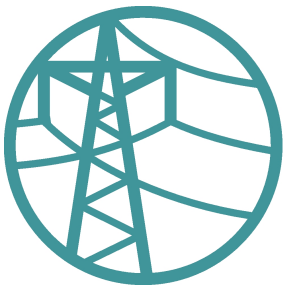


WYDZIAŁ
ELEKTROTECHNIKI
I INFORMATYKI
POLITECHNIKI RZESZOWSKIEJ

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]

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. [3]

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*. [4]

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[5],
- Zakres pomiaru wilgotności: od 0% do 100% RH (wilgotność względna), z dokładnością ±3% RH[5],
- Zakres pomiaru ciśnienia atmosferycznego: od 300 hPa do 1100 hPa, z dokładnością do ±1 hPa[5].

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. [6] 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 [7], 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. [8] 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.[9]
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.[10]

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
    println("User: ${user.name}")
  } catch (e: Exception) {
    e.printStackTrace()
  }
}
```

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:

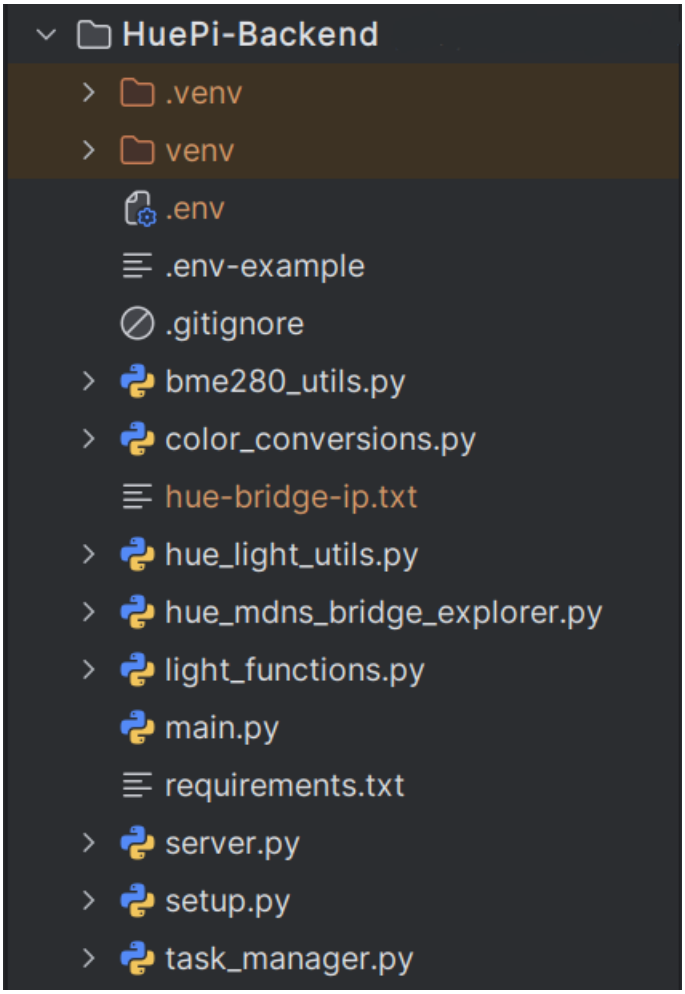
- użytkownik może wykonywać operacje bez konieczności ciągłego przetwarzania danych na urządzeniu mobilnym,
- zmniejszono zapotrzebowanie na zasoby smartfona, takie jak procesor czy bateria,

- 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

```

```
33 if __name__ == '__main__':
34     listener = HueBridgeListener()
35     bridges = get_hue_bridge_ips()
36     for bridge in bridges:
37         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 kom-

plikuje 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

```
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)
```

```
45          return {"message": "OK"}
46
47
48  def turn_on(header, light_id, bridge_ip):
49      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}"
99      r = rq.put(url=light_url, headers=header,
100                 json={
101                     "color": {
102                         "xy": {
103                             "x": x,
104                             "y": y
105                         }
106                     }
107                 }, verify=False)
108     check_response(r)
109     return {"message": "OK"}
110
111
112 def get_full_lights(header, bridge_ip):
113     base_url = f"https://{bridge_ip}/clip/v2/resource"
114     devices_url = f"{base_url}/device"
115
116     # First GET request to retrieve all light devices which
    returns only "rid" and "name"
117     response = rq.get(url=devices_url, headers=header, verify=
    False)
118     lights = []
119
120     check_response(response)
121
122     response_data = response.json()
123     for device in response_data.get("data", []):
124         for service in device.get("services", []):
125             if service.get("rtype") == "light":
126                 light_rid = service["rid"]
127
128                 # Second GET request for detailed light
    information
129                 light_url = f"{base_url}/light/{light_rid}"
130                 light_response = rq.get(url=light_url, headers=
    header, verify=False)
131
132                 if (light_response.status_code != 200 and
    light_response.status_code != 207):
133                     print("Error fetching light details:",
    light_response.status_code, light_response.text)
134                     brightness = None
135                     is_on: bool = None
136                     color_hex = None
137
138                 light_data = light_response.json().get("data",
    [])[0]
139                 brightness = light_data.get("dimming", {}).get("
    brightness")
140                 is_on = light_data.get("on", {}).get("on")
141                 xy_color = light_data.get("color", {}).get("xy")
```

```
142                 color_hex = xy2hex(xy_color["x"], xy_color["y"])
    if xy_color else None
143                 name = device["metadata"]["name"]
144
145                 light_info = {
146                     "rid": light_rid,
147                     "name": name,
148                     "brightness": brightness,
149                     "isOn": is_on,
150                     "color": f"#{color_hex}",
151                 }
152                 lights.append(light_info)
153     return lights
154
155
156 def get_light_details(header, bridge_ip, light_id):
157     light_url = f"https://{bridge_ip}/clip/v2/resource/light/{
    light_id}"
158     response = rq.get(url=light_url, headers=header, verify=
    False)
159
160     print(response.status_code)
161     if (response.status_code != 200 and response.status_code !=
    207):
162         print("Error fetching light details:", response.
    status_code, response.text)
163         return None
164
165     light_data = response.json().get("data", [])[0]  # Get the
    first light data object
166
167     brightness_float_100_range = light_data.get("dimming", {}).
    get("brightness")
168     brightness_float_1_range = round((brightness_float_100_range
    /100), 2)
169     name = light_data.get("metadata", {}).get("name")
170     is_on = light_data.get("on", {}).get("on")
171     xy_color = light_data.get("color", {}).get("xy")
172     color_hex = xy2hex(xy_color["x"], xy_color["y"]) if xy_color
    else None
173
174
175     light_info = {
176         "rid": light_id,
177         "name": name,
178         "brightness": brightness_float_1_range,
179         "isOn": is_on,
180         "color": f"#{color_hex}",
181     }
182
183     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 im-

plementuje mechanizm uwierzytelniania za pomocą klucza API, co zabezpiecza dostęp do funkcji sterujących oświetleniem. Obsługuje różne operacje, takie jak włączanie i wyłączanie 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

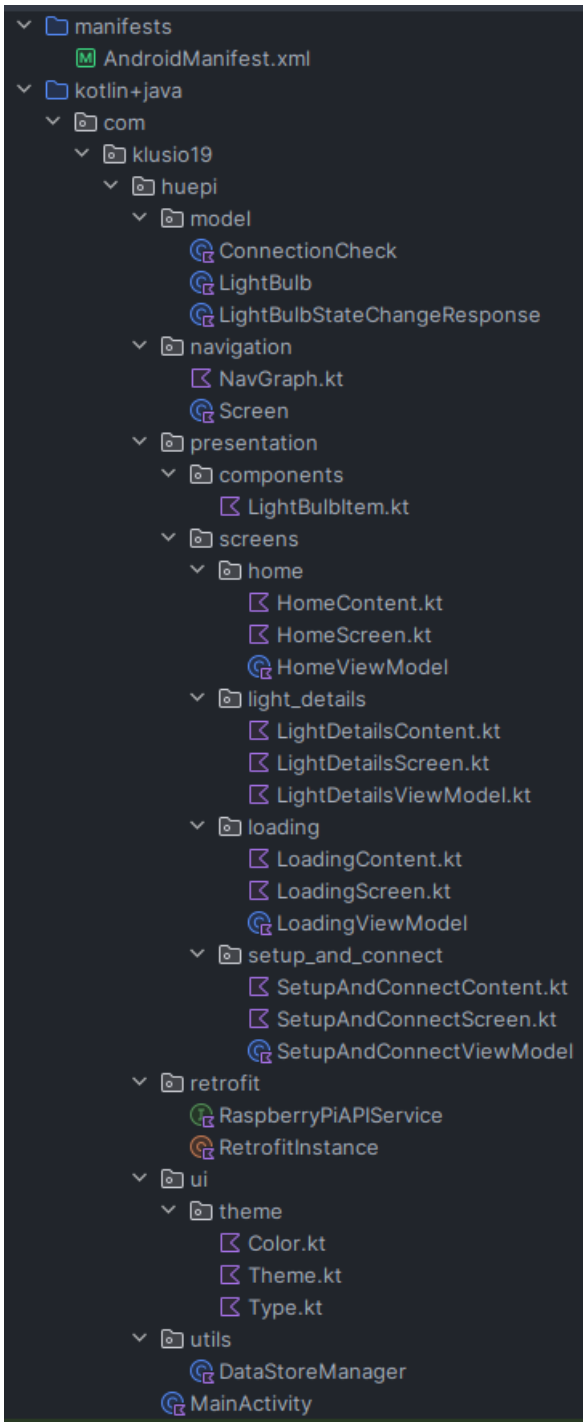
```
import subprocess
import setup
import hue_mdns_bridge_explorer

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)}\"")

    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.

4.2. Aplikacja mobilna

Poniższy rysunek przedstawia strukturę plików projektu.



Rysunek 4.2: Struktura plików projektu aplikacji mobilnej

Aplikacja mobilna HuePi została stworzona w języku Kotlin i wykorzystuje nowoczesne technologie oraz wzorce projektowe stosowane w aplikacjach na system An-

droid. Jej głównym celem jest umożliwienie użytkownikowi wygodnego sterowania inteligentnym oświetleniem Philips Hue poprzez komunikację z serwerem uruchomionym na Raspberry Pi.

Architektura aplikacji

Aplikacja została zaprojektowana zgodnie z wzorcem *MVVM (Model-View-ViewModel)*, który zapewnia lepszą separację logiki biznesowej od interfejsu użytkownika. Struktura kodu została podzielona na trzy główne warstwy:

- **Model** – zawiera definicje klas danych, takich jak reprezentacje żarówek (`LightBulb.kt`) oraz odpowiedzi serwera na zmiany stanu świateł.
- **View** – oparta na **Jetpack Compose**, odpowiada za warstwę interfejsu użytkownika, umożliwiając dynamiczne tworzenie widoków w sposób deklaratywny.
- **ViewModel** – pośredniczy między widokiem a warstwą modelu, zarządza stanem aplikacji i obsługuje logikę biznesową.

Interfejs użytkownika – Jetpack Compose

Do budowy interfejsu aplikacji wykorzystano **Jetpack Compose**, który umożliwia deklaratywne definiowanie komponentów UI oraz elastyczne zarządzanie stanem. Dzięki temu aplikacja jest bardziej responsywna, a kod jest czytelniejszy i łatwiejszy w utrzymaniu w porównaniu do tradycyjnego podejścia opartego na plikach XML.

Komunikacja z serwerem Raspberry Pi

Aplikacja wykorzystuje bibliotekę **Retrofit**, aby komunikować się z serwerem uruchomionym na Raspberry Pi, który zarządza mostkiem Philips Hue oraz czujnikiem BME280. Wysyłane żądania HTTP pozwalają na:

- włączanie i wyłączanie żarówek,
- zmianę jasności oraz koloru świateł,
- odczytywanie aktualnego stanu żarówek,

Obsługa nawigacji

Za zarządzanie przejściami między ekranami odpowiada moduł **Jetpack Compose Navigation**, który definiuje dostępne ścieżki oraz obsługuje przechodzenie między poszczególnymi ekranami aplikacji. Każdy ekran aplikacji został wydzielony do

osobnego modułu w katalogu `presentation/screens`, co ułatwia organizację kodu oraz jego rozbudowę.

Przechowywanie danych

Do zarządzania lokalnymi ustawieniami aplikacji wykorzystano bibliotekę **DataStore Preferences**, która zastępuje starszy mechanizm *SharedPreferences*. Pozwala to na bezpieczne i wydajne przechowywanie preferencji użytkownika, takich jak ostatnio wybrana żarówka czy stan połączenia z serwerem.

Struktura aplikacji

Aplikacja została podzielona na logiczne moduły, w tym:

- `model` – definiuje struktury danych wykorzystywane w aplikacji,
- `presentation` – zawiera komponenty UI oraz ekrany aplikacji,
- `retrofit` – obsługuje komunikację z serwerem przy użyciu Retrofit,
- `navigation` – odpowiada za system nawigacji pomiędzy ekranami,
- `utils` – zawiera klasy pomocnicze, takie jak `DataStoreManager` do obsługi ustawień użytkownika,
- `ui/theme` – definiuje motyw graficzny aplikacji (kolory, typografia, styl komponentów UI).

Aplikacja `HuePi` łączy nowoczesne technologie Androida, takie jak **Jetpack Compose**, **MVVM**, **Retrofit** i **DataStore**, aby zapewnić wygodne sterowanie inteligentnym oświetleniem. Dzięki modularnej budowie kod aplikacji jest czytelny, łatwy do rozwijania i zgodny z najlepszymi praktykami tworzenia aplikacji mobilnych.

Poniżej, w najbliższych podrozdziałach, zostaną omówione poszczególne pliki wchodzące w skład projektu.

4.2.1. MainActivity.kt

```kotlin
package com.klusio19.huepi

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.activity.enableEdgeToEdge
import androidx.core.splashscreen.SplashScreen.Companion.
    installSplashScreen
import androidx.core.view.WindowCompat
import androidx.navigation.compose.rememberNavController
import com.klusio19.huepi.navigation.Screen
import com.klusio19.huepi.navigation.SetupNavGraph
import com.klusio19.huepi.ui.theme.HuePiTheme

class MainActivity : ComponentActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        installSplashScreen()
        enableEdgeToEdge()
        WindowCompat.setDecorFitsSystemWindows(window, false)
        setContent {
            HuePiTheme {
                val navController = rememberNavController()
                    SetupNavGraph(
                        startDestination = Screen.Loading,
                        navController = navController,
                        context = application
                    )
            }
        }
    }
}
```

Plik `MainActivity.kt` stanowi główny punkt wejściowy aplikacji `HuePi`. Jest to jedyna aktywność (*Activity*) w aplikacji, co wynika z zastosowania **Jetpack Compose**, który eliminuje konieczność korzystania z wielu aktywności i zamiast tego pozwala na budowanie całego interfejsu użytkownika w sposób deklaratywny w obrębie jednej aktywności. Głównym zadaniem tej klasy jest inicjalizacja interfejsu użytkownika oraz uruchomienie systemu nawigacji pomiędzy ekranami aplikacji. Główne zadania `MainActivity.kt` to:

- **Inicjalizacja aplikacji** – ustawienie motywu i uruchomienie głównej zawartości aplikacji.
- **Zarządzanie nawigacją** – wykorzystanie Jetpack Navigation do obsługi przejść między ekranami.

- **Obsługa stanu aplikacji** – przekazywanie odpowiednich obiektów `ViewModel` do ekranów, które tego wymagają.

4.2.2. NavGraph.kt

Plik `NavGraph.kt` odpowiada za zarządzanie nawigacją w aplikacji `HuePi`. Wykorzystuje on bibliotekę **Jetpack Compose Navigation**, dostosowaną do deklaratywnego podejścia stosowanego w **Jetpack Compose**. Ponieważ cała aplikacja działa w jednej aktywności, wszystkie ekrany są obsługiwane dynamicznie w ramach systemu nawigacji.

Plik definiuje główny kontener nawigacyjny `NavHost`, który przechowuje listę dostępnych ekranów i zarządza ich zmianą. Struktura nawigacji opiera się na koncepcji **grafu nawigacyjnego**, w którym każda ścieżka (route) reprezentuje jeden z ekranów aplikacji. Dzięki temu podejściu przejścia pomiędzy ekranami realizowane są poprzez dynamiczne wywoływanie funkcji `navigate()` na obiekcie `NavController`, co pozwala na płynną zmianę widoków w aplikacji.

Zasada działania

Główna funkcja `SetupNavGraph()` inicjalizuje nawigację, ustawiając ekran startowy oraz rejestrując poszczególne ścieżki. Każdy ekran jest rejestrowany jako osobna funkcja rozszerzająca `NavGraphBuilder`, co pozwala na lepszą organizację kodu.

```
@Composable
fun SetupNavGraph(startDestination: Screen, navController:
   NavHostController, context: Application) {
    NavHost(navController = navController, startDestination =
   startDestination) {
        loadingRoute(context, navController)
        setupAndConnectRoute(navController)
        homeRoute(navController)
        lightRoute(context)
    }
}
```

W ramach `NavHost` rejestrowane są funkcje obsługujące poszczególne ekrany, takie jak:

- `loadingRoute()` – obsługuje ekran ładowania i przekierowuje użytkownika do odpowiedniego ekranu po zakończeniu procesu inicjalizacji.

- `setupAndConnectRoute()` – odpowiada za ekran konfiguracji połączenia z mostkiem Philips Hue i serwerem Raspberry Pi.

- `homeRoute()` – zarządza ekranem głównym, który wyświetla listę dostępnych żarówek.

- `lightRoute()` – odpowiada za ekran szczegółów dotyczących konkretnej żarówki, pozwalając użytkownikowi na zmianę jej stanu, jasności i koloru.

Obsługa nawigacji w ekranach

Każda z funkcji odpowiadających za poszczególne ekrany implementuje własną logikę nawigacji. Na przykład ekran konfiguracji (`setupAndConnectRoute()`) reaguje na zdarzenie poprawnej walidacji połączenia i po krótkim opóźnieniu przechodzi do ekranu głównego:

```
fun NavGraphBuilder.setupAndConnectRoute(navController:
    NavHostController) {
     composable<Screen.SetupAndConnect> {
         val viewModel: SetupAndConnectViewModel = viewModel()
         LaunchedEffect(Unit) {
             viewModel.navigationEvent.collect { shouldNavigate
    ->
                 if (shouldNavigate) {
                     delay(1000L)
                     navController.navigate(Screen.Home)
                 }
             }
         }
         SetupAndConnectScreen(...)
     }
}
```

W innych przypadkach nawigacja może być bardziej dynamiczna – np. ekran szczegółów żarówki (`lightRoute()`) pobiera argumenty przekazane w ścieżce nawigacyjnej (identyfikator żarówki), a następnie inicjalizuje odpowiedni `ViewModel`:

```
fun NavGraphBuilder.lightRoute(context: Application) {
     composable<Screen.LightDetails> { backStackEntry ->
         val args = backStackEntry.toRoute<Screen.LightDetails>()
         val viewModelFactory = LightDetailsViewModelFactory(
    context, args.rid)
         val viewModel: LightDetailsViewModel = viewModel(factory
     = viewModelFactory)
         LightDetailsScreen(...)
     }
}
```

Plik `NavGraph.kt` odpowiada za zarządzanie ruchem w aplikacji, definiując ścieżki między ekranami i obsługując przejścia. Dzięki zastosowaniu modularnej struktury oraz powiązania ekranów z odpowiednimi `ViewModel`, aplikacja może dynamicznie reagować na zmiany stanu i zapewniać użytkownikowi płynne doświadczenie. Podejście to

zwiększa elastyczność kodu i pozwala na łatwiejszą rozbudowę w przyszłości.

4.2.3. Screen.kt

```kotlin
package com.klusio19.huepi.navigation

import kotlinx.serialization.Serializable

@Serializable
sealed class Screen {
    @Serializable
    data object Loading : Screen()

    @Serializable
    data object SetupAndConnect : Screen()

    @Serializable
    data object Home : Screen()

    @Serializable
    data class LightDetails(val rid: String) : Screen()
}
```

Plik `Screen.kt` definiuje wszystkie ekrany dostępne w aplikacji `HuePi` i określa ich strukturę w systemie nawigacji. Wykorzystuje do tego *sealed class*, która grupuje ekrany jako zamknięty zbiór obiektów. Dzięki temu każdy ekran jest jednoznacznie identyfikowalny i łatwy do obsługi w grafie nawigacyjnym.

W pliku wyróżniono cztery ekrany:

- `Loading` – ekran ładowania aplikacji,
- `SetupAndConnect` – ekran konfiguracji połączenia,
- `Home` – ekran główny, który wyświetla listę żarówek,
- `LightDetails` – ekran szczegółów żarówki, który wymaga identyfikatora światła jako argumentu.

Dzięki takiemu podejściu nawigacja w aplikacji jest bardziej czytelna i bezpieczna, a przekazywanie danych między ekranami odbywa się w uporządkowany sposób.

4.2.4. RetrofitInstance.kt

```kotlin
package com.klusio19.huepi.retrofit

import com.squareup.moshi.Moshi
import com.squareup.moshi.kotlin.reflect.
    KotlinJsonAdapterFactory
import okhttp3.Interceptor
import okhttp3.OkHttpClient
import okhttp3.Request
import okhttp3.logging.HttpLoggingInterceptor
import retrofit2.Retrofit
import retrofit2.converter.moshi.MoshiConverterFactory

object RetrofitInstance {
    fun getClient(baseUrl: String, raspberryApiKey: String):
    RaspberryPiAPIService {
        val moshi = Moshi.Builder()
            .add(KotlinJsonAdapterFactory())
            .build()

        val logging = HttpLoggingInterceptor().setLevel(
    HttpLoggingInterceptor.Level.BODY)

        val okHttpClient = OkHttpClient.Builder()
            .addInterceptor(logging)
            .addInterceptor(
                Interceptor { chain ->
                    val request: Request = chain.request()
                        .newBuilder()
                        .header("api-key", raspberryApiKey)
                        .build()
                    chain.proceed(request)
                }
            ).build()

        val raspberryPiAPIService: RaspberryPiAPIService by lazy
    {
            Retrofit.Builder()
                .baseUrl(baseUrl)
                .addConverterFactory(MoshiConverterFactory.
    create(moshi))
                .client(okHttpClient)
                .build()
                .create(RaspberryPiAPIService::class.java)
        }

        return raspberryPiAPIService
    }
}
```

Plik `RetrofitInstance.kt` odpowiada za konfigurację klienta **Retrofit**, który umożliwia aplikacji `HuePi` komunikację z serwerem Raspberry Pi. Jest to kluczowy element warstwy sieciowej, ponieważ definiuje sposób, w jaki aplikacja wysyła żądania do API i przetwarza odpowiedzi.

Funkcjonalność

- Tworzy instancję `Retrofit` z podanym adresem URL i kluczem API serwera Raspberry Pi.
- Używa **Moshi** jako konwertera do obsługi formatu JSON.
- Dodaje **Interceptor**, który automatycznie dołącza nagłówek z kluczem API do każdego zapytania.
- Wykorzystuje **HttpLoggingInterceptor**, który ułatwia debugowanie żądań HTTP.

Plik `RetrofitInstance.kt` centralizuje konfigurację klienta HTTP, zapewniając jednolite miejsce inicjalizacji połączenia z serwerem. Dzięki temu inne moduły aplikacji mogą łatwo korzystać z API bez konieczności każdorazowego definiowania konfiguracji Retrofitu.

4.2.5. RaspberryPiAPIService.kt

Plik `RaspberryPiAPIService.kt` definiuje interfejs API, który umożliwia aplikacji `HuePi` komunikację z serwerem Raspberry Pi poprzez żądania HTTP. Retrofit wykorzystuje ten interfejs do automatycznego generowania kodu obsługującego połączenia sieciowe, co upraszcza wysyłanie zapytań oraz odbieranie odpowiedzi. Interfejs zawiera metody odpowiadające operacjom wykonywanym na inteligentnym oświetleniu Philips Hue, takim jak pobieranie listy żarówek, włączanie i wyłączanie świateł, zmiana jasności i kolorów oraz zarządzanie dynamiczną zmianą barwy w zależności od temperatury. Każda metoda korzysta z adnotacji `@GET`, definiując odpowiednie ścieżki endpointów serwera. Dzięki zastosowaniu `Response<T>`, każda odpowiedź z serwera jest bezpiecznie obsługiwana, umożliwiając łatwą kontrolę poprawności zwracanych danych. Moduł ten stanowi kluczowy element warstwy sieciowej aplikacji, zapewniając przejrzystą i efektywną komunikację z serwerem.

Przykładowa metoda inerfejsu:

```
@GET("/get-details/{rid}")
suspend fun getLightBulbDetails(
    @Path("rid") rid: String
): Response<LightBulb?>
```

4.2.6. DataStoreManager.kt

Plik `DataStoreManager.kt` odpowiada za przechowywanie i pobieranie danych konfiguracyjnych związanych z połączeniem aplikacji `HuePi` z serwerem Raspberry Pi. Do tego celu wykorzystano **DataStore Preferences**, które zastępuje starszy mechanizm *SharedPreferences*, oferując bardziej wydajny i bezpieczny sposób zarządzania danymi.

Głównym zadaniem `DataStoreManager.kt` jest zapis i odczyt kluczowych informacji dotyczących serwera Raspberry Pi:

- Adres URL serwera – przechowywany, aby użytkownik nie musiał ponownie wpisywać adresu IP przy każdym uruchomieniu aplikacji.
- Klucz API – zapisany lokalnie, aby aplikacja mogła automatycznie uwierzytelniać się przy komunikacji z serwerem.

Zasada działania

Plik definiuje obiekt `preferenceDataStore`, który jest powiązany z kontekstem aplikacji i zarządza przechowywaniem preferencji w lokalnym magazynie danych. Dzięki temu aplikacja może zapisywać i odczytywać ustawienia nawet po jej ponownym uruchomieniu.

Przykładowo, funkcja zapisu adresu serwera działa w następujący sposób:

```
suspend fun saveRaspberryPiUrl(ipAddressToSave: String) {
    context.preferenceDataStore.edit { preferences ->
        preferences[RASPBERRY_PI_URL] = ipAddressToSave
    }
}
```

Podczas zapisu dane są umieszczane w `DataStore`, a każda wartość jest powiązana z unikalnym kluczem. W podobny sposób działa pobieranie wartości – aplikacja sprawdza, czy adres URL został zapisany, a jeśli nie, zwraca wartość domyślną. Plik `DataStoreManager.kt` pozwala na automatyczne zapamiętywanie danych o serwerze Raspberry Pi, dzięki czemu użytkownik nie musi ręcznie wprowadzać ich przy każdym uruchomieniu aplikacji. Jest to kluczowe dla wygody użytkowania oraz zapewnienia płynnej komunikacji z systemem inteligentnego oświetlenia.

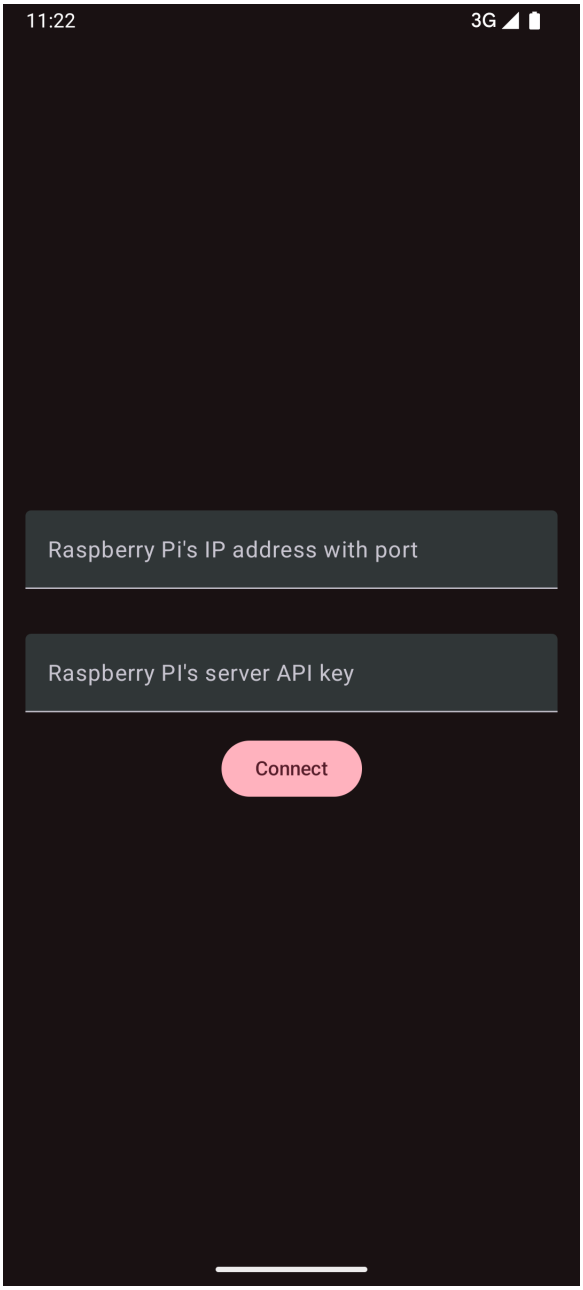
4.2.7. Ekran SetupAndConnect

Ekran `SetupAndConnect` jest pierwszym ekranem, na który trafia użytkownik po uruchomieniu aplikacji `HuePi`. Jego głównym celem jest umożliwienie wprowadzenia da-

nych niezbędnych do nawiązania połączenia z serwerem uruchomionym na **Raspberry Pi**, który zarządza mostkiem Philips Hue. Użytkownik musi podać:

- Adres IP wraz z numerem portu serwera Raspberry Pi.
- Klucz API wymagany do autoryzacji żądań wysyłanych do serwera.

Po wprowadzeniu tych danych aplikacja sprawdza poprawność podanych wartości, zapisuje je lokalnie przy użyciu **DataStore Preferences** i próbuje nawiązać połączenie z serwerem.



Rysunek 4.3: Ekran setup_and_connect

Struktura plików ekranu

Ekran został podzielony na trzy pliki:

- **`SetupAndConnectScreen.kt`** – główny kontener ekranu, który ustawia tło aplikacji i wywołuje funkcję odpowiedzialną za zawartość interfejsu.
- **`SetupAndConnectContent.kt`** – definiuje wygląd i układ komponentów interfejsu użytkownika, takich jak pola tekstowe i przyciski.
- **`SetupAndConnectViewModel.kt`** – zarządza logiką biznesową, w tym walidacją danych wejściowych oraz nawiązywaniem połączenia z serwerem.

Podział ten zgodny jest z architekturą *MVVM* (Model-View-ViewModel), która pozwala na oddzielenie logiki aplikacji od interfejsu użytkownika, co ułatwia testowanie i utrzymanie kodu.

Walidacja danych wejściowych

Aby uniknąć błędów użytkownika, ekran zawiera prostą walidację danych. Weryfikowane są dwa kluczowe elementy:

- Adres IP i port – musi składać się z co najmniej 13 znaków.
- Klucz API – nie może być pusty.

Przykładowa implementacja walidacji znajduje się w pliku `SetupAndConnectViewModel.kt`:

```
fun validateInputs() {
    _isIpAddressValid.value = _ipNumbersTextValue.value.length >= 13
    _isApiKeyValid.value = _apiKeyTextValue.value.isNotBlank()
}
```

Jeśli podane wartości są nieprawidłowe, użytkownik otrzymuje odpowiedni komunikat błędu.

Proces łączenia z serwerem

Po poprawnym wprowadzeniu danych użytkownik naciska przycisk „Connect”, który inicjuje proces weryfikacji połączenia z serwerem. Aplikacja:

1) Formatuje podany adres IP do poprawnej postaci.
2) Tworzy instancję klienta Retrofit do komunikacji z serwerem.

3) Wysyła żądanie testowe sprawdzające dostępność serwera.

4) Jeśli połączenie się powiedzie, zapisuje adres i klucz API w lokalnym magazynie danych (`DataStoreManager.kt`).

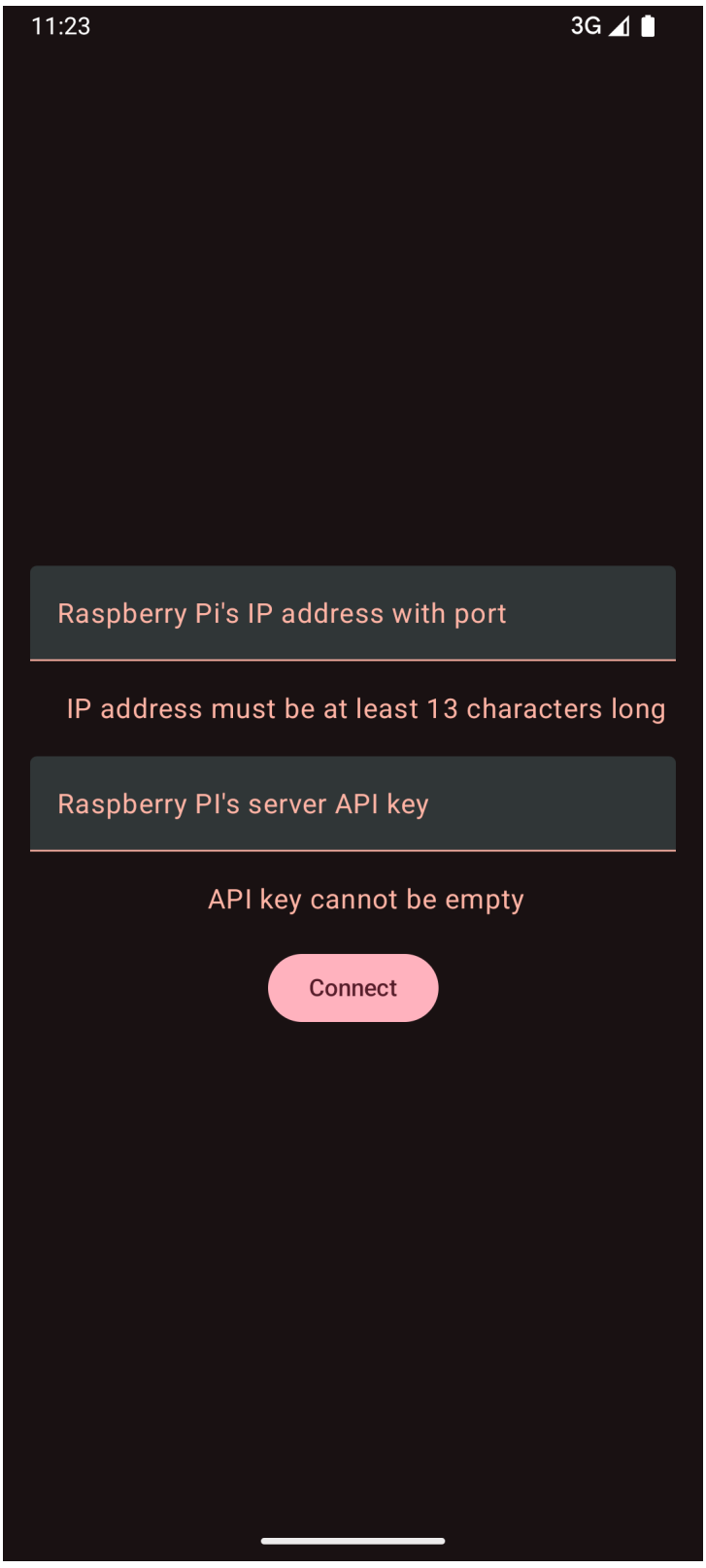
5) Przekierowuje użytkownika do ekranu głównego aplikacji.

Fragment kodu odpowiedzialny za wysyłanie żądania testowego:

```
val response = raspberryPiAPIService.checkConnection()
if (response.isSuccessful && response.body() != null) {
    if (response.body()!!.message == "OK") {
        val saveUrlDeferred = async { dataStoreManager.
    saveRaspberryPiUrl(formattedRaspberryUrl) }
        val saveApiKeyDeferred = async { dataStoreManager.
    saveRaspberryPiApiKey(apiKey) }
        saveUrlDeferred.await()
        saveApiKeyDeferred.await()
        _navigationEvent.send(true)
    }
}
```

Interfejs użytkownika

Ekran zawiera dwa pola tekstowe do wprowadzania adresu IP oraz klucza API, a także przycisk „Connect”, który inicjuje proces łączenia. W przypadku błędnych danych użytkownik otrzymuje komunikaty o błędach wyświetlane pod polami tekstowymi.



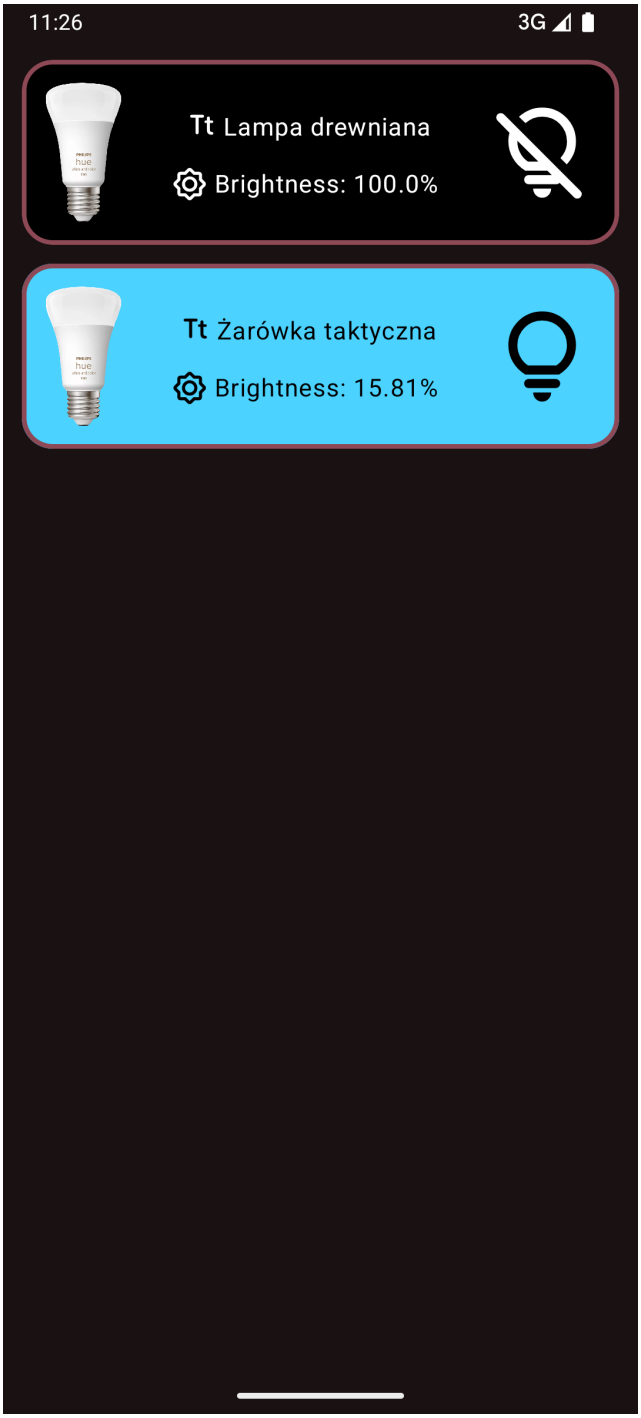
Rysunek 4.4: Błędne dane w polach tekstowych

Podczas nawiązywania połączenia przycisk zmienia się w animowany wskaźnik postępu, co sygnalizuje użytkownikowi trwającą operację.

Ekran `SetupAndConnect` jest kluczowym elementem aplikacji, ponieważ pozwala na nawiązanie połączenia z serwerem Raspberry Pi, bez którego dalsze funkcjonalności aplikacji nie byłyby dostępne. Dzięki zastosowaniu walidacji danych oraz przechowywaniu konfiguracji w **DataStore Preferences**, użytkownik nie musi każdorazowo wprowadzać adresu IP i klucza API, co znacząco poprawia wygodę korzystania z aplikacji.

4.2.8. Ekran Home

Ekran `Home` pełni rolę głównego widoku aplikacji `HuePi`, w którym użytkownik może przeglądać listę dostępnych żarówek Philips Hue. Wyświetlane informacje obejmują nazwę żarówki, jej aktualny kolor, poziom jasności oraz stan (włączona/wyłączona). Użytkownik ma również możliwość odświeżenia listy urządzeń oraz przejścia do ekranu szczegółów konkretnej żarówki.



Rysunek 4.5: Ekran z listą żarówek

Struktura plików ekranu

Ekran został podzielony na trzy główne pliki:

- **`HomeScreen.kt`** – główny kontener ekranu, który obsługuje układ interfejsu oraz funkcję *pull-to-refresh*.
- **`HomeContent.kt`** – definiuje wygląd i zachowanie listy żarówek, obsługuje przypadek, gdy lista jest pusta.
- **`HomeViewModel.kt`** – zarządza pobieraniem listy żarówek oraz ich stanem, wykorzystując do tego **Retrofit** i **DataStore Preferences**.

Podział ten zgodny jest z architekturą *MVVM* (Model-View-ViewModel), dzięki czemu logika biznesowa jest odseparowana od interfejsu użytkownika, co poprawia czytelność kodu i ułatwia jego testowanie.

Interfejs użytkownika

Lista żarówek wyświetlana jest w postaci `LazyColumn`, co pozwala na efektywne renderowanie długich list. Jeśli lista jest pusta, użytkownik widzi komunikat „No light bulbs found!”. Każdy element listy wyświetlany jest przy użyciu komponentu `LightBulbItem`, który prezentuje informacje o stanie żarówki.

```
LazyColumn(
    modifier = modifier.fillMaxSize(),
    contentPadding = PaddingValues(12.dp),
    verticalArrangement = Arrangement.spacedBy(12.dp)
) {
    items(lightBulbsList) { lightBulb ->
        LightBulbItem(
            color = Color(lightBulb.color.toColorInt()),
            lightBulbName = lightBulb.name,
            brightnessLevel = lightBulb.brightness,
            lightBulbOn = lightBulb.isOn,
            rid = lightBulb.rid,
            onLightBulbClicked = onLightBulbClicked
        )
    }
}
```

Ekran obsługuje także mechanizm *pull-to-refresh*, który umożliwia użytkownikowi ręczne odświeżenie listy żarówek.

```
PullToRefreshBox(
    state = pullToRefreshState,
    isRefreshing = isRefreshing,
    onRefresh = onRefresh,
    modifier = Modifier.fillMaxSize()
```

```
) {
    when {
        isRefreshing -> {}
        lightBulbsList.isNullOrEmpty() -> {
            Box(contentAlignment = Alignment.Center, modifier =
    Modifier.fillMaxSize()) {
                Text("No light bulbs found!", style =
    MaterialTheme.typography.bodyLarge)
            }
        }
        else -> {
            HomeContent(
                lightBulbsList = lightBulbsList,
                onLightBulbClicked = onLightBulbClicked,
                modifier = Modifier
                    .statusBarsPadding()
                    .navigationBarsPadding()
            )
        }
    }
}
```

Pobieranie listy żarówek

Dane o żarówkach pobierane są z serwera Raspberry Pi poprzez moduł **Retrofit**. Plik `HomeViewModel.kt` zarządza pobieraniem listy świateł i ich przechowywaniem w stanie aplikacji. Aby uniknąć ponownego wpisywania adresu IP i klucza API, dane te są odczytywane z **DataStore Preferences**.

```
fun fetchLightBulbs() {
    viewModelScope.launch {
        _isRefreshing.value = true
        try {
            val raspberryIpAddress = dataStoreManager.
    getRaspberryPiUrl()
            val raspberryApiKey = dataStoreManager.
    getRaspberryPiApiKey()

            val raspberryPiAPIService = RetrofitInstance.
    getClient(
                baseUrl = raspberryIpAddress,
                raspberryApiKey = raspberryApiKey
            )

            val lightBulbs = raspberryPiAPIService.
    getAllLightsDetails().body()
            _lightBulbsList.value = lightBulbs
        } catch (e: Exception) {
            _lightBulbsList.value = null
        } finally {
            _isRefreshing.value = false
        }
    }
}
```

Ekran `HomeScreen` stanowi główne centrum sterowania dla użytkownika. Pozwala na szybkie przeglądanie aktualnego stanu oświetlenia oraz nawigowanie do ekranu szczegółów konkretnej żarówki. Dzięki zastosowaniu mechanizmu *pull-to-refresh*, użytkownik ma pełną kontrolę nad aktualizacją wyświetlanych danych.

4.2.9. Ekran LightDetails

Ekran `LightDetailsScreen` umożliwia użytkownikowi sterowanie wybraną żarówką Philips Hue. Po przejściu z ekranu głównego użytkownik ma dostęp do szczegółowych informacji o żarówce oraz może zmieniać jej stan, jasność, kolor i włączyć tryb dynamicznej zmiany barwy w zależności od temperatury otoczenia.

Struktura plików ekranu

Ekran został podzielony na trzy pliki:

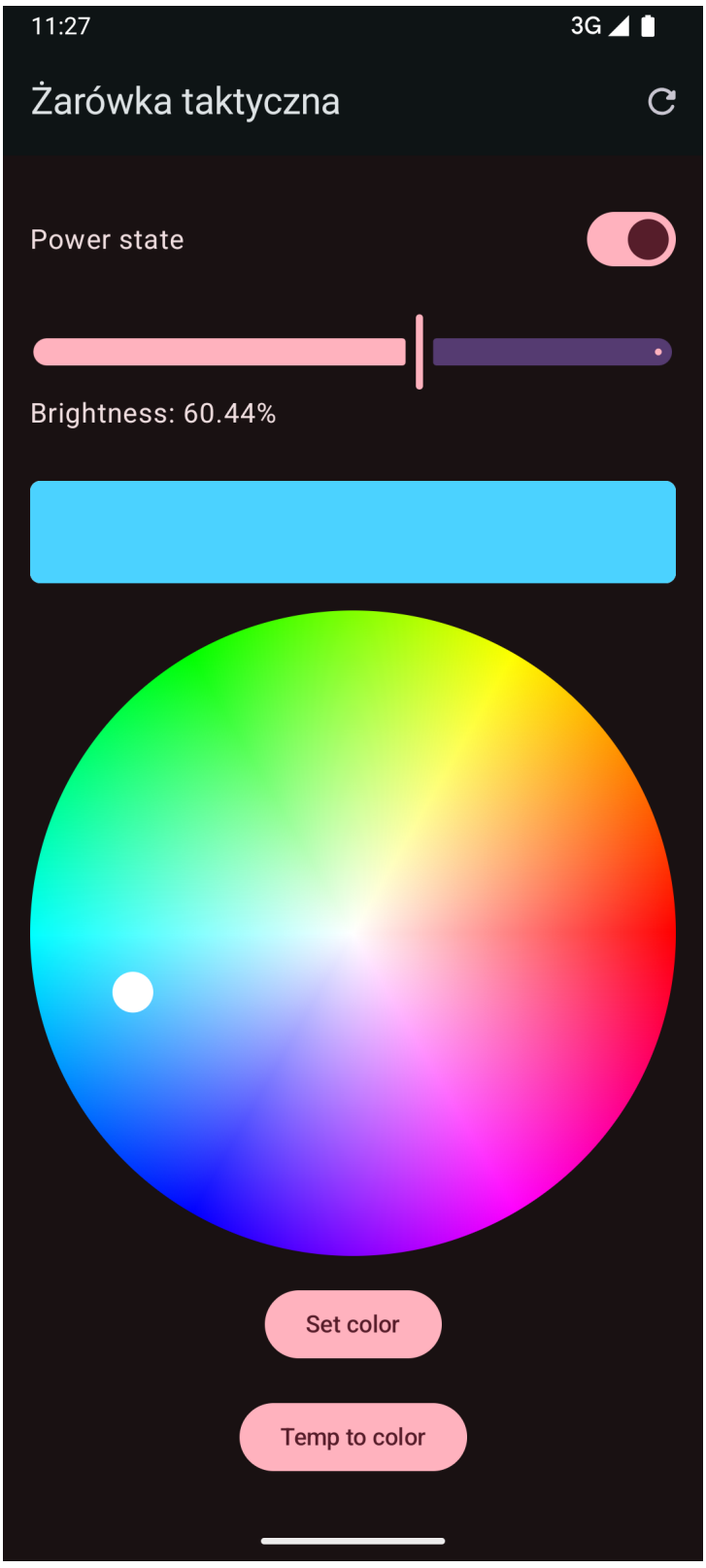
- **`LightDetailsScreen.kt`** – główny kontener ekranu, zarządza układem interfejsu i obsługuje mechanizm odświeżania danych.
- **`LightDetailsContent.kt`** – definiuje widok ekranu, zawierający komponenty do sterowania żarówką (przełącznik zasilania, suwak jasności, wybór koloru).
- **`LightDetailsViewModel.kt`** – zarządza pobieraniem danych o żarówce oraz obsługą interakcji użytkownika, np. wysyłaniem żądań do serwera Raspberry Pi w celu zmiany ustawień światła.

Podział ten zgodny jest z architekturą *MVVM* (Model-View-ViewModel), co zapewnia oddzielenie logiki biznesowej od warstwy interfejsu użytkownika.

Interfejs użytkownika

Główne elementy ekranu to:

- **Przełącznik zasilania** – umożliwia włączenie i wyłączenie żarówki.
- **Suwak jasności** – pozwala dostosować poziom jasności światła.
- **Koło wyboru koloru** – umożliwia użytkownikowi zmianę barwy światła przy użyciu selektora kolorów.
- **Tryb zmiany barwy w zależności od temperatury** – opcjonalna funkcja, która pozwala dostosować kolor światła na podstawie danych z czujnika temperatury.



Rysunek 4.6: Ekran ustawień danej żarówki

Dodatkowo użytkownik może odświeżyć dane o żarówce za pomocą ikony odświeżania w górnej części ekranu.

Pobieranie danych o żarówce

Dane o żarówce są pobierane z serwera Raspberry Pi za pomocą **Retrofit**. Widok aktualizuje się dynamicznie na podstawie zmian w stanie aplikacji.

```
fun fetchLightBulb() {
    viewModelScope.launch {
        _isFetchingData.value = true
        try {
            raspberryIpAddress = dataStoreManager.
   getRaspberryPiUrl()
            raspberryApiKey = dataStoreManager.
   getRaspberryPiApiKey()

            raspberryPiAPIService = RetrofitInstance.getClient(
                baseUrl = raspberryIpAddress,
                raspberryApiKey = raspberryApiKey
            )

            val lightBulb = raspberryPiAPIService.
   getLightBulbDetails(rid).body()
            _lightBulb.value = lightBulb
        } catch (e: Exception) {
            _lightBulb.value = null
        } finally {
            _isFetchingData.value = false
        }
    }
}
```

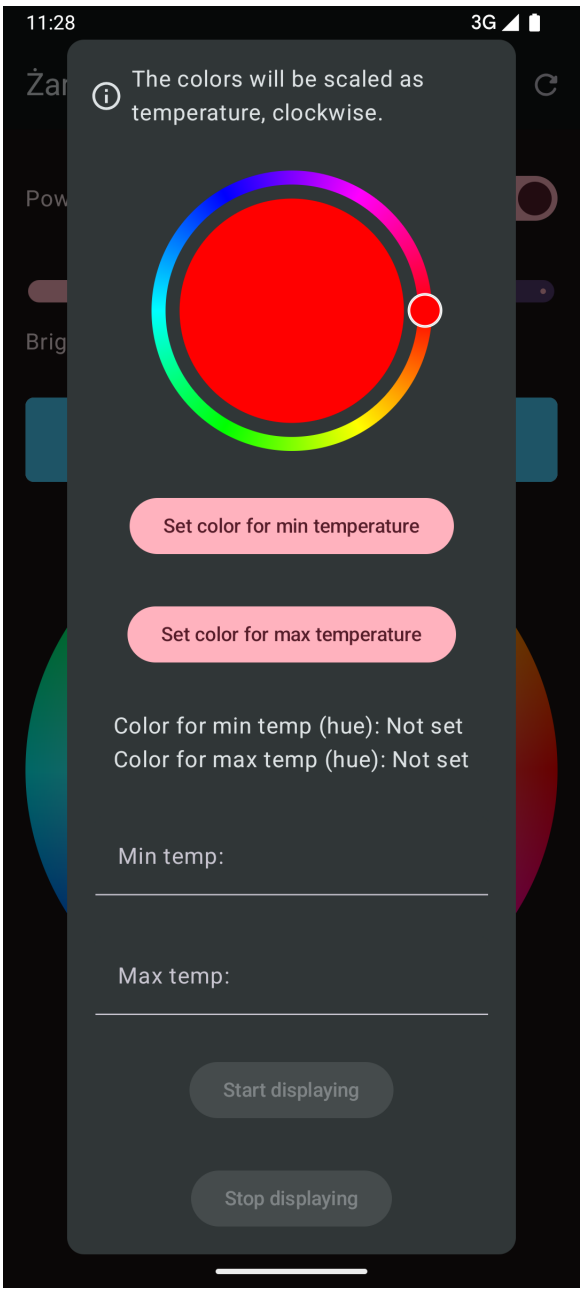
Obsługa zmiany jasności i koloru

Użytkownik może dostosować jasność oraz kolor światła za pomocą odpowiednich kontrolek interfejsu. Każda zmiana jest natychmiast wysyłana do serwera.

```
fun changeBrightness(level: Float) {
    viewModelScope.launch {
        val response = raspberryPiAPIService.setBrightness(
            rid = rid,
            level = level
        )
    }
}

fun setColor(h: Float, s: Float, v: Float) {
    viewModelScope.launch {
        val response = raspberryPiAPIService.setColor(
            rid = rid,
            h = h,
            s = s,
            v = v
        )
    }
}
```

Tryb zmiany barwy w zależności od temperatury

Jedną z kluczowych funkcji ekranu `LightDetailsScreen` jest możliwość aktywowania trybu dynamicznej zmiany barwy światła w zależności od temperatury otoczenia. Funkcja ta została zaimplementowana w formie **okna dialogowego**, które pojawia się po naciśnięciu przycisku `Temp to color`.



Rysunek 4.7: Okno dialogowe trybu wyświetlania koloru na podstawie temperatury

Po otwarciu okna użytkownik musi ustawić następujące parametry:

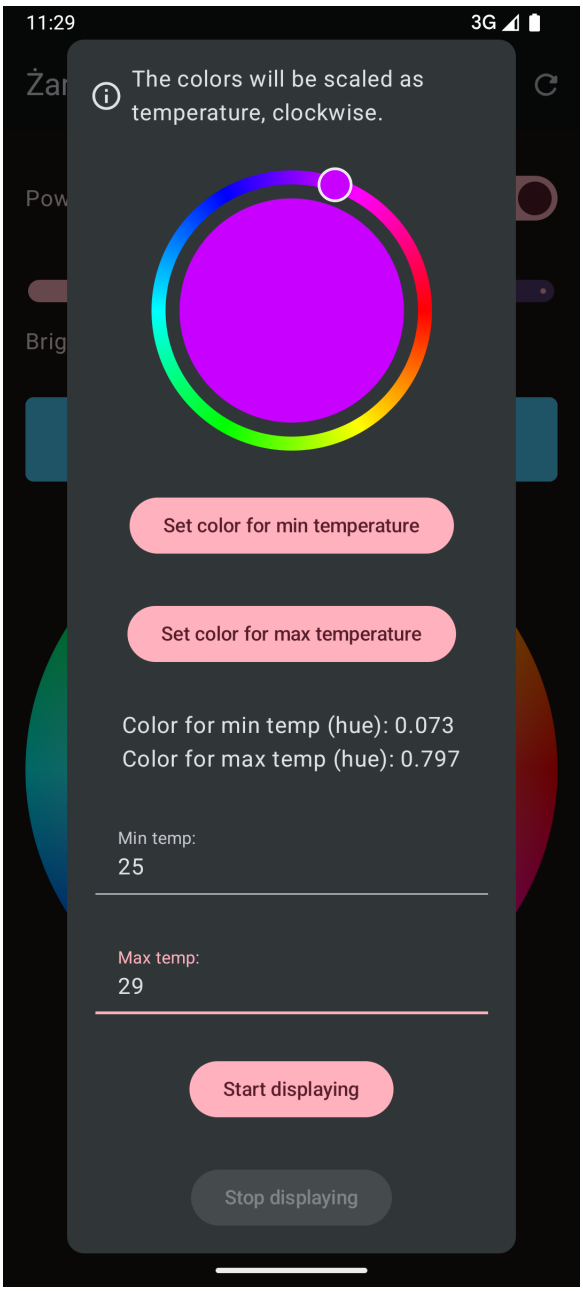
- **Minimalna temperatura** – określa najniższą wartość temperatury, przy której

światło ma przyjąć wybrany kolor.

- **Maksymalna temperatura** – określa najwyższą wartość temperatury, przy której światło ma przyjąć inny wybrany kolor.

- **Kolor dla minimalnej temperatury** – użytkownik wybiera kolor, który będzie wyświetlany, gdy temperatura osiągnie wartość minimalną.

- **Kolor dla maksymalnej temperatury** – użytkownik wybiera kolor, który będzie wyświetlany, gdy temperatura osiągnie wartość maksymalną.



Rysunek 4.8: Okno dialogowe trybu wyświetlania koloru na podstawie temperatury z uzupełnionymi parametrami

Aby uprościć konfigurację, użytkownik wybiera kolory na podstawie jednego parametru – wartości **H (Hue)** z przestrzeni barw **HSV**. Oznacza to, że nie wybiera on pełnego koloru RGB, lecz jedynie jego pozycję na palecie barw HSV.

Sposób obliczania koloru

Gdy funkcja zostanie aktywowana, kolor światła będzie zmieniał się w sposób płynny pomiędzy wartościami wybranymi przez użytkownika. Wartość koloru jest obliczana

na podstawie aktualnej temperatury otoczenia przy użyciu **interpolacji liniowej** w przestrzeni HSV. Interpolacja odbywa się zgodnie z ruchem wskazówek zegara na palecie HSV. Oznacza to, że jeśli użytkownik wybierze dla temperatury minimalnej kolor czerwony ($H = 0.0$) i dla temperatury maksymalnej kolor zielony ($H = 0.33$), to w temperaturach pośrednich kolor będzie stopniowo przesuwał się przez pomarańczowy, żółty aż do zielonego.

```
fun startTempToColorTask(hueMin: Float, hueMax: Float, tempMin:
    Float, tempMax: Float) {
     viewModelScope.launch {
         val response = raspberryPiAPIService.
    startTempToColorTask(
              rid = rid,
              hueMin = hueMin,
              hueMax = hueMax,
              tempMin = tempMin,
              tempMax = tempMax
         )
     }
}
```

Wartości te są przesyłane do serwera Raspberry Pi, który na podstawie odczytów z czujnika temperatury oblicza i aktualizuje kolor światła w czasie rzeczywistym.

Tryb dynamicznej zmiany koloru w zależności od temperatury pozwala użytkownikowi na automatyczne dostosowanie oświetlenia do warunków otoczenia. Dzięki intuicyjnemu interfejsowi, użytkownik może łatwo skonfigurować zakres temperatur i odpowiadające im kolory, co sprawia, że światło zmienia się płynnie zgodnie z ruchem wskazówek zegara na palecie HSV. Jest to zaawansowana funkcjonalność, która wprowadza dodatkowy poziom automatyzacji w ekosystemie Philips Hue.

5. Podsumowanie

Projekt `HuePi` to kompleksowe rozwiązanie umożliwiające użytkownikowi sterowanie inteligentnym oświetleniem Philips Hue poprzez serwer uruchomiony na **Raspberry Pi**. System został zaprojektowany w sposób modularny, wykorzystując nowoczesne technologie zarówno po stronie serwera, jak i aplikacji mobilnej.

5.1. Główne założenia projektu

Głównym celem projektu było stworzenie aplikacji, która pozwala użytkownikowi na wygodne zarządzanie oświetleniem w sposób bardziej elastyczny niż przy użyciu standardowej aplikacji Philips Hue. Kluczowe założenia obejmowały:

- Możliwość sterowania światłami Hue poprzez dedykowany serwer działający na Raspberry Pi.
- Rozszerzenie funkcjonalności ekosystemu Hue o obsługę czujnika temperatury BME280.
- Intuicyjny interfejs użytkownika, bazujący na **Jetpack Compose** i wzorcu **MVVM**.
- Implementację dynamicznej zmiany koloru światła w zależności od temperatury otoczenia.
- Wsparcie dla pamięci lokalnej (**DataStore Preferences**), pozwalającej na przechowywanie konfiguracji połączenia.

5.2. Technologie użyte w projekcie

W projekcie wykorzystano szereg nowoczesnych technologii, które zapewniają wysoką wydajność i skalowalność systemu:

- **FastAPI** – framework użyty do implementacji serwera REST API na Raspberry Pi.
- **Retrofit** – biblioteka odpowiedzialna za komunikację aplikacji mobilnej z serwerem.

- **Jetpack Compose** – nowoczesny system budowy interfejsu użytkownika w aplikacji mobilnej.
- **DataStore Preferences** – mechanizm zapisywania konfiguracji aplikacji na urządzeniu mobilnym.
- **mDNS** – technologia użyta do automatycznego wykrywania mostka Philips Hue w sieci lokalnej.

5.3. Architektura systemu

Projekt został podzielony na dwie główne części:

- **Serwer na Raspberry Pi** – zarządza połączeniem z mostkiem Hue, obsługuje czujnik temperatury i udostępnia API do komunikacji z aplikacją mobilną.
- **Aplikacja mobilna** – pozwala użytkownikowi na sterowanie światłami, przechowuje konfigurację połączenia i dynamicznie pobiera dane z serwera.

Struktura kodu w obu częściach została zaprojektowana zgodnie z dobrymi praktykami programistycznymi, co zapewnia wysoką czytelność i możliwość dalszej rozbudowy systemu.

5.4. Funkcjonalności aplikacji

Aplikacja mobilna dostarcza użytkownikowi bogaty zestaw funkcji:

- **Łączenie z serwerem** – użytkownik może skonfigurować połączenie poprzez podanie adresu IP i klucza API.
- **Lista żarówek** – wyświetlanie wszystkich dostępnych świateł w ekosystemie Hue.
- **Sterowanie żarówką** – możliwość włączania i wyłączania światła, regulacji jasności oraz zmiany koloru.
- **Dynamiczna zmiana koloru** – tryb, w którym kolor światła dostosowuje się do temperatury otoczenia.
- **Odświeżanie danych** – użytkownik może ręcznie aktualizować informacje o dostępnych żarówkach i ich stanie.

5.5. Dalszy rozwój

Projekt `HuePi` udowodnił, że istnieje możliwość rozszerzenia ekosystemu Philips Hue o nowe funkcjonalności, które nie są dostępne w natywnej aplikacji producenta. Dzięki wykorzystaniu serwera pośredniczącego na Raspberry Pi możliwe jest wprowadzenie niestandardowych funkcji, takich jak zmiana koloru światła w zależności od temperatury. Możliwości dalszego rozwoju projektu obejmują:

- Wprowadzenie harmonogramów oświetlenia na podstawie danych z czujników.
- Integrację z dodatkowymi urządzeniami z ekosystemu Philips Hue
- Optymalizację komunikacji między aplikacją a serwerem poprzez WebSockety zamiast REST API.
- Wsparcie dla sterowania głosowego przy użyciu asystentów, takich jak Google Assistant czy Alexa.

5.6. Wnioski

Projekt `HuePi` to nowoczesne rozwiązanie, które łączy elastyczność systemów inteligentnego oświetlenia z możliwościami rozszerzonej automatyki domowej. Dzięki dobrze zaprojektowanej architekturze, modularności kodu oraz zastosowaniu nowoczesnych technologii, aplikacja zapewnia intuicyjne i wygodne sterowanie oświetleniem Philips Hue. W przyszłości projekt może zostać rozwinięty o dodatkowe funkcje, co jeszcze bardziej zwiększy jego użyteczność w codziennym użytkowaniu.

Załączniki

Link do repozytorium projektu https://github.com/Klusio19/HuePi

Literatura

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

[2] https://www.philips-hue.com/pl-pl/explore-hue/apps/bridge

[3] https://www.home-assistant.io/docs/automation/

[4] https://home.google.com/about-google-home/

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

[6] https://fastapi.tiangolo.com/#interactive-api-docs

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

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

[9] https://developer.android.com/develop/ui/compose/why-adopt#less-code

[10] https://square.github.io/retrofit/

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: Raspberry Pi, Philips Hue, Android, Aplikacja mobilna

Projekt HuePi to aplikacja mobilna umożliwiająca sterowanie oświetleniem Philips Hue przez serwer na Raspberry Pi. Wykorzystuje nowoczesne technologie, takie jak FastAPI, Retrofit i Jetpack Compose. Umożliwia zarządzanie światłami, regulację jasności, zmianę kolorów oraz dynamiczne dostosowanie barwy do temperatury otoczenia.

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: Raspberry Pi, Philips Hue, Android, Mobile app

The HuePi project is a mobile app for controlling Philips Hue lighting via a Raspberry Pi server. It employs modern technologies such as FastAPI, Retrofit, and Jetpack Compose. The app enables light management, brightness control, color adjustments, and dynamic color adaptation based on ambient temperature.