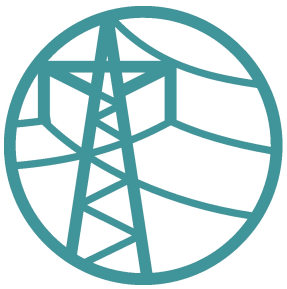


WYDZIAŁ
ELEKTROTECHNIKI
I INFORMATYKI
POLITECHNIKI RZESZOWSKIEJ

Jakub Kusal

HuePi - aplikacja mobilna do obsługi żarówek Philips Hue

Praca dyplomowa inżynierska

Opiekun pracy:
dr inż. Mariusz Mączka

Rzeszów, 2025

Spis treści

1. Wstęp

Używanie urządzeń typu *Smart Home* cieszy się dziś coraz większym zainteresowaniem. [1] Nie jest to już tylko ciekawostka technologiczna, która musi wiązać się z dużymi wydatkami. Urządzenia tego typu stają się coraz bardziej przystępne cenowo, a producenci oferują coraz to większy wybór samych urządzeń końcowych, jak i urządzeń integrujących całą resztę. Do takich urządzeń możemy zaliczyć m.in.: kamery, zamki, głośniki, wyświetlacze, termostaty, gniazdka, żarówki itp. Widać więc, że spośród całej gamy urządzeń, mamy do wyboru opcje takie, które pomagają zautomatyzować i wspomóc życie ludzi, jak również takie, które służą rozrywce.

Tym ostatnim typem urządzeń zajęto się w tym projekcie. Jego celem było stworzenie aplikacji mobilnej Android, umożliwiającej sterowanie żarówkami producenta Philips, z serii Philips Hue. Jest to cały ekosystem smart urządzeń, nie tylko żarówek. W jego skład wchodzą m.in. również: paski LED, lampy stojące, lampki "choinkowe", przyciski, gniazdka, kamery, czujniki ruchu, czujniki zmierzchu.

Mimo ogromnej gamy urządzeń, można jednak zauważyć, iż nie występują w tym systemie żadnego rodzaju czujniki temperatury, czy sensory innych warunków atmosferycznych. Tego typu akcesorium, mogłoby rozszerzyć, już jakże szeroki wachlarz możliwości ekosystemu Philips Hue, dzięki któremu korzystanie z np. żarówek stałoby się jeszcze bardziej ekscytujące. Niniejszy projekt celuje w wypełnienie luki powstałej z faktu braku występowania wyżej wymienionych sensorów i przedstawienie konceptu, który pokazuje, jak natywny ekosystem Philips Hue mógłby się rozwinąć.

2. Cel oraz zakres projektu

Celem projektu, było stworzenie aplikacji mobilnej na systemy operacyjne Android, przeznaczonej do sterowania żarówkami Philips Hue. Nie stanowi ona pełnoprawnego oprogramowania do sterowania każdym produktem z ekosystemu Philips Hue. Poza podstawowymi funkcjonalnościami, ma ona ukazać możliwości, o jakie możnaby rozszerzyć natywne ekosystemy Philips Hue. Oprócz klasycznego sterowania, tj. ustawienie koloru, jasności, włączenia i wyłączenia, aplikacja miała umożliwiać wykorzystanie informacji o temperaturze, w celu wyświetlania kolorów na podstawie tejże temperatury.

2.1. Podobne oprogramowanie

Pomysł na tego typu aplikację nie jest nowy. Istnieje wiele systemów, aplikacji i oprogramowania, które implementują funkcjonalności w zaprezentowanej tutaj aplikacji i mają podobną filozofię. Przykłady niektórych z nich to:

2.1.1. Philips Hue (aplikacja producenta)

Philips Hue to oficjalna aplikacja mobilna producenta, przeznaczona głównie do zarządzania inteligentnym oświetleniem z rodziny Hue. Udostępniana jest na urządzenia z systemem *Android* oraz *iOS*,a jej podstawowym zadaniem jest umożliwienie pełnej konfiguracji i kontroli wszystkich elementów tego ekosystemu, włącznie z tworzeniem scen, harmonogramów oraz automatyzacji.
Aplikacja komunikuje się z mostkiem Philips Hue, który jest fizycznym urządzeniem pełniącym rolę pośrednika pomiędzy oświetleniem a siecią lokalną. Bezpośrednia integracja z mostkiem umożliwia:

- dodawanie nowych żarówek lub innych akcesoriów (np. czujników ruchu, gniazdek) do ekosystemu,
- tworzenie i edytowanie tzw. *scen świetlnych* (zestawów preferowanych ustawień światła, takich jak kolor, jasność czy temperatura barwowa),
- planowanie harmonogramów, pozwalających automatycznie włączać lub wyłączać światło o wybranych porach dnia,
- sterowanie głosowe (po odpowiedniej konfiguracji z asystentami pokroju Ama-

zon Alexa, Apple Siri czy Google Assistant),

- konfigurowanie prostych reguł automatyzacji, które mogą reagować np. na wykrycie ruchu.

Korzystanie z aplikacji Philips Hue pozwala w łatwy sposób zapanować nad całym ekosystemem oświetlenia inteligentnego, jednakże nie wspiera ona pomiaru parametrów środowiskowych, takich jak temperatura czy wilgotność. W efekcie, choć zaspokaja większość typowych potrzeb związanych z inteligentnym oświetleniem, nie rozszerza funkcjonalności systemu o dodatkowe odczyty czy też automatyzacje bazujące na czynnikach zewnętrznych (m.in. z użyciem czujników temperatury).

2.1.2. Home Assistant

Home Assistant to otwartoźródłowe oprogramowanie przeznaczone do lokalnego zarządzania i automatyzacji urządzeń w inteligentnym domu. Można je wdrożyć zarówno na komputerach jednopłytkowych (np. Raspberry Pi), jak i na serwerach czy w środowisku kontenerowym (np. Docker). Dzięki temu jest rozwiązaniem elastycznym pod względem wymagań sprzętowych i sposobu instalacji.
Dużą zaletą Home Assistant jest bogaty ekosystem integracji, który obejmuje również wsparcie dla oświetlenia Philips Hue. Oprócz podstawowego sterowania, możliwe jest konfigurowanie zaawansowanych reguł automatyzacji, reagujących np. na informacje z rozmaitych czujników (temperatury, wilgotności czy ruchu). Platforma oferuje wbudowany edytor scen i skryptów, pozwalający na tworzenie wieloetapowych akcji wyzwalanych przez określone zdarzenia (bądź harmonogramy czasowe).
Home Assistant zapewnia także dostęp do panelu webowego, dzięki któremu w wygodny sposób można zarządzać urządzeniami, przeglądać ich status, a także dostosowywać interfejs użytkownika zgodnie z własnymi preferencjami. Ze względu na swoją otwartość i rozbudowaną społeczność, umożliwia wprowadzanie dodatkowych wtyczek (*add-ons*) i komponentów, co sprawia, że możliwości systemu są praktycznie nieograniczone. Duża liczba gotowych rozwiązań udostępnianych w formie repozytoriów społecznościowych pozwala szybko uruchamiać nowe funkcjonalności lub modyfikować istniejące.

2.1.3. Google Home

Google Home to platforma i aplikacja mobilna rozwijana przez firmę Google, służąca do zarządzania urządzeniami inteligentnego domu. W odróżnieniu od rozwiązań stricte samodzielnych, aplikacja ta ściśle integruje się z ekosystemem usług Google

oraz asystentem głosowym Google Assistant, co pozwala sterować wieloma sprzętami za pomocą komend głosowych.

Podobnie jak inne rozwiązania z tej kategorii, Google Home obsługuje oświetlenie Philips Hue, umożliwiając jego konfigurację, tworzenie tzw. pokoi (ang. *rooms*) i grupowanie urządzeń w praktyczny sposób. Pozwala także w prosty sposób zaprogramować podstawowe sceny świetlne czy harmonogramy włączania i wyłączania. Ze względu na powiązania z kontem Google, cała konfiguracja pozostaje dostępna z różnych urządzeń (smartfon, tablet, głośniki Nest itp.), co ułatwia zdalne sterowanie oświetleniem oraz innymi elementami inteligentnego domu (np. termostatami, głośnikami czy kamerami). W przeciwieństwie do bardziej zaawansowanych platform otwartoźródłowych, Google Home koncentruje się głównie na wygodzie i prostocie użytkowania. Oferuje co prawda możliwość definiowania automatyzacji – tzw. *Routines* – jednak nie pozwala na tak rozbudowane scenariusze integracji, jak systemy pokroju Home Assistant. Niemniej jednak dzięki swojemu podejściu „wszystko w jednym" i ścisłemu powiązaniu z usługami Google, Google Home jest dobrym wyborem dla użytkowników szukających łatwego w konfiguracji i intuicyjnego rozwiązania do codziennego sterowania oświetleniem Philips Hue, jak również innymi urządzeniami typu *Smart Home*.

3. Wykorzystany sprzęt i technologie

Projekt można podzielić na 2 główne części: serwer uruchomiony na Raspberry Pi 3B wraz z czujnikiem oraz aplikacja moblina.

3.1. Serwer na Raspberry Pi

3.1.1. Raspberry Pi 3B

Raspberry Pi 3B to komputer jednopłytkowy opracowany przez fundację Raspberry Pi, który od momentu wprowadzenia na rynek zdobył ogromną popularność zarówno wśród hobbystów, jak i profesjonalistów. Jego niewielkie wymiary (85.6 mm x 56.5 mm) oraz niska cena sprawiają, że stanowi doskonałe rozwiązanie do tworzenia różnorodnych projektów technologicznych, takich jak systemy *Smart Home*, urządzenia IoT (Internet of Things), serwery multimedialne, a nawet jako komputer edukacyjny. Raspberry Pi 3B jest wyposażony w czterordzeniowy procesor ARM Cortex-A53 o taktowaniu 1.2 GHz, co w połączeniu z 1 GB pamięci RAM pozwala na płynne działanie lekkich systemów operacyjnych, takich jak Raspberry Pi OS, Ubuntu czy inne dystrybucje Linuxa zoptymalizowane pod kątem architektury ARM. Wbudowana karta Wi-Fi obsługująca standard 802.11n oraz moduł Bluetooth 4.1 umożliwiają bezprzewodową komunikację z innymi urządzeniami oraz sieciami lokalnymi bez konieczności stosowania dodatkowych adapterów.
Płyta oferuje liczne interfejsy wejścia i wyjścia, co czyni ją wyjątkowo wszechstronną w zastosowaniach praktycznych. Na wyposażeniu znajdują się m.in.:

- 40-pinowy złącze GPIO (*General Purpose Input/Output*), które umożliwia podłączanie i sterowanie szeroką gamą urządzeń peryferyjnych, takich jak czujniki, diody LED, przyciski czy serwomechanizmy,
- 4 porty USB 2.0, umożliwiające podłączenie akcesoriów takich jak klawiatura, mysz, kamera czy pendrive,
- złącze HDMI do podłączenia monitora, co pozwala na wykorzystanie urządzenia jako miniaturowego komputera stacjonarnego,
- złącze CSI do kamer i DSI do ekranów, co otwiera możliwości budowy systemów monitoringu czy interaktywnych wyświetlaczy,

- interfejsy komunikacyjne, takie jak I^2C, *SPI* i *UART*, służące do komunikacji z różnymi modułami i sensorami.

Jednym z przykładów modułów, które mogą być zintegrowane z Raspberry Pi 3B, jest czujnik BME280. Urządzenie to umożliwia precyzyjne pomiary temperatury, wilgotności oraz ciśnienia atmosferycznego. Dzięki swoim kompaktowym wymiarom i wsparciu dla interfejsu I^2C, czujnik BME280 jest często wykorzystywany w projektach związanych z monitorowaniem warunków środowiskowych, systemami *Smart Home* czy urządzeniami prognozującymi pogodę. Raspberry Pi 3B, w połączeniu z takim czujnikiem, może odczytywać dane w czasie rzeczywistym, co pozwala na ich dalsze przetwarzanie oraz wykorzystanie w aplikacjach automatyzacji lub raportowania. Dzięki wielu dostępnym bibliotekom, integracja czujnika z urządzeniem jest stosunkowo prosta i szybka.
Raspberry Pi 3B wspiera również wiele oprogramowania pozwalającego na wygodną integrację z urządzeniami *Smart Home*. Na przykład, instalacja platform takich jak poprzednio omawiany Home Assistant umożliwia tworzenie zaawansowanych reguł automatyzacji oraz centralne zarządzanie wieloma urządzeniami, w tym oświetleniem Philips Hue. Możliwość uruchamiania serwerów lokalnych (np. HTTP czy MQTT) sprawia, że Raspberry Pi może pełnić rolę centrum sterowania lub bramy komunikacyjnej w bardziej rozbudowanych projektach.
Dzięki aktywnej społeczności użytkowników, Raspberry Pi 3B jest dobrze udokumentowany, a w sieci dostępne są liczne poradniki i przykłady implementacji projektów. Wszechstronność i niska cena tego komputera sprawiają, że jest on chętnie wybierany zarówno do celów edukacyjnych, jak i komercyjnych. Raspberry Pi 3B może być zatem kluczowym elementem projektów opartych na inteligentnym oświetleniu, takich jak *HuePi*, w których służy jako serwer API przetwarzający dane z sensorów oraz komunikujący się z żarówkami w ekosystemie Philips Hue.

3.1.2. Czujnik BME280

Czujnik BME280 to wielofunkcyjny sensor środowiskowy opracowany przez firmę Bosch, przeznaczony do pomiaru trzech podstawowych parametrów: temperatury, wilgotności oraz ciśnienia atmosferycznego. Ze względu na swoją wszechstronność, kompaktowe rozmiary oraz wysoką precyzję pomiarów, znajduje szerokie zastosowanie w projektach związanych z automatyką domową, Internetem Rzeczy (IoT) oraz urządzeniami monitorującymi warunki środowiskowe.

BME280 wspiera interfejsy komunikacyjne I^2C oraz *SPI*, co czyni go łatwym do integracji z różnymi mikrokontrolerami i komputerami jednopłytkowymi, takimi jak Raspberry Pi. Dzięki temu użytkownik ma dużą elastyczność w wyborze platformy sprzętowej. Czujnik charakteryzuje się niskim zużyciem energii, co sprawia, że idealnie nadaje się do zastosowań w systemach zasilanych bateryjnie.
Specyfikacja czujnika obejmuje:

- Zakres pomiaru temperatury: od -40°C do +85°C, z dokładnością do ±1°C[2],
- Zakres pomiaru wilgotności: od 0% do 100% RH (wilgotność względna), z dokładnością ±3% RH[2],
- Zakres pomiaru ciśnienia atmosferycznego: od 300 hPa do 1100 hPa, z dokładnością do ±1 hPa[2].

Czujnik jest wykorzystywany w aplikacjach takich jak:

- monitorowanie warunków środowiskowych w budynkach (np. wilgotność w pomieszczeniach mieszkalnych),
- systemy pogodowe, w tym prognozowanie zmian ciśnienia,
- automatyzacja domowa, np. w połączeniu z systemami sterowania oświetleniem, które mogą reagować na zmiany temperatury,
- urządzenia IoT, zbierające dane do analizy i przesyłające je do systemów chmurowych lub lokalnych serwerów.

Czujnik BME280 jest często wykorzystywany w projektach prototypowych i edukacyjnych dzięki szerokiej dostępności bibliotek programistycznych. Możliwość dokładnych pomiarów w połączeniu z łatwością integracji sprawia, że czujnik ten idealnie nadaje się do implementacji w projektach takich jak *HuePi*, w których dane np. o temperaturze mogą być przetwarzane w celu dostosowania funkcji systemu, np. sterowania oświetleniem.

3.1.3. FastAPI

FastAPI to nowoczesny framework do tworzenia aplikacji webowych oraz API w języku Python. Został zaprojektowany z myślą o wysokiej wydajności, prostocie użytkowania oraz integracji z typowaniem statycznym w Pythonie. Dzięki zastosowaniu

standardu ASGI (Asynchronous Server Gateway Interface), FastAPI wspiera obsługę zapytań w trybie asynchronicznym, co czyni go szczególnie przydatnym w aplikacjach wymagających dużej liczby jednoczesnych zapytań lub intensywnej komunikacji z zewnętrznymi usługami. FastAPI jest oparty na narzędziu *Starlette* (do obsługi zapytań i trasowania) oraz *Pydantic* (do walidacji i serializacji danych). Oferuje funkcjonalności, które czynią go jednym z najpopularniejszych frameworków do budowy API, takich jak:

- Automatyczne generowanie dokumentacji API na podstawie definicji punktów końcowych (*endpoints*) i typów danych. Dokumentacja jest dostępna w formatach *Swagger UI* oraz *ReDoc*, co ułatwia testowanie i integrację API.

- Wsparcie dla typowania w Pythonie, co umożliwia precyzyjne określenie wejściowych i wyjściowych danych dla punktów końcowych. Typowanie pozwala również na automatyczną walidację danych i zmniejsza ryzyko błędów programistycznych.

- Obsługa asynchroniczności dzięki natywnemu wsparciu dla konstrukcji `async` i `await`, co sprawia, że framework doskonale sprawdza się w aplikacjach wymagających współbieżności, takich jak przetwarzanie danych w czasie rzeczywistym czy obsługa mikroserwisów.

- Łatwa integracja z systemami autoryzacji i uwierzytelniania, np. przy użyciu protokołów OAuth2 czy JWT (*JSON Web Tokens*).

- Rozbudowane wsparcie dla operacji związanych z przetwarzaniem żądań HTTP, takich jak pobieranie danych z parametrów, nagłówków czy plików przesyłanych przez użytkowników.

FastAPI znajduje zastosowanie w wielu obszarach, takich jak:

- Tworzenie mikroserwisów – dzięki asynchroniczności i wysokiej wydajności framework świetnie nadaje się do budowy modułowych aplikacji o niewielkiej złożoności.

- Systemy IoT – FastAPI może być używane jako interfejs do komunikacji między urządzeniami a centralnym serwerem.

- Tworzenie RESTful API – pozwala na szybkie i efektywne tworzenie punktów końcowych dla aplikacji webowych czy mobilnych.

- Integracja z systemami ML/AI – FastAPI jest często wykorzystywane do udostępniania modeli uczenia maszynowego w formie usług API, umożliwiających ich łatwą integrację z innymi aplikacjami.

Dzięki intuicyjnej składni, bogatej dokumentacji oraz dużej społeczności użytkowników, FastAPI stał się jednym z najchętniej wybieranych narzędzi do budowy nowoczesnych aplikacji webowych i API w Pythonie.

3.2. Aplikacja mobilna

3.2.1. Język Kotlin

Kotlin to nowoczesny, wieloplatformowy język programowania opracowany przez firmę JetBrains. Jego pierwsza stabilna wersja została wydana w 2016 roku [3], a już w 2017 roku został oficjalnie ogłoszony przez Google jako język wspierany do tworzenia aplikacji na system Android. Kotlin został zaprojektowany jako nowoczesna alternatywa dla Javy, oferująca większą produktywność, czytelność kodu oraz nowoczesne funkcje, które redukują ryzyko błędów programistycznych.
Kotlin jest językiem zorientowanym obiektowo z elementami programowania funkcyjnego. Wyróżnia go pełna interoperacyjność z kodem napisanym w Javie, co umożliwia łatwą migrację istniejących projektów. Dzięki temu programiści mogą stopniowo wprowadzać Kotlin do swoich aplikacji, korzystając z bibliotek i narzędzi napisanych w Javie.
Jednym z kluczowych powodów popularyzacji Kotlina jest fakt, że Google wymaga jego użycia do pracy z Jetpack Compose – nowoczesnym narzędziem do tworzenia interfejsów użytkownika w aplikacjach na system Android. Dzięki Jetpack Compose, aplikacje mogą być projektowane szybciej i w bardziej intuicyjny sposób, co sprawia, że Kotlin staje się praktycznie nieodzownym elementem ekosystemu Android.
Do najważniejszych cech języka Kotlin należą:

- **Bezpieczeństwo typów** – Kotlin eliminuje ryzyko wystąpienia błędów typu `NullPointerException` dzięki systemowi *nullable types*, który wymaga jawnego oznaczenia zmiennych mogących przechowywać wartość `null`.

- **Krótszy i czytelniejszy kod** – Kotlin wprowadza nowoczesne konstrukcje składniowe, takie jak wyrażenia lambda, funkcje rozszerzające czy destrukturyzacja, co znacząco zmniejsza ilość kodu wymaganego do implementacji określonych funkcji w porównaniu z Javą.

- **Wsparcie dla programowania funkcyjnego** – język umożliwia tworzenie wyrażeń lambda, operacje na kolekcjach oraz wykorzystanie funkcji wyższego rzędu, co zwiększa elastyczność kodu.

- **Pełna kompatybilność z JVM** – Kotlin działa na maszynie wirtualnej Javy (*Java Virtual Machine*) i może wykorzystywać wszystkie istniejące biblioteki oraz frameworki napisane w Javie.

- **Obsługa współbieżności** – Kotlin wprowadza natywne wsparcie dla współbieżności w postaci *coroutines*, które umożliwiają łatwiejsze i bardziej efektywne zarządzanie współbieżnymi zadaniami.

- **Wieloplatformowość** – dzięki Kotlin Multiplatform język pozwala na tworzenie wspólnego kodu, który może działać na różnych platformach, takich jak Android, iOS, JVM, JavaScript, a nawet natywne aplikacje desktopowe.

Kotlin znajduje zastosowanie w wielu dziedzinach:

- **Tworzenie aplikacji mobilnych** – Kotlin jest obecnie najpopularniejszym językiem do tworzenia aplikacji na system Android dzięki swojej prostocie, wydajności i wsparciu przez Google.

- **Tworzenie aplikacji serwerowych** – Kotlin może być używany z frameworkami takimi jak Ktor czy Spring Boot do budowy nowoczesnych API i aplikacji webowych.

- **Programowanie wieloplatformowe** – Kotlin Multiplatform umożliwia tworzenie wspólnego kodu, który działa zarówno na systemach Android, jak i iOS, co znacząco obniża koszty i czas rozwoju aplikacji.

- **Aplikacje desktopowe** – dzięki Kotlin/Native można tworzyć aplikacje działające na systemach operacyjnych takich jak Windows, macOS czy Linux.

Kotlin zdobył uznanie wśród programistów dzięki swojej nowoczesności, elastyczności i możliwościom. Używa go ponad 60% profesjonalnych programistów Android. [4] Jego integracja z narzędziami JetBrains, takimi jak IntelliJ IDEA, oraz wsparcie przez Google sprawiają, że jest to język przyszłościowy, szczególnie w kontekście tworzenia aplikacji mobilnych.

3.2.2. Jetpack Compose

Jetpack Compose to nowoczesny framework opracowany przez Google, służący do tworzenia interfejsów użytkownika (*UI*) w aplikacjach na system Android. Jest to narzędzie oparte na deklaratywnym podejściu do budowy interfejsów, które pozwala tworzyć dynamiczne i złożone układy w sposób bardziej intuicyjny i efektywny niż tradycyjne podejście z wykorzystaniem XML.
Jednym z kluczowych wymagań do pracy z Jetpack Compose jest użycie języka Kotlin, który dzięki swojej nowoczesnej składni i wsparciu dla programowania funkcyjnego doskonale współgra z deklaratywnym stylem tworzenia interfejsów. Jetpack Compose integruje się bezpośrednio z istniejącym ekosystemem Androida, co umożliwia stopniową migrację aplikacji opartych na tradycyjnych układach XML. Do najważniejszych cech Jetpack Compose należą:

- **Deklaratywne podejście** – interfejsy są definiowane jako funkcje w języku Kotlin, co pozwala na proste i czytelne opisywanie widoków oraz ich zależności.

- **Reaktywność** – framework automatycznie odświeża interfejs użytkownika w odpowiedzi na zmiany w stanie aplikacji, co eliminuje potrzebę ręcznego zarządzania aktualizacjami widoków.

- **Komponenty wielokrotnego użytku** – Jetpack Compose oferuje bibliotekę gotowych komponentów, takich jak przyciski, pola tekstowe czy listy, które można łatwo dostosowywać do potrzeb projektu.

- **Integracja z Jetpack** – Compose współpracuje z innymi bibliotekami wchodzącymi w skład ekosystemu Jetpack, takimi jak *Navigation*, *LiveData* czy *ViewModel*, co ułatwia zarządzanie stanem aplikacji i nawigacją.

- **Obsługa wieloplatformowa** – Jetpack Compose jest rozwijany także w wersji *Compose Multiplatform*, co umożliwia tworzenie interfejsów użytkownika nie

tylko dla Androida, ale również dla innych platform, takich jak *desktop* czy przeglądarki internetowe.

- **Szybka iteracja** – dzięki funkcji *hot reload* programiści mogą natychmiast zobaczyć efekty wprowadzonych zmian w kodzie interfejsu, co znacznie przyspiesza proces tworzenia aplikacji.

Jetpack Compose rozwiązuje wiele problemów związanych z tradycyjnym tworzeniem interfejsów w Androidzie,takich jak skomplikowane zarządzanie układami w XML czy konieczność ręcznego obsługiwania zmian w widokach. Umożliwia budowanie nowoczesnych i estetycznych interfejsów przy jednoczesnym zmniejszeniu ilości kodu oraz uproszczeniu procesu tworzenia aplikacji.
W porównaniu z tradycyjnym podejściem opartym na Javie i XML, Jetpack Compose w połączeniu z językiem Kotlin oferuje znacznie większą produktywność i przejrzystość kodu. W Kotlinie deklaratywne definiowanie interfejsów pozwala uniknąć zbędnego kodu „klejącego" (tzw. *boilerplate code*), który był często niezbędny w Javie do wiązania widoków XML z logiką aplikacji. Kotlin wprowadza funkcje rozszerzające, lambdy i programowanie funkcyjne, co czyni kod bardziej zwięzłym i łatwiejszym w utrzymaniu. Jetpack Compose całkowicie eliminuje potrzebę używania XML, upraszczając proces tworzenia złożonych interfejsów i redukując możliwość wystąpienia błędów związanych z niezgodnością kodu XML i logiki w Javie. Dzięki tym cechom Jetpack Compose i Kotlin razem redefiniują sposób budowania aplikacji na Androida, kładąc nacisk na nowoczesność, elastyczność i szybkość tworzenia oprogramowania.

3.2.3. DataStore Preferences

DataStore Preferences to nowoczesna biblioteka Android Jetpack opracowana przez Google, służąca do przechowywania niewielkich ilości danych w sposób wydajny, bezpieczny i zgodny z zasadami programowania asynchronicznego. Stanowi następcę starszego mechanizmu *SharedPreferences*, eliminując wiele jego ograniczeń, takich jak synchroniczność i ryzyko blokowania głównego wątku (*UI thread*).
Biblioteka DataStore oferuje dwa tryby przechowywania danych:

- **Preferences DataStore** – przeznaczone do przechowywania prostych danych w formacie klucz-wartość, podobnie jak *SharedPreferences*.

- **Proto DataStore** – pozwalające na przechowywanie bardziej złożonych struk-

tur danych przy użyciu protokołu Protobuf (zalecane w przypadku bardziej rozbudowanych wymagań dotyczących danych).

Główne cechy DataStore Preferences to:

- **Asynchroniczność** – operacje zapisu i odczytu danych są wykonywane asynchronicznie przy użyciu Kotlin Coroutines, co zapobiega blokowaniu wątku głównego i poprawia wydajność aplikacji.
- **Reaktywność** – dane są dostępne jako strumienie *Flow*, dzięki czemu można je obserwować i automatycznie reagować na ich zmiany w czasie rzeczywistym.
- **Bezpieczeństwo typów** – dzięki jawnemu definiowaniu typów danych, DataStore zmniejsza ryzyko błędów związanych z niezgodnością typów, co było częstym problemem w *SharedPreferences*.
- **Prosta implementacja** – DataStore Preferences wykorzystuje intuicyjne API i jest w pełni zintegrowane z ekosystemem Android Jetpack, co umożliwia łatwą integrację z innymi komponentami, takimi jak ViewModel czy LiveData.

Przykłady użycia DataStore Preferences

Zapis danych: Dane są przechowywane w pliku konfiguracyjnym w sposób asynchroniczny i bezpieczny:

```
suspend fun savePreference(key: Preferences.Key<String>, value:
    String) {
    dataStore.edit { preferences ->
        preferences[key] = value
    }
}
```

Odczyt danych: Dane są dostępne jako strumień *Flow*, co umożliwia ich obserwację w czasie rzeczywistym:

```
val preferenceFlow: Flow<String?> = dataStore.data.map {
    preferences ->
    preferences[PreferencesKeys.SOME_KEY]
}
```

Zastosowania DataStore Preferences

DataStore Preferences jest idealnym rozwiązaniem do przechowywania niewielkich ustawień aplikacji, takich jak:

- preferencje użytkownika, np. ustawienia motywu (ciemny/jasny),

- preferencje językowe aplikacji,

- zapamiętywanie stanu aplikacji, np. ostatnio wybranej zakładki.

Porównanie z SharedPreferences

DataStore Preferences oferuje wiele usprawnień w porównaniu z *SharedPreferences*:

- Operacje są wykonywane asynchronicznie, co zapobiega przycinaniu interfejsu użytkownika podczas zapisu lub odczytu danych.

- Dzięki wykorzystaniu strumieni *Flow*, DataStore umożliwia automatyczne reagowanie na zmiany danych w czasie rzeczywistym, podczas gdy *SharedPreferences* wymagało ręcznej aktualizacji widoków.

- DataStore Preferences wspiera nowoczesne podejście do programowania z użyciem Kotlin Coroutines, co czyni go bardziej zgodnym z nowoczesnymi standardami tworzenia aplikacji na Androida.

DataStore Preferences to nowoczesne i wydajne narzędzie do przechowywania danych w aplikacjach Android,które eliminuje ograniczenia starszych rozwiązań, takich jak *SharedPreferences*. Dzięki asynchroniczności, reaktywności i bezpieczeństwu typów, DataStore Preferences pozwala tworzyć bardziej wydajne i stabilne aplikacje, które lepiej spełniają wymagania współczesnych użytkowników.

3.2.4. Biblioteka Retrofit

Retrofit to nowoczesna biblioteka open-source opracowana przez firmę Square, która służy do komunikacji z serwerami RESTful poprzez wykonywanie żądań HTTP. Retrofit ułatwia integrację aplikacji Android z zewnętrznymi API, zapewniając prosty i wydajny sposób na wykonywanie zapytań sieciowych oraz przetwarzanie ich odpowiedzi.

Główną zaletą biblioteki Retrofit jest jego modułowość i łatwość w użyciu. Biblioteka automatyzuje wiele aspektów komunikacji sieciowej, takich jak serializacja i deserializacja danych, obsługa różnych typów żądań HTTP (`GET`, `POST`, `PUT`, `DELETE`) oraz zarządzanie nagłówkami i parametrami zapytań.

Cechy biblioteki Retrofit

- **Automatyczna konwersja danych** – Retrofit obsługuje różne formaty danych, takie jak JSON czy XML, przy użyciu adapterów serializujących (np.

Gson, Moshi). Konwersja danych wejściowych i wyjściowych jest w pełni zautomatyzowana.

- **Obsługa interfejsów** – zapytania sieciowe są definiowane w postaci interfejsów, co pozwala na przejrzysty i modularny kod.
- **Łatwe konfigurowanie** – Retrofit umożliwia łatwe zarządzanie podstawowymi ustawieniami żądań, takimi jak adres bazowy serwera, nagłówki czy parametry.
- **Obsługa asynchroniczności** – Retrofit wspiera natywną obsługę Kotlin Coroutines i `RxJava`, co umożliwia asynchroniczne wykonywanie zapytań bez blokowania głównego wątku.
- **Rozszerzalność** – biblioteka pozwala na łatwe dodawanie własnych adapterów do obsługi niestandardowych typów danych.

Przykłady użycia

Definiowanie interfejsu API:

```
import retrofit2.http.GET
import retrofit2.http.Path

interface ApiService {
  @GET("users/{id}")
  suspend fun getUser(@Path("id") userId: Int): User
}
```

Tworzenie instancji Retrofit:

```
import retrofit2.Retrofit
import retrofit2.converter.gson.GsonConverterFactory

val retrofit = Retrofit.Builder()
  .baseUrl("https://api.example.com/")
  .addConverterFactory(GsonConverterFactory.create())
  .build()

val apiService: ApiService = retrofit.create(ApiService::class.
   java)
```

Wykonywanie zapytań:

```
import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.launch

CoroutineScope(Dispatchers.IO).launch {
  try {
    val user = apiService.getUser(1)
    // Przetwarzanie odpowiedzi
```

```
9       println("User: ${user.name}")
10    } catch (e: Exception) {
11      e.printStackTrace()
12    }
13  }
```

Zastosowania biblioteki Retrofit

Retrofit jest szeroko stosowany w aplikacjach Android do:

- komunikacji z API RESTful, np. pobierania danych użytkownika, postów czy obrazów,
- przesyłania danych w aplikacjach korzystających z zapytań `POST` lub `PUT`,
- integracji aplikacji z zewnętrznymi serwisami, takimi jak usługi pogodowe, mapy czy systemy płatności.

Zalety biblioteki Retrofit w porównaniu z innymi rozwiązaniami

W porównaniu z tradycyjnym mechanizmem `HttpURLConnection` czy biblioteką `Volley`, Retrofit oferuje większą modularność, automatyzację przetwarzania danych oraz wsparcie dla nowoczesnych narzędzi, takich jak Kotlin Coroutines. Dzięki temu programiści mogą znacznie szybciej i łatwiej integrować aplikacje Android z zewnętrznymi usługami sieciowymi, tworząc przy tym kod bardziej przejrzysty i łatwy w utrzymaniu.

4. Budowa projektu

Projekt składa się z dwóch głównych części: części serwera na Raspberry Pi oraz części aplikacji mobilnej. Aplikacja mobilna jest niejako *frontendem* dla wykonywanych operacji, podczas gdy faktyczna logika oraz przetwarzanie danych realizowane są przez program uruchomiony na Raspberry Pi. Komunikacja pomiędzy aplikacją mobilną a Raspberry Pi odbywa się za pomocą zapytań API wysyłanych do serwera działającego na Raspberry Pi.

Zdecydowano się na taki model działania całego projektu, w celu odciążenia aplikacji mobilnej zainstalowanej na smartfonie użytkownika od ciągłych zapytań do serwera. Bezpośrednie przetwarzanie danych przez aplikację mobilną mogłoby negatywnie wpłynąć na wydajność urządzenia oraz zużycie jego akumulatora. Raspberry Pi, będące zwykle podłączone do stałego źródła zasilania, stanowi bardziej odpowiednią jednostkę do obsługi intensywnych operacji przetwarzania oraz komunikacji z urządzeniami w ekosystemie Philips Hue. Serwer uruchomiony na Raspberry Pi pełni kilka kluczowych funkcji:

- Obsługuje komunikację z mostkiem Philips Hue, realizując operacje takie jak zmiana koloru, jasności czy stanu oświetlenia.
- Przetwarza dane z czujnika BME280, takie jak temperatura, które mogą być wykorzystywane do wyświetlania kolorów.
- Udostępnia odpowiednio zaprojektowane punkty końcowe API (*endpoints*), które umożliwiają aplikacji mobilnej przesyłanie zapytań w celu realizacji wybranych operacji.
- Zarządza zadaniami w tle, takimi jak dynamiczna zmiana koloru światła w zależności od temperatury, co wymaga ciągłego monitorowania danych.

Aplikacja mobilna pełni natomiast rolę interfejsu użytkownika, umożliwiając intuicyjne sterowanie żarówkami Philips Hue. Dzięki zastosowanemu modelowi architektury:

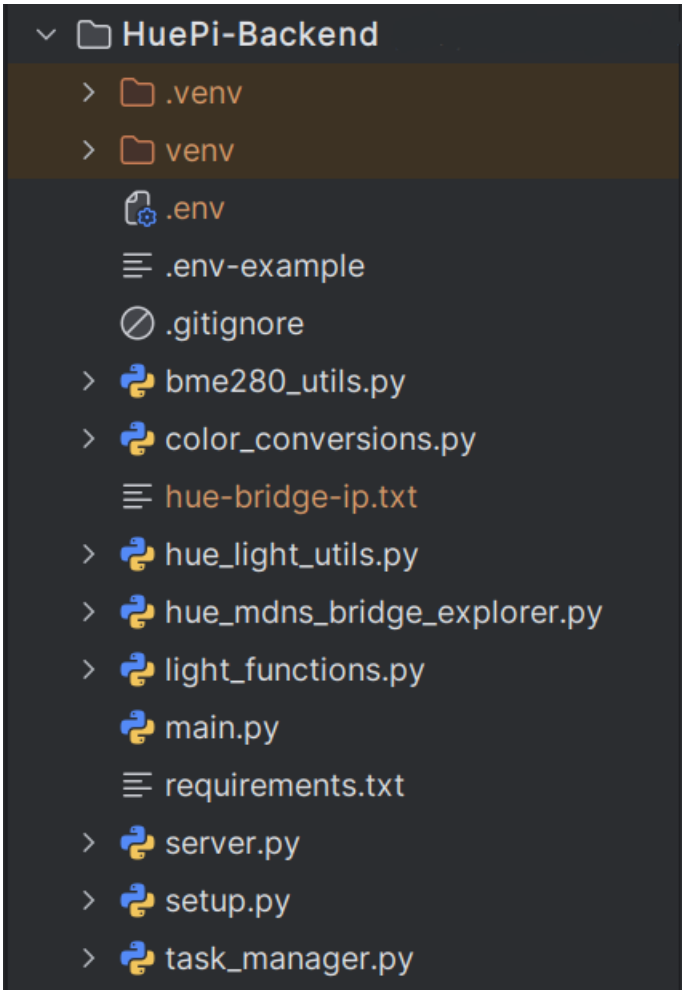
- użytkownik może wykonywać operacje bez konieczności ciągłego przetwarzania danych na urządzeniu mobilnym,
- zmniejszono zapotrzebowanie na zasoby smartfona, takie jak procesor czy bateria,

- zapewniono większą skalowalność systemu, ponieważ serwer na Raspberry Pi może być łatwo rozbudowany o dodatkowe funkcjonalności bez potrzeby modyfikowania aplikacji mobilnej.

Wybrany model architektury zapewnia elastyczność oraz niezawodność całego systemu, przy jednoczesnym wykorzystaniu zalet urządzeń takich jak Raspberry Pi, które doskonale nadają się do pracy jako serwer w niewielkich projektach IoT (*Internet of Things*). Dzięki temu możliwe było stworzenie rozwiązania, które jest zarówno efektywne, jak i przyjazne dla użytkownika końcowego.

4.1. Serwer Raspberry Pi

Struktura plików odpowiadająca za serwer na Raspberry Pi wygląda tak jak na rysunku poniżej:



Rysunek 4.1: Struktura plików serwera Raspberry Pi

4.1.1. .env-example/.env

```
hue-application-key=abcdefg123456
server-api-key=xyz123
```

Pliki `.env` oraz `.env-example` są istotnym elementem konfiguracji serwera uruchamianego na Raspberry Pi, zapewniającym bezpieczeństwo oraz łatwość w zarządzaniu kluczami i danymi dostępowymi. Plik `.env-example` to przykładowy plik wzorcowy, który definiuje, jakie dane konfiguracyjne powinny znaleźć się w pliku `.env`. Zawiera klucze środowiskowe oraz przykładowe wartości, służące jako instrukcja dla użytkownika, który konfiguruje serwer. Plik `.env` to właściwy plik konfiguracyjny, używany podczas uruchamiania serwera. Zawiera rzeczywiste wartości kluczy, które są wymagane do działania serwera. W projekcie pełni następujące funkcje:

- `hue-application-key` – przechowuje klucz aplikacji Philips Hue, który jest niezbędny do wykonywania zapytań do API mostka Philips Hue. Klucz ten pozwala na autoryzację oraz wykonywanie jakichkolwiek poleceń.
- `server-api-key` – klucz autoryzacyjny wymagany przez serwer FastAPI uruchomiony na Raspberry Pi. Każde żądanie do serwera Raspberry Pi musi zawierać ten klucz w nagłówku, aby zostało zaakceptowane. Mechanizm ten zabezpiecza serwer przed nieautoryzowanym dostępem.

Pliki `.env` i `.env-example` są kluczowymi elementami konfiguracji serwera w projekcie. Dzięki nim zarządzanie wrażliwymi danymi, takimi jak klucze API, jest łatwe, bezpieczne i zgodne z nowoczesnymi standardami tworzenia oprogramowania. Plik `.env-example` pełni rolę dokumentacji dla użytkowników, podczas gdy plik `.env` zawiera rzeczywiste wartości wymagane do uruchomienia serwera.

4.1.2. setup.py

```
    import hue_mdns_bridge_explorer

    hue_bridge_explorer = hue_mdns_bridge_explorer.
HueBridgeListener()

    def init():
        print("Looking for Philips Hue bridges...")
        hue_bridge_ip_addresses_list = hue_mdns_bridge_explorer.
get_hue_bridge_ips()
        if len(hue_bridge_ip_addresses_list) > 1:
            print("More than 1 Hue bridge found! Terminating..."
)
```

```
            quit(0)
        elif len(hue_bridge_ip_addresses_list) == 0:
            print("No Hue bridge found! Terminating...")
            quit(0)
        else:
            with open("hue-bridge-ip.txt", "w") as file:
                file.write(str(hue_bridge_ip_addresses_list[0]))
                print(f"Hue Bridge found! Its IP address set to:
    {str(hue_bridge_ip_addresses_list[0])}")
```

Plik `setup.py` pełni kluczową rolę w procesie inicjalizacji serwera na Raspberry Pi, umożliwiając automatyczne wykrycie mostka Philips Hue w sieci lokalnej i zapisanie jego adresu IP w pliku konfiguracyjnym. Jest to istotny krok w przygotowaniu środowiska pracy, ponieważ adres IP mostka jest wymagany do komunikacji z jego API.

Funkcjonalność pliku `setup.py`

- **Wykrywanie mostków Philips Hue:** Plik wykorzystuje skrypt `hue_mdns_bridge_explorer.py`, który przy pomocy protokołu mDNS przeszukuje sieć lokalną w celu odnalezienia mostków Philips Hue.

- **Zapis adresu IP:** Po wykryciu mostka jego adres IP jest zapisywany w pliku `hue-bridge-ip.txt`, który jest później używany przez inne skrypty, takie jak `server.py`, do komunikacji z mostkiem.

- **Walidacja wyników:** Plik sprawdza, ile mostków zostało wykrytych. Jeśli znaleziono więcej niż jeden mostek lub żadnego, proces zostaje zatrzymany, a użytkownik jest o tym informowany.

Działanie krok po kroku

- **Wykrywanie mostków:** Funkcja `init()` korzysta z metody `get_hue_bridge_ips()` zaimplementowanej w pliku `hue_mdns_bridge_explorer.py`, aby znaleźć dostępne mostki Hue w sieci lokalnej.

- **Sprawdzenie liczby mostków:** Jeśli nie znaleziono żadnych mostków lub wykryto ich więcej niż jeden, skrypt kończy działanie, wyświetlając stosowny komunikat.

- **Zapis adresu IP:** W przypadku wykrycia dokładnie jednego mostka jego adres IP jest zapisywany w pliku `hue-bridge-ip.txt`, aby mógł być później używany przez inne skrypty.

Plik `setup.py` automatyzuje proces inicjalizacji serwera, zapewniając poprawne skonfigurowanie środowiska pracy. Dzięki automatycznemu wykrywaniu mostka Hue izapisowi jego adresu IP, minimalizuje ryzyko błędów ręcznej konfiguracji oraz upraszcza proces uruchamiania projektu.

4.1.3. hue_mdns_bridge_explorer.py

```
from zeroconf import ServiceBrowser, Zeroconf
import socket
import time

class HueBridgeListener:
    def __init__(self):
        self.hue_bridge_ips = []

    def remove_service(self, zeroconf: Zeroconf, type_: str, name: str) -> None:
        pass

    def add_service(self, zeroconf: Zeroconf, type_: str, name: str) -> None:
        info = zeroconf.get_service_info(type_, name)
        if info:
            ip_ = socket.inet_ntoa(info.addresses[0])
            self.hue_bridge_ips.append(ip_)

    def update_service(self, zeroconf: Zeroconf, type_: str, name: str) -> None:
        pass

def find_hue_bridges():
    zeroconf = Zeroconf()
    _listener = HueBridgeListener()
    browser = ServiceBrowser(zeroconf, "_hue._tcp.local.", _listener)
    time.sleep(5)
    zeroconf.close()
    return _listener.hue_bridge_ips

def get_hue_bridge_ips():
    _bridges = find_hue_bridges()
    return _bridges

if __name__ == '__main__':
    listener = HueBridgeListener()
    bridges = get_hue_bridge_ips()
    for bridge in bridges:
        print(bridge)
```

Plik `hue_mdns_bridge_explorer.py` odpowiada za wyszukiwanie mostków Philips Hue w sieci lokalnej przy użyciu protokołu mDNS (*Multicast DNS*). Jest kluczowym komponentem systemu, ponieważ umożliwia dynamiczne wykrywanie urządzeń,

eliminując potrzebę ręcznego wprowadzania adresu IP mostka.

Funkcjonalność pliku `hue_mdns_bridge_explorer.py`

- **Przeszukiwanie sieci lokalnej w poszukiwaniu mostków Hue:** Wykorzystuje protokół mDNS do odnajdywania urządzeń, które ogłaszają swoje usługi pod nazwą `_hue._tcp.local.`

- **Pobieranie adresów IP wykrytych mostków:** Gdy mostek Hue zostanie wykryty, jego adres IP jest dodawany do listy dostępnych urządzeń.

- **Obsługa dynamicznych zmian w sieci:** Skrypt nasłuchuje zmiany w dostępnych usługach, dzięki czemu może na bieżąco aktualizować listę wykrytych urządzeń.

Działanie krok po kroku

- **Inicjalizacja klasy `HueBridgeListener`:** Klasa ta jest odpowiedzialna za nasłuchiwanie usług w sieci lokalnej i zapisywanie adresów IP wykrytych mostków Hue.

- **Obsługa zdarzeń:** Metody `add_service()`, `remove_service()` oraz `update_service()` reagują na zmiany w dostępnych usługach sieciowych.

- **Wyszukiwanie mostków:** Funkcja `find_hue_bridges()` używa obiektu `Zeroconf` do przeszukiwania sieci pod kątem usług `_hue._tcp.local.` i pobiera ich adresy IP.

- **Udostępnienie listy mostków:** Funkcja `get_hue_bridge_ips()` zwraca listę wykrytych adresów IP mostków Philips Hue.

- **Uruchomienie skryptu:** Jeśli plik jest uruchamiany jako główny moduł, skrypt wypisuje na ekranie znalezione mostki.

Plik `hue_mdns_bridge_explorer.py` jest wykorzystywany przez inne komponenty systemu, takie jak `setup.py`, do dynamicznego wykrywania mostka Hue. Dzięki zastosowaniu protokołu mDNS użytkownik nie musi ręcznie konfigurować adresu IP urządzenia, co znacząco ułatwia proces instalacji i konfiguracji.

4.1.4. color_conversions.py

```
from rgbxy import Converter
from rgbxy import GamutC
from colorsys import hsv_to_rgb

converter = Converter(GamutC)


def hsv2rgb(h, s, v):
    """Converts hsv(0-1, 0-1, 0-1) to rgb in 0-255 range"""
    return tuple(round(i * 255) for i in hsv_to_rgb(h, s, v))


def hsv2xy(h, s, v):
    """Converts hsv (0-1, 0-1, 0-1 ranges) to Philips' Hue xy
   values (0-1, 0-1)"""
    _v = v
    if v <= 0.001:
        _v = 0.01
    r, g, b = hsv2rgb(h, s, _v)
    x, y = converter.rgb_to_xy(r, g, b)
    return x, y


def rgb2xy(r, g, b):
    """Converts rgb(0-255, 0-255, 0-255) to Philips' Hue xy
   values"""
    if r == 0 and g == 0 and b == 0:
        return 0, 0
    x, y = converter.rgb_to_xy(r, g, b)
    return x, y

def xy2hex(x, y):
    """Converts Philips' Hue xy to hex"""
    hex = converter.xy_to_hex(x, y)
    return hex

if __name__ == '__main__':
    print(rgb2xy(255, 0, 255))
```

Plik `color_conversions.py` zawiera zestaw funkcji umożliwiających konwersję kolorów pomiędzy różnymi modelami, takimi jak HSV, RGB i XY. Jest to niezbędne w kontekście integracji z systemem Philips Hue, ponieważ mostek Hue operuje na współrzędnych barwowych `x` i `y`, zamiast standardowych wartości RGB czy HSV. Do tego różne modele żarówek Philips Hue posiadają różne tzw. *gamuty* co jeszcze bardziej komplikuje konwersję kolorów. Do pomocy przy konwersji została użyta bibliteka **rgbxy**. Główne funkcje w tym pliku odpowiadają za konwersję kolorów z modelu HSV na RGB, a następnie na wartości `x, y`, które mogą być przesłane do API Philips Hue. Dodatkowo umożliwiają przekształcenie barw zapisanych w formacie RGB bezpośrednio do

współrzędnych `xy`, co pozwala na precyzyjne odwzorowanie kolorów w inteligentnym oświetleniu. Oprócz tego dostępna jest funkcja zamieniająca wartości `x, y` na reprezentację heksadecymalną, co może być przydatne do wizualizacji aktualnych ustawień oświetlenia. Plik ten zapewnia kompatybilność między różnymi modelami kolorów i eliminuje konieczność ręcznego przeliczania wartości, co znacząco ułatwia implementację funkcji zmiany barw w systemie sterowania oświetleniem.

4.1.5. bme280_utils.py

```
import bme280
import smbus2

port = 1
address = 0x76
bus = smbus2.SMBus(port)

bme280.load_calibration_params(bus, address)


def read_temperature():
    data = bme280.sample(bus, address)
    return round(data.temperature, 3)


def read_pressure():
    data = bme280.sample(bus, address)
    return round(data.pressure, 3)


def read_humidity():
    data = bme280.sample(bus, address)
    return round(data.humidity, 3)
```

Plik `bme280_utils.py` zawiera funkcje umożliwiające odczyt danych z czujnika BME280, który mierzy temperaturę, ciśnienie atmosferyczne oraz wilgotność. Komunikacja z czujnikiem odbywa się za pośrednictwem magistrali I²C przy użyciu biblioteki `bme280`. Skrypt inicjalizuje czujnik, ładuje parametry kalibracyjne, a następnie dostarcza trzy główne funkcje: `read_temperature()`, `read_pressure()` oraz `read_humidity()`, które zwracają aktualne wartości pomiarowe. Dane te mogą być wykorzystywane np. do dynamicznej zmiany ustawień oświetlenia w zależności od temperatury lub do monitorowania warunków środowiskowych. Plik ten pełni kluczową rolę w integracji systemu z rzeczywistymi danymi atmosferycznymi, pozwalając na ich łatwe pobieranie i dalsze przetwarzanie przez inne moduły aplikacji.

4.1.6. hue_light_utils.py

```python
import requests as rq
import json
import urllib3
from color_conversions import xy2hex
urllib3.disable_warnings(urllib3.exceptions.
    InsecureRequestWarning)


def check_response(response: rq.Response):
    if (response.status_code != 200) and (response.status_code
    != 207):
        print(f'There is something wrong with the Philips Hue
    API call! Status code: {response.status_code}')


def powered_on(header, light_id, bridge_ip):
    light_url = f"https://{bridge_ip}/clip/v2/resource/light/{
    light_id}"
    r = rq.get(url=light_url, headers=header, verify=False)
    check_response(r)
    response_json = r.json()
    light_on = response_json['data'][0]['on']['on']
    if light_on:
        return True
    else:
        return False


def switch_power(header, light_id, bridge_ip):
    light_url = f"https://{bridge_ip}/clip/v2/resource/light/{
    light_id}"
    payload_on = json.dumps({
        "on": {
            "on": True
        }
    })

    payload_off = json.dumps({
        "on": {
            "on": False
        }
    })
    if powered_on(header, light_id, bridge_ip):
        r = rq.put(url=light_url, headers=header, data=
    payload_off, verify=False)
        check_response(r)
        return {"message": "OK"}
    else:
        r = rq.put(url=light_url, headers=header, data=
    payload_on, verify=False)
        check_response(r)
        return {"message": "OK"}


def turn_on(header, light_id, bridge_ip):
    if powered_on(header, light_id, bridge_ip):
```

```
50          return {"message": "Already turned on!"}
51      else:
52          light_url = f"https://{bridge_ip}/clip/v2/resource/light
    /{light_id}"
53          payload = json.dumps({
54              "on": {
55                  "on": True
56              }
57          })
58          r = rq.put(url=light_url, headers=header, data=payload,
    verify=False)
59          check_response(r)
60          return {"message": "OK"}
61
62
63  def turn_off(header, light_id, bridge_ip):
64      if not powered_on(header, light_id, bridge_ip):
65          return {"message": "Already turned off!"}
66      else:
67          light_url = f"https://{bridge_ip}/clip/v2/resource/light
    /{light_id}"
68          payload = json.dumps({
69              "on": {
70                  "on": False
71              }
72          })
73          r = rq.put(url=light_url, headers=header, data=payload,
    verify=False)
74          check_response(r)
75          return {"message": "OK"}
76
77
78  def change_brightness(header, light_id, level, bridge_ip):
79      level_int = (round(level, 2) * 100)
80
81      if level_int > 100:
82          level_int = 100
83      elif level_int < 0:
84          level_int = 0
85
86      light_url = f"https://{bridge_ip}/clip/v2/resource/light/{
    light_id}"
87      payload = json.dumps({
88          "dimming": {
89              "brightness": level_int
90          }
91      })
92      r = rq.put(url=light_url, headers=header, data=payload,
    verify=False)
93      check_response(r)
94      return {"message": "OK"}
95
96
97  def change_color(header, light_id, bridge_ip, x, y):
98      light_url = f"https://{bridge_ip}/clip/v2/resource/light/{
    light_id}"
```

```
    r = rq.put(url=light_url, headers=header,
                json={
                    "color": {
                        "xy": {
                            "x": x,
                            "y": y
                        }
                    }
                }, verify=False)
    check_response(r)
    return {"message": "OK"}


def get_full_lights(header, bridge_ip):
    base_url = f"https://{bridge_ip}/clip/v2/resource"
    devices_url = f"{base_url}/device"

    # First GET request to retrieve all light devices which returns only "rid" and "name"
    response = rq.get(url=devices_url, headers=header, verify=False)
    lights = []

    check_response(response)

    response_data = response.json()
    for device in response_data.get("data", []):
        for service in device.get("services", []):
            if service.get("rtype") == "light":
                light_rid = service["rid"]

                # Second GET request for detailed light information
                light_url = f"{base_url}/light/{light_rid}"
                light_response = rq.get(url=light_url, headers=header, verify=False)

                if (light_response.status_code != 200 and light_response.status_code != 207):
                    print("Error fetching light details:", light_response.status_code, light_response.text)
                    brightness = None
                    is_on: bool = None
                    color_hex = None

                light_data = light_response.json().get("data", [])[0]
                brightness = light_data.get("dimming", {}).get("brightness")
                is_on = light_data.get("on", {}).get("on")
                xy_color = light_data.get("color", {}).get("xy")
                color_hex = xy2hex(xy_color["x"], xy_color["y"]) if xy_color else None
                name = device["metadata"]["name"]

                light_info = {
```

```
                    "rid": light_rid,
                    "name": name,
                    "brightness": brightness,
                    "isOn": is_on,
                    "color": f"#{color_hex}",
                }
                lights.append(light_info)
    return lights


def get_light_details(header, bridge_ip, light_id):
    light_url = f"https://{bridge_ip}/clip/v2/resource/light/{light_id}"
    response = rq.get(url=light_url, headers=header, verify=False)

    print(response.status_code)
    if (response.status_code != 200 and response.status_code != 207):
        print("Error fetching light details:", response.status_code, response.text)
        return None

    light_data = response.json().get("data", [])[0]  # Get the first light data object

    brightness_float_100_range = light_data.get("dimming", {}).get("brightness")
    brightness_float_1_range = round((brightness_float_100_range/100), 2)
    name = light_data.get("metadata", {}).get("name")
    is_on = light_data.get("on", {}).get("on")
    xy_color = light_data.get("color", {}).get("xy")
    color_hex = xy2hex(xy_color["x"], xy_color["y"]) if xy_color else None


    light_info = {
        "rid": light_id,
        "name": name,
        "brightness": brightness_float_1_range,
        "isOn": is_on,
        "color": f"#{color_hex}",
    }

    return light_info
```

Plik `hue_light_utils.py` zawiera zestaw funkcji odpowiedzialnych za komunikację serwera z mostkiem Philips Hue poprzez jego API. Definiuje zapytania HTTP umożliwiające sterowanie inteligentnym oświetleniem, w tym włączanie i wyłączanie świateł, zmianę jasności, kolorów oraz pobieranie szczegółowych informacji o stanie poszczególnych źródeł światła. Wszystkie operacje są realizowane za pomocą żądań `GET` i `PUT`,

wysyłanych do odpowiednich endpointów mostka Philips Hue. Plik wykorzystuje bibliotekę `requests` do obsługi połączeń HTTP, a także umożliwia konwersję danych kolorystycznych do formatu `xy`, wymaganego przez API Hue. Dodatkowo zawiera funkcję sprawdzającą odpowiedzi serwera i informującą o ewentualnych błędach w komunikacji. Moduł ten stanowi kluczowy element systemu sterowania oświetleniem, pozwalając na łatwą integrację funkcji związanych z inteligentnym oświetleniem oraz zapewniając płynną komunikację między serwerem a urządzeniami Hue.

4.1.7. light_functions.py

```
def translate_temperature_to_hsv_color(input_temp, temp_min,
   temp_max, hsv_color_min, hsv_color_max):
if input_temp <= temp_min:
    return hsv_color_min
if input_temp >= temp_max:
    return hsv_color_max

# Determine the range of the input temperatures
temp_span = temp_max - temp_min

# Scale the temperature into a 0-1 range
value_scaled = float(input_temp - temp_min) / float(temp_span)

# Handle the circular nature of the hue
if hsv_color_min <= hsv_color_max:
    # Normal case, no wrapping needed
    hue_color_span = hsv_color_max - hsv_color_min
    return hsv_color_min + (value_scaled * hue_color_span)
else:
    # Wrapping case
    hue_color_span = (1 - hsv_color_min) + hsv_color_max
    hue = hsv_color_min + (value_scaled * hue_color_span)
    if hue > 1:  # Wrap around the circle
        hue -= 1
    return hue
```

Plik `light_functions.py` zawiera funkcję umożliwiającą dynamiczne przekształcanie temperatury otoczenia na wartość koloru w przestrzeni HSV. Funkcja `translate_temperature_to_hsv_color()` przyjmuje wartości minimalnej i maksymalnej temperatury oraz odpowiadające im kolory w modelu HSV. Na tej podstawie dokonuje interpolacji, aby dla danej temperatury zwrócić odpowiednią wartość koloru, uwzględniając cykliczną naturę skali barw HSV. Moduł ten umożliwia tworzenie dynamicznych efektów świetlnych opartych na rzeczywistych danych pomiarowych, co pozwala na inteligentne dostosowanie oświetlenia do aktualnych warunków otoczenia.

4.1.8. task_manager.py

```
class TaskManager:
def __init__(self):
    self.tasks = {}

def start_task(self, light_id: str, task: callable, *args, **
   kwargs):
    if light_id in self.tasks:
        raise ValueError(f"A task is already running for
   light_id {light_id}.")
    self.tasks[light_id] = True
    task(*args, **kwargs)

def stop_task(self, light_id: str):
    if light_id in self.tasks:
        self.tasks[light_id] = False
        del self.tasks[light_id]

def is_task_running(self, light_id: str) -> bool:
    return self.tasks.get(light_id, False)


task_manager = TaskManager()
```

Plik `task_manager.py` zawiera implementację klasy `TaskManager`, która odpowiada za zarządzanie zadaniami wykonywanymi w tle. Klasa ta umożliwia uruchamianie, zatrzymywanie oraz monitorowanie zadań związanych ze sterowaniem oświetleniem Philips Hue. `TaskManager` przechowuje informacje o aktualnie wykonywanych procesach dla poszczególnych świateł, zapobiegając ich jednoczesnemu uruchamianiu. Jest to szczególnie istotne w kontekście dynamicznych funkcji, takich jak automatyczna zmiana koloru w zależności od temperatury. Klasa ta jest wykorzystywana w pliku `server.py`, gdzie pozwala na kontrolowanie działania długotrwałych procesów związanych z oświetleniem. Dzięki niej możliwe jest bezpieczne i kontrolowane zarządzanie stanem świateł, bez ryzyka kolizji między operacjami.

4.1.9. server.py

Plik `server.py` stanowi główną część serwera aplikacji, który został zbudowany przy użyciu frameworka FastAPI. Serwer ten obsługuje żądania HTTP i umożliwia komunikację między klientem a mostkiem Philips Hue, umożliwiając sterowanie inteligentnym oświetleniem. Plik integruje różne moduły, w tym `hue_light_utils.py` do wysyłania żądań do mostka Hue, `bme280_utils.py` do pobierania danych z czujnika BME280 oraz `task_manager.py` do zarządzania długotrwałymi procesami. Serwer implementuje mechanizm uwierzytelniania za pomocą klucza API, co zabezpiecza dostęp do funkcji sterujących oświetleniem. Obsługuje różne operacje, takie jak włączanie i wy-

łączanie świateł, zmiana ich jasności, kolorów oraz dynamiczne dostosowywanie barwy światła na podstawie temperatury.

Przykładowa funkcja: `turn_on()`

Jednym z endpointów dostępnych w serwerze jest funkcja `turn_on()`, która odpowiada za włączanie światła o określonym identyfikatorze.

```
@app.get("/turn-on/{light_id}")
async def turn_on(light_id: str, api_key: str = Security(
    get_api_key)):
     _header = {"hue-application-key": hue_api_key}
     response = hue_light_utils.turn_on(header=_header, light_id=
    light_id, bridge_ip=hue_bridge_ip_address)
     if response.get("message") == "OK":
         return {"message": "Turned on!"}
     elif response.get("message") == "Already turned on!":
         return {"message": "Already turned on!"}
     else:
         return {"message": "Something's wrong!"}, 500
```

Funkcja ta przyjmuje identyfikator światła jako parametr ścieżki oraz sprawdza poprawność klucza API. Następnie tworzy nagłówek uwierzytelniający, który jest wymagany przez mostek Hue, i wywołuje metodę `turn_on()` zdefiniowaną w `hue_light_utils.py`. Na podstawie odpowiedzi API zwraca komunikat informujący o powodzeniu operacji lub ewentualnym błędzie.

Plik `server.py` pełni rolę centralnego elementu aplikacji, umożliwiającego komunikację między klientem a mostkiem Philips Hue. Dzięki zastosowaniu FastAPI i podziałowi na moduły zapewnia wydajną obsługę żądań oraz elastyczność w zarządzaniu inteligentnym oświetleniem.

4.1.10. main.py

```
import subprocess
import setup
import hue_mdns_bridge_explorer

if __name__ == "__main__":
     hue_bridge_explorer = hue_mdns_bridge_explorer.
    HueBridgeListener()
     setup.init()

     command = [
         "uvicorn",
         "server:app",
         "--host", "0.0.0.0",
         "--port", "8000",
     ]

     print(f"Running command: \"{' '.join(command)}\"")
```

```
17
18      subprocess.run(command, check=True)
```

Plik `main.py` pełni rolę punktu wejściowego do uruchomienia serwera aplikacji. Jego zadaniem jest przygotowanie środowiska, w tym inicjalizacja konfiguracji mostka Philips Hue, a następnie uruchomienie serwera FastAPI. Na początku skrypt wywołuje funkcję `init()` z pliku `setup.py`, która automatycznie wykrywa adres IP mostka Philips Hue i zapisuje go w pliku konfiguracyjnym. Następnie definiowana jest komenda do uruchomienia serwera przy użyciu frameworka `Uvicorn`, który jest asynchronicznym serwerem HTTP przeznaczonym do obsługi aplikacji FastAPI. Serwer zostaje uruchomiony na wszystkich interfejsach sieciowych urządzenia (`0.0.0.0`) na porcie 8000. Po wykryciu mostka Hue i zapisaniu jego adresu IP, skrypt uruchamia serwer FastAPI, który nasłuchuje na porcie 8000. Dzięki temu serwer jest gotowy do obsługi zapytań związanych ze sterowaniem oświetleniem.

Plik `main.py` pełni funkcję inicjalizacyjną i uruchamiającą serwer aplikacji. Automatyzuje proces konfiguracji mostka Philips Hue oraz uruchamia serwer FastAPI, zapewniając gotowe środowisko do obsługi żądań HTTP związanych ze sterowaniem inteligentnym oświetleniem.

5. Podsumowanie i wnioski końcowe

$1 \div 3$ stron merytorycznie podsumowanie najważniejszych elementów pracy oraz wnioski wynikające z osiągniętego celu pracy. Proponowane zalecenia i modyfikacje oraz rozwiązania będące wynikiem realizowanej pracy.

Ostatni akapit podsumowania musi zawierać wykaz własnej pracy dyplomanta i zaczynać się od sformułowania: „Autor za własny wkład pracy uważa: . . . ”.

Załączniki

Według potrzeb zawarte i uporządkowane uzupełnienie pracy o dowolny materiał źródłowy (wydruk programu komputerowego, dokumentacja konstrukcyjno-technologiczna, konstrukcja modelu – makiety – urządzenia, instrukcja obsługi urządzenia lub stanowiska laboratoryjnego, zestawienie wyników pomiarów i obliczeń, informacyjne materiały katalogowe itp.).

Literatura

[1] https://www.statista.com/forecasts/887613/number-of-smart-homes-in-the-smart-home-market-in-the-world

[2] https://www.mouser.com/datasheet/2/783/BST-BME280-DS002-1509607.pdf

[3] https://kotlinlang.org/docs/faq.html

[4] https://developer.android.com/kotlin

[5] Jakubczyk T., Klette A.: Pomiary w akustyce. WNT, Warszawa 1997.

[6] Barski S.: Modele transmitancji. Elektronika praktyczna, nr 7/2011, str. 15-18.

[7] Czujnik S200. Dokumentacja techniczno-ruchowa. Lumel, Zielona Góra, 2001.

[8] Pawluk K.: Jak pisać teksty techniczne poprawnie, Wiadomości Elektrotechniczne, Nr 12, 2001, str. 513-515.

POLITECHNIKA RZESZOWSKA im. I. Łukasiewicza Rzeszów, 2025
Wydział Elektrotechniki i Informatyki

STRESZCZENIE PRACY DYPLOMOWEJ INŻYNIERSKIEJ

HUEPI - APLIKACJA MOBILNA DO OBSŁUGI ŻARÓWEK PHILIPS HUE

Autor: Jakub Kusal, nr albumu: EF-169571
Opiekun: dr inż. Mariusz Mączka
Słowa kluczowe: (max. 5 słów kluczowych w 2 wierszach, oddzielanych przecinkami)

Treść streszczenia po polsku

RZESZOW UNIVERSITY OF TECHNOLOGY Rzeszow, 2025
Faculty of Electrical and Computer Engineering

BSC THESIS ABSTRACT

HUEPI - MOBILE APP FOR MANAGING PHILIPS HUE BULBS

Author: Jakub Kusal, nr albumu: EF-169571, I s
Supervisor: Mariusz Mączka, BEng, PhD
Key words: (max. 5 słów kluczowych w 2 wierszach, oddzielanych przecinkami)

Treść streszczenia po angielsku