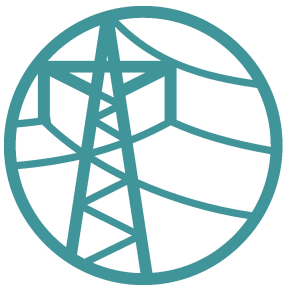


WYDZIAŁ
ELEKTROTECHNIKI
I INFORMATYKI
POLITECHNIKI RZESZOWSKIEJ

Rafał Stępień

Książkowy dziennik w chmurze

Praca dyplomowa inżynierska

Opiekun pracy:
dr inż. Mariusz Mączka

Rzeszów, 2025

Spis treści

1. Wstęp **6**

2. Podobne aplikacje **7**

2.1. Bookmory 7

2.2. TV Time 7

3. Środowiska i technologie **8**

3.1. Android Studio 8

3.2. Kotlin 8

3.3. Jetpack Compose 9

3.4. Material Design 10

3.5. Supabase 10

3.6. Cloudinary 11

3.7. GitHub 11

4. Architektura bazy danych Supabase **12**

4.1. Schemat bazy danych 12

4.2. Normalizacja 14

4.2.1. Pierwsza postać normalna 14

4.2.2. Druga postać normalna 14

4.2.3. Trzecia postać normalna 14

4.2.4. Czwarta postać normalna 15

4.3. Funkcje bazy danych 15

5. Architektura i struktura aplikacji **18**

5.1. Struktura projektu 18

5.1.1. Folder data 19

5.1.2. Folder di 19

5.1.3. Folder navigation 20

5.1.4. Folder presentation 20

5.1.5. Folder utils 22

5.2. Wstrzykiwanie zależności (Dependency Injection) 22

5.2.1. Konfiguracja Hilt 23

5.2.2. Moduły Hilt 24

5.2.3. Przykłady wstrzykiwania zależności z Hilt 25

5.3. MVVM (Model-View-ViewModel) 27
5.3.1. Model 28
6. Podsumowanie **30**
Literatura **31**

1. Wstęp

W dobie rosnącej cyfryzacji i dynamicznego rozwoju technologii mobilnych, coraz więcej osób poszukuje nowoczesnych narzędzi wspierających codzienne czynności, w tym także zarządzanie swoimi pasjami i zainteresowaniami. Jedną z takich pasji, cieszącą się niezmienną popularnością, jest czytanie książek. Wraz z rosnącą liczbą dostępnych tytułów zarządzanie osobistą biblioteką może stać się problematyczne.

Niniejsza praca dotyczy zaprojektowania i implementacji aplikacji mobilnej o nazwie „BookTracker", stworzonej z wykorzystaniem technologii Jetpack Compose oraz zintegrowanej z internetową bazą danych Supabase. Aplikacja oferuje użytkownikom możliwość oznaczania książek jako posiadanych lub przeczytanych, śledzenia nadchodzących premier literackich oraz zarządzania osobistą biblioteką w sposób przejrzysty i dostępny z każdego miejsca, dzięki wykorzystaniu technologii chmurowych. Tematyka ta została wybrana ze względu na rosnącą popularność aplikacji wspierających organizację życia codziennego oraz potrzebę zapewnienia użytkownikom narzędzi umożliwiających wygodne i efektywne zarządzanie ich kolekcjami literackimi.

Zakres pracy obejmuje zaprojektowanie kluczowych funkcji aplikacji, implementację interfejsu użytkownika oraz integrację z bazą danych w chmurze. Głównym celem pracy jest stworzenie aplikacji mobilnej, która w przejrzysty sposób umożliwi użytkownikom zarządzanie osobistą biblioteką książek z wykorzystaniem technologii chmurowych.

2. Podobne aplikacje

W dzisiejszych czasach niemal każda aplikacja ma już swoje odpowiedniki na rynku. Podczas tworzenia aplikacji postawiono więc na skupienie się na grupie docelowej użytkowników, którzy oczekują konkretnych rozwiązań.

2.1. Bookmory

Bookmory to przykład wielu istniejących aplikacji dzięki którym można zarządzać swoją biblioteką, do jej odpowiedników można zaliczyć jeszcze kilka aplikacji takich jak: StoryGraph, Bookshelf, czy Bookly. Każda z wymienionych aplikacji spełnia podobne zadania, a dla większości użytkowników wybór pomiędzy nimi nie ma większego znaczenia, ponieważ różnice między nimi są minimalne.

Istnienie tych aplikacji zainspirowało stworzenie alternatywy, która będzie różniła się od swoich poprzedników unikalną funkcjonalnością i skupieniem się na konkretnej grupie odbiorców.

2.2. TV Time

TV Time to popularna aplikacja służąca do śledzenia postępu oglądania seriali i filmów. Pomimo tego, że aplikacja ta nie ma możliwości zarządzania książkami i skupia się wyłącznie na produkcjach filmowych, to funkcjonalność śledzenia seriali stała się inspiracją do stworzenia alternatywnej aplikacji do śledzenia postępu czytania książek. Główną różnicą do istniejących już rozwiązań jest skupienie się na seriach książek wydawanych w tomach i ułatwieniu użytkownikom obserwowania i zarządzania książkami, które są ze sobą powiązane.

3. Środowiska i technologie

Wybór odpowiednich środowisk i technologii jest kluczowym elementem każdego projektu informatycznego, ponieważ wpływa zarówno na efektywność pracy, jak i na jakość końcowego rozwiązania. Ważnym jest przedstawienie wybranych rozwiązań i uzasadnienie dlaczego były one wybrane zamiast alternatywnych opcji.

3.1. Android Studio

Android Studio to oficjalne zintegrowane środowisko programistyczne (IDE) do tworzenia aplikacji na system Android bazujące na IntelliJ IDEA firmy JetBrains, stworzone i rozwijane przez firmę Google. Jest to kompleksowe narzędzie, które oferuje zestaw funkcji, mających na celu ułatwienie tworzenia, debugowania i publikowania aplikacji.

Środowisko to jest nieustannie aktualizowane przez Google, aby wspierać najnowsze wersje systemu Android oraz dodawać nowe funkcje. Android Studio obsługuje programowanie w językach takich jak Java, Kotlin (rekomendowany przez Google), a także w mniejszym stopniu w C++.

Kluczowym elementem jest edytor kodu, który obsługuje autouzupełnianie, refaktoryzację oraz podpowiedzi kontekstowe, co znacząco ułatwia pisanie czystego i efektywnego kodu.

Android Studio zostało wybrane ze względu na to, że jest najpełniejszym narzędziem do tworzenia aplikacji na system android i w przeciwieństwie do np. Visual Studio Code, można praktycznie od razu rozpocząć pracę zamiast zajmowania się instalowaniem rozszerzeń i potrzebnych komponentów. Android Studio oferuje wszystkie niezbędne funkcje w jednym pakiecie.

3.2. Kotlin

Kotlin to nowoczesny, statycznie typowany język programowania, który został stworzony przez firmę JetBrains i jest oficjalnie wspierany przez Google do tworzenia natywnych aplikacji na system Android. Kotlin jest zaprojektowany z myślą o prostocie, bezpieczeństwie i interoperacyjności z Javą - język ten jest w pełni kompatybilny z istniejącym ekosystemem Javy, co umożliwia łatwą integrację z istniejącym kodem i bibliotekami.

Kotlin oferuje wiele nowoczesnych funkcji, które czynią go atrakcyjnym wyborem. Jego kluczowe cechy to m.in. zwięzła składnia, która pozwala na pisanie czytelnego i mniej podatnego na błędy kodu, oraz zaawansowane mechanizmy, takie jak obsługa funkcji wyższych rzędów, rozszerzenia klas czy programowanie funkcyjne. Wbudowane mechanizmy bezpieczeństwa, takie jak system typów zapobiegający błędom typu null pointer exception (tzw. "null safety"), znacząco zwiększają niezawodność aplikacji.

Kotlin wspiera także współbieżność dzięki korutynom, które są lekkim mechanizmem współbieżności umożliwiającym pisanie asynchronicznego kodu w czytelnym i intuicyjnym stylu. Korutyny działają w ramach istniejących wątków, wykorzystując mechanizmy wstrzymywania i wznawiania, co pozwala na efektywne zarządzanie zasobami systemowymi bez potrzeby blokowania wątków. To podejście znacząco upraszcza tworzenie wydajnych aplikacji, zwłaszcza tych, które intensywnie korzystają z asynchronicznej komunikacji sieciowej, przetwarzania dużych ilości danych czy operacji wejścia/wyjścia.

3.3. Jetpack Compose

Jetpack Compose to nowoczesny framework interfejsu użytkownika stworzony przez Google, który pozwala na tworzenie aplikacji na Androida w sposób deklaratywny. Zamiast używać tradycyjnych plików XML do definiowania widoków, Jetpack Compose umożliwia definiowanie interfejsu w kodzie Kotlin, co prowadzi do tworzenia prostszych, bardziej zwięzłych i łatwiejszych w utrzymaniu aplikacji.

Jedną z największych zalet Jetpack Compose jest możliwość dynamicznego reagowania na zmiany stanu aplikacji. Dzięki podejściu opartemu na deklaratywnej reaktywności, widoki automatycznie aktualizują się w odpowiedzi na zmiany danych, co eliminuje konieczność ręcznego zarządzania aktualizacjami interfejsu użytkownika.

W Jetpack Compose funkcje kompozycyjne (ang. composable functions) stanowią podstawę tworzenia interfejsu użytkownika. Są to specjalne funkcje oznaczone adnotacją @Composable, które pozwalają na deklaratywne definiowanie i łączenie elementów UI. Funkcje kompozycyjne w Jetpack Compose działają na zasadzie deklaratywnego określania struktury i zawartości interfejsu użytkownika, zamiast szczegółowego opisywania sposobu jego wyświetlania. Co umożliwia koncentrację na logice aplikacji, a nie na szczegółach implementacji interfejsu. Każda funkcja kompozycyjna może zawierać inne funkcje kompozycyjne, tworząc w ten sposób złożone i hierarchiczne

struktury UI w sposób naturalny i łatwy do zrozumienia. Takie podejście umożliwia tworzeniee intuicyjnych, skalowalnych i łatwych w utrzymaniu aplikacji.

Jetpack Compose został wybrany, ponieważ jest rozwiązaniem w pełni zintegrowanym z Androidem, co zapewnia najlepszą optymalizację i wydajność w tworzeniu aplikacji natywnych na tę platformę.

3.4. Material Design

Material Design to język projektowania opracowany przez Google, który definiuje zasady estetyki, interakcji i użyteczności aplikacji. W Jetpack Compose Material Design stanowi podstawę projektowania interfejsów użytkownika, oferując gotowe komponenty, takie jak przyciski, pola tekstowe czy karty, które są zgodne z jego wytycznymi. Dzięki temu możliwe jest szybkie i intuicyjne tworzenie estetycznych, responsywnych i spójnych interfejsów, z łatwą personalizacją wyglądu aplikacji poprzez modyfikację motywów, kolorów i stylów.

3.5. Supabase

Supabase to nowoczesna platforma typu Backend-as-a-Service (backend jako usługa), która umożliwia tworzenie aplikacji z wykorzystaniem bazy danych PostgreSQL. Jest to narzędzie do budowy aplikacji, bez potrzeby zarządzania infrastrukturą serwerową.

Kluczowym elementem Supabase jest integracja z PostgreSQL, która umożliwia dostęp do bazy danych, oferującej funkcje takie jak zaawansowane zapytania SQL, funkcje typu trigger oraz bezpieczeństwo na poziomie wiersza (RLS). Supabase automatycznie tworzy RESTful API na podstawie tabel w bazie danych, co pozwala na szybkie wdrażanie aplikacji.

Supabase oferuje również intuicyjne narzędzia do zarządzania bazą danych, które umożliwiają łatwe projektowanie i modyfikowanie struktury tabel oraz relacji między nimi. Dzięki wbudowanemu panelowi administracyjnemu, użytkownicy mogą przeglądać dane, zarządzać użytkownikami oraz monitorować aktywność w bazie danych, co znacząco upraszcza proces rozwoju aplikacji.

3.6. Cloudinary

Cloudinary to platforma do zarządzania multimediami w chmurze, która umożliwia przechowywanie, optymalizację i dostarczanie obrazów, wideo oraz innych plików multimedialnych. Dzięki zaawansowanym funkcjom, takim jak automatyczna optymalizacja, zmiana rozmiaru, kadrowanie i konwersja formatów, Cloudinary pozwala na łatwe dostosowanie zasobów do różnych urządzeń i platform.

3.7. GitHub

GitHub to powszechnie używana platforma do zarządzania kodem źródłowym i współpracy w zespołach programistycznych, oparta na systemie kontroli wersji Git. Umożliwia śledzenie zmian w kodzie, zarządzanie historią projektu oraz łatwą współpracę wielu programistów nad jednym projektem.

4. Architektura bazy danych Supabase

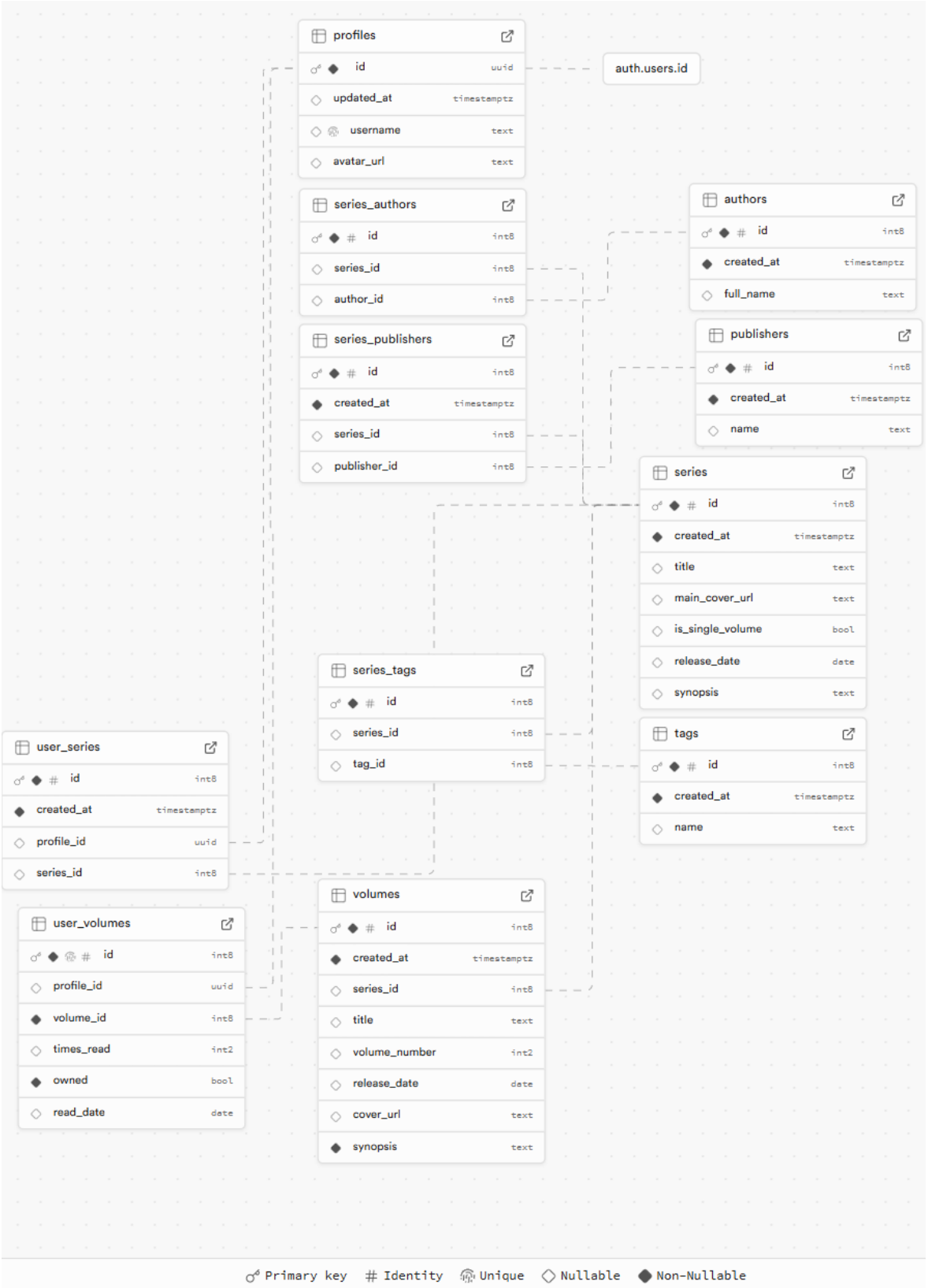
W tym rozdziale przedstawiona zostanie struktura bazy danych PostgreSQL i przykładowe funkcje wykorzystywane do pobierania danych z bazy.

4.1. Schemat bazy danych

Schemat Public bazy danych utworzony przez autora składa się z 11 tabel. Tabela profiles zawiera klucz główny public.profiles.id, który odpowiada kluczowi głównemu auth.users.id w tabeli users w schemacie auth utworzonym automatycznie przez Supabase odpowiedzialnym za autentykację użytkownika.

Za główną tabelę bazy danych można określić tabelę series, w której znajdują się informacje o seriach książek. Seria książek w kontekście tego rozwiazania to zbiór powiązanych ze sobą tematycznie lub fabularnie tomów, które stanowią część większego cyklu. Tabela ta zawiera atrybuty opisujące serię, takie jak tytuł, URL okładki czy streszczenie, a także pola wykorzystywane w logice, takie jak is_single_volume, które pozwalają określić, czy dana pozycja stanowi pełnoprawną serię, czy jest to pojedyncza książka, niezwiązana z żadnym cyklem.

W przypadku sytuacji „wiele do wielu” na przykładzie tabeli user_series, rozwiązanie polega na przechowywaniu relacji między użytkownikami a ich obserwowanymi seriami książek. Tabela ta zawiera pola takie jak profile_id i series_id, które umożliwiają przypisanie użytkownika do jednej lub wielu serii książek. Dzięki temu jeden użytkownik może obserwować wiele serii, a jedna seria może być obserwowana przez wielu użytkowników. Więcej bardziej szczegółowych informacji na temat bazy danych jest ukazane na diagramie ERD wygenerowanym w Supabase na rysunku 4.1.



Rysunek 4.1: Schemat bazy danych

4.2. Normalizacja

Normalizacja bazy danych to proces organizowania danych w taki sposób, aby zminimalizować redundancję i zapewnić integralność danych. Celem jest poprawienie struktury bazy w sposób, który ułatwia jej zarządzanie i utrzymanie, eliminując potencjalne problemy związane z nieefektywnym przechowywaniem informacji.

4.2.1. Pierwsza postać normalna

Pierwsza postać normalna (1NF) na podstawie kodu z listingu 1

```
1     CREATE TABLE public.series (
2       id bigint GENERATED BY DEFAULT AS IDENTITY NOT NULL,
3       title text NULL,
4       ...
5       CONSTRAINT Series_pkey PRIMARY KEY (id)
6       );
```

Listing 1: kod tworzenia tabeli series

Tabela series spełnia pierwszą postać normalną, ponieważ kolumny takie jak title zawierają atomowe wartości, a klucz główny (id) zapewnia unikalność każdego rekordu,

4.2.2. Druga postać normalna

Druga postać normalna (2NF) na podstawie kodu z listingu 2

```
1   CREATE TABLE public.volumes (
2     id bigint GENERATED BY DEFAULT AS IDENTITY NOT NULL,
3     series_id bigint NULL,
4     title text NULL,
5     ...
6     CONSTRAINT volumes_pkey PRIMARY KEY (id),
7     CONSTRAINT volumes_series_id_fkey FOREIGN KEY (series_id)
    REFERENCES series(id)
8     );
```

Listing 2: kod tworzenia tabeli volumes

Tabela volumes spełnia drugą postać normalną, ponieważ jest w pierwszej postaci normalnej oraz wszystkie kolumny, które nie są częścią klucza głównego, są w pełni zależne od klucza głównego (id). Kolumna series_id jest kluczem obcym, a kolumny jak title zależą od id, co eliminuje zależności częściowe.

4.2.3. Trzecia postać normalna

Trzecia postać normalna (3NF) na podstawie kodu z listingu 3

```
1   CREATE TABLE public.user_series (
2     id bigint GENERATED BY DEFAULT AS IDENTITY NOT NULL,
3     profile_id uuid NULL,
```

```
    series_id bigint NULL,
    ...
    CONSTRAINT user_series_pkey PRIMARY KEY (id),
    CONSTRAINT user_series_profile_id_fkey FOREIGN KEY (
  profile_id) REFERENCES profiles(id),
    CONSTRAINT user_series_series_id_fkey FOREIGN KEY (series_id
  ) REFERENCES series(id)
);
```

Listing 3: kod tworzenia tabeli user_series

Tabela user_series spełnia trzecią postać normalną, ponieważ jest w drugiej postaci normalnej, a dodatkowo nie występują zależności przejściowe. Kolumny, które nie są częścią klucza głównego, zależą tylko od klucza głównego (id), a nie od innych kolumn, co zapewnia brak zbędnych zależności między danymi. Tabela ta spełnia również BCNF, ponieważ każda kolumna w niej zależy bezpośrednio od głównego identyfikatora (id).

4.2.4. Czwarta postać normalna

Czwarta postać normalna (4NF) na podstawie kodu z listingu 4

```
    CREATE TABLE public.series_authors (
    id bigint GENERATED BY DEFAULT AS IDENTITY NOT NULL,
    series_id bigint NULL,
    author_id bigint NULL,
    CONSTRAINT series_author_pkey PRIMARY KEY (id),
    CONSTRAINT series_author_author_id_fkey FOREIGN KEY (
  author_id) REFERENCES authors(id) ON UPDATE CASCADE ON DELETE
    CASCADE,
    CONSTRAINT series_author_series_id_fkey FOREIGN KEY (
  series_id) REFERENCES series(id) ON UPDATE CASCADE ON DELETE
   CASCADE
);
```

Listing 4: kod tworzenia tabeli series_authors

Tabela series_authors spełnia czwartą postać normalną, ponieważ nie zawiera zależności wielowartościowych. Relacja między series_id a author_id jest wyrażona w sposób atomowy, gdzie każda kombinacja serii i autora jest reprezentowana przez pojedynczy wiersz. Tabela nie wprowadza redundancji związanej z wieloma wartościami przypisanymi do jednej kolumny.

4.3. Funkcje bazy danych

Funkcje bazy danych stanowią ważny element w zarządzaniu danymi, umożliwiając efektywne wykonywanie różnych operacji w obrębie samej bazy danych. Funkcje

PostgreSQL pozwalają na grupowanie złożonych zapytań, usprawnienie przetwarzania danych oraz zwiększenie wydajności operacji. Dzięki funkcjom bazy danych możliwe jest wykonywanie operacji specyficznych dla aplikacji bezpośrednio na poziomie bazy danych, co pozwala na zmniejszenie obciążenia aplikacji i serwera.

Kod przykładowej funkcji bazy danych znajduje się w listingu 5

```
1   CREATE OR REPLACE FUNCTION public.get_volume_by_id(
    p_volume_id bigint)
2   RETURNS TABLE(id bigint, title text, cover_url text,
    volume_number smallint, user_volume_id bigint, release_date
    timestamp with time zone, times_read smallint, owned boolean,
    synopsis text, read_date timestamp with time zone)
3   LANGUAGE plpgsql
4    AS $function$
5    BEGIN
6      RETURN QUERY
7      SELECT
8        v.id,
9        v.title,
10       v.cover_url,
11       v.volume_number,
12       uv.id AS user_volume_id,
13       v.release_date,
14       COALESCE(uv.times_read, 0) AS times_read,
15       COALESCE(uv.owned, false) AS owned,
16       v.synopsis,
17       uv.read_date
18     FROM
19       volumes v
20     LEFT JOIN
21       user_volumes uv ON v.id = uv.volume_id
22     WHERE
23       v.id = p_volume_id;
24   END;
25   $function$;
26
```

Listing 5: kod funkcji get_volume_by_id

funkcja ta pozwala na podstawie identyfikatora pobrać z bazy danych informacje o wybranym tomie książki (volume), z bazy danych. Funkcja zwraca wynik w postaci teabeli zawierającej dane o tomie oraz dodatkowe informacje związane z użytkownikiem jeżeli są dostępne, takie jak pobrane z tabeli user_volumes, times_read informujące o tym ile razy użytkownik przeczytał dany tom. Może wystąpić sytuacja, w której pobierany tom nie ma odpowiadającego rekordu w tabeli user_volumes, ponieważ użytkownik nie oznaczył go jako przeczytanego ani posiadanego. W takim przypadku dane użytkownika zostaną uzupełnione domyślnymi wartościami, takimi jak 0 dla liczby przeczytań czy false dla statusu posiadania, co jest zrealizowane z

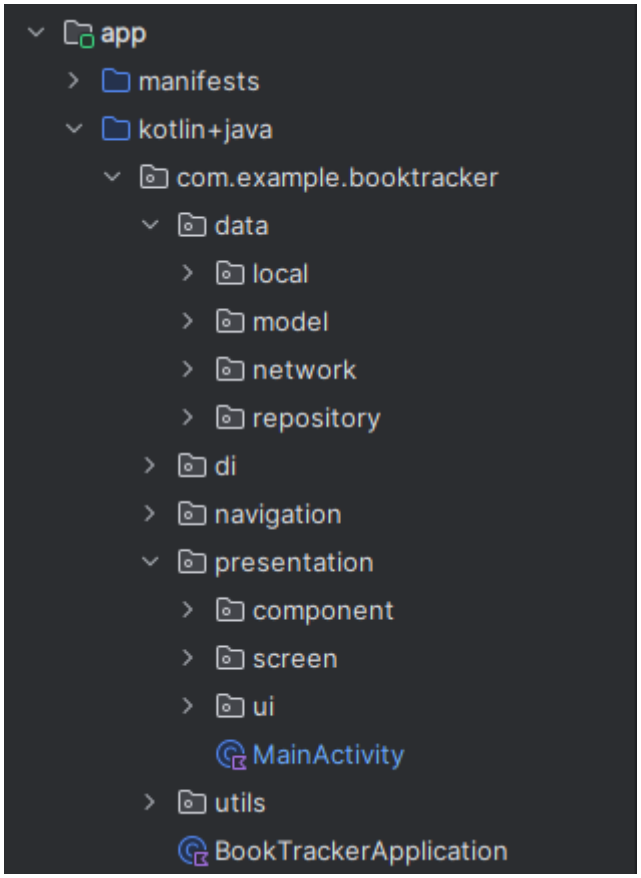
wykorzystaniem funkcji COALESCE, której użycie można zaobserować w liniach 14 i 15.

5. Architektura i struktura aplikacji

Architektura aplikacji została oparta na wzorcu MVVM, co pozwala na efektywne zarządzanie stanem i separację logiki biznesowej od interfejsu użytkownika. Właściwa organizacja plików w projekcie jest kluczowa dla utrzymania porządku i skalowalności aplikacji, umożliwiając łatwe zarządzanie komponentami i ich zależnościami.

5.1. Struktura projektu

Aplikacja została wykonana z wykorzystaniem narzędzia Android Studio, a struktura projektu została zaprojektowana zgodnie z rekomendowanymi zasadami architektury MVVM (Model-View-ViewModel), co zapewnia czytelność, modułowość oraz łatwość w utrzymaniu i rozwoju kodu. Struktura plików projektu przedstawiona jest na rysunku 5.2



Rysunek 5.2: Struktura plików aplikacji

5.1.1. Folder data

Folder "data"pełni rolę zbiorczą dla klas i funkcji odpowiedzialnych za zarządzanie danymi, w tym ich przechowywanie oraz pobieranie z bazy danych.

a) Folder "local"jest przeznaczony do przechowywania danych lokalnych, takich jak tokeny autoryzacyjne, ustawienia aplikacji czy inne informacje, które nie wymagają zdalnej synchronizacji, z wykorzystaniem biblioteki DataStore, która zapewnia bezpieczne i efektywne przechowywanie danych w formie klucz-wartość lub preferencji,

b) Folder "model"przechowuje klasy DTO (Data Transfer Object), które są serializowanymi obiektami wykorzystywanymi do przechowywania danych pobranych z bazy danych. Klasom tym przypisuje się odpowiednie struktury, które umożliwiają prawidłowe reprezentowanie i manipulowanie danymi w aplikacji,

c) w folderze "network"przechowywane są klasy DAO (Data Access Object), które odpowiadają za komunikację z bazą danych Supabase. Klasy te pełnią rolę abstrakcji, umożliwiając wykonywanie operacji na danych, takich jak pobieranie, zapisywanie czy aktualizowanie rekordów, bezpośrednio w bazie. Dzięki zastosowaniu wzorca DAO, operacje te są oddzielone od logiki aplikacji, co zapewnia lepszą modularność, łatwiejsze zarządzanie dostępem do danych oraz umożliwia łatwiejszą wymianę technologii dostępu do danych w przyszłości,

d) Folder "repository"jest odpowiedzialny za przechowywanie klas repozytoriów, które pełnią rolę pośredników pomiędzy warstwą danych a resztą aplikacji. Repozytoria w tym folderze odpowiadają za realizację operacji dostępu do różnych źródeł danych. Oddzielają one logikę biznesową aplikacji od szczegółów technicznych, zapewniając łatwiejsze zarządzanie danymi oraz umożliwiając przyszłe zmiany w źródłach danych bez wpływu na resztę aplikacji. Repozytoria ułatwiają testowanie, zapewniają modularność aplikacji i poprawiają jej elastyczność, umożliwiając łatwą wymianę technologii dostępu do danych.

5.1.2. Folder di

Folder "di"służy do przechowywania modułów odpowiedzialnych za konfigurację zależności aplikacji. Wykorzystuje bibliotekę Hilt, która umożliwia definiowanie

sposobu tworzenia i dostarczania wymaganych obiektów, takich jak klienty API, repozytoria czy inne kluczowe komponenty. Moduły w tym folderze zapewniają centralne miejsce do zarządzania zależnościami, co ułatwia organizację projektu i poprawia czytelność kodu.

5.1.3. Folder navigation

Folder "navigation"zarządza całą logiką nawigacji w aplikacji, co umożliwia łatwe i efektywne przejścia pomiędzy ekranami. Zawiera definicję ekranów jako obiektów typu sealed class, co pozwala na centralne zarządzanie wszystkimi dostępnymi widokami w aplikacji.

Dzięki wykorzystaniu biblioteki navigation-compose, folder "navigation"umożliwia łatwą konfigurację tras i przejść, eliminując konieczność ręcznego zarządzania stanem nawigacji. Obsługuje różne scenariusze, takie jak logowanie, rejestracja, czy przeglądanie biblioteki, zapewniając płynność i spójność w nawigacji. W połączeniu z komponentem NavHost, aplikacja może efektywnie zarządzać stanem nawigacyjnym, przechodzić między ekranami i dostosowywać animacje przejść. Dzięki tej strukturze, aplikacja jest bardziej modularna, a logika nawigacyjna pozostaje łatwa do utrzymania i rozbudowy.

5.1.4. Folder presentation

Folder "presentation"zawiera pliki odpowiadające za warstwę prezentacji aplikacji. Znajdują się w nim zarówno widoki, które definiują interfejs użytkownika, jak i modele widoków, które dostarczają dane i logikę potrzebną do poprawnego działania widoków. Elementy te współpracują ze sobą, aby zapewnić spójne i przejrzyste doświadczenie użytkownika.

Folder "component"zawiera funkcje kompozycyjne, które są wykorzystywane w więcej niż jednym widoku. Przykład takiej funkcji przedstawiono w poniższym listingu 6.

```
1  @Composable
2  fun SeriesProgressIndicator(ownedProgress: Float,
    readingProgress: Float, height: Dp) {
3    Box(
4      modifier = Modifier
5        .fillMaxWidth()
6        .height(height)
7        .background(MaterialTheme.colorScheme.surfaceVariant)
8    ) {
9      Box(
10       modifier = Modifier
11         .fillMaxWidth(ownedProgress)
```

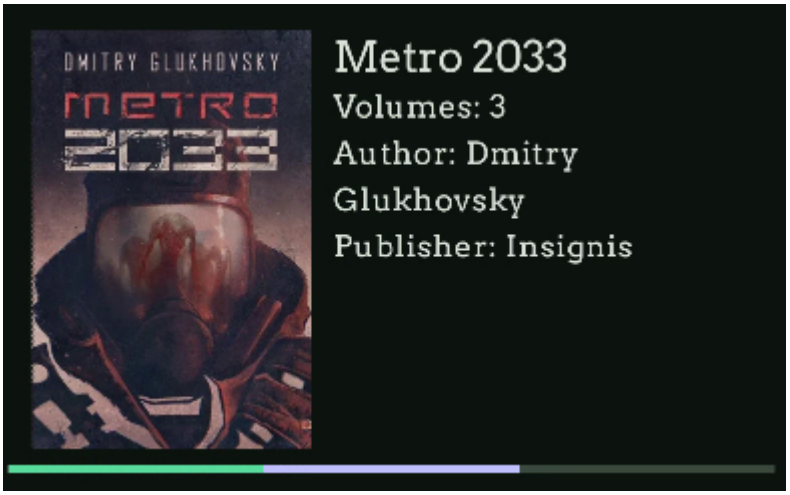
```
            .height(height)
            .background(
              MaterialTheme.colorScheme.tertiary
            )
        )

        Box(
          modifier = Modifier
            .fillMaxWidth(readingProgress)
            .height(height)
            .background(
              MaterialTheme.colorScheme.primary
            )
        )
    }
}
```

Listing 6: kod funkcji SeriesProgressIndicator

Kod odpowiada za wyświetlanie paska postępu dla danej serii, który pokazuje zarówno postęp posiadanych, jak i aktualnie czytanych książek. Pasek jest wyświetlany w formie dwóch nałożonych na siebie prostokątów o różnych szerokościach, zależnych od wartości ownedProgress i readingProgress. Kod paska postępu znajduje się w folderze "component", ponieważ jest on wykorzystywany przez komponent kart na ekranie biblioteki i w nagłówku serii, co widać na poniższym rysunku 5.3

(a) Karta serii



(b) Nagłówek ekranu serii

Rysunek 5.3: SeriesProgressIndicator wykorzystany w różnych komponentach

Kolejnym folderem jest folder "screen", który zawiera podfoldery dla poszczególnych ekranów. Każdy z tych podfolderów zawiera plik główny kodu ekranu, a opcjonalnie model widoku oraz folder "component"do przechowywania komponentów specy-

ficznych dla danego ekranu.

Ostatni folder, "ui", dzieli się na dwa podfoldery: "theme", w którym przechowywane są pliki definiujące style kolorów, typografii oraz motywu aplikacji, oraz "viewmodel", zawierający pliki modeli widoków, które są współdzielone między kilkoma ekranami.

5.1.5. Folder utils

Folder utils zawiera funkcje pomocnicze, które wspierają różne operacje w aplikacji, takie jak walidacja danych wejściowych. Funkcje te są uniwersalne i mogą być wykorzystywane w różnych częściach projektu, co pozwala na unikanie powielania kodu. Dzięki temu zapewniają one centralizację logiki, co ułatwia utrzymanie aplikacji, poprawia jej czytelność oraz umożliwia łatwiejsze testowanie. Przykłady funkcji w tym folderze to metody sprawdzające poprawność adresu e-mail, hasła czy nazwy użytkownika. Wszystkie funkcje w utils są zaprojektowane w sposób modularny, co pozwala na ich elastyczne zastosowanie w całym projekcie. Przykładowa funkcja znajduje się listingu 7

```
1  fun validateEmail(email: String, context: Context): String? {
2      val emailRegex = "^[A-Za-z0-9+_.-]+@[A-Za-z0-9.-]+\$".
   toRegex()
3          return if (email.isEmpty()) {
4              context.getString(R.string.email_empty)
5          } else if (!email.matches(emailRegex)) {
6              context.getString(R.string.invalid_email)
7          } else null
8      }
```

Listing 7: kod funkcji validateEmail

5.2. Wstrzykiwanie zależności (Dependency Injection)

Wstrzykiwanie zależności (ang. Dependency Injection) jest jednym z kluczowych wzorców projektowych, który pomaga w utrzymaniu czystości kodu, jego modularności i testowalności. Wstrzykiwanie zależności odgrywa istotną rolę w zarządzaniu zależnościami między komponentami aplikacji, takimi jak modele widoków, repozytoria, czy usługi zewnętrzne.

Deklaratywne tworzenie interfejsu użytkownika w Jetpack Compose znacząco ułatwia pracę w tworzeniu aplikacji. Jednak wraz z rosnącą złożonością, pojawia się potrzeba efektywnego zarządzania zależnościami, szczególnie w kontekście komponen-

tów, które muszą współpracować ze sobą. W tym celu wykorzystuje się narzędzia do wstrzykiwania zależności, takie jak Hilt. Jest to nowoczesne narzędzie, które upraszcza proces zarządzania zależnościami w aplikacjach Android, eliminując konieczność ręcznego tworzenia i przekazywania obiektów. Hilt bazuje na popularnym rozwiązaniu o nazwie Dagger, które jest wykorzystywane do automatycznego i efektywnego zarządzania zależnościami w dużych aplikacjach. Dzięki Hilt, integracja z Androidem staje się łatwiejsza, a proces konfiguracji i wstrzykiwania zależności bardziej intuicyjny i mniej czasochłonny.

5.2.1. Konfiguracja Hilt

Konfiguracja Hilt dla tworzonej aplikacji polega na uruchomieniu kontenera wstrzykiwania zależności przez dodanie adnotacji @HiltAndroidApp do klasy aplikacji. Adnotacja ta informuje Hilt o konieczności wygenerowania kodu potrzebnego do zarządzania zależnościami w całej aplikacji. Przykład konfiguracji został zaprezentowany w poniższym listingu 8:

```
1  @HiltAndroidApp
2    class BookTrackerApplication: Application() {
3    }
```

Listing 8: Konfiguracja Hilt w aplikacji

Dzięki zastosowaniu @HiltAndroidApp, Hilt automatycznie uruchamia kontener DI (Dependency Injection) podczas startu aplikacji, umożliwiając wstrzykiwanie zależności w różnych komponentach, takich jak modele widoków czy funkcje kompozycyjne.

Adnotacja @AndroidEntryPoint jest kluczowym elementem konfiguracji Hilt. Informuje ona system, że dana klasa powinna być objęta mechanizmem wstrzykiwania zależności. Dzięki temu Hilt automatycznie przygotowuje kontener DI i udostępnia zadeklarowane zależności, które można wstrzykiwać za pomocą adnotacji @Inject. Kod konfiguracji tej adnotacji w klasie głównej aplikacji został zaprezentowany w listingu 9.

```
1  @AndroidEntryPoint
2    class MainActivity : ComponentActivity() {...}
```

Listing 9: Przykład użycia @AndroidEntryPoint w MainActivity

Dzięki tej konfiguracji Hilt może automatycznie wstrzykiwać wszystkie zależności w komponentach podrzędnych, używanych w aplikacji. Ułatwia to zarządzanie złożonością kodu oraz umożliwia bardziej zwięzłą implementację aplikacji.

5.2.2. Moduły Hilt

Jednym z kluczowych elementów Hilt jest definiowanie modułów za pomocą adnotacji @Module. Moduły te pozwalają precyzyjnie określić, jakie obiekty powinny być dostarczane i w jakim cyklu życia mają istnieć. W poniższym listingu 10 znajduje się przykład definiowania modułu dostarczającego klienta Supabase.

```
1  @InstallIn(SingletonComponent::class)
2  @Module
3  object SupabaseModule {
4    @Provides
5    @Singleton
6    fun provideSupabaseClient(): SupabaseClient {
7        return createSupabaseClient(
8            supabaseUrl = BuildConfig.SUPABASE_URL,
9            supabaseKey = BuildConfig.API_KEY
10       ) {
11           install(Auth)
12           install(Postgrest)
13       }
14   }
15
16   @Provides
17   @Singleton
18   fun provideSupabaseAuth(client: SupabaseClient): Auth {
19       return client.auth
20   }
21 }
```

Listing 10: Przykład modułu klienta Supabase

Adnotacja @InstallIn(...) określa, gdzie dany moduł będzie instalowany, a przez to, gdzie dostarczane zależności będą dostępne. Przykładowo, @InstallIn(SingletonComponent::class) z linii pierwszej wskazuje, że zależności zdefiniowane w module będą współdzielone w całej aplikacji. Adnotacja @Provides Informuje Hilt, jak utworzyć daną zależność. W powyższym przykładzie funkcja provideSupabaseClient() definiuje, jak stworzyć instancję SupabaseClient. Adnotacja @Singleton zapewnia, że obiekt będzie tworzony tylko raz podczas działania aplikacji.

Innym przykładem zależności, którą można wstrzykiwać do różnych komponentów aplikacji, są lokalnie przechowywane dane zarządzane za pomocą DataStore. Umożliwia to scentralizowane zarządzanie preferencjami użytkownika oraz łatwy dostęp do nich w różnych częściach aplikacji. Moduł dla DataStore definiuje sposób dostarczania obiektu zarządzającego preferencjami użytkownika i jest skonfigurowany do działania w kontekście całej aplikacji. Kod tego modułu znajduje się w listingu 11.

```
1 @Module
```

```
@InstallIn(SingletonComponent::class)
class DataStoreModule {
    @Provides
    @Singleton
    fun provideUserPreferences(@ApplicationContext context:
   Context): UserPreferences {
        return UserPreferences(context)
    }
}
```

Listing 11: Przykład modułu DataStore

W aplikacjach opartych na wzorcu repozytorium, zarządzanie danymi odbywa się poprzez interfejs, który określa operacje, jakie mogą być wykonywane na danych. Implementacja tego interfejsu dostarcza konkretne mechanizmy do pozyskiwania danych z różnych źródeł. Wstrzykiwanie zależności w Hilt pozwala na łatwą konfigurację repozytorium, umożliwiając podmianę jego implementacji, co może być przydatne podczas testów lub w sytuacjach, gdzie różne źródła danych wymagają różnych implementacji. Przykład implementacji modułu repozytorium znajduje się w listingu 12

```
  @InstallIn(SingletonComponent::class)
  @Module
  abstract class SeriesRepositoryModule {
    @Binds
    @Singleton
    abstract fun bindSeriesRepository(impl: SeriesRepositoryImpl
   ) : SeriesRepository
  }
```

Listing 12: Przykład modułu repozytorium SeriesRepository

5.2.3. Przykłady wstrzykiwania zależności z Hilt

Repozytorium to kluczowy komponent, który abstrahuje logikę uwierzytelniania od reszty aplikacji. W AuthenticationRepositoryImpl wykorzystujemy wcześniej zdefiniowane zależności, takie jak Auth i UserPreferences. Dzięki adnotacji @Inject Hilt automatycznie dostarcza te zależności. Kod repozytorium znajduje się w listingu 13:

```
  class AuthenticationRepositoryImpl @Inject constructor(
    private val auth: Auth,
    private val userPreferences: UserPreferences
) : AuthenticationRepository {

    override suspend fun signIn(email: String, password: String)
   : Boolean {
        return runCatching {
            auth.signInWith(Email) {
                this.email = email
                this.password = password
```

```
            }
            true
        }.getOrDefault(false)
    }

    override suspend fun saveToken() {
        val accessToken = auth.currentAccessTokenOrNull()
        if (accessToken != null) {
            userPreferences.saveUserSession(accessToken)
        }
    }

    ...

}
```

Listing 13: Implementacja AuthenticationRepository

Powyższy kod przedstawia definicję klasy AuthenticationRepositoryImpl, w której wstrzykiwane są zależności: klient auth z modułu SupabaseModule oraz preferencje użytkownika z modułu DataStoreModule. Na przykładzie dwóch metod pokazano, jak można korzystać z tych zależności po ich wstrzyknięciu. Metoda signIn wykorzystuje klienta auth do obsługi logowania użytkownika, natomiast metoda saveToken zapisuje token sesji w preferencjach użytkownika przy użyciu userPreferences.

AuthViewModel wykorzystuje AuthenticationRepository, które zostało wstrzyknięte za pomocą Hilt. Dzięki temu ViewModel jest odpowiedzialny wyłącznie za przetwarzanie danych dla interfejsu użytkownika, bez konieczności zarządzania szczegółami implementacyjnymi repozytorium. Kod ViewModel przedstawiono w listingu 14:

```
@HiltViewModel
class AuthViewModel @Inject constructor(
  private val authenticationRepository:
 AuthenticationRepository
): ViewModel() {
  fun onSignIn() {
      viewModelScope.launch{
          _signInResult.value = authenticationRepository.
 signIn(
              email = _email.value,
              password = _password.value
          )
          if (_signInResult.value!!) {
              authenticationRepository.saveToken()
          }
      }
  }
  ...
}
```

Listing 14: Implementacja AuthViewModel

Dzięki Hilt, AuthViewModel otrzymuje AuthenticationRepository bez konieczności jego ręcznego tworzenia. To pozwala na oddzielenie logiki aplikacji od szczegółów implementacyjnych repozytorium. W przypadku potrzeby zmiany na inne źródło danych wystarczy zmienić tylko moduł Hilt. AuthViewModel pozostałby bez zmian, ponieważ nie jest odpowiedzialny za tworzenie instancji repozytorium.

Funkcja SignInScreen odpowiada za wyświetlenie ekranu logowania. Dzięki funkcji hiltViewModel() instancja AuthViewModel jest automatycznie tworzona i dostarczana przez Hilt. W ten sposób uniknięto ręcznego zarządzania cyklem życia ViewModel, a kod staje się bardziej przejrzysty. Kod funkcji znajduje się w listingu 15:

```
@Composable
fun SignInScreen(
  toSignUpScreen: (() -> Unit),
  toHomeScreen: (() -> Unit),
  signInViewModel: AuthViewModel = hiltViewModel(),
) {
  ...
}
```

Listing 15: Implementacja AuthViewModel

W tej funkcji signInViewModel jest wykorzystywany do zarządzania stanem ekranu i obsługi zdarzeń, takich jak próba logowania. Dzięki temu kod w funkcji Composable pozostaje skupiony na interfejsie użytkownika.

5.3. MVVM (Model-View-ViewModel)

Model-View-ViewModel to popularny wzorzec architektoniczny stosowany w tworzeniu aplikacji, który pomaga w organizacji kodu i rozdzieleniu odpowiedzialności pomiędzy różne warstwy aplikacji. Jego głównym celem jest zwiększenie modularności, łatwości testowania oraz oddzielenie logiki biznesowej od interfejsu użytkownika.

a) Model reprezentuje warstwę danych i logiki biznesowej. Jest odpowiedzialny za zarządzanie danymi, które mogą pochodzić z różnych źródeł, takich jak bazy danych, API czy pliki lokalne. Model nie zawiera żadnej logiki związanej z interfejsem użytkownika ani sposobem prezentacji danych,

b) View (Widok) odpoawiada za warstwę prezentacji. Widok to interfejs użytkownika (UI), który jest odpowiedzialny za wyświetlenie danych i odbieranie interakcji od użytkownika. Widok powinien jedynie reagować na dane dostarczane przez ViewModel. W android widokiem są komponenty Jetpack Compose,

c) ViewModel to warstwa pomiędzy modelem a widokiem. pobiera dane z modelu i przekształca je w taki sposób, aby były gotowe do wyświetlenia w widoku. Ponadto zarządza stanem widoku (np. przechowywaniem stanu aplikacji w przypadku zmiany orientacji ekranu).

5.3.1. Model

Data Transfer Object (DTO) to wzorzec projektowy, który służy do przenoszenia danych między różnymi warstwami aplikacji. DTO są zazwyczaj prostymi obiektami, które zawierają dane, ale nie posiadają logiki biznesowej. Głównym celem DTO jest przenoszenie danych w sposób efektywny i umożliwiający łatwe zarządzanie. Przykładowy kod DTO jest zaprezentowany w poniższym listingu 16

```
@Serializable
data class Series(
    val id: Int,
    val created_at: String,
    val title: String,
    val main_cover_url: String,
    val is_single_volume: Boolean,
    val release_date: LocalDate,
    val synopsis: String,
    var total_volumes_released: Int,
    var is_following: Boolean
)
```

Listing 16: Data Transfer Object Series

Powyższy kod przedstawia klasę Series. Zawiera ona podstawowe informacje o serii książek, takie jak unikalny identyfikator (id), tytuł serii (title), data wydania (release_date), streszczenie fabuły (synopsis), a także dodatkowe informacje jak liczba wydanych tomów (total_volumes_released) oraz status obserwowania serii przez użytkownika (is_following). Wszystkie te dane są przechowywane w formacie, który może być łatwo przesyłany między różnymi warstwami aplikacji, takimi jak repozytoria, ViewModel czy interfejs użytkownika.

Klasa Series jest oznaczona adnotacją @Serializable, co sprawia, że może być łatwo przekształcana do i z formatu JSON. Serializacja jest kluczowa, ponieważ w przypadku aplikacji komunikujących się z zewnętrznymi API, jak np. Supabase, dane są często przesyłane w formacie JSON. Aby odpowiedź z API mogła zostać zamapowana na obiekt w Kotlinie, musi być ona odpowiednio zserializowana.

Warstwa dostępu do danych (DAO - Data Access Object) odpowiada za komunikację z bazą danych lub zewnętrznymi źródłami danych, takimi jak API. Zastosowanie

wzorca DAO pozwala na izolowanie logiki dostępu do danych od reszty aplikacji, co ułatwia zarządzanie kodem, testowanie oraz utrzymanie aplikacji. Klasa DAO odpowiada za wykonanie operacji związanych z pobieraniem, zapisywaniem lub aktualizowaniem danych w bazie danych lub serwisie zewnętrznym.

6. Podsumowanie

Autor za wkład własny uważa stworzenie struktury bazy danych, funkcji PostgreSQL do pobierania z niej danych i zaprogramowanie aplikacji na system android

Literatura

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

STRESZCZENIE PRACY DYPLOMOWEJ INŻYNIERSKIEJ

KSIĄŻKOWY DZIENNIK W CHMURZE

Autor: Rafał Stępień, nr albumu: EF-169625
Opiekun: dr inż. Mariusz Mączka
Słowa kluczowe: (max. 5 słów kluczowych w 2 wierszach, oddzielanych przecinkami)

Treść streszczenia po polsku

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

BSC THESIS ABSTRACT

CLOUD-BASED BOOK JOURNAL

Author: Rafał Stępień, nr albumu: EF-169625
Supervisor: Dr. Eng. Mariusz Mączka
Key words: (max. 5 słów kluczowych w 2 wierszach, oddzielanych przecinkami)

Treść streszczenia po angielsku