Fachhochschule
Dortmund

Bachelorarbeit

Oleksii Baida
Matrikelnummer 7210384

Sicherheits- & Steuerungssytem für das Haus

Bericht

29. Januar 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 erfolgt die detaillierte Darstellung der technischen Informationen zu den verwendeten Komponenten und Technologien.

2.1 Hardware

2.1.1 Arduino

2.1.2 ESP8266

2.1.3 Raspberry Pi

2.1.4 ESP32

2.1.5 M5Stick

2.1.6 BME680

2.1.7 VCNL

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. Somit ist das System stark zentralisiert. 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
- Bidirektionale Kommunikation mit den IoT-Geräten

Die Benutzerdaten und Konfigurationen werden in der lokalen Datenbank auf dem Server gespeichert.

TOOODOOOO wird noch erweitert

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:

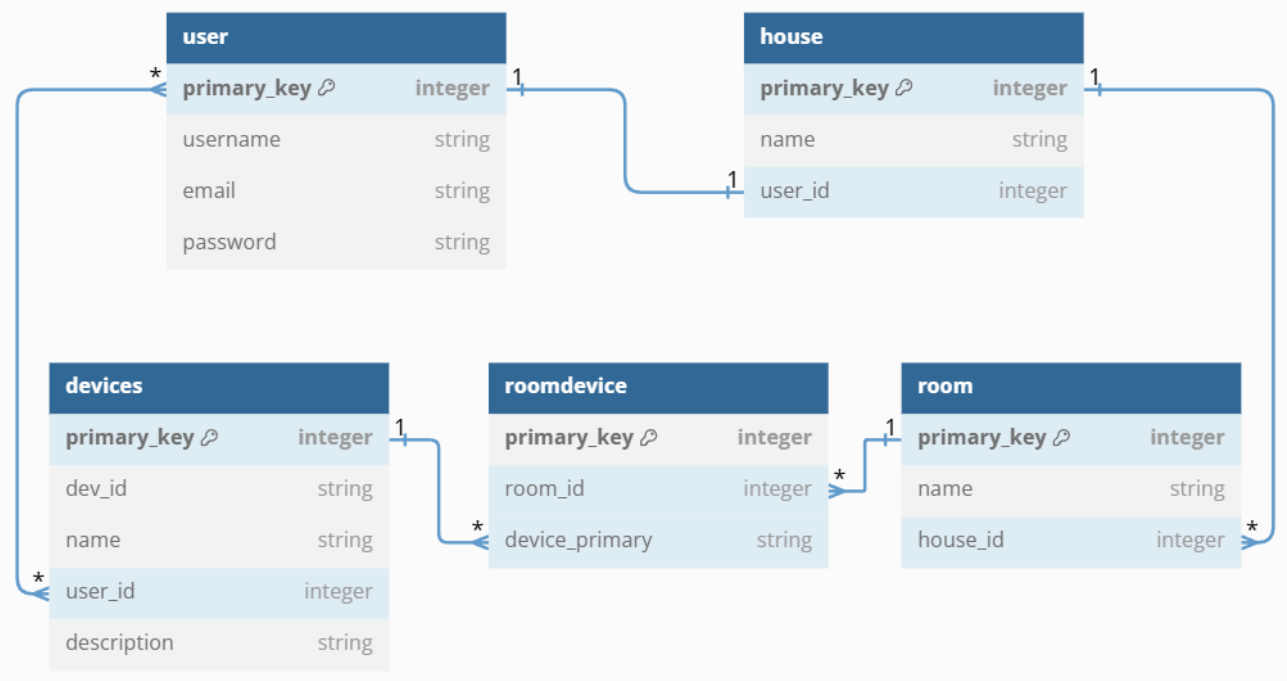

Abbildung 1: Datenbank

- **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. Das Erzeugen der Sessions wird im nächsten Paragraph beschrieben.

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 4) 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.

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.

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[1] 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`“ 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.

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.

[1] JSON Web Token

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 eine entsprechende Fehlermeldung geloggt und eine Fehlermeldung wird zurückgegeben. Sind alle Daten vollständig, werden die Daten 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.

4.2.3 Frontend

4.2.4 Integration der Komponenten

MQtt client, erhalten Daten von Broker etc

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

Links zur verwendeten Software:

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

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

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

Linux-Packete:

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

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

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

ESP- und Arduino-Bibliotheken

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

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

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

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

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

Python-Bibliotheken

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

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

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

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

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

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

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

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

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

Abbildungsverzeichnis

Tabellenverzeichnis

8 Programmcode

```python
from sqlalchemy.ext.asyncio import create_async_engine, AsyncSession
from sqlalchemy.orm import sessionmaker
from app.config import Config
from fastapi import Depends
from . import models
from .base import Base

logger = Config.logger_init()
logger.info("START DB")
__all__=["Base", "models"]

async_engine = create_async_engine(Config.SQLALCHEMY_DATABASE_URL)

async_session = sessionmaker(
    async_engine, expire_on_commit=False, class_=AsyncSession
)

async def create_tables():
    async with async_engine.begin() as connection:
        await connection.run_sync(Base.metadata.create_all)
        await connection.commit()
        logger.debug("TABLES CREATED")

async def get_session():
    async with async_session() as session:
        yield session
```

Listing 4: db: __init__.py