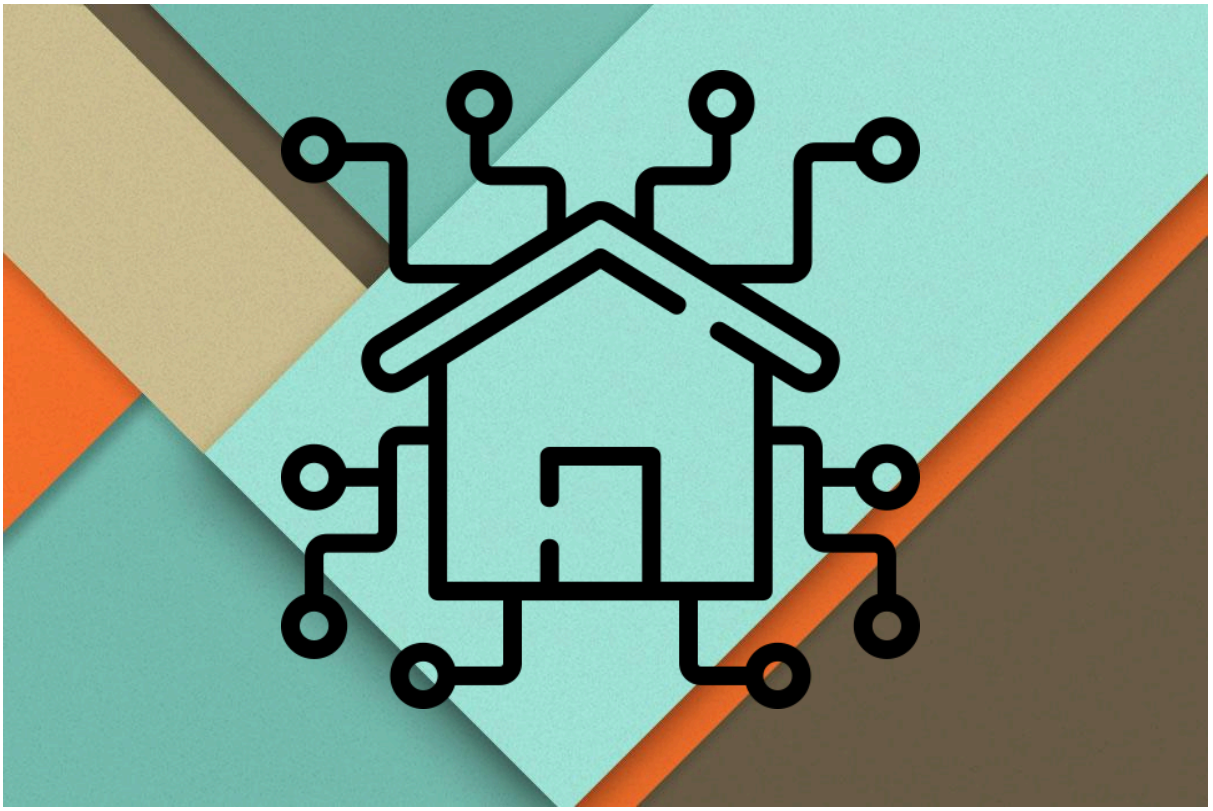


Smart Home Dashboard

30.10.2024

Ziel ist es, eine Plattform zu entwickeln, die es ermöglicht, Smart Home-Geräte wie Steckdosen, Lampen und andere Geräte zentral zu steuern und zu verwalten.

Gruppenmitglieder:

Tim Sommer - 6070379
Paula Bauer - 7918601
Chris David Kaufmann - 7940617

Abstrakt

Das Smart Home Dashboard ist eine Webanwendung, die wir im Rahmen unseres Webprogrammierungsprojekts entwickelt haben. Die Anwendung ermöglicht die Verwaltung und Steuerung von Smart-Home-Geräten, Räumen und Routinen und wurde so konzipiert, dass Benutzer ihre Geräte übersichtlich organisieren und steuern können.Die Anwendung bietet sowohl eine REST API als auch eine GraphQL API, um flexible und effiziente Schnittstellen für die Interaktion mit den Daten des Smart Home Systems zur Verfügung zu stellen.Das Frontend der Anwendung wurde als Single Page Application (SPA) mit Angular umgesetzt, was eine direkte und flüssige Interaktion innerhalb der Benutzeroberfläche ermöglicht. Diese Struktur reduziert die Ladezeiten, da sie ohne regelmäßige Seitenaktualisierungen auskommt, was die Bedienung vereinfacht und beschleunigt. Das Design der Benutzeroberfläche konzentriert sich darauf, die Verwaltung von Geräten, Räumen und Routinen übersichtlich zu gestalten und eine einfache Navigation zu gewährleisten.

Im Backend kommt das Spring-Boot-Framework zum Einsatz, das sowohl die REST- als auch die GraphQL-APIs bereitstellt und die Anwendungslogik unterstützt. Für die Datenhaltung und Verwaltung der Geräte- und Benutzerdaten wurde eine PostgreSQL Datenbank integriert. Die Architektur der Anwendung ist auf einfache Skalierbarkeit und Erweiterbarkeit ausgelegt. Die Verwendung einer relationalen Datenbank erleichtert die Verwaltung und Abfrage von Informationen und ermöglicht eine klare Strukturierung der Daten.

Diese Kombination schafft eine robuste Basis, die eine reibungslose Integration verschiedener Smart Home Geräte ermöglicht und eine langfristige Erweiterbarkeit der Plattform sicherstellt.

Abstract

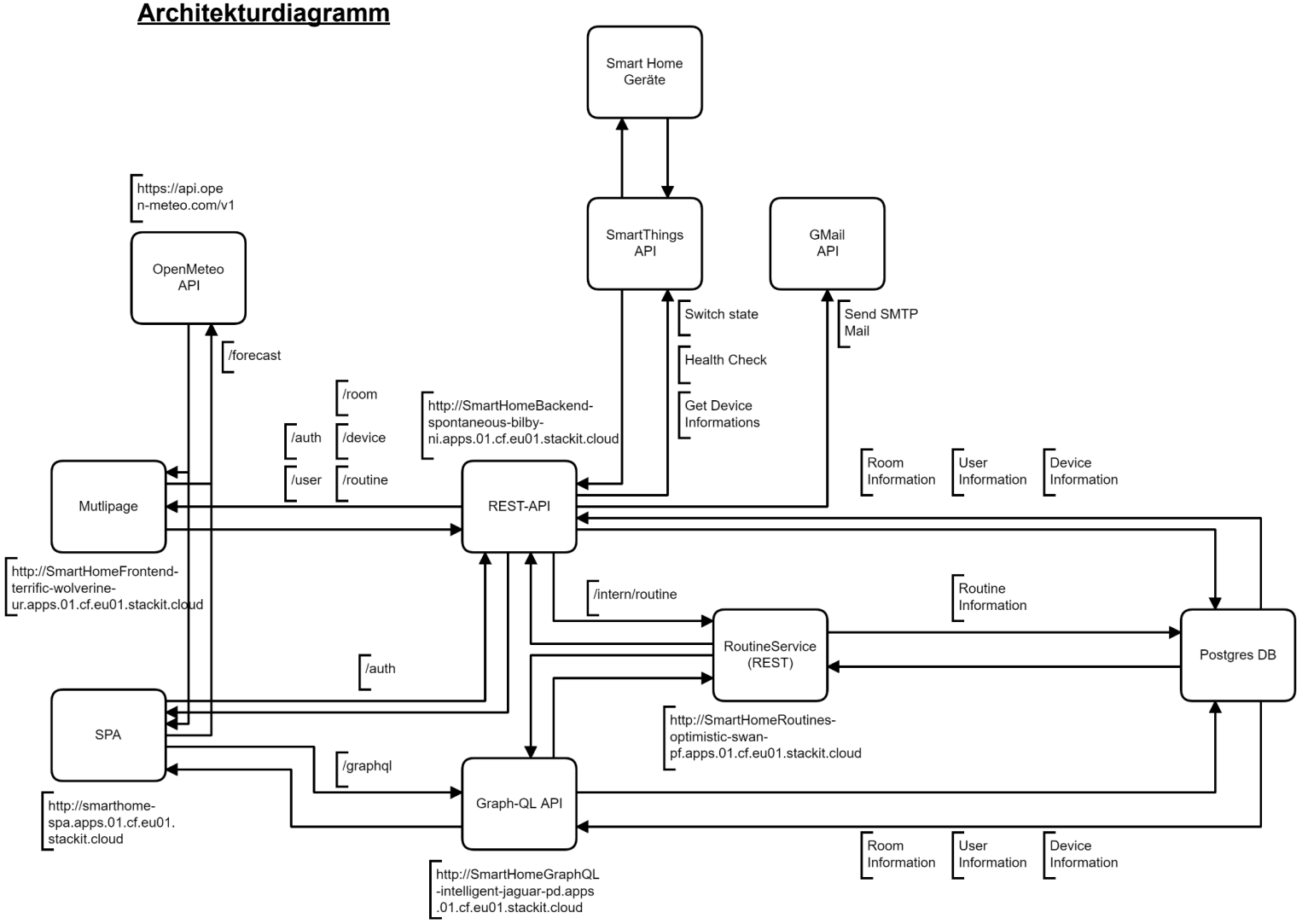
The Smart Home Dashboard is a web application that we developed as part of our web programming project. The application enables the management and control of smart home devices, rooms and routines and is designed to allow users to organize and control their devices clearly.The application offers both a REST API and a GraphQL API to provide flexible and efficient interfaces for interacting with the smart home system data.

The front end of the application was implemented as a Single Page Application (SPA) with Angular, which enables direct and fluid interaction within the user interface. This structure reduces loading times as it does not require regular page refreshes, which simplifies and speeds up operation. The design of the user interface focuses on making the management of devices, rooms and routines clear and ensuring easy navigation.

The Spring Boot framework is used in the backend, which provides both the REST and GraphQL APIs and supports the application logic. A PostgreSQL database was integrated for data storage, management and retrieval of information and enables clear structuring of the data.

This combination creates a robust basis that enables the smooth integration of various smart home devices and ensures the long-term expandability of the platform.

Architekturdiagramm



Der Webservice ist modular aufgebaut und ermöglicht eine effiziente Integration und Verwaltung von Smart Home-Geräten. Durch die klare Trennung der Komponenten wird Flexibilität und Skalierbarkeit gewährleistet, was das System ideal für moderne Wohnkonzepte macht

1. Frontend-Komponenten

Multipage (http://SmartHomeFrontend-terrific-wolverine-ur.apps.01.cf.eu01.stackit.cloud): Diese Komponente repräsentiert eine klassische mehrseitige Webanwendung. Sie interagiert direkt mit der REST-API und bezieht Informationen zu /auth, /room, /device, /user, und /routine. Auch Daten von der OpenMeteo API (Wetterdaten) werden hier verwendet. Das Multipage-Frontend bietet eine benutzerfreundliche Oberfläche, um mit den Smart Home-Geräten und den vom Backend bereitgestellten Daten zu interagieren.

SPA (Single Page Application - http://smarthome-spa.apps.01.cf.eu01.stackit.cloud): Diese Anwendung ist eine Single Page Application und kommuniziert hauptsächlich mit der GraphQL API über den /graphql-Endpunkt. Die SPA ladet einmalig den gesamten Content und aktualisiert sie dynamisch, ohne die gesamte Seite neu zu laden. Auch hier liegt der

Fokus auf der Kommunikation mit dem Backend und der Anzeige von Smart Home-Daten und -Funktionen.

2. Backend-Komponenten

REST-API (http://SmartHomeBackend-spontaneous-billy-ni.apps.01.cf.eu01.stackit.cloud): Die REST-API dient als zentrale Schnittstelle für die Kommunikation zwischen den Frontend-Komponenten und dem Backend. Sie bietet verschiedene Endpunkte, darunter:

/auth: Authentifizierung der Benutzer.

/room: Bereitstellung von Informationen zu Räumen im Smart Home.

/device: Verwaltung von Smart Home-Geräten.

/user: Bereitstellung von Benutzerdaten.

/routine: Verwaltung von Routinen im System.

Zusätzlich greift die REST-API auf externe Dienste zu, wie z. B. die SmartThings API (zur Steuerung von Smart Home-Geräten) und die GMail API (zum Versenden von E-Mails).

GraphQL API (http://SmartHomeGraphQL-intelligent-jaguar-pd.apps.01.cf.eu01.stackit.cloud): Die GraphQL API ist eine alternative Schnittstelle, die vor allem von der SPA verwendet wird. Im Gegensatz zu REST erlaubt GraphQL flexible Abfragen, sodass Clients nur die benötigten Daten abrufen können. Diese API liefert Informationen zu Räumen, Nutzern, Geräten und Routinen.

RoutineService (REST - http://SmartHomeRoutines-optimistic-swan-pf.apps.01.cf.eu01.stackit.cloud): Der RoutineService ist eine eigenständige Komponente, die speziell für die Verwaltung und Verarbeitung von Routinen entwickelt wurde. Er kommuniziert mit der REST-API über den /intern/routine-Endpunkt und greift auf die Postgres-Datenbank zu, um Routine-Informationen zu speichern und zu verarbeiten.

3. Externe APIs

OpenMeteo API (https://api.open-meteo.com/v1): Diese API stellt Wetterdaten zur Verfügung. Das Multipage-Frontend greift über den /forecast-Endpunkt auf diese API zu, vermutlich um wetterabhängige Routinen zu steuern oder Wetterinformationen für den Benutzer anzuzeigen.

SmartThings API: Diese API dient zur Steuerung und Überwachung der Smart Home-Geräte. Die REST-API verwendet Endpunkte wie Switch state (Gerätestatus ändern), Health Check (Überprüfung des Zustands der Geräte) und Get Device Informations (Abrufen von Geräteinformationen). Sie ist eine zentrale Schnittstelle für die Integration der Smart Home-Geräte ins System. GMail API: Diese API wird verwendet, um über die REST-API E-Mails zu versenden, z. B. zur Benachrichtigung von Nutzern über bestimmte Ereignisse im Smart Home. Hier wird ein Send SMTP Mail-Endpunkt genutzt.

4. Datenbank

Postgres DB: Die Datenbank speichert alle wichtigen Informationen wie Room Information, User Information, Device Information und Routine Information. Diese Daten werden vom RoutineService und der REST-API abgerufen und aktualisiert, um eine zentrale Datenspeicherung für das System zu gewährleisten.

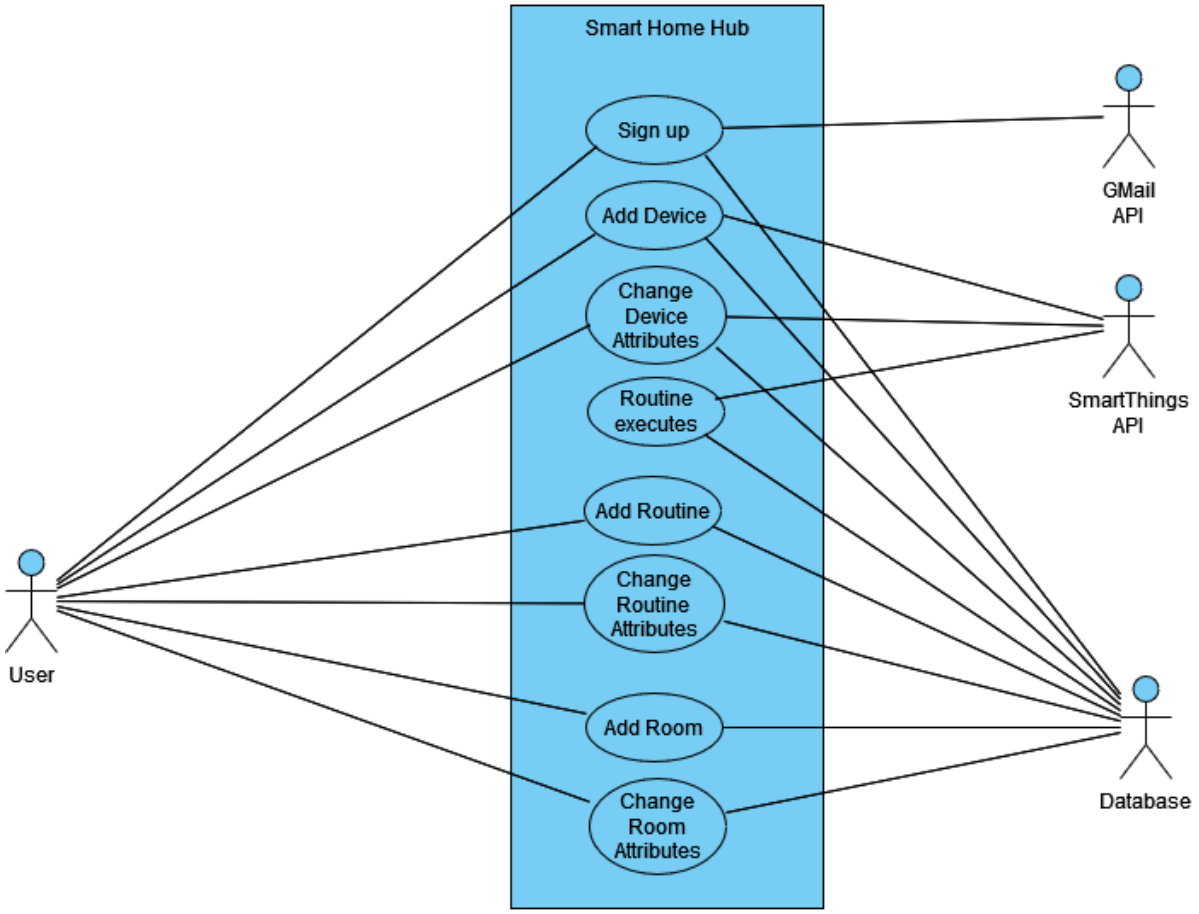
5. Interaktionen und Datenflüsse

Das Diagramm zeigt deutlich die Verbindungen zwischen den verschiedenen Frontend-Anwendungen, APIs und externen Diensten. Die REST-API fungiert als Hauptdrehscheibe für die Kommunikation zwischen den Frontend- und Backend-Komponenten.

Die SPA greift auf die GraphQL API zu, während die Multipage-Anwendung sowohl mit der REST-API als auch mit der OpenMeteo API interagiert.

Die SmartThings API und die GMail API werden durch die REST-API angesteuert, um Smart Home-Geräte zu steuern und Benachrichtigungen zu versenden.

Der RoutineService ist eine separate Komponente, die mit der Postgres-Datenbank verbunden ist, um Routine-Informationen zu verarbeiten und zu speichern.



Erfüllte Anforderungen:

Benutzerfreundlichkeit:

Die Frontend-Anwendungen (Multipage und SPA) bieten intuitive Benutzeroberflächen, die eine einfache Interaktion mit den Smart Home-Geräten ermöglichen.

Modularität:

Die Architektur ist modular aufgebaut, was eine einfache Erweiterung und Anpassung des Systems ermöglicht.

Datenintegration:

Die Integration externer APIs (SmartThings, OpenMeteo, GMail) funktioniert reibungslos, um zusätzliche Funktionen wie Wetterdaten bereitzustellen. */ nur post & Get*

Flexibilität der Datenabfragen:

Die Verwendung von GraphQL ermöglicht flexible und gezielte Datenabfragen, was die Effizienz der Datenverarbeitung erhöht.

Zentrale Datenverwaltung:

Die Postgres-Datenbank speichert alle relevanten Informationen zentral, was die Datenverwaltung vereinfacht.

Automatisierung von Routinen:

Nutzer können benutzerdefinierte Routinen erstellen, die auf bestimmten Bedingungen basieren, wie z. B. Zeitplänen.

Skalierbarkeit:

Die Architektur unterstützt die einfache Hinzufügung neuer Geräte und Funktionen, was das System zukunftssicher macht.

Erhöhte Sicherheit:

Bei Registrierung gibt es einen Verifizierungsprozess mit Verifizierungs Token-Mail an Nutzer, welcher Link aufrufen muss.

Nicht erfüllte Anforderungen:

Eventgesteuerte Routinen:

Aufgrund von Zeitmangel war es nicht möglich, weitere Routinen wie die ereignisgesteuerte Automatisierung einzubinden. Daher sind nur zeitgesteuerte Routinen implementiert.

Keine Echtzeit-Datenübertragung:

Die SmartThings API hat eine Beschränkung der Anfragen pro Minute, weshalb keine Echtzeit-Datenübertragung implementiert werden konnte. Unsere Lösung ist eine Update-Funktion, die die Daten alle 30 Sekunden aktualisiert, um so einen möglichst aktuellen Stand abbilden zu können.

REST API

Device Endpoint

Methode	Endpunkt	Description	Data in	Data out

POST	/device	Gerät erstellen	Header:
			{”Authorization”:”1234”
			}
			{
			”device”:{
			"device_id":"35df5c2a"
			,
			"name": "Wohnzimmer Steckdose",
			"typeID":1,
			"location": 3
			}
			}
				200
				{
				”message”:”Device created”
				}
				400
				{
				”reason”:”Wrong token”
				}

DELETE	/device/{device_id}	Gerät löschen	Header:
			{”Authorization”:”1234”
			}
				200
				{
				”message”:”Deletion complete”
				}
				400
				{
				”reason”:”Wrong token”
				}

PUT	/device/{device_id}	Gerät ändern	Header:
			{”Authorization”:”1234”
			}
			{
			”change”:{
			”new-value”:”Flur”
			“field”:”location”
			}
			}
				200
				{
				”message”:”Change complete”
				}
				400
				{
				”reason”:”Passwort to short”
				}

GET	/device	Alle Geräte abrufen	Header:
			{”Authorization”:”1234”
			}
				200
				{
				"devices":[{
				"device_id":"1AS241",
				"name":"Computer",
				"typeID": 2,
				"type":"Outlet",
				"typeIcon":"outlet",
				"location": 2,
				"status":"online",
				"state": "off"
				},
				{ ...}
				]
				}
				400
				{
				”reason”:”Wrong token”

				}

GET	/device/{device_id}	Einzelnes Gerät abrufen	Header:
			{”Authorization”:”1234”
			}
				200
				{
				"device_id":"1AS241",
				"name":"Computer",
				"typeID": 2,
				"type":"Outlet",
				"typeIcon":"outlet",
				"location": 2,
				"status":"online",
				"state": "off"
				}
				400
				{
				”reason”:”Wrong token”
				}

POST	/device/{device_id}/switch/{on/off}	Gerät schalten	Header:
			{”Authorization”:”1234”
			}
				200
				{
				”message”:”Accepted”
				}
				400
				{
				”reason”:”Connection Error”
				}

GET	/device/{device_id}/health-check	Gerät Status abfragen	Header:
			{”Authorization”:”1234”
			}
				200
				{
				”message”:”Offline”
				}
				400
				{
				”reason”:”Connection Error”
				}

GET	/device/smartthings	Gibt alle SmartThings Geräte zurück, welche noch nicht assigned wurden	Header:
			{”Authorization”:”1234”
			}
				200
				{
				"devices":[
				{
				"device_id":"35df5c2a-ec4a-4283-9c9a-af7ec3d62464",
				"name":"Wohnzimmer Steckdose"
				},
				{ … }
				]
				}
				400
				{
				”reason”:”Connection Error”
				}

Auth Endpoint

Methode	Endpunkt	Description	Data in	Data out

POST	/auth	anmelden	{
			"email":"max@musterma
			nn.com",
			"password":"pw"
			}
				200
				{
				”token”:”123456”
				}
				400
				{
				”reason”:”Wrong
				credentials”
				}

POST	/auth/sign-up	registrieren	{
			"firstName":"Max",
			"lastName":"Mustermann
			",
			"email":"max@musterma
			nn.com",
			"passwort":"pw",
			"pat": "14165asd"
			}
				200
				{
				”message”:”Sign out
				complete”
				}
				
				400
				{
				”reason”:”Mail already
				exist”
				}

POST	/auth/validate
	-token
		Validierung
		und
		Aktualisierung
		des Token
			Header:
			{”Authorization”:”1234”}
				200
				{
				”token”:”1546323”
				}
				
				400
				{
				”reason”:”Wrong token”
				}

POST	/auth/validate
	-email/{token
	}
		Validierung
		der Email
		nach SignUp
				200
				
				400
				{
				”reason”:”Wrong token”
				}

GET	/auth	alive		200 “I am Alive”

User Endpoint:

Methode	Endpunkt	Description	Data in	Data out

GET	/user	Eigenen Account abrufen	Header:
			{”Authorization”:”1234”}
				200
				{
				"firstName":"Max",
				"lastName":"Mustermann",
				"email":"max@mustermann.com",
				"passwort": "",
				"pat": "14165asd"
				}
				400
				{
				”reason”:”Wrong token”
				}

DELETE	/user	Account löschen	Header:
			{”Authorization”:”1234”}
				200
				{
				”message”:”Successfully deleted”
				}
				400
				{
				”reason”:”Wrong token”
				}

PUT	/user	Account ändern	Header:
			{”Authorization”:”1234”}
			{
			”change”:{
			”new-value”:”Schulz”
			“field”:”lastname”
			}
			}
				200
				{
				”message”:”Field changed”
				}
				400
				{
				”reason”:”Wrong token”
				}

Room Endpoint:

Methode	**Endpunkt**	**Description**	**Data in**	**Data out**

GET	/room	Alle Räume abrufen	Header:
			{”Authorization”:”1234”}
				200
				{
				"rooms":[
				{
				"room_id": "room01",
				"name":"Wohnzimmer"
				},
				{ … }
				]
				}
				400
				{
				”reason”:”Wrong token”
				}

GET	/room/{room_id}	Raum abrufen	Header:
			{”Authorization”:”1234”}
				200
				{
				"room_id": "room01",
				"name":"Wohnzimmer"
				}
				400
				{ ”reason”:”Wrong token”}

POST	/room	Raum anlegen	{
			"name":"Wohnzimmer",
			}
				200
				{
				”message”:”Room created”
				}
				400
				{
				”reason”:”Name already exists”
				}

DELETE	/room/{room_id}	Raum löschen	Header:
			{”Authorization”:”1234”}
				200
				{
				”message”:”Room deleted”
				}
				400
				{
				”reason”:”Wrong token”
				}

PUT	/room/{room_id}	Raum ändern	Header:
			{”Authorization”:”1234”}
			{”change”:{
			”new-value”:”Schlafzimmer” ,
			“field”:”name”
			}}
				200
				{
				”message”:”Field changed”
				}
				400
				{
				”reason”:”Wrong token”
				}

Routine Endpoint:

Methode	**Endpunkt**	**Description**	**Data in**	**Data out**

POST	/routine	Routine erstellen	Header:
			{”Authorization”:”1234”}
			{
			"routine":{
			"name":"Guten Morgen",
			"actions":[
			{"device_id":"light01",
			"action":"on"},
			{"device_id":"outlet1",
			"action":"off",}
			],
			"trigger":{
			"type":"time",
			"value":"07:00"
			},
			"state":"on
			}
			}
				200
				{
				”message”:"Routine created"
				}
				
				400
				{
				"reason":"Invalid device_id"
				}

GET	/routine/{routine_id}	Einzelne Routine abrufen	Header:
			{”Authorization”:”1234”}
				200
				{
				"id":"routine01",
				"name":"Guten Morgen",
				"actions":[...(s.o.)],
				"trigger":{...(s.o.)},
				"state":"on
				}
				400
				{
				"reason":"Routine not found"
				}

PUT	/routine/{routine_id}	Routine ändern	Header:
			{”Authorization”:”1234”}
			{
			"id":"routine01",
			"name":"Guten Abend",
			"actions":[...(s.o.)],
			"trigger":{...(s.o.)},
			"state":"on
			}
				200
				{
				”message”:"Routine updated"
				}
				400
				{
				"reason":"Invalid action"
				}

DELETE	/routine/{routine_id}	Routine löschen	Header:
			{”Authorization”:”1234”}
				200
				{
				”message”:"Routine deleted"
				}
				400
				{
				"reason":"Unauthorized"
				}

GET	/routine	Alle Routinen abrufen	Header: {”Authorization”:”1234”}	200
				[
				{
				"id":"routine01",
				"name":"Guten Morgen",
				"actions":[
				{
				"id": 34,
				"device_id":"light01",
				"action":"on"
				},
				{…}
				],
				"trigger":{…},
				"state":"on
				},
				{...}
				]
				400
				{
				"reason":"Invalid token"
				}

GraphQL API

Diese Dokumentation beschreibt die verschiedenen Queries und Mutations der GraphQL API sowie die dazugehörigen Datentypen.

Queries

1. deviceById(id: ID!): Device
    - Beschreibung: Ruft ein Gerät anhand seiner ID ab.
    - Rückgabe: Ein Device-Objekt.
2. allDevices: [Device]!
    - Beschreibung: Ruft alle Geräte ab.
    - Rückgabe: Eine Liste von Device-Objekten.
3. deviceTypeById(id: ID!): DeviceType
    - Beschreibung: Ruft einen Gerätetyp anhand seiner ID ab.
    - Rückgabe: Ein DeviceType-Objekt.
4. allDeviceTypes: [DeviceType]!
    - Beschreibung: Ruft alle Gerätetypen ab.
    - Rückgabe: Eine Liste von DeviceType-Objekten.
5. roomById(id: ID!): Room
    - Beschreibung: Ruft einen Raum anhand seiner ID ab.
    - Rückgabe: Ein Room-Objekt.

6. allRooms: [Room]!
    - Beschreibung: Ruft alle Räume ab.
    - Rückgabe: Eine Liste von Room-Objekten.
7. routineById(id: ID!): Routine
    - Beschreibung: Ruft eine Routine anhand ihrer ID ab.
    - Rückgabe: Ein Routine-Objekt.
8. allRoutines: [Routine]!
    - Beschreibung: Ruft alle Routinen ab.
    - Rückgabe: Eine Liste von Routine-Objekten.
9. userInfo: User
    - Beschreibung: Ruft Informationen über den angemeldeten Benutzer ab.
    - Rückgabe: Ein User-Objekt.

Mutations

- createDevice(id: ID!, name: String!, typeId: ID!, roomId: ID!): Boolean
    - Beschreibung: Erstellt ein neues Gerät.
    - Argumente:
        - id: Die ID des Geräts.
        - name: Der Name des Geräts.
        - typeId: Die ID des Gerätetyps.
        - roomId: Die ID des Raums.
    - Rückgabe: true, wenn das Gerät erfolgreich erstellt wurde, sonst false.
- updateDevice(id: ID!, name: String, typeId: ID, roomId: ID): Device
    - Beschreibung: Aktualisiert ein bestehendes Gerät.
    - Argumente:
        - id: Die ID des Geräts.
        - name (optional): Der neue Name des Geräts.
        - typeId (optional): Die neue ID des Gerätetyps.
        - roomId (optional): Die neue ID des Raums.
    - Rückgabe: Das aktualisierte Device-Objekt.
- switchDevice(id: ID!, status: String!): Device
    - Beschreibung: Schaltet den Status eines Geräts um.
    - Argumente:
        - id: Die ID des Geräts.
        - status: Der neue Status des Geräts.
    - Rückgabe: Das Device-Objekt mit dem aktualisierten Status.
- deleteDevice(id: ID!): Boolean
    - Beschreibung: Löscht ein Gerät.
    - Argumente:
        - id: Die ID des Geräts.
    - Rückgabe: true, wenn das Gerät erfolgreich gelöscht wurde, sonst false.
- createRoutine(name: String!, actions: [ActionInput]!, triggerTime: String!): Boolean
    - Beschreibung: Erstellt eine neue Routine.
    - Argumente:
        - name: Der Name der Routine.
        - actions: Eine Liste von Aktionen.
        - triggerTime: Die Auslösezeit der Routine.

    - Rückgabe: true, wenn die Routine erfolgreich erstellt wurde, sonst false.
- updateRoutine(id: ID!, name: String, actions: [ActionInput], triggerTime: String, state: Boolean): Routine
    - Beschreibung: Aktualisiert eine bestehende Routine.
    - Argumente:
        - id: Die ID der Routine.
        - name (optional): Der neue Name der Routine.
        - actions (optional): Eine neue Liste von Aktionen.
        - triggerTime (optional): Die neue Auslösezeit der Routine.
        - state (optional): Der neue Status der Routine.
    - Rückgabe: Das aktualisierte Routine-Objekt.
- switchRoutine(id: ID!, state: Boolean): Boolean
    - Beschreibung: Schaltet den Status einer Routine um.
    - Argumente:
        - id: Die ID der Routine.
        - state: Der neue Status der Routine.
    - Rückgabe: true, wenn der Status erfolgreich umgeschaltet wurde, sonst false.
- deleteRoutine(id: ID!): Boolean
    - Beschreibung: Löscht eine Routine.
    - Argumente:
        - id: Die ID der Routine.
    - Rückgabe: true, wenn die Routine erfolgreich gelöscht wurde, sonst false.
- createRoom(name: String!): Boolean
    - Beschreibung: Erstellt einen neuen Raum.
    - Argumente:
    - name: Der Name des Raums.
    - Rückgabe: true, wenn der Raum erfolgreich erstellt wurde, sonst false.
- updateRoom(id: ID!, name: String!): Room
    - Beschreibung: Aktualisiert einen bestehenden Raum.
    - Argumente:
        - id: Die ID des Raums.
        - name: Der neue Name des Raums.
    - Rückgabe: Das aktualisierte Room-Objekt.
- deleteRoom(id: ID!): Boolean
    - Beschreibung: Löscht einen Raum.
    - Argumente:
        - id: Die ID des Raums.
    - Rückgabe: true, wenn der Raum erfolgreich gelöscht wurde, sonst false.
- updateUser(firstName: String, lastName: String, email: String, pat: String, isVerified: Boolean): User
    - Beschreibung: Aktualisiert die Informationen eines Benutzers.
    - Argumente:
        - firstName (optional): Der neue Vorname.
        - lastName (optional): Der neue Nachname.
        - email (optional): Die neue E-Mail-Adresse.
        - pat (optional): Der neue PAT-Wert.
        - isVerified (optional): Der neue Verifizierungsstatus.
    - Rückgabe: Das aktualisierte User-Objekt.

Datentypen

- Device
    - Beschreibung: Repräsentiert ein Gerät.
    - Felder:
        - id: Die ID des Geräts.
        - name: Der Name des Geräts.
        - type: Der Typ des Geräts.
        - room: Der Raum, in dem sich das Gerät befindet.
        - status: Der Status des Geräts.
        - state: Der Zustand des Geräts.
- DeviceType
    - Beschreibung: Repräsentiert einen Gerätetyp.
    - Felder:
        - id: Die ID des Gerätetyps.
        - name: Der Name des Gerätetyps.
        - icon: Das Icon des Gerätetyps.
- Routine
    - Beschreibung: Repräsentiert eine Routine.
    - Felder:
    - id: Die ID der Routine.
    - name: Der Name der Routine.
    - actions: Eine Liste von Aktionen, die die Routine ausführt.
    - triggerTime: Die Auslösezeit der Routine.
    - state: Der Zustand der Routine.
- Room
    - Beschreibung: Repräsentiert einen Raum.
    - Felder:
        - id: Die ID des Raums.
        - name: Der Name des Raums.
- Action
    - Beschreibung: Repräsentiert eine Aktion innerhalb einer Routine.
    - Felder:
        - id: Die ID der Aktion.
        - deviceID: Die ID des Geräts, auf das die Aktion abzielt.
        - deviceName: Der Name des Geräts.
        - setTo: Der Zustand, auf den das Gerät gesetzt werden soll.
- User
    - Beschreibung: Repräsentiert den angemeldeten Benutzer
    - Felder:
        - id: Die ID des Users
        - firstName: Der Vorname des Benutzers
        - lastName: Der Nachname des Benutzers
        - email: Die Email des Benutzers
        - pat: Der Personal Access Token des Benutzers für den Zugriff auf SmartThings
        - isVerified: Der Status der Email Verifizierung des Users

Backend: REST

Allgemeine Erklärung

Das Design des REST-Backends folgt dem Prinzip der Trennung von Verantwortlichkeiten und der Verwendung von standardisierten HTTP-Methoden zur Interaktion mit dem Frontend. Das Backend ist in verschiedene Controller-Klassen unterteilt, die jeweils für spezifische Domänenbereiche wie Geräte, Räume und Routinen zuständig sind. Diese Controller-Klassen verwenden Spring Boot und sind mit entsprechenden Annotations versehen, um die REST-Endpunkte zu definieren und zu konfigurieren.

Allgemeine Rückgabetypen

Die Rückgabetypen an das Frontend sind in der Regel ResponseEntity-Objekte, die den HTTP-Statuscode und den eigentlichen Rückgabewert enthalten. Diese Rückgabewerte können einfache Nachrichten (MessageAnswer, MessageReason), Datenobjekte (DeviceGetResponse, RoomDTO) oder Listen von Objekten (AllDevices, AllRooms) sein. Durch die Verwendung von ResponseEntity kann das Backend präzise HTTP-Statuscodes wie 200 OK, 400 Bad Request oder 404 Not Found zurückgeben, was eine klare Kommunikation des Ergebnisses der Anfrage ermöglicht.

Verwendung von Spring und Annotations

Spring Boot wird verwendet, um die Anwendung zu konfigurieren und zu starten. Die wichtigsten Annotations sind:

@RestController: Markiert eine Klasse als REST-Controller, der HTTP-Anfragen entgegennimmt und Antworten zurückgibt.
@RequestMapping: Definiert die Basis-URL für alle Methoden in einer Controller-Klasse.
@GetMapping, @PostMapping, @PutMapping, @DeleteMapping: Spezifizieren die HTTP-Methode und den URL-Pfad für eine bestimmte Methode.
@RequestBody: Bindet den Inhalt des HTTP-Requests an ein Java-Objekt.
@RequestHeader: Bindet den Wert eines HTTP-Headers an eine Methode.
@PathVariable: Bindet einen Teil des URL-Pfads an eine Methode.

Diese Annotationen ermöglichen eine klare und deklarative Definition der REST-Endpunkte und deren Verhalten. Durch die Verwendung von Spring Boot und seinen Annotations wird die Entwicklung und Wartung des Backends erheblich vereinfacht, da viele Boilerplate-Konfigurationen automatisch gehandhabt werden.

Singleton, @Autowired und @Service

Die Annotation @Autowired wird im Spring verwendet, um Abhängigkeiten automatisch zu injizieren. Dies bedeutet, dass Spring zur Laufzeit automatisch die benötigten Bean-Instanzen bereitstellt und sie in die entsprechenden Felder, Konstruktoren oder Setter-Methoden einfügt. Dies erleichtert die Verwaltung von Abhängigkeiten und fördert die lose Kopplung zwischen den Komponenten.

@Service

Die Annotation @Service wird verwendet, um eine Klasse als Service-Komponente zu kennzeichnen. Eine Service-Komponente enthält die Geschäftslogik der Anwendung und wird in der Regel von Controller-Klassen aufgerufen. Durch die Verwendung von @Service wird die Klasse als Spring-Bean registriert, sodass sie von Spring verwaltet und bei Bedarf automatisch injiziert werden kann.

Singleton-Prinzip

Das Singleton-Prinzip stellt sicher, dass eine Klasse nur eine einzige Instanz hat und bietet einen globalen Zugriffspunkt auf diese Instanz. In Spring wird das Singleton-Design Muster standardmäßig für Beans verwendet. Somit kann mithilfe der oben genannten Annotations sichergestellt werden, dass das Singleton Prinzip eingehalten wird. Dies spart Ressourcen und stellt sicher, dass der Zustand der Klassen konsistent bleibt.

Logischer Aufbau der Geräte Komponente

Device-Klasse

Die Device-Klasse repräsentiert ein einzelnes Gerät im Smart-Home-System. Sie enthält Attribute wie id, name, type, room, status und state. Diese Attribute sind notwendig, um die Identität, den Typ, den Standort und den aktuellen Zustand des Geräts zu beschreiben. Die Device-Klasse ist eine einfache POJO (Plain Old Java Object), die als Datenmodell dient. Sie enthält Getter- und Setter-Methoden für jedes Attribut, um den Zugriff und die Manipulation der Daten zu ermöglichen.

DeviceService-Schnittstelle und Implementierung

Die DeviceService-Schnittstelle definiert die grundlegenden Operationen, die auf Geräten ausgeführt werden können. Dazu gehören Methoden wie getDeviceById, getAllDevices, createDevice, updateDevice und deleteDevice.

DeviceServicePostgres Klasse

Die Klasse DeviceServicePostgres implementiert die DeviceService Schnittstelle und verwendet eine PostgreSQL-Datenbank zur Speicherung und Verwaltung der Geräteinformationen.

DeviceController

Der DeviceController ist eine Spring-Controller-Klasse, die als Schnittstelle zwischen dem Client und dem Backend dient. Er definiert die Endpunkte für die verschiedenen Operationen, die auf Geräten ausgeführt werden können. Der DeviceController verwendet die DeviceService-Schnittstelle, um die Geschäftslogik auszuführen. Jede Methode im Controller ist mit einer HTTP-Methode (GET, POST, PUT, DELETE) und einem entsprechenden Endpunkt (/devices, /devices/{id}) annotiert. Die Methoden des Controllers

sind dafür verantwortlich, die Anfragen des Clients zu verarbeiten, die entsprechenden Service-Methoden aufzurufen und die Antworten an den Client zurückzugeben.

Anbindung an SmartThingsAPI

Die Integration der SmartThingsAPI ermöglicht es, Geräte im Smart-Home-System zu steuern und deren Status zu überwachen. Die SmartThingsAPI bietet eine Schnittstelle, um mit den Geräten zu kommunizieren, die über die SmartThings-Plattform verbunden sind.

SmartThings-Schnittstelle und Implementierung

Die SmartThings-Schnittstelle definiert die grundlegenden Operationen, die über die SmartThingsAPI ausgeführt werden können. Dazu gehören Methoden wie getAllDevices, getDevice, setDeviceStatus und getDeviceFullStatus.

Die Klasse SmartThingsImpl implementiert die SmartThings-Schnittstelle und verwendet HTTP-Clients, um Anfragen an die SmartThingsAPI zu senden und Antworten zu verarbeiten.
Beispielhaft hierzu ist die implementierung der setDeviceStatus Methode:

```
@Override
    public boolean setDeviceStatus(String status, String deviceID,
String capabilityId, String accessToken) {
        final HttpPost httpPost = new HttpPost(API_URL + "/" + deviceID
+ "/commands");
        final String json = "{\"commands\":[{\"capability\":\"" +
capabilityId + "\",\"command\":\"" + status + "\"}]}";
        StringEntity entity;
        try {
            entity = new StringEntity(json);
        } catch (UnsupportedEncodingException e) {
            e.printStackTrace();
            return false;}
        httpPost.setEntity(entity);
        httpPost.setHeader("Authorization", "Bearer " + accessToken);
        httpPost.setHeader("Content-type", "application/json");
        try (CloseableHttpClient client = HttpClients.createDefault()) {
            String response = client.execute(httpPost, new
MyResponseHandler());
            SetStatusResponse deviceResponse =
objectMapper.readValue(response, SetStatusResponse.class);
            return
deviceResponse.getResults().get(0).getStatus().equals("ACCEPTED");
        } catch (Exception e) {
            e.printStackTrace();
            return false;}}
```

DeviceController und SmartThings-Integration

Der DeviceController verwendet die SmartThings-Klasse, um die Geräte über die SmartThingsAPI zu steuern und deren Status zu überwachen. Diese sind hier aus sicht der API auf dem gleichen Endpunkt wie auch schon die oben beschriebenen DeviceController Endpunkte

Logischer Aufbau der Room-Komponente

Die Room-Klasse repräsentiert einen Raum im System. Sie enthält verschiedene Attribute, die die Eigenschaften eines Raums beschreiben, wie Name und ID, sowie Methoden zum Zugriff und zur Manipulation dieser Attribute. Für den Zugriff steht hier ein zu dem oben genannten Controller ähnlicher RoomController bereit, welcher mithilfe der RoomService-Klasse die CRUD Operationen abbildet. Hierzu kommt auch noch die besondere Funktion des Schaltens von allen Devices in einem Raum, welcher mittels des DeviceService realisiert wird.
Damit ein Device immer einem Raum zugeordnet wird, wird ein Standard-Raum erzeugt, welchem bei Löschen eines Raumes alle Devices aus dem gelöschten Raum zugeordnet werden.

Logischer Aufbau der User Komponente

Die User Klasse repräsentiert einen Benutzer im System. Sie enthält verschiedene Attribute, die die Eigenschaften eines Benutzers beschreiben, sowie Methoden zum Zugriff und zur Manipulation dieser Attribute. Hierbei werden Passwörter mit einem von SpringSecurity zur Verfügung gestellten Hash Algorithmus gehashed und bei Überprüfung werden diese Hashes miteinander auf gleichheit verglichen. So wird eine Benutzung eines Klarpasswortes verhindert.
Mittels des, wie oben beschriebenen, dazugehörigen Controller und Service Klasse werden die CRUD operationen und die Datenbank Speicherung dargestellt

Logischer Aufbau der Authentifizierung

Der AuthService im Smart Home Dashboard verwendet JSON Web Tokens (JWT) zur Authentifizierung und Autorisierung von Benutzern. JWTs sind kompakte, URL-sichere Tokens, die Informationen über den Benutzer enthalten und digital signiert sind, um die Integrität und Authentizität der Daten zu gewährleisten. Somit kann jeder Controller den ihm übergebenen Header richtig entschlüsseln und die User-Informationen aus diesem auslesen. Dies erleichtert im Gegensatz zu einem eigens geschriebenen Token die Sicherheit und die Einfachheit der Datenauslesung.

RoutineService Schnittstelle

Der RoutineService des Smart Home Dashboards ist eine Schnittstelle, die verschiedene Operationen zur Verwaltung von Routinen definiert. Dieser kommuniziert mit der separaten Komponenten RoutineService, welche aufgrund von Konsistenz ausgelagert wurden. Diese Klasse erstellt für diese Kommunikation http-Request. Die Antwort wird hier dann entweder direkt an das Frontend weitergegeben oder es wird nach HTTP-Status 200 geprüft und darauf folgend ein Boolean zurückgegeben

RoutineService Komponente

Aufgrund der Schwierigkeit mit der Konsistenz der Daten und der Ausführung der Routinen muss das Management der Routinen ausgelagert werden. Hierzu ist die RoutineService Komponente implementiert worden, damit sowohl das REST-Backend als auch das GraphQL-Backend mit Routinen arbeiten können.
Diese Komponente stellt hierzu REST Endpunkte zur Verfügung, mit denen intern kommuniziert werden kann.
Hierzu gibt es Endpunkte zum Hinzufügen, Löschen, Ändern und An/Ausschalten.

Zur Speicherung wird über die RoutineServicePostgres Implementierung umgesetzt. Diese Klasse verwendet eine PostgreSQL-Datenbank zur Speicherung und Verwaltung der Routineninformationen.

RoutineScheduler

Die RoutineScheduler Schnittstelle definiert die grundlegenden Methoden zur Aktivierung und Deaktivierung von Routinen. Mit einer Methode in RoutineServicePostgres lädt die Komponente zum Start initial aus der Datenbank und schaltet diese, je nach gespeicherten Status, an oder aus.

```
private static void initializeRoutine() {
        for (Routine routine : scheduledRoutines) {

          RoutineScheduler routineScheduler = new
RoutineSchedulerImpl(routine.getActions());
            routine.setRoutineScheduler(routineScheduler);
            if (routine.isState()) {
                routine.activateRoutine();}
```

Wenn eine Routine aktiviert wird, dann wird auf dem RoutinenScheduler ein ScheduledFuture erstellt, welche dann die nötigen Aktionen der Routine ausführt. Hierfür wird eine initiale Verzögerung berechnet, wann die Aktion zum ersten Mal ausgeführt wird und eine weitere Verzögerung, welche genau einen Tag entspricht zur täglichen Wiederholung. Dieser Code ist auszugsweise folgend dargestellt:

```
@Override
public void activateDailyRoutine(LocalTime routineTime) {
    LocalDateTime now = LocalDateTime.now();
    LocalDateTime nextRun =
now.withHour(routineTime.getHour()).withMinute(routineTime.getMinute()).
withSecond(0);
    if (now.compareTo(nextRun) > 0) {
        nextRun = nextRun.plusDays(1);
    }
    Duration duration = Duration.between(now, nextRun);
    long initialDelay = duration.getSeconds();

    scheduledTask = scheduledExecutorService.scheduleAtFixedRate(() -> {
        for (Action action : actions) {
            smartThings.executeAction(action);
        }
    }, initialDelay, TimeUnit.DAYS.toSeconds(1), TimeUnit.SECONDS);
}
```

```
@Override
public void deactivateDailyRoutine() {
    if (scheduledTask != null) {
        scheduledTask.cancel(false);
    }}
```

Zur allgemeinen Verwaltung von User, Authentifizierung und SmartThings werden die gleichen Klassen verwendet wie im REST-Backend. Somit ist die Konsistenz der Interfaces Komponenten übergreifend gegeben, womit die Wartung erleichtert wird.

FRONTEND: Multipage

Das Frontend des Smart Home Dashboards ist eine Webanwendung, die verschiedene Funktionen zur Verwaltung und Steuerung von Smart-Home-Geräten, Benutzern, Räumen und Routinen bietet. Die Anwendung verwendet HTML, CSS und JavaScript und integriert verschiedene Bibliotheken und Frameworks wie jQuery und Bootstrap.

Die Multipage-Anwendung ist so aufgebaut, dass zuerst die Login-Seite mit den Funktionen für Login, Registrierung und Authentifizierung erscheint.
Auf jeder Seite wird "java.first" implementiert, das prüft, ob die Authentifizierung korrekt abgelaufen ist. Hier wird dann ein Token abgerufen, welcher im Local Storage gespeichert wird, damit keine automatische Abmeldung erfolgt.

HTML-Dateien:

homepage.html: Die Hauptseite der Anwendung, die das Dashboard und die Navigationselemente enthält.
user-info.html: Eine Unterseite zur Anzeige und Bearbeitung von Benutzerinformationen.

CSS-Dateien:

style.css: Enthält die Stile für das gesamte Dashboard, einschließlich Layout, Farben und Schriftarten.

JS-Dateien:

In den JavaScript-Dateien sind die Funktionen und Aufrufe an das Backend enthalten. Hier werden mit AJAX-Anfragen die Endpunkte der zuvor definierten API angesprochen. Die Kommunikation erfolgt hier ausschließlich über die REST Schnittstelle. Beispielhaft werden einige verwendete Logiken aufgeführt, die für die Funktionen elementar sind. Diese werden jeweils anhand eines speziellen Beispiels bei einzelnen Geräten genauer erklärt. Sie finden jedoch auch bei Räumen und Routinen Anwendung.

Update

Um einen aktuellen Stand anzeigen zu können, ist eine Funktion implementiert, die alle 30 Sekunden die Daten über einen GET-Aufruf aus dem Backend abruft und diese über eine "display devices"-Funktion anzeigt. Die externe API SmartThings begrenzt die minütlichen Abfragen, weshalb die Daten nur alle 30 Sekunden abgefragt werden können.
Die beschriebene Funktionalität wird durch die Funktion updateDevices() implementiert. Diese Funktion führt einen AJAX-GET-Aufruf an den Backend-Endpunkt durch, um die aktuellen Gerätedaten abzurufen, und übergibt diese dann an die displayDevices()-Funktion zur Anzeige

Edit

Die Anforderung, schon bestehende Geräte zu ändern, wird durch eine Bearbeitungsmaske ermöglicht. Hier werden über einen GET-Aufruf die aktuellen Daten abgerufen und in Input-Felder vorausgefüllt. Zudem wird für eine leichtere Bedienung, beispielsweise bei Räumen, eine Dropdown-Liste mit allen vorhandenen Räumen erstellt, aus denen der gewünschte Raum ausgewählt werden kann. Um alle bereits vorhandenen Räume und Gerätetypen anzuzeigen, werden hierfür separate GET-Anfragen geschrieben und die Ergebnisse eingefügt.

Beim Abspeichern der Änderungen prüft eine Funktion, ob Werte geändert wurden. Nur tatsächlich modifizierte Daten werden per POST-Anfrage an das Backend gesendet, was eine effiziente Ressourcennutzung gewährleistet.

```
function updateDeviceIfDifferent(field, newValue, oldValue, device_id) {
    if (newValue !== oldValue) {
        updateField(field, newValue, device_id);
    }}
```

toggle

Für jedes Gerät wird eine Checkbox erstellt, die je nach Gerätestatus als markiert oder nicht markiert angezeigt wird. Wenn der Benutzer die Checkbox ändert, wird die Funktion

handleCheckboxChange aufgerufen. Diese Funktion prüft, ob das Gerät ein- oder ausgeschaltet werden soll, und ruft anschließend die Funktion handleChange auf, die den eigentlichen Schaltvorgang durchgeführt.
Die Funktion handleChange sendet eine POST-Anfrage an die API, um den Gerätestatus zu aktualisieren.

```
function handleChange(isChecked, device_id, uniqueId, checkboxElement){
    $.ajax({
        url:
'https://smarthomebackend-spontaneous-bilby-ni.apps.01.cf.eu01.stackit.c
loud/api/device/' + device_id + '/switch/' + (isChecked ? 'on' : 'off'),
        method: 'POST',
        headers: {
            'Authorization': localStorage.getItem('authToken')
        },
        success: function(response) {
            console.log('Device updated successfully:', response);
            onSuccess(device_id, uniqueId);
        },
        error: function(error) {
            console.error('Error updating device:', error);
        }});}
```

Nach dem erfolgreichen Schalten überprüft die onSuccess-Funktion erneut den aktuellen Status des Geräts. Dies geschieht durch eine GET-Anfrage mit der Geräte-ID an den Server. Der Server sendet daraufhin den aktuellen Zustand des Geräts zurück. Basierend auf dieser Antwort wird die Checkbox entsprechend aktiviert oder deaktiviert.
Diese Funktion gewährleistet, dass der in der Benutzeroberfläche angezeigte Zustand des Geräts stets mit dem tatsächlichen Status auf dem Server übereinstimmt.

SPA

Die Single Page Application (SPA) des Smart Home Dashboards ist eine moderne Webanwendung, die mit Angular entwickelt wurde. Sie bietet eine benutzerfreundliche Oberfläche zur Verwaltung und Steuerung von Smart-Home-Geräten, Benutzern, Räumen und Routinen. Exemplarisch der Aufbau einer Komponente der SPA:

```
smarthome-dashboard-spa/
├── src/
│   ├── app/
│   │   ├── add-room/
│   │   │   ├──add-room.component.css
│   │   │   ├──add-room.component.html
│   │   │   ├──add-room.component.spec.ts
│   │   │   └──add-room.component.ts
```

Grundlegend wurde die SPA nach dem gleichen Prinzip wie die Multipage aufgebaut. Hierbei wurden dazu zwei verschiedene Seiten gebaut, die entweder mit dem /auth oder

dem /home Endpunkt angesprochen werden. Die /auth-Seite ist für die Authentifizierung der Benutzer zuständig, wo sich Nutzer anmelden oder registrieren können. Nach erfolgreicher Anmeldung werden sie zur /home-Seite weitergeleitet, welche das Hauptinterface der Anwendung darstellt.

Auf der /home-Seite werden die Kernfunktionen der Anwendung bereitgestellt. Die Seite ist modular aufgebaut und enthält Komponenten wie die Sidebar und den Hauptinhalt. Durch die Nutzung von Routing und Lazy Loading werden die einzelnen Module effizient geladen, was die Performance der Anwendung verbessert.

Das Design folgt dem Prinzip der klaren Trennung von Verantwortlichkeiten. Die Authentifizierung Logik ist vom Hauptteil der Anwendung getrennt, was die Wartbarkeit und Erweiterbarkeit des Codes erleichtert. Zudem fördert die modulare Struktur die Wiederverwendbarkeit von Komponenten und ermöglicht eine skalierbare Architektur für zukünftige Entwicklungen.
Darüber hinaus werden die Endpunkte mithilfe der AuthGuard Klasse geschützt, um unbefugten Zugriff zu verhindern. Die AuthGuard Klasse implementiert die CanActivate Schnittstelle von Angular und überprüft den Authentifizierungsstatus des Benutzers, bevor der Zugriff auf bestimmte Routen gewährt wird. Wenn ein Benutzer nicht angemeldet ist und versucht, auf eine geschützte Route zuzugreifen, wird er automatisch zur /auth/login Seite weitergeleitet.

Die Implementierung des AuthGuard sieht wie folgt aus:

```
// src/app/auth.guard.ts
import { Injectable, inject } from '@angular/core';
import { CanActivate, Router } from '@angular/router';
import { LoginService } from './login/login.service';
@Injectable({
  providedIn: 'root',
})
export class AuthGuard implements CanActivate {
  private loginService = inject(LoginService);
  private router = inject(Router);
  canActivate(): boolean {
    if (this.loginService.isLoggedIn()) {
      return true;
    } else {
      this.router.navigate(['/auth/login']);
      return false;
    }}}
```

Authentifizierung

In diesem Beispiel wird der LoginService verwendet, um zu überprüfen, ob der Benutzer eingeloggt ist. Ist dies nicht der Fall, wird er zur Login-Seite umgeleitet. Dieses Verfahren stellt sicher, dass nur autorisierte Benutzer Zugriff auf geschützte Inhalte haben.

Die Nutzung des AuthGuard erhöht die Sicherheit der Anwendung, indem sie unberechtigte Zugriffe verhindert und einen kontrollierten Zugang zu sensiblen Daten ermöglicht. Durch die Integration in das Routing-System der SPA wird eine nahtlose und sichere Benutzererfahrung gewährleistet.

Der einzige Punkte der nicht von der Multipage auf die SPA übertragen wird, ist die Funktion der Verifizierung der Mail. Dies musste aus Zeitgründen und Erschwernissen im Backend gekürzt werden. Somit kann der User zwar die Sign-Up Funktionalität nutzen, bekommt aber in der dazugehörigen Mail einen Link zur Multipage anstelle der SPA.

Kommunikation mit dem Backend

Mithilfe des Packages Apollo kommuniziert die SPA bei fast allen Anfragen mit dem GraphQL-Backend, welches die Daten und Funktionen für die Anwendung bereitstellt. Apollo fungiert dabei als GraphQL-Client, der es ermöglicht, effizient Daten vom Server zu laden und lokale Caching-Mechanismen zu nutzen. Durch diese Integration können Datenabfragen und -mutationen einfach definiert und verwaltet werden, was die Entwicklung und Wartung des Codes vereinfacht.
Nur bei Anmeldung wird das REST-Backend verwendet, da so eine konsistente Authentifizierung dargestellt werden kann.

Durch den effizienten Aufbau des Apollo Packages ist es möglich, die Authentifizierung einmal in der app.conifg.ts zu definieren. Hierbei wird jeder Request ein Header hinzugefügt, wenn ein Token im localStorage oder SessionStorage vorhanden ist. Hierfür wurde Apollo wie folgt implementiert:

```
provideApollo(() => {
      const httpLink = inject(HttpLink);
      const loginService = inject(LoginService);
      const basic = setContext((operation, context) => ({
        headers: {
          Accept: 'charset=utf-8',
        },
      }));
      const auth = setContext((operation, context) => {
        const token = loginService.getToken();
        if (token === null) {
          return {};
        } else {
          return {
            headers: {
              Authorization: `Bearer ${token}`,
            },
          };
        }});
      return {
        link: ApolloLink.from([
          basic,
          auth,
```

```
          httpLink.create({
            uri:
'https://smarthomegraphql-intelligent-jaguar-pd.apps.01.cf.eu01.stackit.clou
d/graphql',
          }),
        ]),
```

GraphQL

GraphQL ist eine Abfragesprache für APIs und eine Laufzeitumgebung zur Ausführung von Abfragen anhand eines Typ-Systems, das Ihre Daten beschreibt. In der SPA des Smart Home Dashboards wird GraphQL verwendet, um eine flexible und effiziente Schnittstelle für die Interaktion mit den Daten des Smart Home Systems bereitzustellen.

Im Projekt gibt es sowohl REST- als auch GraphQL-APIs, die ähnliche funktionale Service-Klassen verwenden. Diese Service-Klassen, wie z.B. DeviceService und RoomService, bieten ähnliche Funktionalitäten in beiden APIs an. Der Hauptunterschied besteht darin, dass die REST-API JSON-basierte Rückgabetypen verwendet, während die GraphQL-API typisierte Rückgabetypen verwendet, die durch das GraphQL-Schema definiert sind. Zum Beispiel bietet die REST-API Endpunkte wie /api/device und /api/room, die JSON-Daten zurückgeben, während die GraphQL-API Abfragen wie allDevices und allRooms bereitstellt, die typisierte Objekte zurückgeben. Hierbei sind zwar die möglichen Parameter vordefiniert, doch kann bei der Abfrage dynamisch definiert werden, welche von diesen zurückgegeben werden sollen. Trotz dieser Unterschiede in den Rückgabetypen bleibt die zugrunde liegende Logik in den Service-Klassen konsistent, was eine einheitliche Verwaltung und Verarbeitung der Daten ermöglicht.

Grundlegende Konzepte

Schema: Das Schema definiert die Struktur der API, einschließlich der verfügbaren Typen, Abfragen und Mutationen. Es beschreibt, welche Daten abgefragt und welche Operationen ausgeführt werden können.

Query: Eine Query ist eine Abfrage, die Daten vom Server anfordert. Sie spezifiziert, welche Felder und verschachtelten Felder benötigt werden.

Mutation: Eine Mutation ist eine Abfrage, die die Daten auf dem Server verändert (z.B. Erstellen, Aktualisieren oder Löschen von Daten).

Schema-Definition

Das Schema wird in einer .graphqls-Datei definiert, welche in den Resources liegt. Hier ist ein Auszug aus dem Schema, welches Beispielhaft die Mutations und Querys für Geräte zeigt:

```
type Query {
    deviceById(id: ID!): Device
    allDevices: [Device]!
}
type Mutation {
    createDevice(id: ID!, name: String!, typeId: ID!, roomId: ID!):
Boolean
    updateDevice(id: ID!, name: String, typeId: ID, roomId: ID): Device
    switchDevice(id: ID!, status: String!): Device
    deleteDevice(id: ID!): Boolean
  }
type Device {
    id: ID!
    name: String!
    type: DeviceType!
    room: Room!
    status: String
    state: String }
type DeviceType {
    id: ID!
    name: String!
    icon: String }
type Room {
    id: ID!
    name: String! }
```

Controller und Resolver

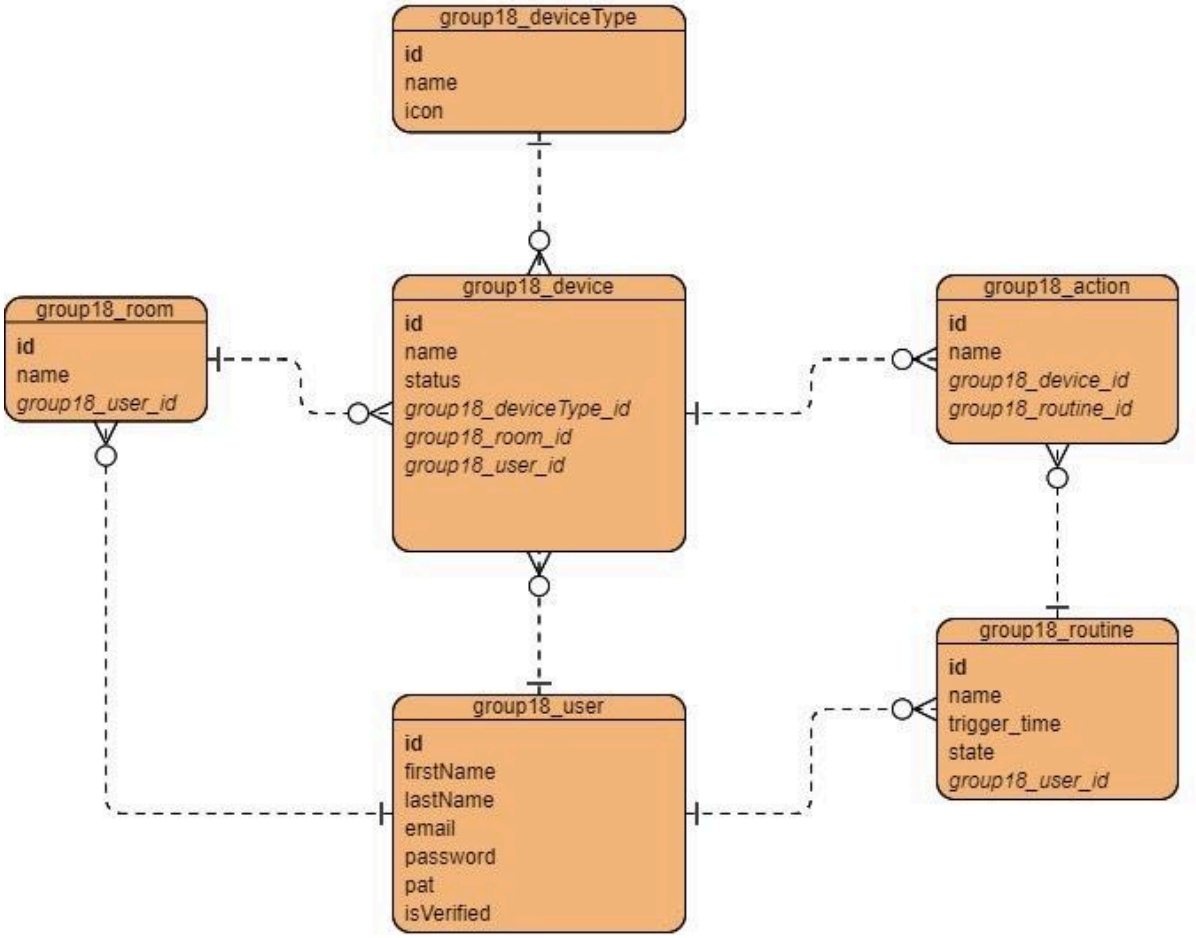
Die Controller-Klassen enthalten sogenannte Resolver-Methoden, die die Daten für die Abfragen und Mutationen bereitstellen. Diese rufen genau wie in dem REST Backend die korrespondierenden Service Klassen auf. Hierbei ist ein entscheidender Unterschied, dass die Methoden bei Erfolg keine separate Objekte wie mit Jackson zurückgeben, sondern hier einfach die “normalen” Objekte zurückgeben. Auch gibt es hier keine ResponseEntity, sondern es wird bei Fehlern ein Runtime Error geworfen, da in GraphQL nur mit sehr hohem Aufwand ein HTTP Code und eine benutzerdefinierte Error-Nachricht erstellt werden können.

AuthInterceptor

Der AuthInterceptor ist ein Spring Interceptor, der eingehende HTTP-Anfragen abfängt und das JWT überprüft. Wenn das Token gültig ist, wird der Benutzer authentifiziert. Somit braucht es nicht, wie im Rest, eine Wiederholung von Code, sondern kann einmal zentral definiert werden.

```
public class AuthInterceptor implements HandlerInterceptor {
    ...
    @Override
    public boolean preHandle(HttpServletRequest request,
HttpServletResponse response, Object handler) throws Exception {
        String token = request.getHeader("Authorization");
        if (token != null && token.startsWith("Bearer ")) {
            token = token.substring(7);
            String email = authService.extractEmail(token);
            User user = userService.getUserByEmail(email);
            authenticatedUser.set(user);
        }
        return true;
    }
    public static User getAuthenticatedUser() {
        return authenticatedUser.get();
    }
    public static void clear() {
        authenticatedUser.remove();
    }}
```

Postgres DB



Dies ist das Entity-Relationship (ER)-Diagramm, das die DB darstellt. Hier werden die einzelnen Entitäten, ihre Attribute und die Beziehungen zwischen ihnen erklärt.

Im Smart Home Dashboard Projekt verwenden wir PostgreSQL-Datenbank-Constraints, um die referenzielle Integrität sicherzustellen. Insbesondere nutzen wir ON DELETE CASCADE Constraints, um sicherzustellen, dass alle zugehörigen Datensätze automatisch gelöscht werden, wenn ein Benutzer, Device oder Routine gelöscht wird. Dies bedeutet, dass z.B. beim Löschen eines Benutzers alle mit diesem Benutzer verknüpften Datensätze, wie z.B. Geräte, Räume und Routinen, ebenfalls entfernt werden.

Die Datenbank basiert auf einer relationalen Modellierung, bei der zentrale Entitäten wie Benutzer, Räume, Geräte, Gerätetypen, Aktionen und Routinen in separaten Tabellen organisiert sind. Beziehungen zwischen diesen Entitäten werden durch Fremdschlüssel hergestellt, was eine flexible und skalierbare Zuordnung von Geräten zu Räumen, Aktionen zu Geräten und Routinen zu Benutzern ermöglicht.

Diese Datenbank befindet sich in der dritten Normalform (3NF). Sie erfüllt die Anforderungen der ersten Normalform (1NF), da alle Attribute atomare Werte enthalten, und die zweite Normalform (2NF), da alle Nicht-Schlüssel-Attribute vollständig vom Primärschlüssel abhängen. Außerdem sind alle Tabellen in der dritten Normalform, da keine transitiven

Abhängigkeiten existieren – jedes Nicht-Schlüssel-Attribut hängt ausschließlich vom Primärschlüssel ab. Dadurch werden Redundanzen minimiert und Datenintegrität gewährleistet.

Reflektion

Reflexion Chris David Kaufmann

In diesem Projekt war ich sowohl für das Frontend als auch für die Dokumentation zuständig. Mein Hauptbeitrag bestand darin, eine ansprechende und intuitive Benutzeroberfläche für die Steuerung der Smart-Home-Geräte zu entwickeln sowie die Projekt- und Code-Dokumentation zu verfassen, um die Plattform verständlich und leicht wartbar zu machen.

Im Frontend-Bereich habe ich strukturiert gearbeitet, indem ich ein klares Designkonzept und die wichtigsten Funktionalitäten definiert habe. Dabei habe ich vor allem AJAX genutzt, um die Kommunikation zwischen Frontend und Backend effizient zu gestalten. Es war entscheidend, dass die Anwendung reaktionsschnell bleibt und Daten in Echtzeit aktualisiert werden können. Bei der Dokumentation habe ich besonderen Wert auf die klare und prägnante Beschreibung der API-Schnittstellen und der Systemarchitektur gelegt, um sicherzustellen, dass das Projekt für zukünftige Entwickler und Nutzer leicht verständlich ist.

Eine meiner Stärken liegt darin, technische Abläufe sowohl im Code als auch in der Dokumentation verständlich und strukturiert darzustellen. Diese Fähigkeit hat dabei geholfen, die Projektarbeit übersichtlich zu halten und die Zusammenarbeit im Team zu erleichtern. Allerdings habe ich festgestellt, dass ich während intensiver Entwicklungsphasen nicht immer genug Zeit für die laufende Dokumentation eingeplant habe, sodass ich am Ende einige Abschnitte nacharbeiten musste.

In einem ähnlichen Projekt würde ich darauf achten, die Dokumentation während der Entwicklung kontinuierlich zu aktualisieren, um spätere Engpässe zu vermeiden. Eine noch engere Zusammenarbeit mit dem Backend-Team von Anfang an wäre ebenfalls hilfreich gewesen, um die API-Integration frühzeitig besser abzustimmen und potenzielle Probleme früher zu erkennen.

Meinen Kommilitonen würde ich empfehlen, die Dokumentation nicht zu unterschätzen. Eine gut strukturierte und verständliche Dokumentation spart Zeit und verbessert die Zusammenarbeit im Team. Zudem sollte von Beginn an auf klare Design- und Entwicklungsrichtlinien geachtet werden, um Missverständnisse und ineffizientes Arbeiten zu vermeiden.

Reflexion Paula Bauer

In unserem Projekt habe ich mich hauptsächlich auf die Entwicklung des Frontends konzentriert. Mein Beitrag bestand darin, die Benutzeroberfläche der Single Page Application (SPA) zu gestalten und sicherzustellen, dass die Interaktion mit der SmartThings API reibungslos verläuft. Besonders wichtig war dabei, dass die Anwendung intuitiv bedienbar ist und die Steuerung der Smart-Home-Geräte schnell und unkompliziert ermöglicht.

Ich habe strukturiert gearbeitet, indem ich zu Beginn des Projekts eine klare Strategie für die Benutzeroberfläche entwickelt habe. Im Nachhinein habe ich jedoch festgestellt, dass ich zu wenig Zeit auf Designentscheidungen verwendet habe. Dieser eingeschränkte Fokus auf

das Design führte dazu, dass die Benutzeroberfläche weniger ansprechend war, als sie hätte sein können. Eine ausgewogenere Priorisierung, bei der Designaspekte stärker berücksichtigt werden, wäre in manchen Phasen des Projekts effizienter gewesen.

Zu meinen Stärken zählt meine Fähigkeit, moderne Webtechnologien einzusetzen und eine funktionale, saubere Benutzeroberfläche zu entwickeln. Da der Fokus in diesem Projekt jedoch eher auf der Funktionalität lag, fiel es mir stellenweise schwer, ein ästhetisch ansprechendes Design umzusetzen. Trotzdem habe ich wertvolle Erfahrung gesammelt, indem ich grundlegende CSS-Techniken erlernt und erstmals intensiver mit Designaspekten gearbeitet habe. Gleichzeitig konnte ich bei der Entwicklung der funktionalen Elemente unterstützen, was mir deutlich besser liegt. In Zukunft wäre daher eine andere Aufgabenverteilung sinnvoll, um meine Stärken im Bereich Funktionalität stärker einzubringen.

In einem ähnlichen Projekt würde ich darauf achten, das Design nicht aus den Augen zu verlieren, da wir uns im Verlauf zu sehr auf die Funktionalitäten konzentriert haben. Eine engere Zusammenarbeit mit den Backend-Entwicklern von Anfang an wäre hilfreich, um sicherzustellen, dass Schnittstellen frühzeitig getestet und integriert werden können, ohne dass Designaspekte dabei vernachlässigt werden. Außerdem würde ich regelmäßiger Nutzerfeedback einholen, um das Design schrittweise anzupassen und die Balance zwischen Ästhetik und Funktionalität besser zu halten.

Meinen Kommilitonen würde ich empfehlen, von Anfang an ausreichend Zeit in ein ansprechendes Design zu investieren und es im Projektverlauf nicht aus den Augen zu verlieren. Ein solides Grunddesign sorgt dafür, dass die Anwendung optisch ansprechend bleibt, und erleichtert spätere Anpassungen. Zudem hilft eine enge Abstimmung mit den Backend-Entwicklern, technische Probleme frühzeitig zu lösen und sicherzustellen, dass Design und Funktionalität Hand in Hand gehen.

Reflexion Tim Sommer

In unserem Projekt war ich für das Backend verantwortlich, einschließlich der Implementierung der GraphQL- und REST-Schnittstellen, der PostgreSQL-Datenbank sowie der Anbindung an die SmartThings API. Darüber hinaus habe ich die Gesamtkoordination des Projekts übernommen, was sowohl die technische Leitung als auch die organisatorische Planung umfasste.

Ich habe von Anfang an strukturiert gearbeitet, indem ich die Systemarchitektur geplant, die API-Integrationen konzipiert und die Datenbank modelliert habe. Ein zentraler Aspekt meiner Arbeit war die Implementierung von GraphQL, um flexible und effiziente Abfragen zu ermöglichen, sowie die Integration von REST-Schnittstellen für die Kommunikation mit der SmartThings API und externen Diensten wie dem Mail-Service. Dadurch konnten wir sicherstellen, dass die Plattform in Echtzeit mit den Smart-Home-Geräten kommuniziert und Nutzer beispielsweise Benachrichtigungen per E-Mail erhalten.

Zu meinen Stärken gehört es, Backend-Architekturen zu entwerfen und deren Komponenten effizient miteinander zu verknüpfen. Dies hat mir ermöglicht, die Anforderungen des Projekts in einer skalierbaren und gut strukturierten Art und Weise umzusetzen. Als Projektleiter war es zudem meine Aufgabe, das Team zu koordinieren, Aufgaben zu delegieren und dafür zu

sorgen, dass wir die Meilensteine einhalten. Die Kommunikation im Team war dabei besonders wichtig.

Eine Schwäche, die sich im Verlauf des Projekts zeigte, war meine Neigung, zu viel Verantwortung selbst zu übernehmen, was gelegentlich dazu führte, dass ich andere Aufgaben zurückstellte und weniger Raum für die Eigeninitiative der Teammitglieder ließ. Außerdem stellte sich heraus, dass das API-Design in einigen Punkten nicht vollständig durchdacht war, insbesondere bei den Datentypen, was in bestimmten Bereichen Optimierungspotential aufwies und an wenigen Stellen sogar zu notwendigen Korrekturen führte. Rückblickend wäre eine intensivere Planung hilfreich gewesen, um spätere Änderungen wie die Auslagerung des RoutineServices in eine separate Komponente zu vermeiden. In einem ähnlichen Projekt würde ich stärker auf eine durchdachte Planungsphase achten und die Aufgabenverteilung optimieren, um mehr Vertrauen in das Team zu setzen und solche Anpassungen frühzeitig einzuplanen.

In einem zukünftigen Projekt würde ich außerdem mehr Zeit für die Planung der API-Tests einplanen. Die enge Abstimmung zwischen Frontend und Backend verlief gut, hätte aber noch flüssiger sein können, wenn wir mehr automatisierte Tests für die API-Integrationen von Anfang an eingebaut hätten. Auch das Einholen von Feedback zu den Projektfortschritten hätte regelmäßiger geschehen sollen, um die gesamte Entwicklungsarbeit noch transparenter zu gestalten.

Meinen Kommilitonen würde ich empfehlen, bei Projekten mit komplexen Backend-Strukturen frühzeitig einen klaren Entwicklungsplan zu erstellen und diesen flexibel an die Projektbedürfnisse anzupassen. Zudem ist es wichtig, als Projektleiter nicht nur die technischen Anforderungen im Blick zu behalten, sondern auch die Stärken des Teams zu nutzen und ein kollaboratives Arbeitsumfeld zu fördern. Kontinuierliche Abstimmung und Feedback-Schleifen sind der Schlüssel, um effizient und erfolgreich zu arbeiten.