-Pakete für Raspberry Pi

2.3.7 Python

2.3.8 Asyncio

2.3.9 FastAPI

2.3.10 Uvicorn

2.3.11 SQLAlchemy

3 Konzeption das Systems

Im Rahmen dieses Projektes wurde ein System entwickelt, welches die Funktionalitäten eines Kontroll- und Verwaltungssystems mit denen eines IoT-Systems vereint. Die Integration von Sensordaten und Benutzerinteraktionen stellt einen wesentlichen Aspekt des Systems dar. Die Realisierung erfolgt durch die Kombination verschiedener Technologien und Teilsysteme, darunter eine Webanwendung auf FastAPI, eine MQTT-Kommunikationsschicht und verschiedene Aktoren und Sensoren, die auf Arduino- oder ESP-Module basieren.

3.1 Komponenten des Systems

Das entwickelte System basiert auf einer modularen Architektur, die mehrere Komponenten integriert. Jede dieser Komponenten erfüllt eine spezifische Rolle im Gesamtsystem:

- **Raspberry Pi:** Der zentrale Server, der das lokale WLAN-Netzwerk bereitstellt, den MQTT-Broker hostet und die Webanwendung ausführt.
- **Arduino:** Ausgestattet mit mehreren Sensoren, die Gefahren wie Feuer und Gas erkennen und den Zugang zum Haus sichern. Entsprechende Meldungen werden an den Server gesendet.
- **M5Stick:** Angeschlossen an die Temperatur- und Lichtsensoren und sendet die aufgezeichneten Daten auf den Server.
- **ESP32:** TTTOOOOODDDDOOOO
- **Webserver:** Eine auf FastAPI basierende RESTful API, die Benutzern den Zugriff auf das System und die Steuerung von Geräten ermöglicht.
- **Datenbank:** Eine lokale Datenbank zur Speicherung von Benutzerinformationen und Gerätekonfigurationen.
- **Web-Anwendung:** Eine browserbasierte Benutzeroberfläche, die den Benutzern eine intuitive Steuerung und Visualisierung der Daten ermöglicht.

3.2 Architektur und Datenfluss

Der Raspberry Pi dient als zentraler Server des Systems. Der Minicomputer stellt ein WLAN-Netzwerk zur Verfügung und hostet den Webserver mit der Datenbank sowie den MQTT-Broker. Alle Benutzerinteraktionen, Sensordaten, Datenflüsse und Datenverarbeitungen finden auf dem Server statt. Das System ist somit stark zentralisiert und arbeitet nur lokal. Das bedeutet, dass sich alle Benutzer in einem lokalen Netzwerk mit dem Raspberry Pi befinden müssen.

Die Geräte verbinden sich mit dem WLAN, das vom Raspberry Pi zur Verfügung gestellt wird. Die Sensoren senden ihre Daten an den MQTT-Broker, der auf dem Raspberry Pi läuft. Die Aktoren abonnieren die Command-Topics mit der entsprechenden "Geräte-ID".

Im Kern des Systems befindet sich ein Webserver, der auf dem Raspberry Pi ausgeführt wird. Dieser Webserver stellt eine RESTful-API zur Verfügung. Für den Zugriff zu der API wurde eine Webseite aufgebaut. Die API bietet folgende Funktionen an:

- Authentifizierung und Autorisierung der Benutzer
- Verwaltung der Gerätekonfigurationen und Benutzerprofile

- Bereitstellung von Endpunkten zur Abfrage und Steuerung von IoT-Geräten
- Bereitstellung der Daten von den Geräten in Echtzeit
- Bidirektionale Kommunikation mit den IoT-Geräten

Die Benutzerauthentifizierung erfolgt über JSON Web Tokens (JWT). Bei der Anmeldung wird ein JWT erzeugt, das eine verschlüsselte Benutzer-ID enthält. Dieses Token wird auf der Seite des Benutzers gespeichert und bei jeder Anfrage an den Server gesendet, um die Identität des Benutzers zu verifizieren. JWTs haben eine festgelegte Lebensdauer, nach deren Ablauf eine erneute Anmeldung oder eine Token-Erneuerung erforderlich ist. Die Verwendung von JWT ermöglicht eine sichere, zustandslose Authentifizierung ermöglicht, da der Server keine Sitzungsdaten speichern muss.

Die Benutzerdaten und Gerätekonfigurationen werden in einer lokalen Datenbank gespeichert, die mit `SQLite` implementiert wird. `SQLite` hat den Vorteil, dass die gesamte Datenbank in einer einzigen Datei gespeichert wird, wodurch der Ressourcenverbrauch des Servers minimiert wird. Der Zugriff auf die Datenbank erfolgt über `SQLAlchemy`, eine leistungsfähige ORM-Bibliothek für Python. `SQLAlchemy` ermöglicht asynchrone Datenbankzugriffe, was die Leistung und Skalierbarkeit der Anwendung verbessert.

Die Geräte senden ihre Daten an den zentralen MQTT-Broker. Um die eingehenden Nachrichten effizient verarbeiten zu können, wird ein MQTT-Client mit Hilfe der Bibliothek `aiomqtt` implementiert. Diese Bibliothek verwendet Python `asyncio`, was eine nicht-blockierende, asynchrone Verarbeitung ermöglicht. Dadurch können Nachrichten empfangen und verarbeitet werden, ohne den Hauptprozess zu blockieren, was die Reaktionsfähigkeit des Systems verbessert. Es ermöglicht auch die parallele Verwendung mehrerer MQTT-Clients, was die Skalierbarkeit des Systems optimiert, was besonders für Systeme mit mehreren Geräten wichtig ist.

Die Gerätedaten müssen den Benutzern in Echtzeit zur Verfügung gestellt werden. Zu diesem Zweck wird `WebSocket` verwendet. Es ermöglicht eine kontinuierliche und bidirektionale Datenübertragung zwischen Server und Client. Dadurch wird sichergestellt, dass die Daten sofort und ohne wiederholte Anfragen an den Benutzer übermittelt werden.

Der Benutzer benötigt eine Schnittstelle, um auf das System zugreifen zu können. Dazu wird eine Webseite in Form einer SPA[2] entwickelt. Eine SPA lädt einmal die grundlegende HTML-, CSS- und JavaScript-Struktur und aktualisiert dann dynamisch nur die relevanten Inhalte, ohne die gesamte Seite neu zu laden. Der Hauptvorteil einer SPA liegt in der verbesserten Nutzererfahrung, da Seitenwechsel nahezu verzögerungsfrei erfolgen. Außerdem wird die Serverlast reduziert, da nur die benötigten Daten über eine API geladen werden, anstatt komplette HTML-Dokumente zu übertragen. Dies führt zu einer schnelleren und reaktionsfähigeren Anwendung, was insbesondere bei Anwendungen mit Echtzeitanforderungen von Vorteil ist.

Die für die SPA entwickelte API kann auch von anderen Anwendungen, z. B. mobilen Anwendungen, genutzt werden. Dadurch können die Anwendungen auf einer gemeinsamen Backend-Struktur aufbauen, was die Wartung vereinfacht und die Wiederverwendbarkeit des Codes erhöht.

Die vorgestellte Architektur bietet eine effiziente, skalierbare und reaktionsfähige Struktur. Der Zugriff auf das System erfolgt über die API. Durch die Verwendung asynchroner Methoden werden Blockierungen vermieden. `WebSocket` stellt die empfangenen Daten ohne serverseitige Speicherung sofort zur Verfügung. Die SPA bietet eine schnelle und benutzerfreundliche Schnittstelle mit direktem Zugriff auf die API-Endpunkte. Diese Architektur garantiert eine hohe Performance und vereinfacht sowohl die Wartung als auch zukünftige Erweiterungen des Systems.

[2] engl. Single Page Application

4 Implementierung und praktische Umsetzung

4.1 Einrichtung der Hardwarekomponenten

4.1.1 Arduino

TOODOO wird leicht geändert und erweitert.

Der Arduino mit den angeschlossenen Sensoren stellt das Sichterheitskomponente des Systems dar. Dieses Teilsystem erkennt die Gefahren von Gas und Feuer und alarmiert den Benutzer. Außerdem stellt der Arduino der gesicherte Zugang zu dem Haus durch die Eingabe einer PIN. Die Anschließung der Sensoren und Aktoren an Arduino ist in der PA1 [1] beschrieben.

Der Arduino kann nicht direkt mit einem WLAN verbunden werden, da er kein WLAN-Modul besitzt. Für diesen Zweck habe ich ein ESP8266-Modul verwendet. Der wird mit dem Arduino durch eine UART-Schnittstelle angeschlossenen und mit dem WLAN verbunden. Die Kommunikation zwischen Arduino und ESP8266 erfolgt über die serielle Schnittstelle. Der Arduino gibt die entsprechenden Kommanden durch Serial an ESP8266 weiter und der ESP8266 führt die Befehle aus. Die detaillierte Beschreibung zur Verbindung von Arduino und ESP8266 ist in dem Bericht zu meiner Projektarbeit 2 [2] Kapitel 4.1 zu finden.

Der Arduino sendet und empfängt die MQTT-Nachrichten via ESP8266. Ein Mal pro Sekunde sendet der Arduino eine MQTT-Nachricht in das Topic ßtatus/¡ID¿/ïn dem die Bereitschaft der angeschlossenen Sensoren geteilt wird. Bei der Erkennung einer Gefahr wird sofort der Alarm ausgelöst und eine MQTT-Nachricht in das Topic älarm/¡ID¿"mit dem entsprechenden Text gesendet.

4.1.2 Raspberry Pi

4.1.3 Setup der ESP-Module

In diesem Projekt werden drei verschiedene ESP-Module verwendet: ESP8266, ESP32 und der auf dem ESP32 basierende M5Stick. Jedes Modul erfüllt spezifische Aufgaben, aber alle Module müssen sowohl mit dem WLAN als auch mit dem MQTT-Broker verbunden sein. Daher ist die Implementierung der Funktion `setup()` für alle drei Module ähnlich aufgebaut.

Beim Start des ESP-Moduls wird die Funktion `setup()` (Listing **??**) aufgerufen. Zunächst wird es versucht, die WLAN-Zugangsdaten aus dem EEPROM auszulesen (Zeile 5-10). Das EEPROM wird mit einer Größe von 128 Byte initialisiert. Dies ist notwendig, bevor Daten aus dem Speicher gelesen werden können. Es werden zwei Felder vom Typ `char` wurden mit den in den eckigen Klammern angegebenen Längen erstellt (Zeile 6-7), um die aus dem EEPROM gelesenen Daten zu speichern. Beide Felder sind zunächst mit Nullen gefüllt. Der Name des WLAN-Netzes (SSID) wird ab Adresse 0 des EEPROM ausgelesen und mit der Funktion `EEPROM.get(0, eeprom_ssid)` im Char-Feld gespeichert. Das Passwort wird ab Adresse 32 aus dem EEPROM ausgelesen.

Anschließend prft die Boolean-Funktion `is_valid_string()` (Listing **??**), ob der übergebene String-Parameter eine Länge größer als 0 und kleiner als `max_length`, sowie eine korrekte Terminierung hat. Zusätzlich gibt die Funktion den Wert `False` zurück, wenn der String ein Standard-EEPROM-Zeichen mit dem Wert `0xFF` enthält.

Sind die gültigen Werte im EEPROM gespeichert, versucht das Modul, sich mit diesen Zugangsdaten mit dem WLAN zu verbinden (Zeile 14). Die Funktion `connect_wifi` (Listing **??**) schaltet das WLAN-Modul in den Modus SStationüm, was die Verbindung mit anderen WLAN-Netzwerken erlaubt. Danach wird es versucht mit dem WLAN mit übergebenen Zugangsdaten zu verbinden. In der Variable `wifi_repeat` ist die Anzahl der Versuche zur Erstellung der

Verbindung gespeichert. Jede Sekunde wird den Status der Verbindung überprüft. Wenn die Verbindung erfolgreich erstellt wurde, gibt die Funktion `True` aus.

Nach erfolgreicher Verbindung mit dem WLAN, wird die Funktion `connect_mqtt` (Listing **??**) aufgerufen. Zunächst wird die Funktion `set_topics` aufgerufen, die die Topics zum Senden und Empfangen der Nachrichten erstellt. Dabei werden die Topics mit `DEVICE_ID` für jedes Gerät individuell formuliert. Der `mqttClient` ist ein Objekt der Klasse `PubSubClient`. Die Funktion `setServer()` erhält als Parameter die IP-Adresse des MQTT-Brokers sowie den Port, auf dem der Broker lauscht. Diese Parameter sind als Konstanten definiert. Die Callback-Funktion wird beim Empfang einer MQTT-Nachricht ausgeführt und ist für jedes Modul unterschiedlich aufgebaut. Anschließend verbindet sich das Modul mit dem MQTT-Broker und abonniert das `SUBSCRIBE_TOPIC`.

Falls die aus dem EEPROM ausgelesenen Daten ungültig sind oder keine erfolgreiche WLAN-Verbindung aufgebaut werden kann, wird die Funktion setup_ap() (Listing **??**) aufgerufen. Diese Funktion versetzt das WLAN-Modul in den Modus „Access Point“ und stellt einen Webserver bereit. Die Konfiguration des Access Points erfolgt über die Befehle WiFi.softAP() und WiFi.softAPConfig(), wobei die Parameter des Access Points in Listing **??** definiert sind.

Der Webserver wird durch das Objekt `server` der Klasse `AsyncWebServer` zur Verfügung gestellt. Der Webserver wird mit dem Befehl `server.begin()` gestartet. Die HTTP-Route wird innerhalb der Funktion server.on() festgelegt. Beim Aufruf der Root-Route (/") wird eine HTTP-GET-Anfrage bearbeitet, die mit einer HTML-Seite beantwortet wird. Diese Seite, deren Inhalt in der Variablen `html_page` (Listing **??**) gespeichert ist, enthält ein Formular zur Eingabe der WLAN-Zugangsdaten.

Wenn der Benutzer auf die Taste „Submit“ drückt, wird eine HTTP-POST-Anfrage an den Server gesendet. Die Zugangsdaten werden aus dem request-Objekt ausgelesen und auf ihre maximale Länge überprüft. Wenn die Daten gültig sind, werden sie im EEPROM gespeichert. Der EEPROM wird initialisiert, die neuen Zugangsdaten werden gespeichert, und die Änderungen mit `EEPROM.commit()` übernommen. Anschließend wird das ESP-Modul mit `ESP.restart()` neu gestartet, um die neuen WLAN-Daten zu verwenden.

Diese Setup-Konfiguration wird bei allen ESP-Modulen verwendet. Je nach Modul kann diese Funktion erweitert sein. Beispielsweise wird bei Modulen mit Display angezeigt, ob die Verbindung aufgebaut wurde.

4.1.4 ESP8266

Der ESP8266 wird für die Verbindung des Arduinos mit dem WLAN verwendet. Zu seinem Aufgaben gehört das Senden und Empfangen der MQTT-Nachrichten in entsprechenden Topics. Wie in meinem Bericht zur Projektarbeit 2 [2] Kapitel 4.1 beschrieben, ist der ESP8266 über eine UART-Schnittstelle mit dem Arduino verbunden.

Beim Start des Moduls wird die Funktion `setup()` (Listing **??**) ausgeführt. Zusätzlich läuchtet der blaue LED während der Einrichtung des Moduls. Der Diode läuchtet, wenn es keine Spannung auf dem GPIO 1 fällt.

```
digitalWrite(1, LOW);   // Diode einschalten
digitalWrite(1, HIGH);  // Diode ausschalten
```

Nach der erfolgreichen Einrichtung und Verbindung mit dem WLAN sowie MQTT-Broker, schaltet der Diode aus. Bei jedem Pogrammdurchlauf wird die Funktion `readSerialData` (Listing **??**) aufgerufen. Stehen die Daten in der seriellen Schnittstelle zur Verfügung, werden diese zunächst als String ausgelesen und gespeichert. Der String wird auf ihre maximale Länge geprüft. Anschließend muss der String in ein char-Feld konvertiert werden. Um das Topic und den Text der Nachrichten zuzuordnen, wird nach der Position von „:“ gesucht. Alle Zeichen

rechts von „:“ gehören zum Text der Nachricht und werden in der Variablen „`message`“ gespeichert. Aus den Zeichen links von „:“ wird das Topic gebildet. Arduino überträgt nur die Namen der Haupttopics. Das ESP8266 erweitert das von Arduino übergebene Topic mit der ID des Gerätes und formuliert das `PUBLISH_TOPIC` (Zeile 22-32). Wenn das Topic aus irgendeinem Grund nicht korrekt formuliert werden kann, wird die Nachricht im Haupttopic gesendet. Damit ist sichergestellt, dass die Nachricht versendet wird. Sobald das ESP8266 die Daten von dem Arduino empfängt, wird die MQTT-Nachricht an den MQTT-Broker mit dem Befehl „`mqttClient.publish(topic, message)`“ versandt.

Die Verarbeitung der empfangenen Nachricht erfolgt in der Funktion `callback` (Listing **??**). Der ESP8266 abonniert nur das Topic, das in der Variablen `SUBSCRIBE_TOPIC` definiert ist. Beim Empfang der Nachricht aus dem Topic wird geprüft, ob die Nachricht aus dem dem Gerät zugewiesenen Topic stammt. Wenn dies der Fall ist, wird der Text der Nachricht über Serial an Arduino gesendet.

4.1.5 M5Stick

Der M5Stick wird zur Überwachung der Licht- und Luftbedingungen eingesetzt. Die Idee ist, dass der Sensor die aktuelle Wetterbedingungen überwacht und diese an dem Server sendet. Für diesen Zweck soll der Sensor an dem Fenster draußen positioniert werden.

Am Modul sind die Sensoren BME680 und VCNL4040 angebunden. Der BME680 erfasst Umweltdaten: Temperatur, Luftfeuchtigkeit und Gaswiderstand. Der VCNL4040 dient als Lichtsensor und misst Lichtbedingungen: Umgebungshelligkeit (Ambient Light), weißes Licht sowie Annäherung (Proximity). Diese Daten werden an dem Server via MQTT-Nachrichten gesendet.

Für die Verwendung der beiden Sensoren im Programmcode müssen die entsprechenden Bibliotheken importiert werden und die Klassenobjekten für die Sensoren erstellt werden (Listing 1).

```
1  #include <Adafruit_BME680.h>
2  #include <Adafruit_VCNL4040.h>
3  Adafruit_BME680 bme_sensor;
4  Adafruit_VCNL4040 vcnl4040 = Adafruit_VCNL4040();
```

Listing 1: M5Stick: Sensoren

Beim Start des Moduls wird die Funktion `setup()` aufgerufen. Zusätzlich zu der im Listing **??** duchgeführte Einrichtungen müssen noch die beiden Sensoren eingerichted und im Betriebszustand gebracht werden. Dacher werden im Setup die Funktionen `vcnl_setup()` und `bme_setup()` für die Einrichtung der Sensoren aufgerufen.

Listing 2: M5Stick: vcnl_setup

4.1.6 ESP32

4.2 Softwareentwicklung

In diesem Abschnitt wird die Aufbau und Entwicklung der Software für das System beschrieben. Für die Entwicklung wurden Programmiersprachen Python und JavaScript mit mehreren Bibliotheken sowie die Markierungssprache HTML und CSS für die Visualisierung der Webseite verwendet.

Das Programm läuft auf dem Raspberry Pi. Der Programmcode ist in dem Projektordner unter `bachelor/code/spa` zu finden. Das Programm ist modular aufgebaut, wobei jedes Paket (Package) in einem eigenen Unterordner organisiert ist und eine spezifische Aufgabe erfüllt. Die Struktur des Programms sieht wie folgt aus:

```
bachelor/code/spa/
|
|--- app/
|   |--- config/
|   |--- db/
|   |--- mqtt/
|   |--- webserver/
|   |--- __init__.py
|   |--- __main__.py
|
|--- requirements.txt
|--- tailwind.config.js
```

- **app/** — Haputmodul des Programms. Enthält die Startdatei `__main__.py` sowie eine Initialisierungsdatei `__init__.py`, um das gesamte Modul als Package erkennbar zu machen.
- **confif/** — Enthält die Einstellungen für die Datenbankverbindungen, MQTT-Client, Logger und den Webserver.
- **db/** — Verwaltet der Datenbank. Enthält die Funktionen für die Verbindung zur Datenbank, die Erstellung von Tabellen und die Interaktion mit der Datenbank.
- **mqtt/** — Enthält die Logik für die Kommunikation über das MQTT-Protokoll. Enthält Code für den Empfang, die Verarbeitung und das Senden von MQTT-Nachrichten.
- **webserver/** — Stellt die API-Endpunkte und eine Webseite zur Verfügung. Enthält Klassen zur Verwaltung der Benutzerinteraktionen und zur Bereitstellung der Webseite.

Zuerst müssen die benötigten Module auf dem Server installiert werden. Die Module sind in der Datei `requirements.txt` aufgelistet. Mit dem Befehl werden die Module heruntergeladen und installiert:

```
pip install -r requirements.txt
```

4.2.1 Datenbank

Struktur der Datenbank

Die Datenbank für dieses Projekt wurde mit SQLAlchemy implementiert. Ziel war es, eine relationale Datenbank zu erstellen, die modular und flexibel aufgebaut ist, sowie die Möglichkeit bietet die Datenbankoperationen ohne direkten SQL-Abfragen durchzufüchren.

Die Funktionen zur Verwaltung der Datenbank sind in einem Paket zusammengefasst. Dieses Paket bündelt alle relevanten Komponenten, wie die Konfiguration der Datenbankverbindung, das Session-Management sowie die Definition der Datenbankmodelle.Durch diese modulare Struktur kann das Datenbankpaket in anderen Klassen oder Paketen problemlos importiert werden, um eine zentrale und einheitliche Interaktion mit der Datenbank zu ermöglichen.

Die Struktur des Packets sieht wie folgt aus:

```
bachelor/code/spa/db/
|
|--- __init__.py
|--- base.py
|--- database.db
|--- models.py
|--- queries.py
```

Die Datei `database.db` stellt die Datenbank dar. Es enthält alle Tabellen und Daten, die eingetragen wurden. Die Abbildung 1 zeigt die in der Datenbank gelegte Tabellen und deren Verknüpfungen:

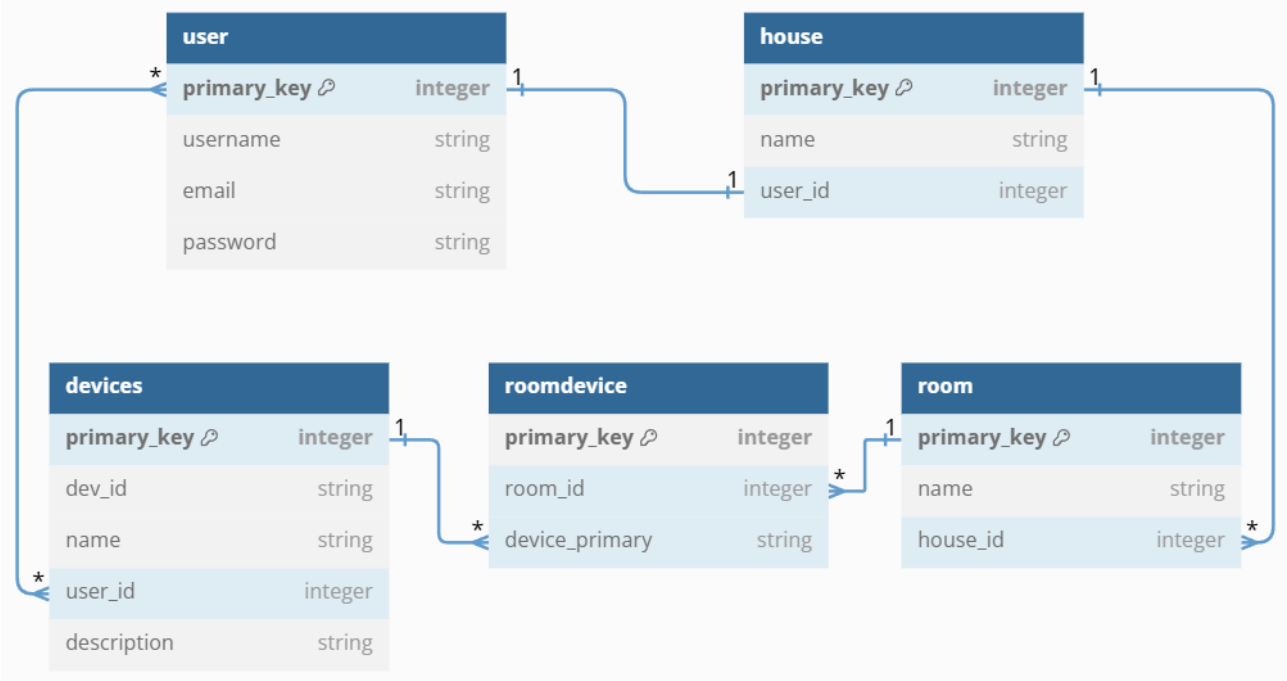

Abbildung 1: Datenbank

- **user** — Enthält die Benutzerdaten.
  - ***primary_key:*** Ein Primärschlüssel vom Typ Integer, der automatisch inkrementiert wird.
  - ***username:*** Der Benutzername, der für jeden Benutzer eindeutig ist.
  - ***email:*** Die E-Mail-Adresse des Benutzers, ebenfalls einzigartig.
  - ***password:*** Das Passwort des Benutzers, das verschlüsselt gespeichert wird.

  Ein Benutzer kann mehrere Häuser und Geräte haben.
- **house** — Speichert die vom Benutzer erstellten Häuser.

  - ***primary_key:*** Ein Primärschlüssel vom Typ Integer, der automatisch inkrementiert wird.
  - ***name:*** Der vom Benutzer erstellte Name des Hauses.
  - ***user_id:*** Ein Fremdschlüssel, der auf den Primärschlüssel der Tabelle user verweist, um anzugeben, welchem Benutzer das Haus gehört.

  Ein Haus gehört zu einem bestimmten Benutzer. Ein Haus kann mehrere Räume enthalten.

- **room** — Speichert die vom Benutzer erstellten Räume.
  - ***primary_key:*** Ein Primärschlüssel vom Typ Integer, der automatisch inkrementiert wird.
  - ***name:*** Der vom Benutzer erstellte Name des Raums.
  - ***house_id:*** Ein Fremdschlüssel, der auf den Primärschlüssel der Tabelle house verweist.

  Ein Raum gehört immer zu einem bestimmten Haus. Ein Raum kann mehrere Geräte enthalten.

- **devices** — Speichert die vom Benutzer angemeldeten Geräte.
  - ***primary_key:*** Ein Primärschlüssel vom Typ Integer, der automatisch inkrementiert wird.
  - ***dev_id:*** Die eindeutige, vordefinierte ID des Geräts.
  - ***name:*** Der vom Benutzer erstellte Name des Geräts.
  - ***user_id:*** Ein Fremdschlüssel, der auf den Primärschlüssel der Tabelle user verweist und angibt, welchem Benutzer das Gerät gehört.
  - ***description:*** Eine optionale Beschreibung des Geräts.

  Ein Gerät gehört genau einem Benutzer und einem Raum. Ein anderes Gerät mit derselben Geräte-ID kann jedoch von einem anderen Benutzer angemeldet und einem anderen Raum zugeordnet werden.

- **roomdevice** — Speichert die Zuordnung der Geräte zu den Räumen.
  - ***primary_key:*** Ein Primärschlüssel vom Typ Integer, der automatisch inkrementiert wird.
  - ***room_id:*** Ein Fremdschlüssel, der auf den Primärschlüssel der Tabelle room verweist.
  - ***device_primary:*** Ein Fremdschlüssel, der auf den Primärschlüssel der Tabelle device verweist.

Aufbau mit SQLAlchemy

Jetzt müssen diese Tabellen und deren Verknüpfungen in Form der Modellen in SQLAlchemy definiert werden. Die Models sind in der Datei `models.py` (Listing **??**) definiert.

Zunächst müssen die benötigte Module importiert werden. Das Modul `bcrypt` wird für die Entschlüssertung und Verifizierung der Passwörter verwendet. Die Module aus der Bibliothek `sqlalchemy` enthalten die Funktion für die Interaktionen mit der Datenbank. Das Modul `.base` ist die Datei `base.py` (Lisitng 3). Da wird nur ein `declarative_base` importiert, und in der Variable `Base` initialisiert.

```
1  from sqlalchemy.orm import declarative_base
2  Base = declarative_base()
```

Listing 3: db: base.py

Die Tabellenmodelle werden in Klassen definiert, die von der `Base` erben. Dies ist die Standardeinstellung für SQLAlchemy. Die Variable `__tablename__` legt der Name der Tabelle fest. Die Spalten sind als Objekte der Klasse `Column` mit gegebenen Parametern definiert. Die Klasse `UserModel` enthält eine Funktion für die Validierung der Passwörter (Zeile 17). Die Funktion bekommt ein Password-String als Parameter und vergleicht dieser mit dem im Modell gespeicherten Passwort.

In SQLAlchemy, wenn ein `ChlidModel` Modell einen Fremdschlüssel des `ParentModel` Modells beinhaltet, muss eine Beziehung mit dem `ChlidModel` Modell in der `ParentModel` Klasse definiert werden. Dies wird verwendet, um Datenbankkonflikte beim Löschen oder Aktualisieren von Einträgen im Modell `ChlidModel` zu vermeiden.

Die Klasse `HouseModel` enthält neben der Definition der Spalten eine Beziehung zwischen `HouseModel` und `RoomModel` im Attribut `rooms`. Die Beziehung ist so definiert, dass alle zugehörigen Räume gelöscht werden, wenn das Haus gelöscht wird. Außerdem ist in der Klassenvariablen `__table_args__` mit dem Befehl `UniqueConstraint('name', 'user_id')` (Zeile 27) definiert, dass die Kombination der Spalten `name` und `user_id` eindeutig sein muss. Ein Benutzer kann also nicht 2 Häuser mit dem gleichen Namen anlegen.

Die Klassen `RoomModel` und `DeviceModel` haben eine Beziehung zur Klasse `RoomDeviceModel`, und in der Klasse `RoomDeviceModel` werden die Beziehungen zu den beiden Klassen definiert. Damit wird sichergestellt, dass beim Löschen eines Raumes oder eines Gerätes auch die entsprechende Verknüpfung gelöscht wird. Dabei ist zu beachten, dass beim Löschen eines Raumes das zum Raum gehörende Gerät nicht gelöscht wird und umgekehrt.

Interaktionen mit der Datenbank

In der Datei `queries.py` (Listing **??**) sind die benötigten Funktionen für die Interaktion mit der Datenbank definiert. Diese Funktionen ermöglichen das Einfügen, Löschen, Abrufen und Aktualisieren der Daten in der Datenbank.

Jede Funktion bekommt ein `AsyncSession` als erster Parameter. Über diese Session werden die Operationen in der Datenbank duchgeführt. Das erlaubt auch asynchroner Zugriff zur Datenbank.

Als Beispiel werde ich die Struktur der Funktion `add_user()` näher erläutern. Diese Funktion fügt einen neuen Benutzer in die Datenbank ein. Die Funktion erhält als Parameter `AsyncSession` und die Benutzerdaten, d.h. Benutzername, Email und Passwort. Zuerst wird das Passwort verschlüsselt. Anschließend wird ein Objekt `new_user` der Klasse `UserModel` mit den übergebenen Parametern erzeugt. Dieses Objekt wird mit dem Befehl `db_session.add(new_user)` in die Datenbank eingefügt. Nach dem Einfügen muss der Befehl `db_session.commit()` aufgerufen werden, um die Änderungen in der Datenbank zu speichern. Anschließend wird das `new_user` aktualisiert und der Primärschlüssel zurückgegeben.

Beim Arbeiten mit der Datenbank müssen mögliche Fehler behandelt werden. Deshalb enthält jede Funktion einen `Try-Catch-Block`.

Ein `IntegrityError` wird ausgelöst, wenn eine Datenbankoperation gegen eine Integritätsbedingung der Datenbank verletzt. Er wird bei Verletzung einer Fremdschlüsselbeziehung, der Unique Constraints oder beim Einfügen von Daten des falschen Typs erzeugt.

Ein `SQLAlchemyError` umfasst alle Fehler, die von SQLAlchemy ausgelöst werden, wenn die Operation nicht erfolgreich war. Dieser Fehler wird bei Syntaxfehlern in der SQL-Abfrage oder bei Fehlern in der Datenbankverbindung ausgelöst.

Alle anderen möglichen Fehler, die keine SQLAlchemy-Fehler sind, werden im Block Exception gesammelt.

Bei jedem Fehler wird der Fehler geloggt und die Session zurückgesetzt, damit keine unvollständigen Daten in der Datenbank gespeichert werden. Anschließend wird eine HTTPException mit Status und Beschreibung des Fehlers zurückgegeben.

Initialisierung der Datenbank

Beim Import des Moduls db wird die Klasse __init__ (Listing **??**) aufgerufen. In dieser Klasse wird zunächst der Logger initialisiert. Anschließend wird eine asynchrone Engine async_engine für die Verbindung zur Datenbank erzeugt. Mit dem Parameter async_engine wird das Objekt async_session der Klasse Sessionmaker erzeugt. Die Funktion create_tables() initialisiert alle Tabellen in den Klassenmodellen, die von der Klasse Base erben. Die Funktion get_session() gibt eine asynchrone Session aus der async_session zurück. Mit dieser Session wird dann auf die Datenbank zugegriffen.

4.2.2 Webbasierte Schnittstelle (API)

Um die Benutzerinteraktionen mit dem System zu ermöglichen, muss eine Schnittstelle gebaut werden. Diese Schnittstelle muss den gesicherten Zugriff zu dem System bieten, die notwendigen Informationen bereitstellen, auf die Benutzerinteraktionen reagieren und die Daten verwalten. Nach sorgfältiger Abwägung verschiedener Optionen habe ich mich für die Implementierung einer webbasierten Schnittstelle auf Basis von FastAPI entschieden.

Ein entscheidender Vorteil von FastAPI gegenüber anderen Frameworks wie Flask oder Django in diesem Projekt ist die native Unterstützung asynchroner Programmierung. Dadurch kann FastAPI Anfragen effizient bearbeiten, indem sie mehrere Tasks gleichzeitig ausführt, anstatt blockierend auf das Ende einzelner Prozesse zu warten. Dies ist für mein System besonders vorteilhaft, da mehrere Benutzerinteraktionen, Datenverarbeitung und auch MQTT-Kommunikation parallel ablaufen müssen. Darüber hinaus bietet FastAPI automaisch generierte Dokumentation für die API.

Die zentrale Idee hinter der Entwicklung der API besteht darin, dem Benutzer eine effiziente und sichere Möglichkeit zu bieten, IoT-Geräte in einer strukturierten Umgebung zu verwalten. Dies umfasst das Hinzufügen neuer Geräte, die Gruppierung dieser Geräte in Räumen sowie die Verwaltung mehrerer Räume innerhalb eines Hauses. Darüber hinaus ermöglicht die API die kontinuierliche Überwachung und Steuerung der hinzugefügten Geräte.

Der Webserver ist, wie auch anderen Komponenten, in einem Paket namens webserver srukturiert. Die Struktur des Packets ist wie folgt aufgebaut:

```
bachelor/code/spa/webserver
|
|--- __init__.py
|--- routes.py
|--- services.py
|--- templates/
|--- static/
```

Die Kommunikation zwischen Client und Server erfolgt über HTTP-Methoden:

- **GET**: Zum Abrufen von Informationen aus dem System.
- **POST:** Zum Übermitteln neuer Daten oder Benutzerinteraktionen.
- **PUT:** Zur Aktualisierung vorhandener Informationen.

- **DELETE:** Zum Löschen von Daten.

Im Skript `routes.py` sind die API-Endpunkte definiert, die verschiedene Funktionen des Systems bereitstellen. Jeder Endpunkt ist dabei einer bestimmten Aufgabe zugeordnet.

Zur besseren Strukturierung und Modularisierung des Codes wurden die Hilfsfunktionen in das Skript `services.py` ausgelagert. Diese Funktionen enthalten die Anwendungslogik, verarbeiten die Daten und stellen sie zur Verfügung, verwalten die Verbindungen zum WebSocket.

Nach dem Import der erforderlichen Module wird ein Router mit `APIRouter()` erstellt, um die API-Endpunkte zu definieren. Der Router wird anschließend mit der Methode `include_router()` in das Hauptobjekt der FastAPI-Anwendung integriert.

Um auf Anfragen mit HTML-Dateien zu antworten, wird ein `Jinja2Templates`-Objekt erstellt. Dabei wird der Pfad zum Ordner `templates/` angegeben, der die HTML-Vorlagen enthält. Dieses Objekt ermöglicht es, dynamische HTML-Antworten zu generieren, indem Daten aus den Endpunkten in die Vorlagen eingebunden werden.

Jede Funktion, die auf die Datenbank zugreift, benötigt als Parameter eine Instanz der Klasse `AsyncSession`. Im Modul `routes.py` wird dazu die Funktion `get_session()` aus dem Paket `db` importiert, die für die Erstellung und Verwaltung von Datenbanksitzungen zuständig ist. Durch die Verwendung des Abhängigkeitsmechanismus `Depends(get_session)` wird bei jedem Aufruf einer Router-Funktion automatisch eine neue Datenbanksitzung erzeugt und an die entsprechenden Unterfunktionen weitergegeben. Dies ermöglicht eine effiziente und asynchrone Kommunikation mit der Datenbank, wodurch mehrere Anfragen gleichzeitig ohne blockierende Wartezeiten bearbeitet werden können.

Durch die Verwendung von FastAPI-Routern und einer klar strukturierten Modulanordnung bleibt die API übersichtlich, leicht wartbar und erweiterbar. Die Trennung von Präsentationslogik (HTML, CSS) und Daten (API) wird ebenfalls gewährleistet.

Sicherheit

Sicherheit spielt bei diesem System eine wichtige Rolle. Für die API gibt es zwei kritische Sicherheitspunkte: die Übertragung sensibler Daten und die Authentifizierung der Benutzer.

Um sensible Daten, wie Passwörter, sicher zu speichern, wird der Algorithmus `bcrypt` verwendet. Anstatt Passwörter im Klartext zu speichern, wird bei der Registrierung eines Benutzers das Passwort gehasht. Der Hash wird in der Datenbank gespeichert, nicht das Passwort selbst. Bei der Anmeldung wird das eingegebene Passwort mit dem gespeicherten Hash verglichen, ohne dass das Originalpasswort jemals entschlüsselt wird. Dadurch sind die Benutzerdaten auch dann geschützt, wenn die Datenbank kompromittiert wird.

Zur Authentifizierung und Identifizierung des Benutzers nach erfolgreicher Anmeldung wird das JWT-Verfahren[3] verwendet. Die Funktion `services.create_jwt_token()` generiert ein JWT-Token, das an den Benutzer zurückgegeben wird. Dieses Token enthält die Benutzer-ID in verschlüsselter Form, besitzt eine in der Konfiguration definierte Ablaufzeit (`Config.JWT_EXPIRE_TIME`) und wird mit einem geheimen Schlüssel (`Config.JWT_SECRET_KEY`) signiert.

Das Token muss clientseitig gespeichert und bei jeder Anfrage im HTTP-Header mitgesendet werden, um die Authentifizierung des Benutzers sicherzustellen. Der entsprechende Header muss dabei die folgende Struktur aufweisen:

```
headers: {
    "auth": "Bearer <JWT TOKEN>"
}
```

Das Request-Objekt wird an die Funktion `routes.get_token()` übergeben, die das JWT-Token aus dem HTTP-Header extrahiert. Dabei wird geprüft, ob der Header das Feld „`auth`“

[3] JSON Web Token

enthält und ob dessen Wert mit dem Schlüsselwort "Bearer"beginnt. Ist dies nicht der Fall, wird eine Exception ausgelöst, da das Token in einem ungültigen Format vorliegt. Im normalen Ablauf wird das Token aus dem Header extrahiert und zur weiteren Verarbeitung zurückgegeben.

Die Verifizierung des Tokens stellt eine grundlegende Voraussetzung für alle API-Funktionen dar. Nur gültige und nicht abgelaufene Tokens ermöglichen den Zugriff auf geschützte Ressourcen der API.

Nach der Extraktion wird das Token als Parameter an die Funktion `services.verify_token()` übergeben, um seine Gültigkeit zu überprüfen. Zunächst wird das Token entschlüsselt, und die darin enthaltene Benutzer-ID wird extrahiert. Ist das Token abgelaufen, ungültig, fehlerhaft oder fehlen notwendige Informationen, wird eine entsprechende Fehlermeldung ausgegeben und eine HTTP-Exception mit dem entsprechenden Statuscode ausgelöst. Andernfalls wird die Benutzerkennung für nachfolgende Verarbeitungsschritte zur Verfügung gestellt.

Anmeldung

Damit ein Benutzer die Funktionen der API nutzen kann, muss zunächst ein Konto erstellt werden. Dies erfolgt durch Senden eines `POST`-Requests an die Route `/sign_up`. Der Request muss die erforderlichen Parameter `username`, `email` und `password` enthalten. Nach Erhalt der Anfrage wird die Funktion `signup_post()` aufgerufen, die die eingegebenen Daten verarbeitet und die Benutzerregistrierung durchführt.

Die Registrierungsdaten werden als `SignUpModel`-Objekt entgegengenommen und an die Funktion `services.signup_user()` weitergeleitet. Diese Funktion übernimmt die Validierung der Eingaben, indem sie prüft, ob alle erforderlichen Felder ausgefüllt wurden. Wenn ein Wert fehlt, wird die entsprechende Fehlermeldung geloggt und zurückgegeben. Sind alle Daten vollständig, wird der Benutzer mit der Funktion `queries.add_user()` in der Datenbank gespeichert.

Nach erfolgreicher Speicherung wird ein JWT-Token generiert und zurückgegeben, das die Benutzer-ID enthält. Mit diesem Token kann der Benutzer direkt angemeldet werden. Tritt während des Prozesses eine Exception auf, so wird diese geloggt und eine entsprechende Fehlermeldung ausgegeben.

Einloggen Um einem bereits registrierten Benutzer den Zugriff auf die API zu ermöglichen, ist eine Authentifizierung erforderlich. Diese erfolgt durch das Senden eines `POST`-Requests an die Route `/login`. Der Request muss die Parameter `username` und `password` enthalten, die im `user_data`-Objekt übergeben werden. Nach Erhalt der Anfrage wird die Funktion `login_post()` aufgerufen, die die Authentifizierung des Benutzers vornimmt.

Die Benutzerdaten werden als `UserLoginModel` entgegengenommen und an die Funktion `services.auth_user()` übergeben. Diese Funktion überprüft, ob die erforderlichen Felder `username` und `password` ausgefüllt wurden. Ist es nicht der Fall, wird eine entsprechende Fehlermeldung ausgegeben. Anschließend wird mit der Funktion `queries.get_user_data()` geprüft, ob ein Benutzer mit dem angegebenen `username` in der Datenbank existiert. Wird kein entsprechender Datensatz gefunden, wird eine `HTTPException` mit dem Statuscode `401` und der Meldung „Invalid username“ ausgelöst.

Wenn der Benutzer existiert, wird das eingegebene Passwort mit der gespeicherten verschlüsselten Version verglichen. Ist die Prüfung erfolgreich, gibt die Funktion den Primärschlüssel zurück. Andernfalls wird die Fehlermeldung „Wrong password“ zurückgegeben.

Nach erfolgreicher Authentifizierung wird in der Funktion `loin_post()` ein JWT-Token erzeugt und zurückgegeben, das die Benutzerkennung enthält. Tritt während des Prozesses eine Exception auf, wird diese protokolliert und eine entsprechende Fehlermeldung ausgegeben.

Nach dem erfolgreichen Eingloggen und Erhalt des Tokens, kann der Benutzer zu den Funktionen der API zugreifen.

Haus und Raum

Der Benutzer kann seine Geräte nach Räumen und die Räume nach Häusern gruppieren. Dies erleichtert die Verwaltung und Überwachung der Geräte. Besonders im Frontend ist dies sehr nützlich, da es eine klare Strukturierung des Hauses geschaffen wird. Dabei kann ein Benutzer mehrere Häuser besitzen.

Um ein Haus zu anzulegen, muss ein `POST`-Request an die Route `add_house` gesendet werden. Der Request muss im Header ein Bearer-Token und im Body den Namen des Hauses enthalten. Nach Erhalt des Requests wird die Funktion `add_house_post()` aufgerufen. Standardmäßig wird das Token validiert, und die Benutzer-ID wird aus dem Token extrahiert, um das Haus dem entsprechenden Benutzer zuzuordnen. Wenn eine gültige Benutzer-ID aus dem Token erzeugt wurde, wird diese zusammen mit der Instanz der Klasse `AsyncSession` und dem Namen des Hauses an die Funktion `services.create_new_house()` übergeben.

Die Funktion `services.create_new_house()` prüft zunächst, ob ein gültiger Hausname übergeben wurde. Anschließend wird mit der Funktion `queries.add_new_house()` ein neuer Eintrag in der Datenbank angelegt und dem Benutzer zugeordnet. Wenn die Erstellung erfolgreich war, gibt die Funktion eine Bestätigung zurück. Andernfalls oder im Fehlerfall wird eine entsprechende Fehlermeldung generiert und geloggt.

Das Verfahren zum Hinzufügen eines Raumes zu einem House ist im Wesentlichen dasselbe wie bei der Erstellung eines House. An die Route `add_room` wird ein `POST`-Request mit dem Token im Header und mit dem Namen des Raumes sowie der zugehörigen House-ID als `RoomModel` im Inhalt gesendet. Auch hier wird das JWT-Token standardmäßig validiert, und die Benutzer-ID daraus extrahiert.

Die Daten des Raumes und die Benutzer-ID werden an die Funktion `services.add_room()` übergeben. Diese Funktion holt zunächst mit dem Befehl `queries.get_house()` die Daten des Hauses aus der Datenbank, um den Besitzer des Hauses zu bestätigen. Stimmt die übergebene Benutzer-ID mit der in der Datenbank gespeicherten Benutzer-ID des Hausbesitzers überein, wird mit der Funktion `queries.add_new_room()` ein neuer Eintrag in der Datenbank erstellt. Im Erfolgsfall wird eine Bestätigung zurückgegeben.

Wenn der Benutzer nicht berechtigt ist, einen Raum zu dem angegebenen Haus hinzuzufügen, wird eine Meldung über unberechtigten Zugriff geloggt und als Fehlerantwort zurückgegeben. Das Löschen eines Hauses oder Raumes erfolgt ähnlich wie das Hinzufügen: Es wird ein `POST`-Request an die entsprechenden Routen (`/delete_house` bzw. `delete_room`) gesendet, wobei im Header das Bearer-Token und im Inhalt des Requests die zu löschende Haus- bzw. Raum-ID übermittelt wird. Dabei ist zu beachten, dass beim Löschen eines Hauses auch alle zugehörigen Räume gelöscht werden. Dies wird durch die Beziehung `rooms` in `HouseModel` in der Datenbank definiert.

Beim Löschen eines Hauses wird ein Request mit den Benutzer- und House-ID an die Route `/delete_house` gesenden. Nach der Validierung des Tokens und Erzeugeung der Benutzer-ID werden die Daten an die Funktion `services.delete_house()` übergeben. Diese Funktion prüft, ob das Haus dem Benutzer gehört und löscht mit dem Befehl `queries.delete_house()` den zugehörigen Eintrag aus der Datenbank. Die zugehörigen Räume werden ebenfalls gelöscht. Im Fehlerfall wird der entsprechende Fehler geloggt und ausgegeben.

Das Löschen eines Raumes ist ähnlich dem Löschen eines Hauses. Ein Request mit der Raum- und Haus-ID wird an die Route `delete_room` gesendet. Nach der Validierung des Tokens und Erzeugeung der Benutzer-ID werden die Daten an die Funktion `services.delete_room()` übergeben. Wenn der Benutzer der Eigentümer des Hauses mit der übergebenen House-ID ist, wird der Raum aus der Datenbank gelöscht. Tritt ein Fehler auf, wird dieser geloggt und die entsprechende Meldung wird ausgegeben.

Die Route `/get_houses` gibt alle Häuser des Benutzers im JSON-Format zurück. Der Request muss im Header nur das Token enthalten, aus dem die Benutzer-ID erzeugt wird. Die

Benutzer-ID wird an die Funktion `services.get_houses()` übergeben, die Benutzer-ID an die Funktion `queries.get_houses_on_user()` übergibt. Diese Funktion gibt eine Liste von `HouseModel` zurück, die dem Benutzer gehören. Diese Liste wird in eine „List of Dictionaries“ umgewandelt, die alle Häuser, die zugehörigen Räume und die Geräte enthält. Anschließend wird ein JSONResponse-Objekt zurückgegeben.

Device

Die zentrale Aufgabe der API ist die Überwachung der angeschlossenen Geräte und die Benachrichtigung der Benutzer über erkannte Gefahren. Damit dieses System genutzt werden kann, muss der Benutzer die Möglichkeit haben, seine Geräte in das System einzubinden.

Jedes Gerät verfügt über eine eindeutige Geräte-ID, die im Programmcode des Gerätes fest hinterlegt ist und zur Identifikation innerhalb des Systems dient. Die Speicherung der Geräte erfolgt in der Datenbank innerhalb der Tabelle `device` in Form von `DeviceModel`-Objekten.

Die Integration eines neuen Gerätes erfolgt durch das Senden eines `POST`-Requests an die Route `/add_new_device`. Der Header des Requests muss ein gültiges JWT-Token enthalten, aus dem die Benutzer-ID ausgelesen wird. Im Body des Requests muss die eindeutige Geräte-ID angegeben werden. Optional kann ein vom Benutzer erstellter Name und eine Beschreibung des Gerätes sowie die Raum-ID für die Zuordnung des Gerätes zum Raum übergeben werden. Der Name dient ausschließlich der Benutzerfreundlichkeit, indem er eine eindeutige Identifikation des Gerätes ermöglicht. Die übergebenen Daten werden innerhalb der API als Instanz `services.DeviceModel` entgegengenommen und verarbeitet.

Nach Erhalt des Requests wird die Funktion `add_new_device_post` aufgerufen. Zuerst wird das Token validiert und die Benutzer-ID aus dem Token erzeugt. Anschließend werden die Gerätedaten an die Funktion `services.add_new_device` übergeben. Diese prüft zunächst, ob sowohl die Geräte-ID als auch der Gerätename im Request enthalten sind. Fehlt eine dieser Angaben, wird der Vorgang abgebrochen und eine Fehlermeldung zurückgegeben.

Zusätzlich hat der Benutzer die Möglichkeit, das Gerät direkt einem bestimmten Raum innerhalb des Hauses zuzuordnen. Dazu muss im Body des Requests die entsprechende Raum-ID angegeben werden. Die Funktion `services.add_new_device` sucht in diesem Fall, mit dem Befehl `queries.get_house_by_room` nach dem entsprechenden Haus. Anschließend wird geprüft, ob die aus dem Token extrahierte Benutzer-ID mit der des Hauseigentümers übereinstimmt. Ist dies nicht der Fall, wird der Zugriff verweigert, der Vorgang abgebrochen und eine entsprechende Fehlermeldung zurückgegeben.

Anschließend wird mit dem Befehl `queries.add_new_device` das Gerät zunächst ohne direkte Raumzuordnung in der Datenbank gespeichert. Ist eine Raum-ID vorhanden, wird mit dem Befehl `queries.add_room_device` ein Eintrag in der Tabelle `roomdevice` erzeugt, der den Primärschlüssel des Gerätes mit der Raum-ID verknüpft.

Diese strukturierte Vorgehensweise gewährleistet nicht nur eine sichere und konsistente Registrierung neuer Geräte, sondern bietet dem Benutzer auch die Möglichkeit, seine Smart-Home-Geräte flexibel zu verwalten und eindeutig zu identifizieren.

MQTT-Client

Die Geräte senden kontinuierlich ihre Daten über die MQTT-Verbindung an den MQTT-Broker. Jedes Gerät nutzt sein separates Topic, das sich auf die Geräte-ID bezieht. Um die Daten eines bestimmten Geräts zu empfangen, muss ein MQTT-Client das entsprechende Topic abonnieren.

Der MQTT-Client für den Webserver ist im Paket `mqtt` enthalten. In der Datei `client.py` ist die Klasse `MQTTClient` implementiert, die eine asynchrone Schnittstelle zur Verarbeitung von MQTT-Nachrichten bereitstellt.

Die Implementierung basiert auf der Bibliothek `aiomqtt`. Durch die asynchrone Architektur kann der Client effizient auf die eingehenden MQTT-Nachrichten reagieren, ohne blockierende Operationen auszuführen.

Die asynchrone Funktion `start_client()` enthält eine Endlosschleife und erwartet als Parameter eine Liste von Topics, die der Client abonnieren soll. Diese Funktion wird beim Start des Webservers mit dem Befehl `asyncio.create_task()` als asynchrone Aufgabe gestartet. Dadurch wird sichergestellt, dass die MQTT-Verbindung parallel zu anderen asynchronen Aufgaben läuft. Zudem stellt die Endlosschleife sicher, dass die Verbindung nach einer Unterbrechung oder einem Fehler automatisch wiederhergestellt wird.

Innerhalb der asynchronen Schleife wird zunächst mit dem Befehl `async with aiomqtt.Client(...)` die Verbindung zu dem MQTT-Broker aufgebaut. Anschließend werden die übergebenen Topics abonniert. Danach empfängt der Client die MQTT-Nachrichten asynchron über die async for-Schleife, die auf client.messages lauscht. Jede eingehende Nachricht wird an die Funktion `process_message()` weitergeleitet, die für die weitere Verarbeitung verantwortlich ist.

Die Funktion `process_message()` extrahiert zunächst die Geräte-ID aus dem Topic. Anschließend wird der Payload der Nachricht dekodiert und in ein JSON-Objekt data umgewandelt. Diese Daten werden dann über WebSockets an den Webserver weitergeleitet. Dazu wird die Klasse WebsocketHandler aus der Datei `services.py` importiert, und die Daten werden an die Funktion `send_data()` dieser Klasse übergeben.

Durch diese Architektur kann der Webserver in Echtzeit auf neue MQTT-Nachrichten reagieren und die empfangenen Daten effizient an verbundene Clients weiterleiten.

WebSocket

Die Geräte senden kontinuierlich Daten über die MQTT-Verbindung an den MQTT-Broker. Diese Daten müssen verarbeitet und dem Benutzer in Echtzeit angezeigt werden. Zu diesem Zweck wird eine WebSocket-Schnittstelle bereitgestellt, die eine direkte und effiziente bidirektionale Kommunikation ermöglicht.

Die Route `mqtt/device/<device_id>` stellt eine WebSocket-Verbindung zur Verfügung, wobei `device_id` für die eindeutige Geräte-ID steht. Die Klasse `WebsocketHandler` in der Datei `services.py` verwaltet sämtliche WebSocket-Verbindungen und deren Lebenszyklus. Das Feldattribut `active_connections` speichert alle aktiven WebSocket-Verbindungen in Form einer „List of Dictionaries“, wobei jedes Dictionary eine bestehende Verbindung mit der zugehörigen `device_id` sowie dem WebSocket-Objekt enthält.

Die Autorisierung des Benutzers, der eine Verbindung zum WebSocket aufbauen möchte, erfolgt über die Funktion `auth_websocket()`. Diese akzeptiert zunächst die WebSocket-Verbindung und wartet asynchron auf die erste Nachricht. Die erste Nachricht, die an dem WebSocket gesendet wird, dient als Initialisierungsnachricht und muss ein Token erhalten, das ähnlich wie ein HTTP-Header aufgebaut ist:

```
"auth": "Bearer <TOKEN>"
```

Aus dem übergebenen Token wird die Benutzer-ID extrahiert. Anschließend wird mit dem Befehl `queries.verify_user_device(...)` geprüft, ob der authentifizierte Benutzer berechtigt ist, auf das angegebene Gerät zuzugreifen. Fehlt das Token oder besitzt der Benutzer keine Zugriffsrechte auf das Gerät, wird eine entsprechende Fehlermeldung ausgegeben und geloggt.

Nach erfolgreicher Autorisierung des Benutzers wird die Funktion `connect()` aus der Klasse `WebsocketHandler` aufgerufen. Diese fügt die neue Verbindung in die Liste von aktiven Verbindungen hinzu. Die Verbindung bleibt bestehen, bis sie entweder vom Benutzer geschlossen oder durch einen Netzwerkfehler unterbrochen wird.

Die Kommunikation über das WebSocket-Protokoll erfolgt in einer Endlosschleife, die ständig auf eingehende Nachrichten wartet. Wird die Verbindung vom Benutzer oder durch andere

Ereignisse unterbrochen, wird diese mit der Funktion `disconnect` aus der Liste der aktiven Verbindungen entfernt.

Die Funktion `send_data()` ermöglicht das gezielte Senden von Daten über die WebSocket-Schnittstelle. Dabei wird geprüft, ob eine aktive WebSocket-Verbindung mit der entsprechenden `device_id` existiert. Falls ja, werden die zu sendenden Daten als JSON-Objekt über die WebSocket-Schnittstelle an das Gerät übertragen. Tritt ein Übertragungsfehler auf, wird die Verbindung geschlossen und aus der Liste der aktiven Verbindungen entfernt.

Die Implementierung dieser WebSocket-Schnittstelle ermöglicht eine effiziente und latenzarme Datenübertragung zwischen dem MQTT-Broker und den angebundenen Geräten.

4.2.3 Frontend

Das Frontend ist im Form einer Einseitigerapplikation (SPA[4]) implementiert.

4.2.4 Integration der Komponenten

MQtt client, erhalten Daten von Broker etc

[4]Eng. Single Page Applicaiton

5 Funktionsweise des Systems

Hier wird beschrieben wie der Benutzer das System verwendet und wie laufen die Prozesse im System. Es wird ein Diagramm ersetellt ähnlich zu im PA 2.

6 Ergebnisse und Diskussion

Hier wird zusammengefasst, was ich mit diesem Projekt gelernt habe, welche Ziele erreicht und nicht erreicht habe, mögliche Weiterentwicklung des Systems etc

7 Quellen

Literatur

[1] O. Baida, *Anbindung der Sensoren und Aktoren an den Arduino zur Realisierung eines Sicherheitssystems*, Projektarbeit 1, 2024.

[2] O. Baida, Projektarbeit 2 *Sicherheitssystem fur das Haus basierend auf Arduino, ESP8266 & Raspberry Pi* https://github.com/oleksiibaida/PA2.git

[3]

[4] Statista, *„Smart Home - Anzahl der Haushalte in Deutschland 2028“*, Zugegriffen: 13. Januar 2025. [Online]. Verfügbar unter https://de.statista.com/prognosen/885611/anzahl-der-smart-home-haushalte-in-deutschland

[5] J. Breithut, *„Strom und Heizung: Wann ein Smart Home wirklich beim Energiesparen hilft“*, Der Spiegel, 17. Juli 2022. Zugegriffen: 13. Januar 2025. [Online]. Verfügbar unter: https://www.spiegel.de/netzwelt/gadgets/strom-und-heizung-wann-ein-smart-home-wirklich-beim-energiesparen-hilft-a-ffb4b710-

Links zur verwendeten Hardware:

[6] Arduino.cc, *Arduino UNO*, https://docs.arduino.cc/hardware/uno-rev3/

[7] Raspberry Pi Foundation, *Raspberry Pi 1 B+*, https://www.raspberrypi.com/products/raspberry-pi-1-model-b-plus/

[8] Espressif, *ESP8266*, https://www.espressif.com/, https://www.electronicwings.com/sensors-modules/esp8266-wifi-module

[9] Bosh, *BME680 - Datasheet*, https://www.bosch-sensortec.com/media/boschsensortec/downloads/datasheets/bst-bme680-ds001.pdf, Zugriff: 4. Februar 2025.

[10] Vishay, *VCNL4040 - Datasheet*, https://www.vishay.com/docs/84274/vcnl4040.pdf, Zugriff: 5. Februar 2025.

Links zur verwendeten Software:

[11] Dr Andy Stanford-Clark, Arlen Nipper, *Message Queuing Telemetry Transport*, https://mqtt.org/

[12] Guido van Rossum, Python Software Foundation, *Python*, https://www.python.org/

[13] Telegram FZ-LLC, *Telegram Messenger*, https://github.com//telegramdesktop/tdesktop

Linux-Packete:

[14] Jouni Malinen, *hostapd*, https://w1.fi/hostapd/, Zugriff am: 19. September 2024.

[15] Simon Kelley, *dnsmasq*, https://dnsmasq.org/doc.html, Zugriff am: 20. September 2024.

[16] Eclipse Foundation, *Eclipse Mosquitto*, https://mosquitto.org/

ESP- und Arduino-Bibliotheken

[17] Knolleary, *PubSubClient*, https://pubsubclient.knolleary.net/, Zugriff am: 21. Oktober 2024.

[18] ESPWIFI.h, https://arduino-esp8266.readthedocs.io/en/latest/esp8266wifi/readme.html

[19] EEPROM.h, https://docs.arduino.cc/learn/built-in-libraries/eeprom/

[20] Keypad.h https://docs.arduino.cc/libraries/keypad/

[21] R. Scholz, *Syncloop*, Persönliche Mitteilungen

Python-Bibliotheken

[22] Pierre Fersing, Roger Light *paho-mqtt*, https://pypi.org/project/paho-mqtt/, Zugriff am: 21. Oktober 2024.

[23] Open Source, *python-telegram-bot*, https://docs.python-telegram-bot.org/en/v21.6/

[24] Python Software Foundation, *json*, https://docs.python.org/3/library/json.html

[25] Python Software Foundation, *threading*, https://docs.python.org/3/library/threading.html

[26] Python Software Foundation, *queue*, https://docs.python.org/3/library/queue.html

[27] Gerhard Häring, *sqlite3*, https://docs.python.org/3/library/sqlite3.html

[28] Lawrence Hudson, *pyzbar*, https://github.com/NaturalHistoryMuseum/pyzbar/

[29] Intel, *OpenCV*, https://github.com/opencv/opencv-python

[30] Aio-Libs, *aiohttp*, https://github.com/aio-libs/aiohttp

Abbildungsverzeichnis

Tabellenverzeichnis

8 Programmcode