

AKADEMIA NAUK STOSOWANYCH W NOWYM SĄCZU

Wydział Nauk Inżynierjnych
Katedra Informatyki

DOKUMENTACJA PROJEKTOWA PROGRAMOWANIE URZĄDZEŃ MOBILNYCH

Raptor

Autor:
Mateusz Stanek
Dawid Szołdra
Filip Wąchała

Prowadzący:
mgr inż. Dawid Kotlarski

Nowy Sącz 2024

Spis treści

1. Ogólne określenie wymagań projektu	5
1.1. Ogólny zarys wymagań	5
1.2. Wykorzystane czujniki	5
1.3. Zarys interfejsu	5
2. Określenie wymagań szczegółowych	8
2.1. Ogólny opis wymagań projektu	8
2.2. Ogólny zarys narzędzi użytych w projekcie	8
2.2.1. Android Studio	8
2.2.2. Kotlin	9
2.3. Wykorzystanie czujników	10
2.4. Zachowanie w niepożądanych sytuacjach	10
2.5. Dalszy rozwój	10
3. Projektowanie	11
3.1. Założenie programu	11
3.2. Przedstawienie menu	11
3.3. Odczyt i przetwarzanie plików	11
3.4. Struktura bazy danych	11
3.4.1. Ogólny opis	11
3.4.2. Opis pól tabel	12
3.4.2.1. Autorzy	12
3.4.2.2. Albumy	12
3.4.2.3. Piosenki	12
3.4.3. Relacje w bazie	12
3.5. Czujnik światła	13
3.6. Uwierzytelnianie biometrią	13
3.7. Ładowanie obrazów albumów	13
4. Implementacja	15
4.1. Zarządzanie bazą danych	15

4.1.1. Klasa DatabaseManager	15
4.1.2. Klasa LibraryDb	19
4.1.3. Obiekty Dao	19
4.1.4. Tabele	22
4.1.5. Relacje	23
4.1.5.1. AlbumWithSongs	24
4.1.5.3. AlbumWithAuthors i AuthorWithAlbums	24
4.2. Autoryzacja odciskiem palca	28
4.3. Odczyt i przetwarzanie plików	32
4.3.1. Tag Extractor	32
4.3.2. MusicFileLoader	37
4.4. NavHost	41
5. Implementacja	46
5.1. Zarządzanie bazą danych	46
5.1.1. Klasa DatabaseManager	46
5.1.2. Klasa LibraryDb	50
5.1.3. Obiekty Dao	50
5.1.4. Tabele	53
5.1.5. Relacje	54
5.1.5.1. AlbumWithSongs	55
5.1.5.3. AlbumWithAuthors i AuthorWithAlbums	55
5.2. Czujnik światła	56
5.3. Autoryzacja odciskiem palca	59
5.4. Odczyt i przetwarzanie plików	63
5.4.1. Tag Extractor	63
5.4.2. MusicFileLoader	68
5.5. Ładowanie obrazów albumów	72
6. Testowanie	74
7. Podręcznik użytkownika	75
Literatura	96

Spis rysunków	97
Spis tabel	98
Spis listingów	99

1. Ogólne określenie wymagań projektu

1.1. Ogólny zarys wymagań

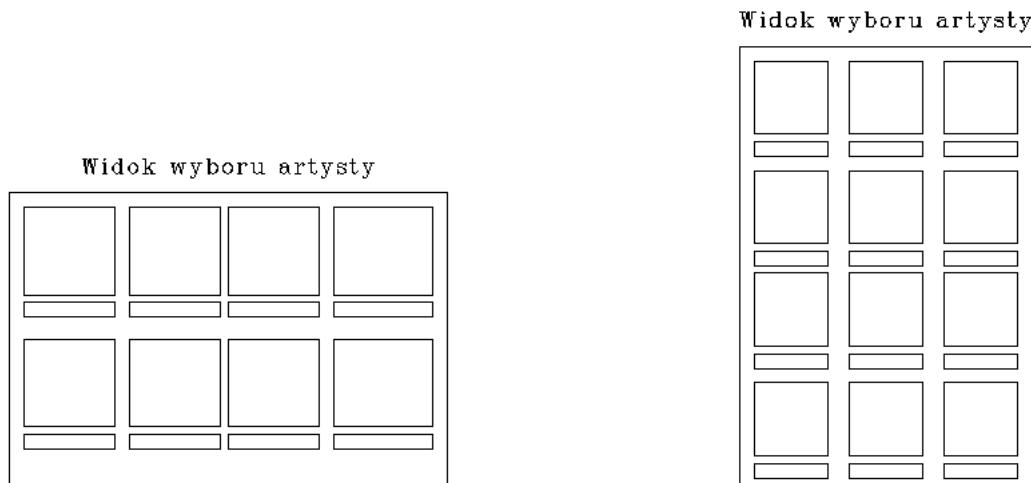
Celem programu jest pełnienie funkcji odtwarzacza muzyki. Program będzie mógł skanować dany folder i jego podfoldery, a w nich zawarty pliki muzyczne i tworzyć na ich podstawie bibliotekę, zapisaną na dysku.

1.2. Wykorzystane czujniki

Program ma na celu wykorzystanie trzech czujników, z którymi użytkownik będzie wchodził w interakcję. Zostaną użyte następujące:

- Żydroskop - Interfejs programu będzie się zmieniał w zależności od orientacji urządzenia.
- Wykrywacz odcisków palca - dostęp do programu powinien być ograniczony dla użytkowników mogących zweryfikować swój odcisk.
- Czujnik światła - Interfejs programu będzie mógł zmieniać swoje kolory w zależności od wykrytego poziomu światła na czujniku

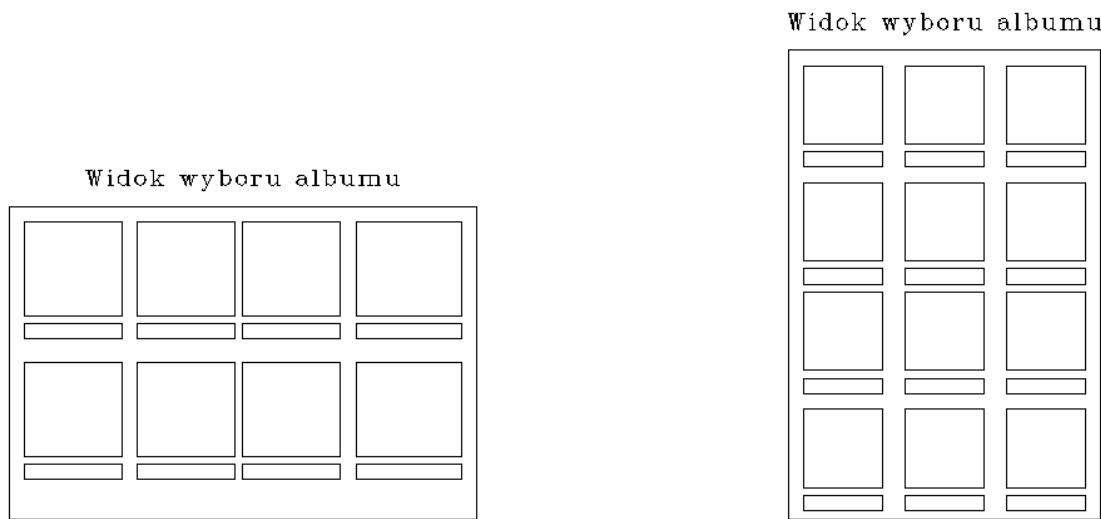
1.3. Zarys interfejsu



Rys. 1.1. Mockup widoku biblioteki - listing wykonawców

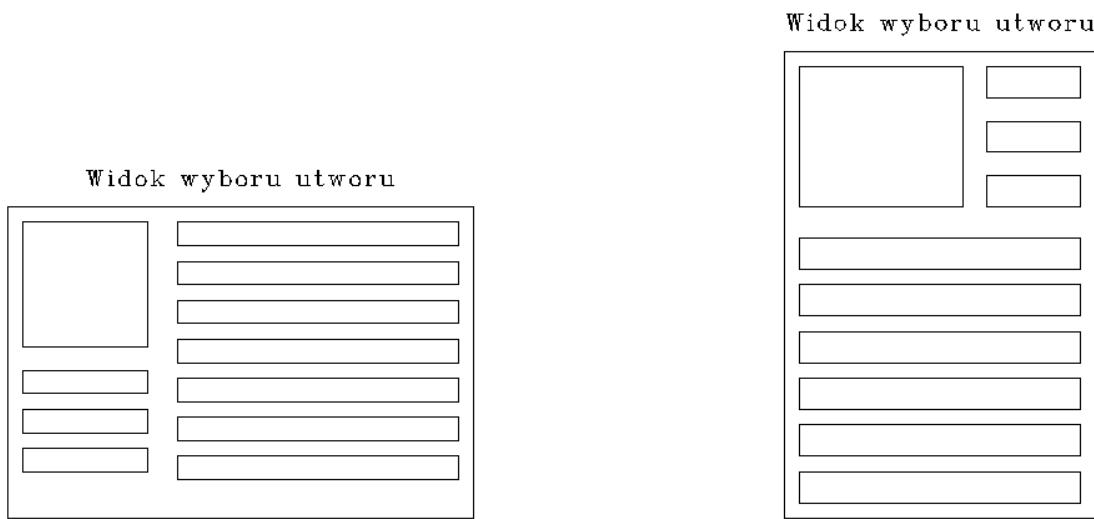
Widok wykonawców, jest przedstawiony na rysunku nr. 1.1. Ten widok będzie ekranem startowym aplikacji. Jako "Kafelki", zwracać będzie się dokument do ułożonych

równomiernie na rysunku kwadratów. Na każdym z nich napisana będzie nazwa danego wykonawcy. Klikanie na jeden z nich przejdzie do widoku albumów danego wykonawcy.



Rys. 1.2. Mockup widoku albumów danego wykonawcy

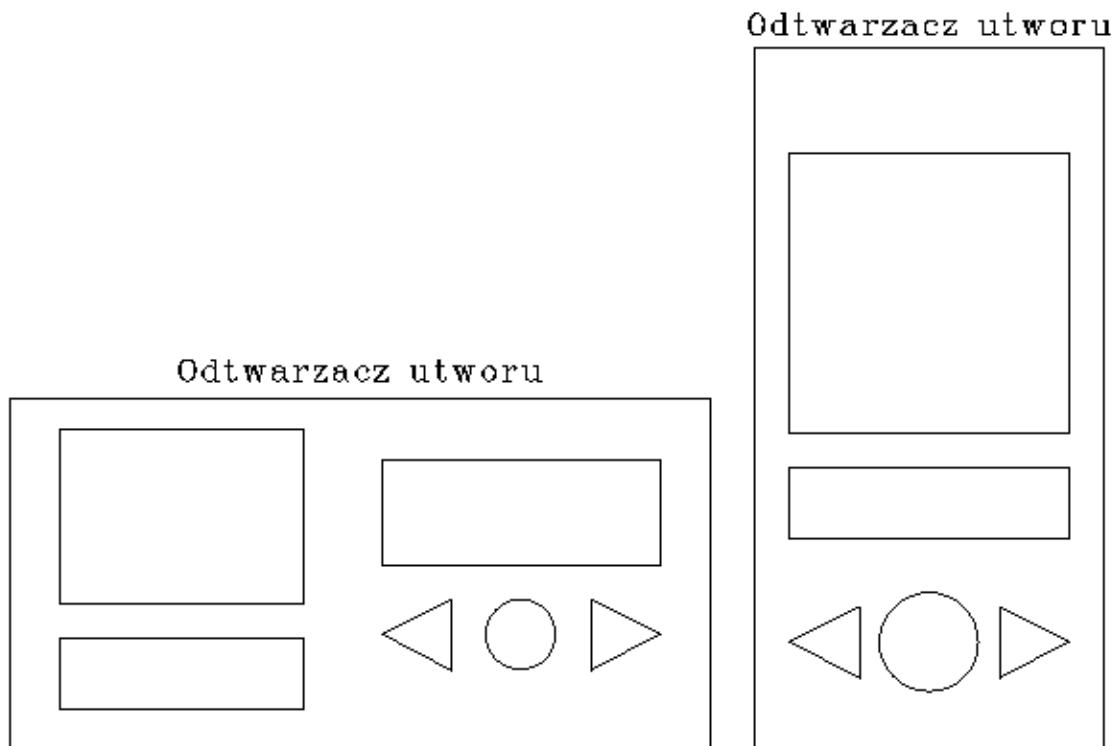
Widok albumów jest przedstawiony na rysunku nr. 1.2. Widok będzie podobny do widoku wykonawców. Różni się on tym, że na "kafelkach", będą pokazane zdjęcia poszczególnych albumów. Pod "kafelkami", znajdują się nazwy danych albumów.



Rys. 1.3. Mockup widoku wyboru utworu

Rysunek nr. 1.3 przedstawia ekran pokazujący się po wybraniu albumu. Po wejściu na jakiś album zaprezentowane zostaną zawarte w nim utwory. W lewym górnym kwadracie to zdjęcie danego albumu, a obok niego jest kilka informacji o albumie jak

wykonawca, data, tytuł, w postaci tekstu. Dłuższe paski zawarte na dole to lista piosenek, w postaci przycisków z napisanymi, tytułami które można kliknąć, aby daną piosenkę włączyć.



Rys. 1.4. Mockup widoku wyboru utworu

Wygląd interfejsu odtwarzacza został zaprezentowany na rysunku nr.???. W widoku horyzontalnym po lewej stronie na górze znajduje się obraz albumu a na dole pod obrazem będzie nazwa utworu, po prawej stronie na górze znajduje się paszek przewijania w formie soundwave który jest wyciągany z pliku muzycznego a na dole pod soundwave znajdują się przyciski pozwalające na manipulację utworem jak np. na zatrzymanie go lub przewinięcie. W widoku horyzontalnym na samym dole znajduje się zdjęcie albumu, poniżej nazwa utworu, potem soundwave a na końcu przyciski manipulacyjne.

2. Określenie wymagań szczegółowych

2.1. Ogólny opis wymagań projektu

Aplikacja jest zaprojektowana w Android Studio w języku Kotlin. Całe UI aplikacji będzie zbudowane na podstawie Frameworka Jetpack Compose^[1]. Używając wbudowanych bibliotek w SDK Androida, będzie mogła odczytywać pliki ze wskazanego folderu. Odczytywanie tagów z plików odbędzie się za pomocą biblioteki Media3^[2]. Jest to oficjalna biblioteka Googlea do obsługi plików medialnych na Androidzie. Wszelki processing audio np. na potrzeby wizualizacji może zostać wykonany za pomocą SDK i wbudowanego modułu AudioProcessor^[3]. Odtwarzaniem pliku będzie zajmowała się biblioteka ExoPlayer^[4], posiadająca częściową integrację z Media3. Informacje o utworach powinny być ładowane do bazy danych. Będzie ona lokalnie, na urządzeniu. Ku temu celu, można użyć biblioteki Room^[5]. Biblioteka ta jest wrapperem do wbudowanych funkcjonalności SQL Androida.

2.2. Ogólny zarys narzędzi użytych w projekcie

2.2.1. Android Studio

Android Studio jest IDE stworzonym przez Google, na bazie IntelliJ IDEA od JetBrains. Jest ono przystosowane, jak z nazwy wynika, do tworzenia aplikacji na Androida. Ku temu celu posiada wiele udogodnień, odróżniających program od typowego edytora jak np. wbudowany emulator Androida, integrujący się z całym środowiskiem, czy preview różnych elementów interfejsu - gdzie elementy te generowane są w kodzie, a nie w osobnym języku jak np. xml - bez potrzeby dekompilacji całej aplikacji.

Android Studio został użyty w projekcie, ponieważ:

- Sam program jest crossplatformowy - nasz zespół używa wielu systemów operacyjnych. Platformy takie jak MAUI, są zespolone z Visual Studio, czyli z Windowsem. Android Studio jest dostępny na wszystkie większe systemy operacyjne, co ułatwia nam pracę.
- Jest to program, zbudowany na podstawie IdeaJ, czyli zagłębiony jest w tym ekosystemie. Oznacza to dostęp do większej ilości pluginów niż np. Visual Studio, nie wspominając o ogólnej możliwości dostosowania ustawień.

Wady korzystania z Android Studio to m.in.

- Duże wykorzystanie zasobów - program lubi zżerać duże ilości RAMu. W tym momencie, mając otwarty mały projekt + emulator, program wykorzystuje ponad 9GB RAMu.

2.2.2. Kotlin

Kotlin został stworzony w 2010 roku przez firmę JetBrains oraz jest on przez nią rozwijany. Kotlin jest wieloplatformowym językiem typowanym statystycznie który został zaprojektowany aby współpracować z maszyną wirtualną Javy. Swoją nazwę zawdzięcza wyspie Kotlin która znajduje się w zatoce finlandzkiej.

Kotlin jest wykorzystywany w projekcie ze względów:

- Jest on wspierany przez Android Studio, razem z Javą i C++. Kotlin ponadto, ma dostęp do nowoczesnych frameworków jak Jetpack Compose
- Jest on *defakto* językiem do programowania na Androida - do niedawna Java mogła cieszyć się tym tytułem, ale od 2019 r. Google ogłosiło Kotlinę jako rekomendowany język do tworzenia aplikacji na Android.

Składnia Kotlina wygląda następująco:

```
1 fun main() {  
2     printf("Czesc to ja, kotlin!")  
3 }
```

Listing 1. kotlin001 - Funkcje

Definicja funkcji wykonywana jest za pomocą "fun".

Zmienne w Kotlinie deklarowane są za pomocą **val** i **var**. Różnica polega na tym, że zmienne oznaczone **val** mogą zostać modyfikowane natomiast zmienne oznaczone **val** już nie.

```
1 fun main() {  
2     var nazwa = "Projekt Android"  
3     val liczba = "777"  
4 }
```

Listing 2. kotlin002 - Zmienne

Kompilator Kotlina posiada funkcję autodedukcji typów, więc w wielu wypadkach typu zmiennej nie trzeba adnotować.

2.3. Wykorzystanie czujników

- Żyroskop - Z racji, że każdy element interfejsu w Jetpack jest generowany kodem, można, przynajmniej na początku, ustawić każdą wersję interfejsu jako osobną funkcję. Następnie, w zależności od wykrytej orientacji, przy użyciu API sensorów[6], można wywoływać odpowiednią funkcję.
- Mikrofon - Funkcja dyktafonu najprawdopodobniej będzie całkiem oddzielnym Activity. Funkcjonalność ta, z natury, jest dosyć oddzielna od reszty aplikacji. Nagrania dyktafonem powinny być zapisywane do osobnego folderu. Można by zintegrować nagrania z resztą aplikacji jako osobnego wykonawcę w widoku biblioteki. Mikrofon będzie nagrywany poprzez moduł MediaRecorder[7]
- Czujnik światła - Android Studio oferuje możliwość definiowania własnych klas zajmujących się kolorystyką. Oznacza to że można używać różnych obiektów w zależności od warunków. Wykrywanie światła będzie się odbywało używając API sensorów[6]

2.4. Zachowanie w niepożądanych sytuacjach

Głównym wyjątkiem, na który może napotkać się aplikacja jest błąd odczytu albo plików, albo tagów z pliku. Kotlin, na szczęście, pozwala na łatwe sprawdzanie wartości null danych zmiennych operatorem ?. W odpowiednich fragmentach kodu dotyczących ładowania plików, będzie sprawdzana poprawność danych i najprawdopodobniej pojawi się pop-up po stronie użytkownika, że wystąpił błąd, a po stronie dewelopera błąd zostanie logowany.

2.5. Dalszy rozwój

Jeżeli praca nad aplikacją będzie się odbywała w przyszłości, należy skupić uwagę na lepszym zarządzaniu biblioteką (auto tagowanie, pobieranie miniatur z internetu, itp.). Ponadto, należy szukać błędów, które nadal zostały w aplikacji.

3. Projektowanie

3.1. Założenie programu

W tym rozdziale przedstawiona zostanie ogólna zasada działania programu.

Głównym celem programu jest odtwarzanie muzyki.

3.2. Przedstawienie menu

Program składa się z trzech okien.

- Okno wyboru autora
- Okno wyboru albumu
- Okno wyboru utworów

Aplikacja włączając się wyświetla menu wyboru autora. Menu przedstawione jest w postaci kafelkowej.

Po wybraniu autora włączane jest menu wyboru albumu.

Po wybraniu albumu otwierane jest menu wyboru piosenek należących do tego utworu.

3.3. Odczyt i przetwarzanie plików

3.4. Struktura bazy danych

3.4.1. Ogólny opis

Baza danych jest złożona z trzech tabel:

- Autorzy - tabela ta ma zawierać wszystkie informacje o autorach z biblioteki użytkownika. Założeniem jest, że każdy autor ma unikalną nazwę, ponieważ nie ma żadnego dobrego sposobu unikalnej identyfikacji autorów z samych lokalnych plików.
- Albumy - tabela ta, oprócz katalogowania albumów, głównie pełni rolę „pośrednika” między piosenkami a autorami. Ważną informacją jaką zawiera każdy rekord, jest odnośnik do okładki danego albumu. Opisane jest to w sekcji nr. 3.4.3. Warto wspomnieć, że albumy każdego autora muszą mieć unikalne

nazwy - problem identyfikacji jest podobny jak przy autorach - lecz nazwy albumów różnych autorów mogą się powtarzać.

- Piosenki - tabela ta zawiera informacje o wszystkich piosenkach w bibliotece, pozyskane z tagów plików.

Detale dotyczące każdej z tabel można przeczytać w sekcji nr. 3.4.2.

3.4.2. Opis pól tabel

3.4.2.1. Autorzy

- **nazwa** - unikalna nazwa autora, jest zarazem kluczem głównym

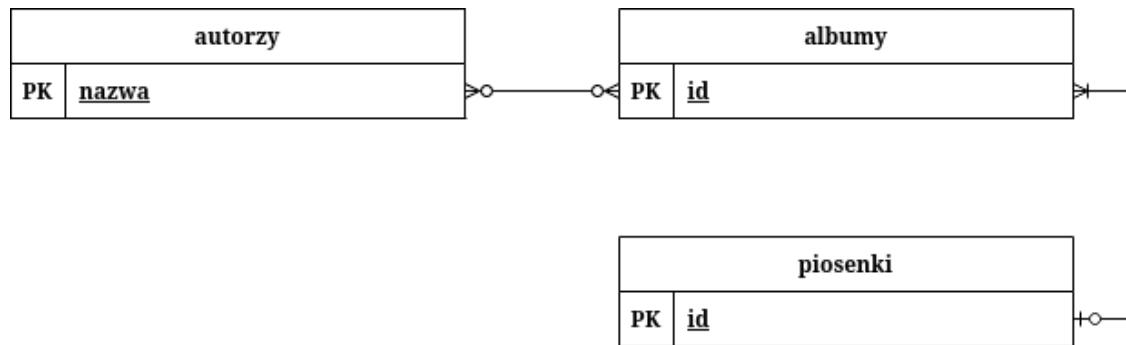
3.4.2.2. Albumy

- **id** - klucz główny, unikatowy identyfikator albumu
- **nazwa** - nazwa albumu, unikalna w obrębie jednego autora
- **tytuł** - tytuł albumu
- **okładka** - ścieżka do pliku z okładką albumu

3.4.2.3. Piosenki

- **id** - klucz główny, unikatowy identyfikator piosenki
- **tytuł** - tytuł piosenki
- **album** - klucz obcy, odniesienie do albumu, do którego należy piosenka
- **ścieżka** - ścieżka do pliku z piosenką

3.4.3. Relacje w bazie



Rys. 3.1. Model relacji bazy

Na rysunku nr. 3.1 ukazany jest uproszczony model bazy danych, biorący tylko pod uwagę komponenty potrzebne do określenia relacji. Autorzy są w relacji M do N z albumami. Każdy autor może mieć wiele albumów, a każdy album wiele autorów. Piosenki z albumami są w relacji 1 do N . Każda piosenka może należeć wyłącznie do jednego albumu, ale każdy album może mieć wiele piosenek.

Warto zauważyć, że piosenki nie bezpośrednio łączą się z autorami. Jeżeli piosenka chce uzyskać swojego autora, musi zrobić to poprzez album.

3.5. Czujnik światła

Zadanie czujnika światła jest dość proste i niezbyt skomplikowane. Celem czujnika jest wykrywanie intensywności światła i na podstawie tej intensywności aktualizacja wyglądu menu poprzez wykorzystanie schematów kolorów oraz motywów. Założenie jest takie, że przy intensywności światła mniejszej niż 50 procent z 10000 lumenów aplikacja będzie przełączać się na tryb ciemny, przy intensywności większej niż 50 procent aplikacja będzie przełączać się na tryb jasny.

3.6. Uwierzytelnianie biometrią

Aplikacja będzie wykorzystywać sensor biometryczny odcisku palca do autoryzacji użytkownika. Przed wejściem do aplikacji, zostanie wywołany ekran uwierzytelniania udostępniony przez system, na tym ekranie będzie okienko na którym będzie znajdowało się miejsce, gdzie będzie można przyłożyć palec. Aplikacja będzie sprawdzać czy odcisk palca znajduje się zapisany na telefonie i będzie wykorzystywać ten odcisk do uwierzytelniania.

3.7. Ładowanie obrazów albumów

Większość plików muzycznych ma przypisany sobie obraz. Obraz ten w praktycznie każdym przypadku jest miniaturką albumu do którego dana piosenka należy. Aplikacja powinna w odpowiednich miejscach je wyświetlać.

Dlatego, że miniaturki mają być dostępne w wielu miejscach w aplikacji, należałoby umieścić odnośniki do nich w bazie danych, konkretnie w tabeli dotyczącej albumów (patrz sekcja nr. 3.4.2.2) - logicznie, jeżeli miniaturka danego albumu jest dostępna, będzie używana w wielu podobnych kontekstach gdzie używane są same informacje o albumie.

Pozostaje kwestia faktycznego pozyskiwania obrazów z pliku. Proces ten odbędzie się za pomocą funkcji `Bitmap` z biblioteki `System.Drawing`.

dzie się podczas etapu ładowania tagów, po załadowaniu faktycznych plików, ale przed załadowaniem informacji do bazy. Miniaturki zostaną wyciągnięte z plików, i zostaną one włożone do głównego katalogu aplikacji jako bitmapy. Przechowywanie miniaturek jako osobne pliki pozwala na łatwiejsze ich pozyskiwanie podczas dalszego działania aplikacji.

4. Implementacja

4.1. Zarządzanie bazą danych

4.1.1. Klasa DatabaseManager

Za zarządzanie bazą danych odpowiedzialna jest klasa `DatabaseManager`, której kod jest zamieszany na listingu nr. 26. Klasa jest wrapperem do bazy danych Room^[5] i do niej akcesorów.

```
1 @Singleton
2 class DatabaseManager @Inject constructor(
3     @ApplicationContext context: Context
4 ) {
5     private val database: LibraryDb = Room.databaseBuilder(
6         context,
7         LibraryDb::class.java, "Library"
8     ).build()
9
10    fun collectAuthorsFlow(): Flow<List<Author>> = database.uiDao()
11        .getAllAuthorsFlow()
12
13    fun collectAlbumsByAuthorFlow(authorName: String): Flow<List<
14        Album>> {
15        return database.uiDao().getAuthorWithAlbums(authorName)
16            .map { it.albums }
17    }
18
19    fun collectSongsByAlbumFlow(albumId: Long): Flow<List<Song>> {
20        return database.uiDao().getAlbumWithSongs(albumId)
21            .map { it.songs }
22    }
23
24    ...
25
26    fun populateDatabase(songs: List<TagExtractor.SongInfo>) {
27        assert(Thread.currentThread().name != "main")
28
29        val dao = database.logicDao()
30
31        fun addAuthors() {
32            songs.fastForEach { song ->
33                //TODO: there should be a distinction between
34                albumartists and regular artists
35                song.albumArtists?.fastForEach { name ->
```

```
33             if(dao.getAuthor(name) == null) {
34                 dao.insertAuthor(Author(name = name))
35             }
36         }
37     }
38 }
39
40
41     fun addAlbumsAndRelations() {
42         // FIXME: xddddddd
43         val distinctAlbumArtistsList = songs
44             .map { Triple(it.album, it.albumArtists, it.
45 coverUri) }
46             .distinct()
47         Log.d(javaClass.simpleName, "Distinct artists set:
48 $distinctAlbumArtistsList")
49
50         distinctAlbumArtistsList.fastForEach {
51             val albumTitle = it.first.toString()
52             val artists = it.second
53             val coverUri = it.third
54
55             val albumId = dao.insertAlbum(Album(
56                 title = albumTitle,
57                 coverUri = coverUri.toString(),
58             ))
59
60             artists?.fastForEach {
61                 dao.insertAlbumAuthorCrossRef(
62                     AlbumAuthorCrossRef(
63                         albumId = albumId,
64                         name = it.toString()
65                     )
66                 }
67             }
68         }
69
70         fun addSongs() {
71             songs.fastForEach { song ->
72                 Log.d(javaClass.simpleName, "NEW SONG\n")
73                 Log.d(javaClass.simpleName, "Album artists: ${song.
74 albumArtists}")
75
76                 val albumWithAuthorCandidates = dao
77                     .getAlbumsByTitle(song.album.toString())
78             }
79         }
80     }
81 }
```

```
74         .map { it.albumId }
75             .map { dao.getAlbumWithAuthors(it) }
76             Log.d(javaClass.simpleName, ""
77                 $albumWithAuthorCandidates")
78
79             var correctAlbum: Album? = null
80             albumWithAuthorCandidates.fastForEach {
81                 Log.d(javaClass.simpleName, "${song.
82                     albumArtists}, ${it.authors}")
83                     //FIXME: theese guys shouldn't be ordered, will
84                     have to refactor a bunch of
85                     // stuff with sets instead of lists
86                     if(song.albumArtists?.sorted() == it.authors.
87                         map { it.name }.sorted()) {
88                         correctAlbum = it.album
89                     }
90
91             }
92
93         }
94     }
95
96     addAuthors()
97     addAlbumsAndRelations()
98     addSongs()
99 }
100 }
```

Listing 3. Struktura klasy DatabaseManager

Na pierwszej linijce można zauważyc adnotację `@Singleton`. Pochodzi ona z biblioteki `Hilt`[8]. Powiadamia ona bibliotekę o tym że klasa jest singletonem, czyli że ma istnieć tylko jej jedna instancja na cały program. Uczyniono to, dlatego że baza danych powinna być jedna na całą aplikację. Menadżer z nią interfejsujący, dlatego że jest używany w wielu innych klasach, też powinien mieć tylko jedną instancję, aby nie marnować pamięci.

Na linijce nr. 2, widać konstruktor klasy, do którego też przy użyciu `Hilt`, wstrzykiwany jest `context`.

Następnie, na linijce nr. 5, widać inicjalizację samego obiektu bazy `database`.

Baza jest reprezentowana przez klasę `LibraryDb`, definicję której można zobaczyć w sekcji 5.1.2

Dalej, do linijki nr. 22 pokazane są metody zwracające różne elementy bazy. Większość z tych metod zwraca `Flow[TODO:]`. Room natywnie obsługuje Flowy, a dlatego że wymusza dostęp do bazy z innych wątków niż główny, większość operacji wykonywanych na bazie odbywa się za pośrednictwem typów `Flow`

Same metody są wrapperami do obiektów `Dao` bazy.Więcej o nich w sekcji 5.1.3. Niektóre obrabiają dane jak np. `collectSongsByAlbumFlow()` na linijce nr. 17., która mapuje zwraca piosenki z wyjściowej klasy relacyjnej.

Metod tych jest więcej, lecz wyglądają one bardzo podobnie. Dla zwięzłości, można je pominąć.

Metoda `populateDatabase()` zadeklarowana na linijce nr. 24, jest odpowiedzialna za ładowanie wyjątkowych plików informacji do bazy. Jako parametr odstaje ona zmienną `songs` typu `List<TagExtractor.SongInfo>` Zadeklarowane są w niej trzy funkcje pomocnicze: `addAuthors()`, `addAlbumsAndRelations()` i `addSongs()`. Wywoływanie są one po kolejno w metodzie głównej.

Funkcja `addAuthors()`, zadeklarowana na linijce nr. 29, jest prosta w swoim działaniu. Lista z `SongInfo` jest iterowana i po kolejno wpisywani są wszyscy autorzy, którzy jeszcze w bazie nie istnieją.

Funkcja `addAlbumsAndRelations()`, zadeklarowana na linijce nr. 41, odpowiada za dodawanie albumów do bazy oraz tworzenie relacji między nimi, a autorami. Dworzona zmienna `distinctAlbumArtistsList` mapuje tylko unikalne pary albumów i autorów (zmienna `coverUri` nie ma znaczenia przy określaniu autorstwa, jest przypisywana tutaj dlatego, że trudno było znaleźć dla niej lepsze miejsce). Dzięki temu początkowemu filtrowaniu, wiadomo, że każdy napotkany album będzie unikalny. Następnie, `distinctAlbumArtistsList` jest iterowana - przy każdej iteracji dodawany jest nowy album do bazy. Metoda `insertAlbum()` zwraca `id` nowo dodanego albumu. Wynik jej jest przypisywany do zmiennej `albumId` na linijce nr. 53. Potem, zostaje przypisywana relacja albumu z autorami. Autorów może być kilku, więc są oni reprezentowani przy każdej iteracji przez listę, która jest iterowana, a relacja zostaje dodawana z nazwą autora i `albumId`.

Funkcja `addSongs()`, zadeklarowana na linijce nr. 67, ma na celu dodanie piosenek do bazy. Ciało funkcji jest w pętli iterującej się przez listę piosenek. Na początku pętli, na linijce nr. 72 deklarowana jest zmienna `albumWithAuthorCandidates`. Jest ona listą relacji album - autorzy wszystkich albumów o tej samej nazwie co ten w danym elemencie listy. Następnie, lista ta jest iterowana i przy każdej iteracji spraw-

dzane jest czy lista autorów w danej relacji jest równa z listą autorów danej piosenki. Jeżeli tak, wartość danego albumu z wybranej relacji jest przypisywana do zmiennej zadeklarowanej na linii nr. 78 `correctAlbum`. Na końcu funkcji, piosenka dodana jest do bazy przy użyciu metody `dao`.

4.1.2. Klasa LibraryDb

Klasa `LibraryDb` jest deklaracją faktycznej instancji bazy danych, która jest implementowana i generowana przez bibliotekę `Room`. Z racji tego, że jest to klasa abstrakcyjna, jej zadaniem jest określenie struktury bazy i jakie komponenty ma ona zawierać

```
1 @Database(entities = [Song::class, Author::class, Album::class,
2                     AlbumAuthorCrossRef::class], version
3 = 1)
4 abstract class LibraryDb : RoomDatabase() {
5     abstract fun logicDao(): LogicDao
6     abstract fun uiDao(): UIDao
7 }
```

Listing 4. Deklaracja bazy `LibraryDb`

Jak widać na listingu nr. 27, na początku klasy należy zamieścić adnotację `@Database`. Powiadamia ona bibliotece `Room` o tym, że następująca klasa jest bazą danych. Parametr `entities` określa wszystkie tabele jakie mają się w klasie zawierać. O tabelach więcej w sekcji nr. 5.1.4. Parametr `version` zajmuje się wersjonowaniem bazy. Jest on ważny przy aktualizacjach aplikacji, aby baza mogła zostać odpowiednio zmieniona.

Na linijce nr. 3 umieszczona jest faktyczna deklaracja klasy. Dziedziczy ona z klasy `RoomDatabase`. Jedyne rzeczy jakie są do dziecięcej klasy dodawane, to metody zwracające obiekty `dao`, opisane w sekcji nr. 5.1.3.

4.1.3. Obiekty Dao

Obiekty `dao` (Data Access Object(s)) to obiekty używane do interakcji z zawartością bazy danych. Głównie używa się ich do dodawania elementów do bazy oraz ich odczytywania. Same obiekty definiuje się jako interfejsy z adnotacją `@Dao`. Są one implementowane przez `Room`. Baza danych w projekcie wykorzystuje dwa interfejsy `dao` - `UIDao`, którego kod zamieszczony jest na listingu nr. 28 oraz `LogicDao`, którego kod zamieszczony jest na listingu nr. 29.

```
1 @Dao
```

```
2 interface UI Dao {  
3     @Query("SELECT * FROM Song")  
4     fun getAllSongs(): Flow<List<Song>>  
5  
6     @Query("SELECT * FROM Song WHERE songId = :songId")  
7     fun collectSongFromId(songId: Long): Flow<Song>  
8  
9     // Get an album with its songs  
10    @Transaction  
11    @Query("SELECT * FROM album WHERE albumId = :albumId")  
12    fun getAlbumWithSongs(albumId: Long): Flow<AlbumWithSongs>  
13  
14    @Query("SELECT * FROM album WHERE albumId = :albumId")  
15    fun getAlbumById(albumId: Long?): Flow<Album>  
16  
17    // Get an album with its authors  
18    @Transaction  
19    @Query("SELECT * FROM album WHERE albumId = :albumId")  
20    fun getAlbumWithAuthors(albumId: Long?): Flow<AlbumWithAuthors  
?>  
21  
22    @Query("SELECT * FROM Author")  
23    fun getAllAuthorsFlow(): Flow<List<Author>>  
24  
25    // Get an author with their albums  
26    @Transaction  
27    @Query("SELECT * FROM author WHERE name = :name")  
28    fun getAuthorWithAlbums(name: String): Flow<AuthorWithAlbums>  
29 }
```

Listing 5. Deklaracja interfejsu UI Dao

```
1 @Dao  
2 interface Logic Dao {  
3     @Insert  
4     fun insertSong(song: Song)  
5  
6     @Insert  
7     fun insertAlbum(album: Album): Long  
8  
9     @Insert(onConflict = OnConflictStrategy.REPLACE)  
10    fun insertAuthor(author: Author)  
11  
12    @Insert(onConflict = OnConflictStrategy.REPLACE)  
13    fun insertAlbumAuthorCrossRef(albumAuthorCrossRef:  
        AlbumAuthorCrossRef)
```

```

14
15     @Query("SELECT * FROM Author")
16     fun getAllAuthors(): List<Author>
17
18     @Transaction
19     @Query("SELECT * FROM album WHERE albumId = :albumId")
20     fun getAlbumWithAuthors(albumId: Long): AlbumWithAuthors
21
22     @Transaction
23     @Query("SELECT * FROM author WHERE name = :name")
24     fun getAuthorWithAlbums(name: String): AuthorWithAlbums
25
26     @Query("SELECT * FROM author WHERE name = :name")
27     fun getAuthor(name: String): Author?
28
29     @Query("SELECT * FROM album WHERE title = :title")
30     fun getAlbumsByTitle(title: String): List<Album>
31
32     @Query("SELECT * FROM album WHERE title = :title LIMIT 1")
33     fun getAlbumByTitle(title: String): Album?
34
35     @Query("SELECT * FROM AlbumAuthorCrossRef WHERE albumId = :albumId AND name = :authorName LIMIT 1")
36     fun getCrossRefByAlbumAndAuthor(albumId: Long, authorName: String): AlbumAuthorCrossRef?
37
38     @Query("SELECT * FROM Song WHERE songId = :songId")
39     fun getSongfromId(songId: Long): Song
40 }

```

Listing 6. Deklaracja interfejsu LogicDao

Dlatego, że baza Room wymaga dostępu do elementów z innego wątku niż główny, LogicDao może być tylko używany w kodzie, o którym wiadomo, że nie jest wykonywany na głównym wątku. UI dao natomiast, służy ekskluzywnie do zwracania Flowów. Większość elementów związanych z interfejsem w reszcie kodu aplikacji już korzysta z Flowów, więc dao to łatwo jest zintegrować.

Typowy sposób w jaki dodaje się element do bazy znajduje się na linijce nr. 4 w kodzie LogicDao, na listingu nr. 29. Funkcja `insertSong()`, zadnotowana jest `@Insert`. Powiadamia to Room, że funkcja ta odpowiedzialna jest za dodawanie elementu. Parametr `song` to piosenka jaka ma być dodana. W następnej funkcji `insertAlbum()` widać, że funkcje `@Insert` mogą zwracać wartości. W tym przypadku funkcja zwraca `id` nowo dodanego albumu. Można też zwrócić uwagę na

metodę `insertAuthor()` na linijce nr. 8, a w szczególności parametr `onConflict` w adnotacji `@Insert`. Wartość parametru `OnConflictStrategy.REPLACE` mówi bibliotece, aby nie pomijała elementów o tych samych wartościach co już są w tabeli, ale zamieniała starsze na te nowe.

Przykład odczytywania elementu jest dobrze zilustrowany na metodzie `getAuthor()` zadeklarowanej na linijce nr. 27. Adnotacja `@Query` przyjmuje parametr `String`, który jest kwerendą SQL jaka ma być wykonana. Kwerenda `SELECT * FROM author WHERE name = :name` wybiera wszystkich autorów, których pole `name` równe jest parametrowi metody `name` (odnoszenie do parametru w kwerendzie poprzedzone jest znakiem „:”). Dlatego że w bazie może być tylko jeden autor z daną nazwą, zwracany jest pojedynczy autor, a nie ich lista. Niektóre metody, jak na linijce nr. 20 `getAlbumWithAuthors()`, używają adnotacji `@Transaction`. W przypadku tej metody, zwraca ona relację, czyli czyta z kilku tabel. Adnotacja `@Transaction` zapewnia, że transakcja jest atomiczna, co za tym idzie, inne wątki nie mogą nagle zmienić wartości jakieś tabeli.

Interfejs `UIdao` działa podobnie jak `LogicDao`, ale zwraca on tylko i wyłącznie `Flowy`, które są natywnie obsługiwane przez `Room`.

4.1.4. Tabele

W bibliotece `Room`, każda tabela to `dataclass` określana adnotacją `@Entity`. Kolumny takiej tabeli to po prostu pola klasy. Klucz danej tabeli jest określany adnotacją `@PrimaryKey`

```
1 @Entity
2 data class Author (
3     @PrimaryKey val name: String
4 )
```

Listing 7. Deklaracja tabeli `Author`

Tabela `Author`, zawarta na listingu nr. 30, określa autorów. Tabela jest prosta, jedynym polem jest `name`, który jest kluczem.

```
1 @Entity
2 data class Album(
3     @PrimaryKey(autogenerate = true) val albumId: Long = 0,
4     val title: String,
5     val coverUri: String?,
```

6)

Listing 8. Deklaracja tabeli Album

Tabela `Album`, zawarta na listingu nr. 31, określa albumy. Kluczem jest zmienna `albumId`. Klucz jest generowany automatycznie, dzięki parametrowi adnotacji `autoGenerate`. Pole `title` określa tytuł, a pole `coverUri` określa adres URI okładki.

```

1
2 @Entity(
3     foreignKeys = [
4         ForeignKey(
5             entity = Album::class,
6             parentColumns = ["albumId"],
7             childColumns = ["albumId"],
8             onDelete = ForeignKey.CASCADE
9         )
10    ],
11
12    indices = [Index(value = ["albumId"])]
13 )
14 data class Song(
15     @PrimaryKey(autoGenerate = true) val songId: Long = 0,
16     val title: String?,
17     val albumId: Long?,
18     val fileUri: String?,
19 )

```

Listing 9. Deklaracja tabeli Song

Klasa ta, zawarta na listingu nr. 32, określa tabelę piosenek. Pole `foreignKeys` w adnotacji `@Entity` określa obce klucze, którymi posługuje się klasa. W tym przypadku określone jest to, że pole w `Song albumId` wskazuje na pole w `Album albumId`. Pole `indices` każe indeksować pola z `albumId` ku polepszeniu szybkości bazy. W ciele klasy, pole `title` to tytuł piosenki. Pole `albumId` określa ID albumu, do którego należy dana piosenka.

4.1.5. Relacje

Relacje w Room są określane jako osobne `dataclassy`. Są one zadeklarowane adnotacją `@Relation` w danej klasie. Ponadto umieszczenie elementu w adnotacji

@Embedded, pozwala klasie „przyswoić” pola danego elementu. Dzięki temu klasa może odnosić się do pól danego elementu tak jakby były one bezpośrednio w klasie. Konieczne jest umieszczenie elementu głównego, od którego będzie relacja wyodziła, do tej adnotacji.

4.1.5.1. AlbumWithSongs

Klasa `AlbumWithSongs` na listingu nr. 33, określa relację albumów i piosenek.

```
1 data class AlbumWithSongs(
2     @Embedded val album: Album,
3     @Relation(
4         parentColumn = "albumId",
5         entityColumn = "albumId"
6     )
7     val songs: List<Song>
8 )
```

Listing 10. Deklaracja relacji `AlbumWithSongs`

Relacja łączy pole `albumId` albumu z polem `albumId` piosenek. Pole `songs` zawiera wszystkie piosenki z tą samą wartością pola `albumId` co faktyczny klucz danego albumu.

```
1 @Entity(primaryKeys = ["albumId", "name"])
2 data class AlbumAuthorCrossRef(
3     val albumId: Long,
4     val name: String
5 )
```

Listing 11. Deklaracja tabeli relacji `AlbumAuthorCrossRef`

Tabela na listingu nr. 34 określa relację M do N między albumami a autorami. Jest to tabela z dwoma kluczami głównymi: `albumId` dla tabeli `Album` i `name` dla tabeli `Author`.

4.1.5.3. AlbumWithAuthors i AuthorWithAlbums

Obie klasy są do siebie bardzo podobne więc zostaną omówione razem.

```
1 data class AlbumWithAuthors(
2     @Embedded val album: Album,
3     @Relation(
```

```

4     parentColumn = "albumId",
5     entityColumn = "name",
6     associateBy = Junction(AlbumAuthorCrossRef::class)
7 )
8     val authors: List<Author>
9 )

```

Listing 12. Deklaracja relacji AlbumWithAuthors

```

1 data class AuthorWithAlbums(
2     @Embedded val author: Author,
3     @Relation(
4         parentColumn = "name",
5         entityColumn = "albumId",
6         associateBy = Junction(AlbumAuthorCrossRef::class)
7     )
8     val albums: List<Album>
9 )

```

Listing 13. Deklaracja relacji AuthorWithAlbums

Na listingu nr. 35 przedstawiona jest klasa `AlbumWithAuthors`. Definiuje ona relację danego albumu z jego autorami. Dlatego, że relacja jest M do N , w adnotacji `@Relation` dodane jest odniesienie do tabeli relacji `AlbumAuthorCrossRef`, opisanej w sekcji nr. 5.1.5.2. Klucze, jakie mają być porównywane są zdefiniowane w parametrach `parentColumn`, dla `id` albumu i `entityColumn` dla nazwy autora. Wynikiem tej relacji jest lista albumów. Sytuacja wygląda podobnie w `AuthorWithAlbums`, na listingu nr. 36. Tym razem to autor jest rodzicem i oczekujemy od relacji listy albumów danego autora.

Czujnik światła Czujnik światła został zaimplementowany za pomocą wbudowanej funkcji. Zadaniem czujnika jest dynamiczna zmiana schematu kolorów aplikacji na podstawie danych otrzymanych z czujnika światła wbudowanego w urządzeniu mobilnym z systemem android.

```

1 @AndroidEntryPoint
2 class MainActivity : FragmentActivity(), SensorEventListener {
3     private lateinit var sensorManager: SensorManager
4     private var lightSensor: Sensor? = null
5     private val _isDarkTheme = mutableStateOf(false)
6     private val isDarkTheme: State<Boolean> = _isDarkTheme
7
8     private val _isAuthenticated = mutableStateOf(false)
9     private val isAuthenticated: State<Boolean> = _isAuthenticated
10
11    override fun onCreate(savedInstanceState: Bundle?) {

```

```
12     super.onCreate(savedInstanceState)
13     val biometricAuthenticator = BiometricAuthenticator(this)
14
15     sensorManager = getSystemService(Context.SENSOR_SERVICE) as
16     SensorManager
17     lightSensor = sensorManager.getDefaultSensor(Sensor.
18     TYPE_LIGHT)
19
20     setContent {
21         val darkTheme by isDarkTheme
22         val authenticated by isAuthenticated
23         RaptorTheme(darkTheme = darkTheme) {
24             Surface(
25                 modifier = Modifier.fillMaxSize(),
26                 color = MaterialTheme.colorScheme.background
27             ) {
28                 if (authenticated) {
29                     MainScreen()
30                 } else {
31                     AuthenticationScreen(
32                         onAuthenticate = {
33                             promptBiometricAuthentication(
34                             biometricAuthenticator
35                         )
36                     }
37                 }
38             }
39         }
40     }
41
42     private fun promptBiometricAuthentication(
43         biometricAuthenticator: BiometricAuthenticator) {
44         biometricAuthenticator.PromptBiometricAuth(
45             title = "Authentication Required",
46             subtitle = "Please authenticate to proceed",
47             negativeButtonText = "Cancel",
48             fragmentActivity = this,
49             onSuccess = {
50                 runOnUiThread {
51                     _isAuthenticated.value = true
52                 }
53             },
54         ),
55     }
56 }
```

```
53     onFailed = {
54     },
55     onError = { errorCode, errorString ->
56     }
57   )
58 }
59
60 override fun onResume() {
61   super.onResume()
62   lightSensor?.let { sensor ->
63     sensorManager.registerListener(this, sensor, SensorManager.
64     SENSOR_DELAY_NORMAL)
65   }
66
67   override fun onPause() {
68     super.onPause()
69     sensorManager.unregisterListener(this)
70   }
71
72   override fun onSensorChanged(event: SensorEvent?) {
73     if (event?.sensor?.type == Sensor.TYPE_LIGHT) {
74       val lightLevel = event.values[0]
75       val maxLightLevel = lightSensor?.maximumRange ?: 10000f
76
77       _isDarkTheme.value = lightLevel < 0.4 * maxLightLevel
78     }
79   }
80
81   override fun onAccuracyChanged(sensor: Sensor?, accuracy: Int)
82   {
83 }
```

Listing 14. Implementacja czujnika światła w `MainActivity.kt`

Na listingu 37 przedstawiona jest funkcja MainScreen. W wierszu 2 mamy `SensorEventListener`, który jest frameworkm w androidzie który pozwala aplikacji na reagowanie na zmiany odczytywane przez czujniki smartfona. Po dodaniu automatycznie tworzony są funkcje `onSensorChanged`, `onResume`, `onPause`, `onAccuracyChanged`. Implementację sensora zaczynamy od utworzenia zmiennych `sensormanager` oraz `lightSensor` w wierszach 3 i 4. Zmienna `sensormanager` odpowiedzialna jest za pobranie menadżera czujników z poziomu systemu. Zmienna `lightSensor` odpowiedzialna jest za uzyskanie referencji do głównego czujnika urządzenia, jeżeli

się nie uda to zwraca wartość null. Zmienna `_isDarkTheme` w wierszu 5 określa czy aplikacja wykorzystuje obecnie tryb ciemny, zmienna `isDarkTheme` w wierszu 6 pomaga jej w tym za pomocą odczytu w UI.

W `SetContent` w wierszu 23 do dynamicznej zmiany kolorów wykorzytywane jest "RaptorTheme(darkTheme = darmTheme)" zdefiniowany w `Theme.kt` w folderze `ui.Theme`.

Poniżej, w wierszach od 60 do 65 znajduje się funkcja `onResume`, która rejestruje słuchacza zdarzeń po wznowieniu działania aplikacji. odpowiedzialne jest za częstotliwość aktualizacji(`SENSOR_DELAY_NORMAL`).

This oznacza implementację `SensorEventListener`.

Funkcja `onPause` odpowiedzialna jest zatrzymanie działania słuchacza zdarzeń w przypadku pracy aplikacji w tle.

Funkcja `onSensorChanged` wywoływana jest za każdym razem gdy uzyskany zostanie nowy odczyt z czujnika systemowego. Zmienna `lightLevel` pobiera aktualny poziom oświetlenia(jednostka to luks). Zmienna `maxLightLevel` pobiera maksymalny zakres pomiarowy czujnika. W przypadku braku przypisana zostanie wartość 10000 luksów. W wierszu 77 znajduje się instrukcja przejścia w tryb ciemny jeżeli obecny wykrywany poziom światła jest mniejszy niż 40 procent. Funkcja `onAccuracyChanged` nie jest tutaj wykorzystywana. Może być wykorzystana do np. zmiany dokładności wykrywania czujnika.

4.2. Autoryzacja odciskiem palca

Autoryzacja odciskiem palca działa w następujących krokach:

1. Sprawdzenie, czy uwierzytelnianie biometryczne jest dostępne, przedstawione na listingu nr 38
2. Zbudowanie monitu biometrycznego, przedstawione na listingu nr 39
3. Obsługa wyniku monitu biometrycznego, przedstawiona na listingu nr 40

```
1 enum class BiometricAuthenticationStatus(val id: Int) {  
2     READY(1),  
3     NOT_AVAILABLE(-1),  
4     TEMPORARY_NOT_AVAILABLE(-2),  
5     AVAILABLE_BUT_NOT_ENROLLED(-3)  
6 }  
7
```

```
8 fun isBiometricAuthAvailable(): BiometricAuthenticationStatus {
9     return when (biometricmanager.canAuthenticate(BIOMETRIC_STRONG)) {
10         BiometricManager.BIOMETRIC_SUCCESS ->
11             BiometricAuthenticationStatus.READY
12         BiometricManager.BIOMETRIC_ERROR_NO_HARDWARE ->
13             BiometricAuthenticationStatus.NOT_AVAILABLE
14         BiometricManager.BIOMETRIC_ERROR_HW_UNAVAILABLE ->
15             BiometricAuthenticationStatus.TEMPORARY_NOT_AVAILABLE
16         BiometricManager.BIOMETRIC_ERROR_NONE_ENROLLED ->
17             BiometricAuthenticationStatus.AVAILABLE_BUT_NOT_ENROLLED
18         else -> BiometricAuthenticationStatus.NOT_AVAILABLE
19     }
20 }
```

Listing 15. Sprawdzenie dostępności uwierzytelnienia biometrycznego

Na zamieszczonym listingu funkcja `BiometricAuthenticationStatus` tworzy stany gotowości uwierzytelniania. Funkcja `isBiometricAuthAvailable` jest odpowiedzialna za sprawdzenie czy telefon obsługuje uwierzytelnianie biometryczne oraz czy w telefonie są zapisane odciski palca. `biometricmanager.canAuthenticate(BIOMETRIC_STRONG)` jest odpowiedzialna za zapytanie systemu, o możliwość użycia uwierzytelniania silnego biometrycznego, w zależności od odpowiedzi systemu zostanie zwrocony odpowiednio zdefiniowany status.

```
1 fun PromptBiometricAuth(
2     title: String,
3     subtitle: String,
4     negativeButtonText: String,
5     fragmentActivity: FragmentActivity,
6     onSuccess: (result: BiometricPrompt.AuthenticationResult) -> Unit
7     ,
8     onFailed: () -> Unit,
9     onError: (errorCode: Int, errorMessage: String) -> Unit,
10 ) {
11     when(isBiometricAuthAvailable()) {
12         BiometricAuthenticationStatus.NOT_AVAILABLE -> {
13             onError(BiometricAuthenticationStatus.NOT_AVAILABLE.id, "Not
14             available on this device")
15             return
16         }
17         BiometricAuthenticationStatus.TEMPORARY_NOT_AVAILABLE -> {
18             onError(BiometricAuthenticationStatus.
19             TEMPORARY_NOT_AVAILABLE.id, "Not available at this moment")
20             return
21     }
22 }
```

```
18     }
19     BiometricAuthenticationStatus.AVAILABLE_BUT_NOT_ENROLLED -> {
20         onError(BiometricAuthenticationStatus.
21             AVAILABLE_BUT_NOT_ENROLLED.id, "Add a fingerprint")
22         return
23     }
24     else -> Unit
25 }
26 biometricPrompt = BiometricPrompt(
27     fragmentActivity,
28     object: BiometricPrompt.AuthenticationCallback() {
29         override fun onAuthenticationSucceeded(result:
30             BiometricPrompt.AuthenticationResult) {
31             super.onAuthenticationSucceeded(result)
32             onSuccess(result)
33         }
34
35         override fun onAuthenticationError(errorCode: Int, errString:
36             CharSequence) {
37             super.onAuthenticationError(errorCode, errString)
38             onError(errorCode, errString.toString())
39         }
40
41         override fun onAuthenticationFailed() {
42             super.onAuthenticationFailed()
43             onFailure()
44         }
45     }
46 )
47 promptinfo = BiometricPrompt.PromptInfo.Builder()
48     .setTitle(title)
49     .setSubtitle(subtitle)
50     .setNegativeButton(negativeButtonText)
51     .build()
52     biometricPrompt.authenticate(promptinfo)
53 }
```

Listing 16. Sprawdzenie monitu biometrycznego

Funkcja PromptBiometricAuth odpowiedzialna jest za tworzenie monitu, który zostanie wyświetlony użytkownikowi podczas włączenia aplikacji. Monit zawiera tytuł, podtytuł oraz przycisk negatywny. Od wiersza 10 do 24 znajduje się instrukcja when która jest odpowiedzialna za sprawdzenie dostępności uwierzytelniania. Następnie w wierszach od 25 do 43 tworzony jest nowy obiekt który będzie obsługiwał okno dialogowe. `object:BiometricPrompt.AuthenticationCallback()`

odpowiedzialne jest za implementację metody obsługi zdarzeń związanych z uwierzytelnianiem. Jeżeli uwierzytelnianie się powiedzie to wywoływana jest metoda `onAuthenticationSucceeded`, która przekazuje wynik jako argument do `PromptBiometricAuth`, co umożliwia odblokowanie aplikacji. `onAuthenticationError` wywoływanie jest w przypadku wystąpienia błędu. `onAuthenticationFailed` jest wywoływanie w przypadku niepowodzenia uwierzytelniania, np. niewłaściwy odcisk palca. Od wiersza 44 do 49 znajduje się prompt builder, odpowiadający za skonfigurowanie danych które będą wyświetlane w monicie, na samym końcu w wierszu 49 wywołana jest metoda.

```
1  private fun promptBiometricAuthentication(biometricAuthenticator: BiometricAuthenticator) {
2      biometricAuthenticator.PromptBiometricAuth(
3          title = "Authentication Required",
4          subtitle = "Please authenticate to proceed",
5          negativeButtonText = "Cancel",
6          fragmentActivity = this,
7          onSuccess = {
8              runOnUiThread {
9                  _isAuthenticated.value = true
10             }
11         },
12         onFailed = {
13     },
14         onError = { errorCode, errorString ->
15     }
16     )
17 }
```

Listing 17. Obsługa wyniku monitu

Funkcja w `MainActivity`, przekazuje dane do tworzonego monitu, w przypadku pozytywnego uwierzytelnienia, `IsAuthenticated` jest ustawiany jako `true`, dzięki czemu aplikacja wie że można opuścić ekran logowania oraz przejść do wczytywania reszty aplikacji. Fragment kodu odpowiedzialny za załadowanie ekranu uwierzytelniania przed przejściem do ekranu głównego znajduje się w `onCreate`, przedstawionym na listingu nr 41.

```
1  if (authenticated) {
2      MainScreen()
3  } else {
4      AuthenticationScreen(
5          onAuthenticate = {
6              promptBiometricAuthentication(biometricAuthenticator)
```

```
7     }
8   )
9 }
10
```

Listing 18. Zawartość onCreate

4.3. Odczyt i przetwarzanie plików

4.3.1. Tag Extractor

Klasa `TagExtractor` jest odpowiedzialna za wyciąganie tagów z piosenek. Każda piosenka zawiera tagi, na które składają się: nazwa artysty, nazwa artystów z albumu, tytuł, data wydania, nazwa albumu, URI piosenki, URI obrazka cover. Deklaracje tych tagów pokazane są na listingu nr 42.

```
1 data class SongInfo(
2   val artists: List<String>?,
3   val albumArtists: List<String>?,
4   val title: String?,
5   val releaseDate: String?,
6   val album: String?,
7   val fileUri: Uri?,
8   val coverUri: Uri?,
9 )
10
```

Listing 19. Deklaracja tagów

Metoda `buildSongInfo()` przedstawiona na listingu nr 43 jest odpowiedzialna za parsowanie metadanych. Otrzymuje

```
1 @OptIn(UnstableApi::class)
2 private fun buildSongInfo(metadata: Metadata, uri: Uri?):
3   SongInfo {
4   val metadataList = mutableListOf<Metadata.Entry>()
5   for(i in 0 until metadata.length()) {
6     metadataList.add(metadata.get(i))
7   }
8   Log.d("${javaClass.simpleName}", "Metadata list: $metadataList")
9
10  when(metadataList[0]) {
11    is VorbisComment -> {
12      fun handleMissingTags(map: MutableMap<String?, Any?>) {
```

```
12         if(map["ALBUMARTIST"] == null) map["ALBUMARTIST"] = List<
13             String>(1,{"Unknown"} )
14
15     val entryMap: MutableMap<String?, Any?> = mutableMapOf()
16
17     // the last element 'picture' screws up the logic and it
18     // only has a mimetype value
19     // which i think is useless
20     metadataList.take(metadataList.size - 1).fastForEach {
21         val entry = it as VorbisComment
22
23         val key = entry.key
24         val value = entry.value
25         when(key) {
26             "ALBUMARTIST", "ARTIST" -> {
27                 if(!entryMap.containsKey(key)) {
28                     entryMap[key] = mutableListOf<String?>(value)
29                 } else {
30                     (entryMap[key] as? MutableList<String?>)?.add(value)
31                 }
32             }
33             else -> {
34                 // assert(!entryMap.containsKey(key))
35                 if(entryMap.containsKey(key)) {
36                     Log.w(javaClass.simpleName, "Unhandled duplicate
37                         key: $key")
38                     return@fastForEach
39                 }
40                 entryMap[key] = value
41             }
42         }
43     }
44     handleMissingTags(entryMap)
45
46     val coverUri = imageManager.extractAlbumimage(
47         uri,
48         entryMap["ALBUMARTIST"] as List<String>,
49         entryMap["ALBUM"] as String
50     )
51
52     return SongInfo(
```

```
53     artists = entryMap["ARTIST"] as? List<String>?,
54     albumArtists = entryMap["ALBUMARTIST"] as List<String>,
55     title = entryMap["TITLE"] as? String?,
56     album = entryMap["ALBUM"] as String?,
57     releaseDate = entryMap["DATE"] as? String?,
58     fileUri = uri,
59     trackNumber = (entryMap["TRACKNUMBER"] as? String?)?.toInt()
60     () ,
61     coverUri = coverUri,
62     ).also {
63         Log.d(javaClass.simpleName, "Vorbis Song: $it")
64     }
65 }
66
67     is Id3Frame -> {
68         fun handleMissingTags(map: MutableMap<String, List<String>>> {
69             if(map["TPE2"] == null) map["TPE2"] = List<String>(1,{ "Unknown" } )
70         }
71
72         val entryMap: MutableMap<String, List<String>> =
73             mutableMapOf()
74             metadataList.fastForEach {
75                 val entry = it as Id3Frame
76                 Log.d(javaClass.simpleName, "Id3 metadata: $entry")
77
78                 when(entry) {
79                     is TextInformationFrame -> {
80                         entryMap[entry.id] = entry.values
81                     }
82
83                     else -> {
84                         Log.w(javaClass.simpleName, "Unimplemented id3 frame:
85                         $entry")
86                     }
87                 }
88
89                 handleMissingTags(entryMap)
90
91                 val coverUri = imageManager.extractAlbumimage(
92                     uri,
93                     entryMap["TPE2"]?: emptyList() ,
```

```
93     entryMap["TALB"]?.get(0).toString()
94
95 )
96
97     return SongInfo(
98         artists = entryMap["TPE1"],
99         albumArtists = entryMap["TPE2"] as List<String>,
100        title = entryMap["TIT2"]?.get(0),
101        releaseDate = entryMap["TDA"]?.get(0),
102        album = entryMap["TALB"]?.get(0),
103        fileUri = uri,
104        trackNumber = entryMap["TRCK"]?.get(0)?.let {
105            return@let it.takeWhile { it != '/' }.toInt()
106        },
107        coverUri = coverUri
108    ).also {
109        Log.d(javaClass.simpleName, "id3 Song: $it")
110    }
111 }
112
113 else -> {
114     metadataList.fastForEach {
115         Log.w(
116             javaClass.simpleName,
117             "Unhanded tag format: ${it::class.simpleName}, metadata: $it"
118         )
119     }
120
121     return SongInfo(
122         null, mutableListOf("Unknown"), null, null, null, null,
123         null, null
124     )
125 }
126 }
```

Listing 20. Metoda buildSongInfo()

Instrukcje w wierszach od 3 do 6 odpowiedzialne są za tworzenie listy metadanych. Następuje inicjalizacja pustej listy `metadataList`. Następnie pętla przeszukuje po `metadata.length`, następnie dodaje każdy wpis do listy. W wierszach od 9 do 120 znajduje się pętla while, odpowiedzialna za warunkowe sprawdzanie formatu danych. Instrukcja składa się z tzech części: `VorbisComment`, `Id3Frame` oraz `else`.

1. **VorbisComment** znajdujące się w wierszach od 10 do 65 odpowiedzialne jest za parsowanie gdy mamy do czynienia z formatem metadanych typu Vorbis. Funkcja **handleMissingTags()** w wierszach od 11 do 13 jest odpowiedzialna za wypełnienie brakujących pól domyślnymi wartościami. Następnie w wierszach od 15 do 42 tworzona jest mapa wejścia oraz parsowanie. W wierszach od 44 do 50 następuje wywołanie funkcji uzupełniającej brakujące tagi oraz ekstrakcja coveru piosenki. Wyodrębnia wbudowany w plik audio cover oraz zapisuje go do pliku w pamięci aplikacji. W wierszach od 52 do 63 tworzony jest obiekt **SongInfo**.
2. **Id3Frame** znajduje się w wierszach od 67 do 111. Odpowiedzialne za parsowanie gdy mamy do czynienia z formatem Id3Frame. Działa podobnie do poprzednika. Mamy funkcję wewnętrzną która tworzy brakujące tagi, tworzona jest mapa oraz następuje parsowanie, wywołanie funkcji tworszącej brakujące tagi oraz ekstrakcja coveru i na koniec tworzony jest obiekt **SongInfo** dla tego formatu.
3. **else** znajduje się w wierszach od 113 do 124. Jest wykonywana gdy natrafimy na nieobsługiwany przez aplikację format. Ostrzeżenie jest zapisywane do logów oraz zwracane jest **SongInfo** zawierające minimalne informacje.

Metoda **TagExtractor** przedstawiona na listingu 44. Metoda jest odpowiedzialna za skanowanie plików audio przekazywanych jako lista **SongFile** oraz wyodrębnienia ich z metadanych.

```
1  @OptIn(UnstableApi::class)
2  fun extractTags(fileList: List<MusicFileLoader.SongFile>): List<
3      SongInfo> {
4      val tagsList = mutableListOf<SongInfo>()
5
6      for(file in fileList) {
7          val mediaItem = MediaItem.fromUri("${file.uri}")
8
9          val trackGroups = MetadataRetriever.retrieveMetadata(context,
10             mediaItem).get()
11
12          if(trackGroups != null) {
13              // Parse and handle metadata
14              assert(trackGroups.length == 1)
15
16              val tags = trackGroups[0]
17                  .getFormat(0)
18                  .metadata
```

```
17     .let {
18         buildSongInfo(
19             it!!,
20             file.uri
21         )
22     }
23
24     tagsList.add(tags)
25 }
26 }
27 return tagsList
28 }
```

Listing 21. Metoda TagExtractor()

Instrukcja w wierszu 3 tworzy pustą listę do której będą wkładane obiekty SongInfo.

Pętla for przechodzi przez każdy plik SongFile oraz tworzy obiekt MediaItem w wierszu 6, pobiera metadane w wierszu 8, instrukcja if w wierszach od 10 do 25 jest odpowiedzialna za walidację i parsowanie, sprawdza, czy trackGroups nie jest null. assert w wierszu 12 zakłada, że metadane będą w jednym tracku. Następnie w tags uzysujemy dostęp do metadanych oraz parsujemy w buildSongInfo. W wierszu 24 dodajemy wynik do tagList. W wierszu 27 zwracamy tagList.

4.3.2. MusicFileLoader

Zadanie MusicFileLoader znalezienie wszystkich plików muzycznych z wybranego przez użytkownika folderu. Na listingu nr 45.

```
1 package com.example.raptor
2
3 import android.content.Context
4 import android.content.Intent
5 import android.net.Uri
6 import android.provider.DocumentsContract
7 import android.util.Log
8 import androidx.activity.compose.ManagedActivityResultLauncher
9 import androidx.activity.compose.
10    rememberLauncherForActivityResult
11 import androidx.activity.result.contract.ActivityResultContracts
12 import androidx.compose.runtime.Composable
13 import androidx.compose.ui.util.fastForEach
14 import androidx.documentfile.provider.DocumentFile
15 import dagger.hilt.android.qualifiers.ApplicationContext
16 import kotlinx.coroutines.flow.MutableStateFlow
```

```
16 import javax.inject.Inject
17
18 /**
19 * Handles the loading of music files in a given directory and its
20 * subdirectories as well as
21 * launching a file picker instance
22 */
23 class MusicFileLoader
24 @Inject constructor( @ApplicationContext private val context:
25 Context) {
26
27     /**
28      * Dataclass representing a song file
29      *
30      * @param filename Name of the file
31      * @param uri 'Uri' corresponding to the file
32      * @param mimeType The type of the file
33      */
34     data class SongFile(val filename: String, val uri: Uri, val
35     mimeType: String)
36
37     /**
38      * Observable list of 'SongFile' currently loaded
39      */
40     var songFileList = MutableStateFlow<List<SongFile>>(emptyList()
41 )
42     private set
43
44     private lateinit var launcher : ManagedActivityResultLauncher<
45 Uri?, Uri?>
46
47     private fun traverseDirs(treeUri: Uri): List<SongFile> {
48         val _songFiles = mutableListOf<SongFile>()
49
50         fun visit(uri: Uri) {
51             val root = DocumentFile.fromTreeUri(context, uri)
52             Log.d(javaClass.simpleName, "Visiting dir: ${root?.name}")
53             val childDirs = root?.listFiles()?.filter { it.isDirectory }
54             childDirs?.fastForEach { visit(it.uri) }
55
56             val songFiles = root?.listFiles()
57             ?.filter {
58                 it.type?.slice(0..4) == "audio"
59             }
60         }
61
62         childDirs?.fastForEach { visit(it.uri) }
63
64         val songFiles = root?.listFiles()
65         ?.filter {
66             it.type?.slice(0..4) == "audio"
67         }
68     }
69 }
```

```
55     ?.map {
56         SongFile(
57             filename = it.name.toString(),
58             uri = it.uri,
59             mimeType = it.type.toString()
60         )
61     }
62
63     Log.d(javaClass.simpleName, "Visited dir ${root?.name},
64 songs: ${songFiles?.map { it
65             .filename
66         }}")
67
68     _songFiles.addAll(songFiles?: emptyList())
69
70     visit(treeUri)
71
72
73     return _songFiles
74 }
75
76 /**
77 * \@Composable function that prepares the file picker launcher
78 *
79 * Must be called before 'launch()'
80 */
81 @Composable
82 fun PrepareFilePicker() {
83     val contentResolver = context.contentResolver
84
85     launcher = rememberLauncherForActivityResult(
86         contract = ActivityResultContracts.OpenDocumentTree()
87     ) { treeUri: Uri? ->
88         treeUri?.let {
89             val permissions = Intent.FLAG_GRANT_READ_URI_PERMISSION
90             or
91                 Intent.FLAG_GRANT_WRITE_URI_PERMISSION
92             contentResolver.takePersistableUriPermission(treeUri,
93             permissions)
94
95             Log.d(javaClass.simpleName, "Selected: $it")
96             songFileList.value = traverseDirs(treeUri)
97         }
98     }
99 }
```

```
97     }
98
99     /**
100    * Launches the file picker
101   */
102  fun launch() {
103    launcher.launch(null)
104  }
105 }
```

Listing 22. Kod MusicFileLoader

Na początku znajduje się klasa danych `SongFile` w wierszu 31. Służy do przechowywania informacji o pojedynczym pliku audio. W wierszach 36 i 37 jest zmienna `songFileList`, służąca do przechowywania listy obiektów `SongFile`. W wierszu 39 znajduje się zmienna `launcher`, który zwraca uri folderu wybranego przez użytkownika. Funkcja `TraverseDirs` w wierszach od 41 do 74 jest odpowiedzialna za rekurencyjne przeszukiwanie folderu określonego przez `treeUri`. Zbiera wszystkie pliki audio znajdujące się w tym folderze. Najpierw tworzona jest lista `_songFiles` która będzie przechowywać wyniki. następnie tworzona jest funkcja wewnętrzna `visit` która wywołuje obiekt reprezentujący wybrany folder, następnie pliki są listowane oraz oddzielane od folderu. Następnie rekurencyjnie wywołujemy dla każdego katalogu `visit(it.uri)`. Następnie w bieżącym katalogu filtrowane są pliki w wierszach od 52 do 54. Następnie z każdego pliku tworzony jest obiekt `SongFile` a następnie dołączany do `_SongFiles`. Na koniec zwracane jest `_SongFiles`. Następnie znajduje się funkcja `PrepareFilePicker` w wierszach od 82 do 97. Funkcja jest odpowiedzialna za przygotowanie launchera. `rememberLauncherForActivityResult` pozwala na stworzenie obiektu launchera który pozwoli na otwarcie `OpenDocumentTree`. Kontrakt `ActivityResultContracts.OpenDocumentTree()` pozwala na wybranie całego folderu. Następnie w wierszach od 88 do 96 znajdują się instrukcje definiujące uprawnienia, aby zachować uprawnienia potrzebne jest

`takePersistableUriPermission()`.

`songFileList.value = traverseDirs(treeUri)` służy do aktualizacji flow, URI folderu przekazywane jest do funkcji `traverseDirs`.

Na koniec w wierszach od 102 do 104 znajduje się funkcja `launch`, odpowiedzialna za uruchomienie selektora folderów.

4.4. NavHost

NavHost to element Androida, jest kontenerem aplikacji, przechowuje aktualny stan nawigacji i jest odpowiedzialny za zarządzanie przejściami i animacjami między fragmentami.

Używany jest głównie w połączeniu z NavController, aby umożliwić użytkownikowi poruszanie się między różnymi ekranami aplikacji.

W aplikacji znajduje się 6 różnych ekranów:

- MainScreen
- MainScreenContent
- AlbumsScreen
- SongsScreen
- AuthenticationScreen
- AudioPlayer

A samych elementów jest 12, nie wliczając że niektóre ekranы posiadają dwie wersje układów w zależności od ustawienia telefonu, np. *AlbumsScreen* pokazany na listingu nr.23.

```
1 @Composable
2 fun AlbumsScreen(
3     navController: NavHostController,
4     author: String,
5 ) {
6     val viewModel: AlbumsScreenViewModel = hiltViewModel<
7         AlbumsScreenViewModel, AlbumsScreenViewModel.Factory>(
8         creationCallback = { it.create(author) }
9     )
10    val albumsAndCovers by viewModel.albumsAndCovers.collectAsState(
11        emptyList()
12    )
13    var selectedAlbum by rememberSaveable { mutableStateOf<Pair<
14        Album, ImageBitmap>?>?>(null) }
15
16    if (selectedAlbum == null) {
17        BoxWithConstraints(
18            modifier = Modifier
19                .fillMaxSize()
20                .paint(
```

```
18             painter = painterResource(id = R.drawable.tans3
19             ),
20             contentScale = ContentScale.Crop,
21             alignment = Alignment.Center
22         ),
23     ) {
24
25         val columns = if (maxWidth < maxHeight) 3 else 5
26
27         LazyVerticalGrid(
28             columns = GridCells.Fixed(columns),
29             modifier = Modifier.fillMaxSize(),
30             contentPadding = PaddingValues(16.dp),
31             verticalArrangement = Arrangement.spacedBy(16.dp),
32             horizontalArrangement = Arrangement.spacedBy(16.dp)
33         ) {
34             items(albumsAndCovers, key = { it.first.albumId })
35             { pair ->
36                 val album = pair.first
37                 val cover = pair.second
38
39                 AlbumTile(
40                     albumName = album.title,
41                     cover = cover,
42                     onClick = {
43                         Log.d("MainActivity", "Album id passed
44                         to navhost: ${album.albumId}")
45                         assert(album.albumId != 0L)
46                         selectedAlbum = pair
47                     },
48                     modifier = Modifier,
49                 )
50             }
51         }
52     }
53 } else {
54     if (isPortrait()) {
55         PortraitView(selectedAlbum = selectedAlbum!!,
56         navController = navController)
57     } else {
58         LandscapeView(selectedAlbum = selectedAlbum!!,
59         navController = navController)
60     }
61 }
```

56 }

Listing 23. Kod Navhost AlbumsScreen

Do zmiany pozycji elementów na ekranie używane są dwie funkcje *PortraitView* pokazany na listingu nr.24 oraz *LandscapeView* przedstawiony na listingu nr.25.

```
1 @Composable
2 fun PortraitView(selectedAlbum: Pair<Album, ImageBitmap>,
3     navController: NavHostController) {
4     Column(
5         horizontalAlignment = Alignment.CenterHorizontally,
6         modifier = Modifier.fillMaxSize().paint(
7             painter = painterResource(id = R.drawable.tans3),
8             contentScale = ContentScale.Crop,
9             alignment = Alignment.Center
10        ),
11    ) {
12        Row(
13            verticalAlignment = Alignment.CenterVertically,
14            modifier = Modifier
15                .fillMaxWidth()
16                .padding(16.dp)
17        ) {
18            Image(
19                bitmap = selectedAlbum.second,
20                contentDescription = "${selectedAlbum.first.title}
21                    cover",
22                    modifier = Modifier
23                    .size(200.dp)
24                    .weight(1f)
25            )
26            Text(
27                text = selectedAlbum.first.title,
28                style = MaterialTheme.typography.headlineMedium,
29                textAlign = TextAlign.Start,
30                color = MaterialTheme.colorScheme.onSurface,
31                modifier = Modifier
32                    .padding(start = 16.dp)
33                    .weight(1f)
34            )
35        SongsScreen(
36            navController = navController,
37            libraryViewModel = hiltViewModel(),
```

```
38         albumId = selectedAlbum.first.albumId
39     )
40 }
41 }
```

Listing 24. Kod Navhost PortraitView

```
1 @Composable
2 fun LandscapeView(selectedAlbum: Pair<Album, ImageBitmap>,
3     navController: NavHostController) {
4     Row(
5         modifier = Modifier.fillMaxSize().paint(
6             painter = painterResource(id = R.drawable.tans3),
7             contentScale = ContentScale.Crop,
8             alignment = Alignment.Center
9         ),
10    ) {
11        Column(
12            modifier = Modifier
13                .fillMaxHeight()
14                .padding(16.dp)
15                .weight(1f)
16        ) {
17            Image(
18                bitmap = selectedAlbum.second,
19                contentDescription = "${selectedAlbum.first.title}
20                cover",
21                modifier = Modifier.size(200.dp)
22            )
23            Text(
24                text = selectedAlbum.first.title,
25                style = MaterialTheme.typography.headlineMedium,
26                textAlign = TextAlign.Center,
27                color = MaterialTheme.colorScheme.onSurface,
28                modifier = Modifier.padding(vertical = 16.dp)
29            )
30        }
31        SongsScreen(
32            navController = navController,
33            libraryViewModel = hiltViewModel(),
34            albumId = selectedAlbum.first.albumId,
35            modifier = Modifier.weight(2f)
36        )
37    }
38 }
```

36 }

Listing 25. Kod Navhost LandscapeView

W tych dwóch funkcjach znajdują się elementy ogólnie używane przez wszystkie ekranы pozwalające na ich wielofunkcyjność, pochodzącą z potrzeby wywoływanie tylko ich części. =====

5. Implementacja

5.1. Zarządzanie bazą danych

5.1.1. Klasa DatabaseManager

Za zarządzanie bazą danych odpowiedzialna jest klasa `DatabaseManager`, której kod jest zamieszczony na listingu nr. 26. Klasa jest wrapperem do bazy danych Room[5] i do niej akcesorów.

```
1 @Singleton
2 class DatabaseManager @Inject constructor(
3     @ApplicationContext context: Context
4 ) {
5     private val database: LibraryDb = Room.databaseBuilder(
6         context,
7         LibraryDb::class.java, "Library"
8     ).build()
9
10    fun collectAuthorsFlow(): Flow<List<Author>> = database.uiDao()
11        .getAllAuthorsFlow()
12
13    fun collectAlbumsByAuthorFlow(authorName: String): Flow<List<
14        Album>> {
15        return database.uiDao().getAuthorWithAlbums(authorName)
16            .map { it.albums }
17    }
18
19    fun collectSongsByAlbumFlow(albumId: Long): Flow<List<Song>> {
20        return database.uiDao().getAlbumWithSongs(albumId)
21            .map { it.songs }
22    }
23
24    ...
25
26    fun populateDatabase(songs: List<TagExtractor.SongInfo>) {
27        assert(Thread.currentThread().name != "main")
28
29        val dao = database.logicDao()
30
31        fun addAuthors() {
32            songs.fastForEach { song ->
33                //TODO: there should be a distinction between
34                albumartists and regular artists
35                song.albumArtists?.fastForEach { name ->
```

```
33             if(dao.getAuthor(name) == null) {
34                 dao.insertAuthor(Author(name = name))
35             }
36         }
37     }
38 }
39
40
41     fun addAlbumsAndRelations() {
42         // FIXME: xddddddd
43         val distinctAlbumArtistsList = songs
44             .map { Triple(it.album, it.albumArtists, it.
45         coverUri) }
46             .distinct()
47         Log.d(javaClass.simpleName, "Distinct artists set:
48             $distinctAlbumArtistsList")
49
50         distinctAlbumArtistsList.fastForEach {
51             val albumTitle = it.first.toString()
52             val artists = it.second
53             val coverUri = it.third
54
55             val albumId = dao.insertAlbum(Album(
56                 title = albumTitle,
57                 coverUri = coverUri.toString(),
58             ))
59
60             artists?.fastForEach {
61                 dao.insertAlbumAuthorCrossRef(
62                     AlbumAuthorCrossRef(
63                         albumId = albumId,
64                         name = it.toString()
65                     )
66                 }
67             }
68         }
69     }
70
71     fun addSongs() {
72         songs.forEach { song ->
73             Log.d(javaClass.simpleName, "NEW SONG\n")
74             Log.d(javaClass.simpleName, "Album artists: ${song.
75         albumArtists}")
76
77             val albumWithAuthorCandidates = dao
78                 .getAlbumsByTitle(song.album.toString())
79         }
80     }
81
82 }
```

```
74         .map { it.albumId }
75             .map { dao.getAlbumWithAuthors(it) }
76             Log.d(javaClass.simpleName, "
77                 $albumWithAuthorCandidates")
78
79             var correctAlbum: Album? = null
80             albumWithAuthorCandidates.fastForEach {
81                 Log.d(javaClass.simpleName, "${song.
82                     albumArtists}, ${it.authors}")
83                     //FIXME: theese guys shouldn't be ordered, will
84                     have to refactor a bunch of
85                     // stuff with sets instead of lists
86                     if(song.albumArtists?.sorted() == it.authors.
87                         map { it.name }.sorted()) {
88                         correctAlbum = it.album
89                     }
90                 }
91             }
92             dao.insertSong(Song(
93                 title = song.title,
94                 albumId = correctAlbum?.albumId,
95                 fileUri = song.fileUri.toString(),
96             ))
97         }
98     }
99 }
100 }
```

Listing 26. Strukatura klasy DatabaseManager

Na pierwszej linijce można zauważyc adnotację `@Singleton`. Pochodzi ona z biblioteki `Hilt`[8]. Powiadamia ona bibliotekę o tym że klasa jest singletonem, czyli że ma istnieć tylko jej jedna instancja na cały program. Uczyniono to, dlatego że baza danych powinna być jedna na całą aplikację. Menadżer z nią interfejsujący, dlatego że jest używany w wielu innych klasach, też powinien mieć tylko jedną instancję, aby nie marnować pamięci.

Na linijce nr. 2, widać konstruktor klasy, do którego też przy użyciu `Hilt`, wstrzykiwany jest `context`.

Następnie, na linijce nr. 5, widać inicjalizację samego obiektu bazy `database`.

Baza jest reprezentowana przez klasę `LibraryDb`, definicję której można zobaczyć w sekcji 5.1.2

Dalej, do linijki nr. 22 pokazane są metody zwracające różne elementy bazy. Większość z tych metod zwraca `Flow[TODO:]`. Room natywnie obsługuje Flows, a dlatego że wymusza dostęp do bazy z innych wątków niż główny, większość operacji wykonywanych na bazie odbywa się za pośrednictwem typów Flow

Same metody są wrapperami do obiektów Dao bazy.Więcej o nich w sekcji 5.1.3. Niektóre obrabiają dane jak np. `collectSongsByAlbumFlow()` na linijce nr. 17., która mapuje zwraca piosenki z wyjściowej klasy relacyjnej.

Metod tych jest więcej, lecz wyglądają one bardzo podobnie. Dla zwięzłości, można je pominać.

Metoda `populateDatabase()` zadeklarowana na linijce nr. 24, jest odpowiedzialna za ładowanie wyjątych z plików informacji do bazy. Jako parametr odstaje ona zmienna `songs` typu `List<TagExtractor.SongInfo>`. Zadeklarowane są w niej trzy funkcje pomocnicze: `addAuthors()`, `addAlbumsAndRelations()` i `addSongs()`. Wywoływanie są one po kolej w metodzie głównej.

Funkcja `addAuthors()`, zadeklarowana na linijce nr. 29, jest prosta w swoim działaniu. Lista z `SongInfo` jest iterowana i po kolej wpisywani są wszyscy autorzy, którzy jeszcze w bazie nie istnieją.

Funkcja `addAlbumsAndRelations()`, zadeklarowana na linijce nr. 41, odpowiada za dodawanie albumów do bazy oraz tworzenie relacji między nimi, a autorami. Tworzona zmienna `distinctAlbumArtistsList` mapuje tylko unikalne pary albumów i autorów (zmienna `coverUri` nie ma znaczenia przy określaniu autorstwa, jest przypisywana tutaj dlatego, że trudno było znaleźć dla niej lepsze miejsce). Dzięki temu początkowemu filtrowaniu, wiadomo, że każdy napotkany album będzie unikalny. Następnie, `distinctAlbumArtistsList` jest iterowana - przy każdej iteracji dodawany jest nowy album do bazy. Metoda `insertAlbum()` zwraca `id` nowo dodanego albumu. Wynik jej jest przypisywany do zmiennej `albumId` na linijce nr. 53. Potem, zostaje przypisywana relacja albumu z autorami. Autorów może być kilku, więc są oni reprezentowani przy każdej iteracji przez listę, która jest iterowana, a relacja zostaje dodawana z nazwą autora i `albumId`.

Funkcja `addSongs()`, zadeklarowana na linijce nr. 67, ma na celu dodanie piosenek do bazy. Ciało funkcji jest w pętli iterującej się przez listę piosenek. Na początku pętli, na linijce nr. 72 deklarowana jest zmienna `albumWithAuthorCandidates`. Jest ona listą relacji album - autorzy wszystkich albumów o tej samej nazwie co ten w danym elemencie listy. Następnie, lista ta jest iterowana i przy każdej iteracji spraw-

dzane jest czy lista autorów w danej relacji jest równa z listą autorów danej piosenki. Jeżeli tak, wartość danego albumu z wybranej relacji jest przypisywana do zmiennej zadeklarowanej na linii nr. 78 `correctAlbum`. Na końcu funkcji, piosenka dodana jest do bazy przy użyciu metody `dao`.

5.1.2. Klasa LibraryDb

Klasa `LibraryDb` jest deklaracją faktycznej instancji bazy danych, która jest implementowana i generowana przez bibliotekę `Room`. Z racji tego, że jest to klasa abstrakcyjna, jej zadaniem jest określenie struktury bazy i jakie komponenty ma ona zawierać

```
1 @Database(entities = [Song::class, Author::class, Album::class,  
           AlbumAuthorCrossRef::class], version  
2 = 1)  
3 abstract class LibraryDb : RoomDatabase() {  
4     abstract fun logicDao(): LogicDao  
5     abstract fun uiDao(): UIDao  
6 }
```

Listing 27. Deklaracja bazy LibraryDb

Jak widać na listingu nr. 27, na początku klasy należy zamieścić adnotację `@Database`. Powiadamia ona bibliotekę `Room` o tym, że następująca klasa jest bazą danych. Parametr `entities` określa wszystkie tabele jakie mają się w klasie zawierać. O tabelach więcej w sekcji nr. 5.1.4. Parametr `version` zajmuje się wersjonowaniem bazy. Jest on ważny przy aktualizacjach aplikacji, aby baza mogła zostać odpowiednio zmieniona.

Na linijce nr. 3 umieszczona jest faktyczna deklaracja klasy. Dziedziczy ona z klasy `RoomDatabase`. Jedyne rzeczy jakie są do dziecięcej klasy dodawane, to metody zwracające obiekty `dao`, opisane w sekcji nr. 5.1.3.

5.1.3. Obiekty Dao

Obiekty `dao` (Data Access Object(s)) to obiekty używane do interakcji z zawartością bazy danych. Głównie używa się ich do dodawania elementów do bazy oraz ich odczytywania. Same obiekty definiuje się jako interfejsy z adnotacją `@Dao`. Są one implementowane przez `Room`. Baza danych w projekcie wykorzystuje dwa interfejsy `dao - UIDao`, którego kod zamieszczony jest na listingu nr. 28 oraz `LogicDao`, którego kod zamieszczony jest na listingu nr. 29.

```
1 @Dao
```

```
2 interface UI Dao {  
3     @Query("SELECT * FROM Song")  
4     fun getAllSongs(): Flow<List<Song>>  
5  
6     @Query("SELECT * FROM Song WHERE songId = :songId")  
7     fun collectSongFromId(songId: Long): Flow<Song>  
8  
9     // Get an album with its songs  
10    @Transaction  
11    @Query("SELECT * FROM album WHERE albumId = :albumId")  
12    fun getAlbumWithSongs(albumId: Long): Flow<AlbumWithSongs>  
13  
14    @Query("SELECT * FROM album WHERE albumId = :albumId")  
15    fun getAlbumById(albumId: Long?): Flow<Album>  
16  
17    // Get an album with its authors  
18    @Transaction  
19    @Query("SELECT * FROM album WHERE albumId = :albumId")  
20    fun getAlbumWithAuthors(albumId: Long?): Flow<AlbumWithAuthors  
?>  
21  
22    @Query("SELECT * FROM Author")  
23    fun getAllAuthorsFlow(): Flow<List<Author>>  
24  
25    // Get an author with their albums  
26    @Transaction  
27    @Query("SELECT * FROM author WHERE name = :name")  
28    fun getAuthorWithAlbums(name: String): Flow<AuthorWithAlbums>  
29 }
```

Listing 28. Deklaracja interfejsu UI Dao

```
1 @Dao  
2 interface Logic Dao {  
3     @Insert  
4     fun insertSong(song: Song)  
5  
6     @Insert  
7     fun insertAlbum(album: Album): Long  
8  
9     @Insert(onConflict = OnConflictStrategy.REPLACE)  
10    fun insertAuthor(author: Author)  
11  
12    @Insert(onConflict = OnConflictStrategy.REPLACE)  
13    fun insertAlbumAuthorCrossRef(albumAuthorCrossRef:  
        AlbumAuthorCrossRef)
```

```
14  
15     @Query("SELECT * FROM Author")  
16     fun getAllAuthors(): List<Author>  
17  
18     @Transaction  
19     @Query("SELECT * FROM album WHERE albumId = :albumId")  
20     fun getAlbumWithAuthors(albumId: Long): AlbumWithAuthors  
21  
22     @Transaction  
23     @Query("SELECT * FROM author WHERE name = :name")  
24     fun getAuthorWithAlbums(name: String): AuthorWithAlbums  
25  
26     @Query("SELECT * FROM author WHERE name = :name")  
27     fun getAuthor(name: String): Author?  
28  
29     @Query("SELECT * FROM album WHERE title = :title")  
30     fun getAlbumsByTitle(title: String): List<Album>  
31  
32     @Query("SELECT * FROM album WHERE title = :title LIMIT 1")  
33     fun getAlbumByTitle(title: String): Album?  
34  
35     @Query("SELECT * FROM AlbumAuthorCrossRef WHERE albumId = :  
36         albumId AND name = :authorName LIMIT 1")  
37     fun getCrossRefByAlbumAndAuthor(albumId: Long, authorName:  
38         String): AlbumAuthorCrossRef?  
39  
40 }
```

Listing 29. Deklaracja interfejsu LogicDao

Dlatego, że baza Room wymaga dostępu do elementów z innego wątku niż główny, LogicDao może być tylko używany w kodzie, o którym wiadomo, że nie jest wykonywany na głównym wątku. UI Dao natomiast, służy ekskluzywnie do zwracania Flowów. Większość elementów związanych z interfejsem w reszcie kodu aplikacji już korzysta z Flowów, więc dao to łatwo jest zintegrować.

Typowy sposób w jaki dodaje się element do bazy znajduje się na linijce nr. 4 w kodzie LogicDao, na listingu nr. 29. Funkcja `insertSong()`, zadnotowana jest `@Insert`. Powiadamia to Room, że funkcja ta odpowiedzialna jest za dodawanie elementu. Parametr `song` to piosenka jaka ma być dodana. W następnej funkcji `insertAlbum()` widać, że funkcje `@Insert` mogą zwracać wartości. W tym przypadku funkcja zwraca `id` nowo dodanego albumu. Można też zwrócić uwagę na

metodę `insertAuthor()` na linijce nr. 8, a w szczególności parametr `onConflict` w adnotacji `@Insert`. Wartość parametru `OnConflictStrategy.REPLACE` mówi bibliotece, aby nie pomijała elementów o tych samych wartościach co już są w tabeli, ale zamieniała starsze na te nowe.

Przykład odczytywania elementu jest dobrze zilustrowany na metodzie `getAuthor()` zadeklarowanej na linijce nr. 27. Adnotacja `@Query` przyjmuje parametr `String`, który jest kwerendą SQL jaka ma być wykonana. Kwerenda `SELECT * FROM author WHERE name = :name` wybiera wszystkich autorów, których pole `name` równe jest parametrowi metody `name` (odnoszenie do parametru w kwerendzie poprzedzone jest znakiem „`:`”). Dlatego że w bazie może być tylko jeden autor z daną nazwą, zwracany jest pojedynczy autor, a nie ich lista. Niektóre metody, jak na linijce nr. 20 `getAlbumWithAuthors()`, używają adnotacji `@Transaction`. W przypadku tej metody, zwraca ona relację, czyli czyta z kilku tabel. Adnotacja `@Transaction` zapewnia, że transakcja jest atomiczna, co za tym idzie, inne wątki nie mogą nagle zmienić wartości jakiejs tabeli.

Interfejs `UIdao` działa podobnie jak `LogicDao`, ale zwraca on tylko i wyłącznie Flows, które są natywnie obsługiwane przez Room.

5.1.4. Tabele

W bibliotece Room, każda tabela to `dataclass` określana adnotacją `@Entity`. Kolumny takiej tabeli to po prostu pola klasy. Klucz danej tabeli jest określany adnotacją `@PrimaryKey`

```
1 @Entity
2 data class Author (
3     @PrimaryKey val name: String
4 )
```

Listing 30. Deklaracja tabeli Author

Tabela `Author`, zawarta na listingu nr. 30, określa autorów. Tabela jest prosta, jedynym polem jest `name`, który jest kluczem.

```
1 @Entity
2 data class Album(
3     @PrimaryKey(autoGenerate = true) val albumId: Long = 0,
4     val title: String,
5     val coverUri: String?,
```

6)

Listing 31. Deklaracja tabeli Album

Tabela **Album**, zawarta na listingu nr. 31, określa albumy. Kluczem jest zmienna **albumId**. Klucz jest generowany automatycznie, dzięki parametrowi adnotacji **autoGenerate**. Pole **title** określa tytuł, a pole **coverUri** określa adres URI okładki.

```

1
2 @Entity(
3     foreignKeys = [
4         ForeignKey(
5             entity = Album::class,
6             parentColumns = ["albumId"],
7             childColumns = ["albumId"],
8             onDelete = ForeignKey.CASCADE
9         )
10    ],
11
12    indices = [Index(value = ["albumId"])]
13 )
14 data class Song(
15     @PrimaryKey(autoGenerate = true) val songId: Long = 0,
16     val title: String?,
17     val albumId: Long?,
18     val fileUri: String?,
19 )

```

Listing 32. Deklaracja tabeli Song

Klasa ta, zawarta na listingu nr. 32, określa tabelę piosenek. Pole **foreignKeys** w adnotacji **@Entity** określa obce klucze, którymi posługuje się klasa. W tym przypadku określone jest to, że pole w **Song albumId** wskazuje na pole w **Album albumId**. Pole **indices** każe indeksować pola z **albumId** ku polepszeniu szybkości bazy. W ciele klasy, pole **title** to tytuł piosenki. Pole **albumId** określa ID albumu, do którego należy dana piosenka.

5.1.5. Relacje

Relacje w Room są określane jako osobne **dataclassy**. Są one zadeklarowane adnotacją **@Relation** w danej klasie. Ponadto umieszczenie elementu w adnotacji

@Embedded, pozwala klasie „przyswoić” pola danego elementu. Dzięki temu klasa może odnosić się do pól danego elementu tak jakby były one bezpośrednio w klasie. Konieczne jest umieszczenie elementu głównego, od którego będzie relacja wyodziła, do tej adnotacji.

5.1.5.1. AlbumWithSongs

Klasa `AlbumWithSongs` na listingu nr. 33, określa relację albumów i piosenek.

```
1 data class AlbumWithSongs(
2     @Embedded val album: Album,
3     @Relation(
4         parentColumn = "albumId",
5         entityColumn = "albumId"
6     )
7     val songs: List<Song>
8 )
```

Listing 33. Deklaracja relacji `AlbumWithSongs`

Relacja łączy pole `albumId` albumu z polem `albumId` piosenek. Pole `songs` zawiera wszystkie piosenki z tą samą wartością pola `albumId` co faktyczny klucz danego albumu.

```
1 @Entity(primaryKeys = ["albumId", "name"])
2 data class AlbumAuthorCrossRef(
3     val albumId: Long,
4     val name: String
5 )
```

Listing 34. Deklaracja tabeli relacji `AlbumAuthorCrossRef`

Tabela na listingu nr. 34 określa relację *M* do *N* między albumami a autorami. Jest to tabela z dwoma kluczami głównymi: `albumId` dla tabeli `Album` i `name` dla tabeli `Author`.

5.1.5.3. AlbumWithAuthors i AuthorWithAlbums

Obie klasy są do siebie bardzo podobne więc zostaną omówione razem.

```
1 data class AlbumWithAuthors(
2     @Embedded val album: Album,
3     @Relation(
```

```
4     parentColumn = "albumId",
5     entityColumn = "name",
6     associateBy = Junction(AlbumAuthorCrossRef::class)
7   )
8   val authors: List<Author>
9 )
```

Listing 35. Deklaracja relacji `AlbumWithAuthors`

```
1 data class AuthorWithAlbums(
2   @Embedded val author: Author,
3   @Relation(
4     parentColumn = "name",
5     entityColumn = "albumId",
6     associateBy = Junction(AlbumAuthorCrossRef::class)
7   )
8   val albums: List<Album>
9 )
```

Listing 36. Deklaracja relacji `AuthorWithAlbums`

Na listingu nr. 35 przedstawiona jest klasa `AlbumWithAuthors`. Definiuje ona relację danego albumu z jego autorami. Dlatego, że relacja jest M do N , w anotacji `@Relation` dodane jest odniesienie do tabeli relacji `AlbumAuthorCrossRef`, opisanej w sekcji nr. 5.1.5.2. Klucze, jakie mają być porównywane są zdefiniowane w parametrach `parentColumn`, dla id albumu i `entityColumn` dla nazwy autora. Wynikiem tej relacji jest lista albumów. Sytuacja wygląda podobnie w `AuthorWithAlbums`, na listingu nr. 36. Tym razem to autor jest rodzicem i oczekujemy od relacji listy albumów danego autora.

5.2. Czujnik światła

Czujnik światła został zaimplementowany za pomocą wbudowanej funkcji. Zadaniem czujnika jest dynamiczna zmiana schematu kolorów aplikacji na podstawie danych otrzymanych z czujnika światła wbudowanego w urządzeniu mobilnym z systemem android.

```
1 @AndroidEntryPoint
2 class MainActivity : FragmentActivity(), SensorEventListener {
3   private lateinit var sensorManager: SensorManager
4   private var lightSensor: Sensor? = null
5   private val _isDarkTheme = mutableStateOf(false)
6   private val isDarkTheme: State<Boolean> = _isDarkTheme
7 }
```

```
8     private val _isAuthenticated = mutableStateOf(false)
9     private val isAuthenticated: State<Boolean> = _isAuthenticated
10
11    override fun onCreate(savedInstanceState: Bundle?) {
12        super.onCreate(savedInstanceState)
13        val biometricAuthenticator = BiometricAuthenticator(this)
14
15        sensorManager = getSystemService(Context.SENSOR_SERVICE) as
16        SensorManager
17        lightSensor = sensorManager.getDefaultSensor(Sensor.
18            TYPE_LIGHT)
19
20        enableEdgeToEdge()
21
22        setContent {
23            val darkTheme by isDarkTheme
24            val authenticated by isAuthenticated
25            RaptorTheme(darkTheme = darkTheme) {
26                Surface(
27                    modifier = Modifier.fillMaxSize(),
28                    color = MaterialTheme.colorScheme.background
29                ) {
30                    if (authenticated) {
31                        MainScreen()
32                    } else {
33                        AuthenticationScreen(
34                            onAuthenticate = {
35                                promptBiometricAuthentication(
36                                    biometricAuthenticator)
37                            }
38                        )
39                    }
40                }
41
42                private fun promptBiometricAuthentication(
43                    biometricAuthenticator: BiometricAuthenticator) {
44                    biometricAuthenticator.PromptBiometricAuth(
45                        title = "Authentication Required",
46                        subtitle = "Please authenticate to proceed",
47                        negativeButtonText = "Cancel",
48                        fragmentActivity = this,
49                        onSuccess = {
```

```
49     runOnUiThread {
50         _isAuthenticated.value = true
51     }
52 },
53 onFailed = {
54 },
55 onError = { errorCode, errorString ->
56 }
57 )
58 }
59
60 override fun onResume() {
61     super.onResume()
62     lightSensor?.let { sensor ->
63         sensorManager.registerListener(this, sensor, SensorManager.
64 SENSOR_DELAY_NORMAL)
65     }
66 }
67
68 override fun onPause() {
69     super.onPause()
70     sensorManager.unregisterListener(this)
71 }
72
73 override fun onSensorChanged(event: SensorEvent?) {
74     if (event?.sensor?.type == Sensor.TYPE_LIGHT) {
75         val lightLevel = event.values[0]
76         val maxLightLevel = lightSensor?.maximumRange ?: 10000f
77
78         _isDarkTheme.value = lightLevel < 0.4 * maxLightLevel
79     }
80 }
81
82     override fun onAccuracyChanged(sensor: Sensor?, accuracy: Int)
83 { }
```

Listing 37. Implementacja czujnika światła w `MainActivity.kt`

Na listingu 37 przedstawiona jest funkcja MainScreen. W wierszu 2 mamy `SensorEventListener`, który jest frameworkm w androidzie który pozwala aplikacji na reagowanie na zmiany odczytywane przez czujniki smartfona. Po dodaniu automatycznie tworzone są funkcje `onSensorChanged`, `onResume`, `onPause`, `onAccuracyChanged`. Implementację sensora zaczynamy od utworzenia zmiennych `sensorma-`

nager oraz lightSensor w wierszach 3 i 4. Zmienna sensormanager odpowiedzialna jest za pobranie menadżera czujników z poziomu systemu. Zmienna lightSensor odpowiedzialna jest za uzyskanie referencji do głównego czujnika urządzenia, jeżeli się nie uda to zwraca wartość null. Zmienna _isdarkTheme w wierszu 5 określa czy aplikacja wykorzystuje obecnie tryb ciemny, zmienna isDarkTheme w wierszu 6 pomaga jej w tym za pomocą odczytu w UI.

W SetContent w wierszu 23 do dynamicznej zmiany kolorów wykorzytywane jest "RaptorTheme(darkTheme = darmTheme)" zdefiniowany w Theme.kt w folderze ui.Theme.

Poniżej, w wierszach od 60 do 65 znajduje się funkcja onResume, która rejestruje słuchacza zdarzeń po wznowieniu działania aplikacji. odpowiedzialne jest za częstotliwość aktualizacji(SENSOR_DELAY_NORMAL).

This oznacza implementację SensorEventListener.

Funkcja onpause odpowiedzialna jest zatrzymanie działania słuchacza zdarzeń w przypadku pracy aplikacji w tle.

Funkcja onSensorChanged wywoływaną jest za każdym razem gdy uzyskany zostanie nowy odczyt z czujnika systemowego. Zmienna lightLevel pobiera aktualny poziom oświetlenia(jednostka to luks). Zmienna maxLightLevel pobiera maksymalny zakres pomiarowy czujnika. W przypadku braku przypisana zostanie wartość 10000 luksów. W wierszu 77 znajduje się instrukcja przejścia w tryb ciemny jeżeli obecny wykrywany poziom światła jest mniejszy niż 40 procent. Funkcja onAccuracyChanged nie jest tutaj wykorzystywana. Może być wykorzystana do np. zmiany dokładności wykrywania czujnika.

5.3. Autoryzacja odciskiem palca

Autoryzacja odciskiem palca działa w następujących krokach:

1. Sprawdzenie, czy uwierzytelnianie biometryczne jest dostępne, przedstawione na listingu nr 38
2. Zbudowanie monitu biometrycznego, przedstawione na listingu nr 39
3. Obsługa wyniku monitu biometrycznego, przedstawiona na listingu nr 40

```
1 enum class BiometricAuthenticationStatus(val id: Int) {  
2     READY(1),  
3     NOT_AVAILABLE(-1),  
4     TEMPORARY_NOT_AVAILABLE(-2),
```

```

5   AVAILABLE_BUT_NOT_ENROLLED (-3)
6 }
7
8 fun isBiometricAuthAvailable(): BiometricAuthenticationStatus {
9   return when (biometricmanager.canAuthenticate(BIOMETRIC_STRONG)) {
10    BiometricManager.BIOMETRIC_SUCCESS ->
11    BiometricAuthenticationStatus.READY
12    BiometricManager.BIOMETRIC_ERROR_NO_HARDWARE ->
13    BiometricAuthenticationStatus.NOT_AVAILABLE
14    BiometricManager.BIOMETRIC_ERROR_HW_UNAVAILABLE ->
15    BiometricAuthenticationStatus.TEMPORARY_NOT_AVAILABLE
16    BiometricManager.BIOMETRIC_ERROR_NONE_ENROLLED ->
17    BiometricAuthenticationStatus.AVAILABLE_BUT_NOT_ENROLLED
18    else -> BiometricAuthenticationStatus.NOT_AVAILABLE
19  }
20 }
```

Listing 38. Sprawdzenie dostępności uwierzytelnienia biometrycznego

Na zamieszczonym listingu funkcja `BiometricAuthenticationStatus` tworzy stany gotowości uwierzytelniania. Funkcja `isBiometricAuthAvailable` jest odpowiedzialna za sprawdzenie czy telefon obsługuje uwierzytelnianie biometryczne oraz czy w telefonie są zapisane odciski palca. `biometricmanager.canAuthenticate(BIOMETRIC_STRONG)` jest odpowiedzialna za zapytanie systemu, o możliwość użycia uwierzytelniania silnego biometrycznego, w zależności od odpowiedzi systemu zostanie zwrócony odpowiednio zdefiniowany status.

```

1 fun PromptBiometricAuth(
2   title: String,
3   subtitle: String,
4   negativeButtonText: String,
5   fragmentActivity: FragmentActivity,
6   onSuccess: (result: BiometricPrompt.AuthenticationResult) -> Unit
7     ,
8   onFailed: () -> Unit,
9   onError: (errorCode: Int, errorMessage: String) -> Unit,
10 ) {
11   when(isBiometricAuthAvailable()) {
12     BiometricAuthenticationStatus.NOT_AVAILABLE -> {
13       onError(BiometricAuthenticationStatus.NOT_AVAILABLE.id, "Not
14         available on this device")
15     }
16     BiometricAuthenticationStatus.TEMPORARY_NOT_AVAILABLE -> {
```

```
16     onError(BiometricAuthenticationStatus.  
17             TEMPORARY_NOT_AVAILABLE.id, "Not available at this moment")  
18     return  
19 }  
20 BiometricAuthenticationStatus.AVAILABLE_BUT_NOT_ENROLLED -> {  
21     onError(BiometricAuthenticationStatus.  
22             AVAILABLE_BUT_NOT_ENROLLED.id, "Add a fingerprint")  
23     return  
24 }  
25     else -> Unit  
26 }  
27 biometricPrompt = BiometricPrompt(  
28     fragmentActivity,  
29     object: BiometricPrompt.AuthenticationCallback() {  
30         override fun onAuthenticationSucceeded(result:  
31             BiometricPrompt.AuthenticationResult) {  
32             super.onAuthenticationSucceeded(result)  
33             onSuccess(result)  
34         }  
35         override fun onAuthenticationError(errorCode: Int, errString:  
36             CharSequence) {  
37             super.onAuthenticationError(errorCode, errString)  
38             onError(errorCode, errString.toString())  
39         }  
40         override fun onAuthenticationFailed() {  
41             super.onAuthenticationFailed()  
42             onFailed()  
43         }  
44     promptinfo = BiometricPrompt.PromptInfo.Builder()  
45         .setTitle(title)  
46         .setSubtitle(subtitle)  
47         .setNegativeButtonText(negativeButtonText)  
48         .build()  
49     biometricPrompt.authenticate(promptinfo)  
50 }
```

Listing 39. Sprawdzenie monitu biometrycznego

Funkcja `PromptBiometricAuth` odpowiedzialna jest za tworzenie monitu, który zostanie wyświetlony użytkownikowi podczas włączenia aplikacji. Monit zawiera tytuł, podtytuł oraz przycisk negatywny. Od wiersza 10 do 24 znajduje się instruk-

cia when która jest odpowiedzialna za sprawdzenie dostępności uwierzytelniania. Następnie w wierszach od 25 do 43 tworzony jest nowy obiekt który będzie obsługiwał okno dialogowe. `object:BiometricPrompt.AuthenticationCallback()` odpowiedzialne jest za implementację metody obsługi zdarzeń związanych z uwierzytelnianiem. Jeżeli uwierzytelnianie się powiedzie to wywoływana jest metoda `onAuthenticationSucceeded`, która przekazuje wynik jako argument do `PromptBiometricAuth`, co umożliwia odblokowanie aplikacji. `onAuthenticationError` wywoływanie jest w przypadku wystąpienia błędu. `onAuthenticationFailed` jest wywoływanie w przypadku niepowodzenia uwierzytelniania, np. niewłaściwy odcisk palca. Od wiersza 44 do 49 znajduje się prompt builder, odpowiedzialny za skonfigurowanie danych które będą wyświetlane w monicie, na samym końcu w wierszu 49 wywołana jest metoda.

```

1  private fun promptBiometricAuthentication(biometricAuthenticator: BiometricAuthenticator) {
2      biometricAuthenticator.PromptBiometricAuth(
3          title = "Authentication Required",
4          subtitle = "Please authenticate to proceed",
5          negativeButtonText = "Cancel",
6          fragmentActivity = this,
7          onSuccess = {
8              runOnUiThread {
9                  _isAuthenticated.value = true
10             }
11         },
12         onFailed = {
13     },
14         onError = { errorCode, errorMessage ->
15     }
16     )
17 }
```

Listing 40. Obsługa wyniku monitu

Funkcja w MainActivity, przekazuje dane do tworzonego monitu, w przypadku pomyslnego uwierzytelnienia, IsAuthenticated jest ustawiany jako true, dzięki czemu aplikacja wie że można opuścić ekran logowania oraz przejść do wczytywania reszty aplikacji. Fragment kodu odpowiedzialny za załadowanie ekranu uwierzytelniania przed przejściem do ekranu głównego znajduje się w `onCreate`, przedstawionym na listingu nr 41.

```

1  if (authenticated) {
2      MainScreen()
3  } else {
```

```

4     AuthenticationScreen(
5         onAuthenticate = {
6             promptBiometricAuthentication(biometricAuthenticator)
7         }
8     )
9 }
10

```

Listing 41. Zawartość onCreate

5.4. Odczyt i przetwarzanie plików

5.4.1. Tag Extractor

Klasa `TagExtractor` jest odpowiedzialna za wyciąganie tagów z piosenek. Każda piosenka zawiera tagi, na które składają się: nazwa artysty, nazwa artystów z albumu, tytuł, data wydania, nazwa albumu, URI piosenki, URI obrazka cover. Deklaracje tych tagów pokazane są na listingu nr 42.

```

1  data class SongInfo(
2      val artists: List<String>?,
3      val albumArtists: List<String>?,
4      val title: String?,
5      val releaseDate: String?,
6      val album: String?,
7      val fileUri: Uri?,
8      val coverUri: Uri?,
9  )
10

```

Listing 42. Deklaracja tagów

Metoda `buildSongInfo()` przedstawiona na listingu nr 43 jest odpowiedzialna za parsowanie metadanych. Otrzymuje

```

1  @OptIn(UnstableApi::class)
2  private fun buildSongInfo(metadata: Metadata, uri: Uri?):
3      SongInfo {
4      val metadataList = mutableListOf<Metadata.Entry>()
5      for(i in 0 until metadata.length()) {
6          metadataList.add(metadata.get(i))
7      }
8      Log.d("${javaClass.simpleName}", "Metadata list: $metadataList")
9      when(metadataList[0]) {

```

```
10     is VorbisComment -> {
11         fun handleMissingTags(map: MutableMap<String?, Any?>) {
12             if(map["ALBUMARTIST"] == null) map["ALBUMARTIST"] = List<
13 String>(1, {"Unknown"} )
14         }
15
16         val entryMap: MutableMap<String?, Any?> = mutableMapOf()
17
18         // the last element 'picture' screws up the logic and it
19         // only has a mimetype value
20         // which i think is useless
21         metadataList.take(metadataList.size - 1).fastForEach {
22             val entry = it as VorbisComment
23
24             val key = entry.key
25             val value = entry.value
26             when(key) {
27                 "ALBUMARTIST", "ARTIST" -> {
28                     if(!entryMap.containsKey(key)) {
29                         entryMap[key] = mutableListOf<String?>(value)
30                     } else {
31                         (entryMap[key] as? MutableList<String?>)?.add(value
32                     )
33                     }
34                 }
35             }
36             else -> {
37                 // assert(!entryMap.containsKey(key))
38                 if(entryMap.containsKey(key)) {
39                     Log.w(javaClass.simpleName, "Unhandled duplicate
key: $key")
40                     return@fastForEach
41                 }
42                 entryMap[key] = value
43             }
44         }
45
46         handleMissingTags(entryMap)
47
48         val coverUri = imageManager.extractAlbumimage(
49             uri,
50             entryMap["ALBUMARTIST"] as List<String>,
51             entryMap["ALBUM"] as String
52         )
53     }
54 }
```

```
51
52     return SongInfo(
53         artists = entryMap["ARTIST"] as? List<String>?,
54         albumArtists = entryMap["ALBUMARTIST"] as List<String>,
55         title = entryMap["TITLE"] as? String?,
56         album = entryMap["ALBUM"] as String?,
57         releaseDate = entryMap["DATE"] as? String?,
58         fileUri = uri,
59         trackNumber = (entryMap["TRACKNUMBER"] as? String?)?.toInt()
60     ),
61     coverUri = coverUri,
62     ).also {
63         Log.d(javaClass.simpleName, "Vorbis Song: $it")
64     }
65
66
67     is Id3Frame -> {
68         fun handleMissingTags(map: MutableMap<String, List<String>>)
69             >> {
70             if (map["TPE2"] == null) map["TPE2"] = List<String>(1, {"Unknown"})
71         }
72
73         val entryMap: MutableMap<String, List<String>> =
74             mutableMapOf()
75             metadataList.fastForEach {
76                 val entry = it as Id3Frame
77                 Log.d(javaClass.simpleName, "Id3 metadata: $entry")
78
79                 when (entry) {
80                     is TextInformationFrame -> {
81                         entryMap[entry.id] = entry.values
82                     }
83
84                     else -> {
85                         Log.w(javaClass.simpleName, "Unimplemented id3 frame: $entry")
86                     }
87                 }
88             }
89
90             handleMissingTags(entryMap)
91
92             val coverUri = imageManager.extractAlbumImage()
```

```
91     uri ,
92     entryMap["TPE2"]?: emptyList() ,
93     entryMap["TALB"]?.get(0).toString()
94
95   )
96
97   return SongInfo(
98     artists = entryMap["TPE1"] ,
99     albumArtists = entryMap["TPE2"] as List<String> ,
100    title = entryMap["TIT2"]?.get(0) ,
101    releaseDate = entryMap["TDA"]?.get(0) ,
102    album = entryMap["TALB"]?.get(0) ,
103    fileUri = uri ,
104    trackNumber = entryMap["TRCK"]?.get(0)?.let {
105      return@let it.takeWhile { it != '/' }.toInt()
106    },
107    coverUri = coverUri
108  ).also {
109    Log.d(javaClass.simpleName, "id3 Song: $it")
110  }
111 }
112
113 else -> {
114   metadataList.fastForEach {
115     Log.w(
116       javaClass.simpleName,
117       "Unhanded tag format: ${it::class.simpleName}, metadata:
$it"
118     )
119   }
120
121   return SongInfo(
122     null, mutableListOf("Unknown"), null, null, null, null,
123     null, null
124   )
125 }
126 }
```

Listing 43. Metoda `buildSongInfo()`

Instrukcje w wierszach od 3 do 6 odpowiedzialne są za tworzenie listy metadanych. Następujeinicjalizacja pustej listy `metadataList`. Następnie pętla przeszukuje po `metadata.length`, następnie dodaje każdy wpis do listy. W wierszach od 9 do 120 znajduje się pętla while, odpowiedzialna za warunkowe sprawdzanie formatu danych.

Instrukcja składa się z tzech części: `VorbisComment`, `Id3Frame` oraz `else`.

1. `VorbisComment` znajdujące się w wierszach od 10 do 65 odpowiedzialne jest za parsowanie gdy mamy do czynienia z formatem metadanych typu Vorbis. Funkcja `handleMissingTags()` w wierszach od 11 do 13 jest odpowiedzialna za wypełnienie brakujących pól domyślnymi wartościami. Następnie w wierszach od 15 do 42 tworzona jest mapa wejścia oraz parsowanie. W wierszach od 44 do 50 następuje wywołanie funkcji uzupełniającej brakujące tagi oraz ekstrakcja coveru piosenki. Wyodrębnia wbudowany w plik audio cover oraz zapisuje go do pliku w pamięci aplikacji. W wierszach od 52 do 63 tworzony jest obiekt `SongInfo`.
2. `Id3Frame` znajduje się w wierszach od 67 do 111. Odpowiedzialne za parsowanie gdy mamy do czynienia z formatem Id3Frame. Działa podobnie do poprzednika. Mamy funkcję wewnętrzną która tworzy brakujące tagi, tworzona jest mapa oraz następuje parsowanie, wywołanie funkcji tworzącej brakujące tagi oraz ekstrakcja coveru i na koniec tworzony jest obiekt `SongInfo` dla tego formatu.
3. `else` znajduje się w wierszach od 113 do 124. Jest wykonywana gdy natrafimy na nieobsługiwany przez aplikację format. Ostrzeżenie jest zapisywane do logów oraz zwracane jest `SongInfo` zawierające minimalne informacje.

Metoda `TagExtractor` przedstawiona na listingu 44. Metoda jest odpowiedzialna za skanowanie plików audio przekazywanych jako lista `SongFile` oraz wyodrębnienia ich z metadanych.

```
1  @OptIn(UnstableApi::class)
2  fun extractTags(fileList: List<MusicFileLoader.SongFile>): List<
3      SongInfo> {
4      val tagsList = mutableListOf<SongInfo>()
5
6      for(file in fileList) {
7          val mediaItem = MediaItem.fromUri("${file.uri}")
8
9          val trackGroups = MetadataRetriever.retrieveMetadata(context,
10             mediaItem).get()
11
12          if(trackGroups != null) {
13              // Parse and handle metadata
14              assert(trackGroups.length == 1)
15
16              val tags = trackGroups[0]
```

```
15     .getFormat(0)
16     .metadata
17     .let {
18         buildSongInfo(
19             it!!,
20             file.uri
21         )
22     }
23
24     tagsList.add(tags)
25 }
26 }
27 return tagsList
28 }
```

Listing 44. Metoda TagExtractor()

Instrukcja w wierszu 3 tworzy pustą listę do której będą wkładane obiekty SongInfo.

Pętla for przechodzi przez każdy plik SongFile oraz tworzy obiekt MediaItem w wierszu 6, pobiera metadane w wierszu 8, instrukcja if w wierszach od 10 do 25 jest odpowiedzialna za walidację i parsowanie, sprawdza, czy trackGroups nie jest null. assert w wierszu 12 zakłada, że metadane będą w jednym tracku. Następnie w tags uzysujemy dostęp do metadanych oraz parsujemy w buildSongInfo. W wierszu 24 dodajemy wynik do tagList. W wierszu 27 zwracamy tagList.

5.4.2. MusicFileLoader

Zadanie MusicFileLoader znalezienie wszystkich plików muzycznych z wybranego przez użytkownika folderu. Na listingu nr 45.

```
1 package com.example.raptor
2
3 import android.content.Context
4 import android.content.Intent
5 import android.net.Uri
6 import android.provider.DocumentsContract
7 import android.util.Log
8 import androidx.activity.compose.ManagedActivityResultLauncher
9 import androidx.activity.compose.
10    rememberLauncherForActivityResult
11 import androidx.activity.result.contract.ActivityResultContracts
12 import androidx.compose.runtime.Composable
13 import androidx.compose.ui.util.forEach
```

```
14 import dagger.hilt.android.qualifiers.ApplicationContext
15 import kotlinx.coroutines.flow.MutableStateFlow
16 import javax.inject.Inject
17
18 /**
19 * Handles the loading of music files in a given directory and its
20     subdirectories as well as
21 * launching a file picker instance
22 */
23 class MusicFileLoader
24 @Inject constructor( @ApplicationContext private val context:
25     Context) {
26
27     /**
28     * Dataclass representing a song file
29     *
30     * @param filename Name of the file
31     * @param uri 'Uri' corresponding to the file
32     * @param mimeType The type of the file
33     */
34     data class SongFile(val filename: String, val uri: Uri, val
35     mimeType: String)
36
37     /**
38     * Observable list of 'SongFile' currently loaded
39     */
40     var songFileList = MutableStateFlow<List<SongFile>>(emptyList())
41
42     private set
43
44     private lateinit var launcher : ManagedActivityResultLauncher<
45         Uri?, Uri?>
46
47     private fun traverseDirs(treeUri: Uri): List<SongFile> {
48         val _songFiles = mutableListOf<SongFile>()
49
50         fun visit(uri: Uri) {
51             val root = DocumentFile.fromTreeUri(context, uri)
52             Log.d(javaClass.simpleName, "Visiting dir: ${root?.name}")
53             val childDirs = root?.listFiles()?.filter { it.isDirectory }
54
55             childDirs?.fastForEach { visit(it.uri) }
56
57             val songFiles = root?.listFiles()
58             ?.filter {
```

```
53         it.type?.slice(0..4) == "audio"
54     }
55     ?.map {
56         SongFile(
57             filename = it.name.toString(),
58             uri = it.uri,
59             mimeType = it.type.toString()
60         )
61     }
62
63     Log.d(javaClass.simpleName, "Visited dir ${root?.name},
64 songs: ${songFiles?.map { it
65             .filename
66         }}")
67
68     _songFiles.addAll(songFiles?: emptyList())
69
70     visit(treeUri)
71
72     return _songFiles
73 }
74
75 /**
76 * \@Composable function that prepares the file picker launcher
77 *
78 * Must be called before 'launch()'
79 */
80
81 @Composable
82 fun PrepareFilePicker() {
83     val contentResolver = context.contentResolver
84
85     launcher = rememberLauncherForActivityResult(
86         contract = ActivityResultContracts.OpenDocumentTree()
87     ) { treeUri: Uri? ->
88         treeUri?.let {
89             val permissions = Intent.FLAG_GRANT_READ_URI_PERMISSION
90             or
91                 Intent.FLAG_GRANT_WRITE_URI_PERMISSION
92             contentResolver.takePersistableUriPermission(treeUri,
93             permissions)
94
95             Log.d(javaClass.simpleName, "Selected: $it")
96             songFileList.value = traverseDirs(treeUri)
97         }
98     }
99 }
```

```
95      }
96    }
97  }
98
99  /**
100 * Launches the file picker
101 */
102 fun launch() {
103   launcher.launch(null)
104 }
105 }
```

Listing 45. Kod MusicFileLoader

Na początku znajduje się klasa danych `SongFile` w wierszu 31. Służy do przechowywania informacji o pojedynczym pliku audio. W wierszach 36 i 37 jest zmienna `songFileList`, służącą do przechowywania listy obiektów `SongFile`. W wierszu 39 znajduje się zmienna `launcher`, który zwraca uri folderu wybranego przez użytkownika. Funkcja `TraverseDirs` w wierszach od 41 do 74 jest odpowiedzialna za rekurencyjne przeszukiwanie folderu określonego przez `treeUri`. Zbiera wszystkie pliki audio znajdujące się w tym folderze. Najpierw tworzona jest lista `_songFiles` która będzie przechowywać wyniki. następnie tworzona jest funkcja wewnętrzna `visit` która wywołuje obiekt reprezentujący wybrany folder, następnie pliki są listowane oraz oddzielane od folderu. Następnie rekurencyjnie wywołujemy dla każdego katalogu `visit(it.uri)`. Następnie w bieżącym katalogu filtrowane są pliki w wierszach od 52 do 54. Następnie z każdego pliku tworzony jest obiekt `SongFile` a następnie dołączany do `_SongFiles`. Na koniec zwracane jest `_SongFiles`. Następnie znajduje się funkcja `PrepareFilePicker` w wierszach od 82 do 97. Funkcja jest odpowiedzialna za przygotowanie launchera. `rememberLauncherForActivityResult` pozwala na stworzenie obiektu launchera który pozwoli na otwarcie `OpenDocumentTree`. Kontrakt `ActivityResultContracts.OpenDocumentTree()` pozwala na wybranie całego folderu. Następnie w wierszach od 88 do 96 znajdują się instrukcje definiujące uprawnienia, aby zachować uprawnienia potrzebne jest

`takePersistableUriPermission().`

`songFileList.value = traverseDirs(treeUri)` służy do aktualizacji flow, URI folderu przekazywane jest do funkcji `traverseDirs`.

Na koniec w wierszach od 102 do 104 znajduje się funkcja `launch`, odpowiadająca za uruchomienie selektora folderów.

5.5. Ładowanie obrazów albumów

Za ładowanie obrazów z plików muzycznych odpowiedzialna jest klasa `ImageManager`, kod której umieszczony jest na listingu nr. 46.

```
1 class ImageManager @Inject constructor(@ApplicationContext private
2     val context: Context) {
3
4     fun extractAlbumimage(
5         uri: Uri?,
6         artistNames: List<String>,
7         albumName: String
8     ): Uri? {
9         val retriever = MediaMetadataRetriever()
10
11
12         retriever.setDataSource(context, uri)
13         val pictureBytes = retriever.embeddedPicture
14
15
16         pictureBytes?.let {
17             bitmapFile.writeBytes(pictureBytes)
18         }
19
20         retriever.release()
21
22         return bitmapFile.toUri()
23     }
24
25     fun getBitmapFromAppStorage(uri: Uri?): ImageBitmap {
26         Log.d(javaClass.simpleName, "Collecting bitmap with uri: "
27             + uri)
28         if(uri != null) {
29             try {
30                 context.contentResolver.openInputStream(Uri.parse(
31                     uri.toString())).use {
32                     return BitmapFactory.decodeStream(it)
33                         .asImageBitmap()
34                 }
35             } catch(e: FileNotFoundException) {
36                 Log.e(javaClass.simpleName, "Can't parse thumbnail
at: " + uri)
37                 return ImageBitmap(1,1)
38             }
39         }
40     }
41 }
```

```
37     } else {
38         return ImageBitmap(1, 1, )
39     }
40 }
41 }
```

Listing 46. Struktura klasy `ImageManager`

Na początku klasy, do konstruktora, wstrzykiwana jest zależność `context` kontekstu aplikacji. Klasa potrzebuje go, dlatego, że wchodzi w szerszą interakcję z funkcjami systemowymi.

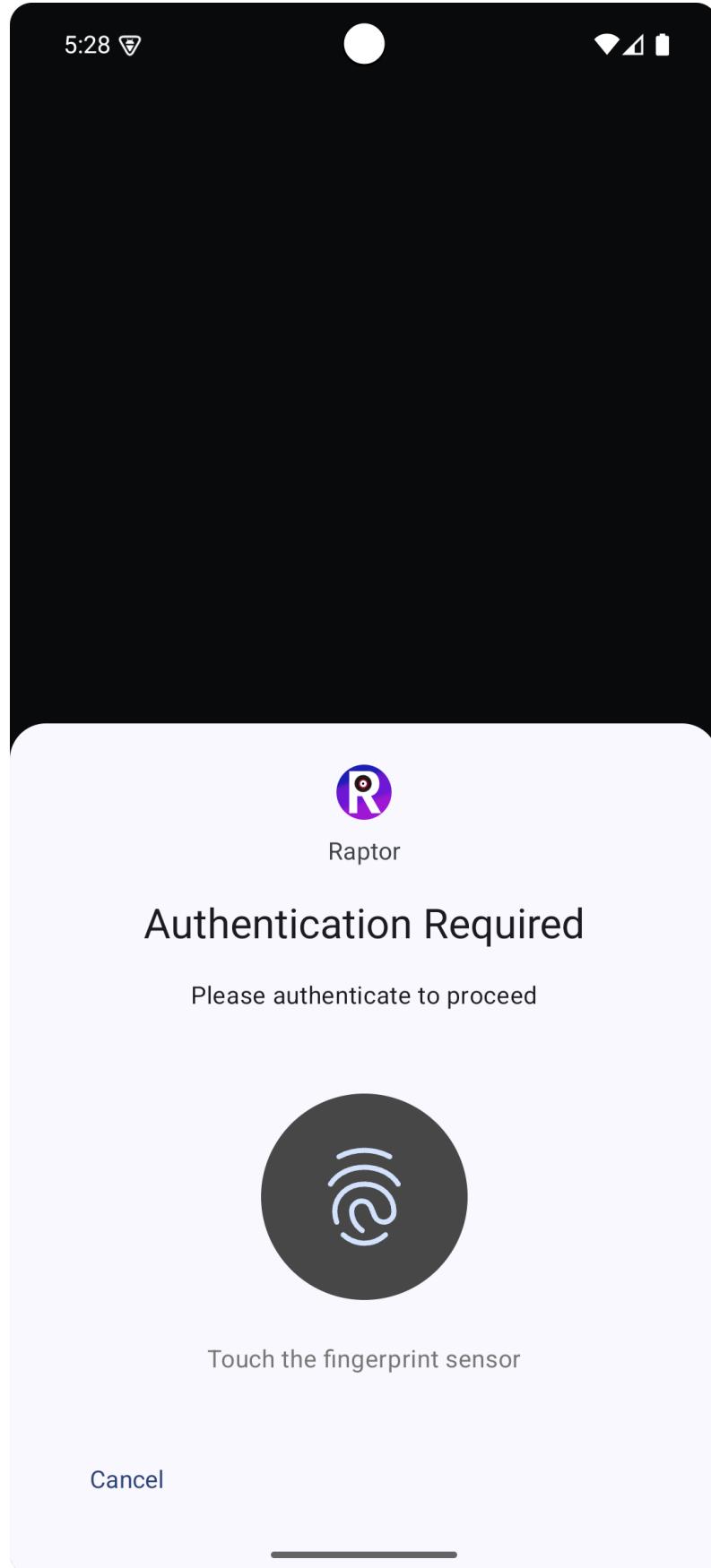
Metoda `extractAlbumimage()` zadeklarowana na linijce nr. 2, odpowiedzialna jest za wyciąganie miniaturki albumu z pliku i umieszczanie jej w katalogu głównym aplikacji. Parametr `uri` to odnośnik do danego pliku muzycznego. Parametry `artistNames` i `albumName` to respektownie lista nazw autorów danej piosenki i nazwa albumu. Dwa ostatnie parametry będą potrzebne do nazwania danej miniaturki później w metodzie. Na początku metody, czyli linijce nr. 7, tworzony jest obiekt `MediaMetadataRetriever` i przypisany do zmiennej `retriever`. Klasa ta służy do wyciągania informacji z plików muzycznych. W tym przypadku pozwoli ona wyciągnąć miniaturkę w danym pliku zawartą. Na linijce nr. 9, ustawiane jest źródło dla `retirevera` jako parametr `uri`. Następnie, tworzona jest zmienna `pictureBytes`, do której przypisywane są surowe bajty tworzące miniaturkę. Zmienna `bitmapFile` reprezentuje instancję pliku do jakiego ma być zapisany bitmap. Struktura nazwy pliku to `<nazwa_artysty1;nazwa_artysty2...>;nazwa_albumu>`. Następnie, `pictureBytes` jest zapisywane do pliku `bitmapFile`. Na końcu funkcji, na linijce nr. 20, zwalniane jest miejsce potrzebne na `retrievera` i na linijce nr. 22 zwracane jest `uri bitmapFile`.

Druga metoda w klasie to `getBitmapFromAppStorage()`, zadeklarowana na linijce nr. 25. Ma ona na celu wczytanie miniaturki z pamięci aplikacji, nie z pliku muzycznego. Parametr `uri` to odnośnik do pliku z miniaturką. Na linijce nr. 27 sprawdzane jest czy parametru `uri` istnieje. Jeżeli tak to otwierany jest plik przy pomocy `uri`, używając API `contentResolver`. Używając zwróconego przez niego strumienia, tworzony jest bitmap, który jest konwertowany do klasy `ImageBitmap`, używanej bezpośrednio przez `Compose`. Jeżeli `uri` nie istnieje, zostaje zwracany placeholderowy `ImageBitmap`, jak widać na linijce nr. 37.

6. Testowanie

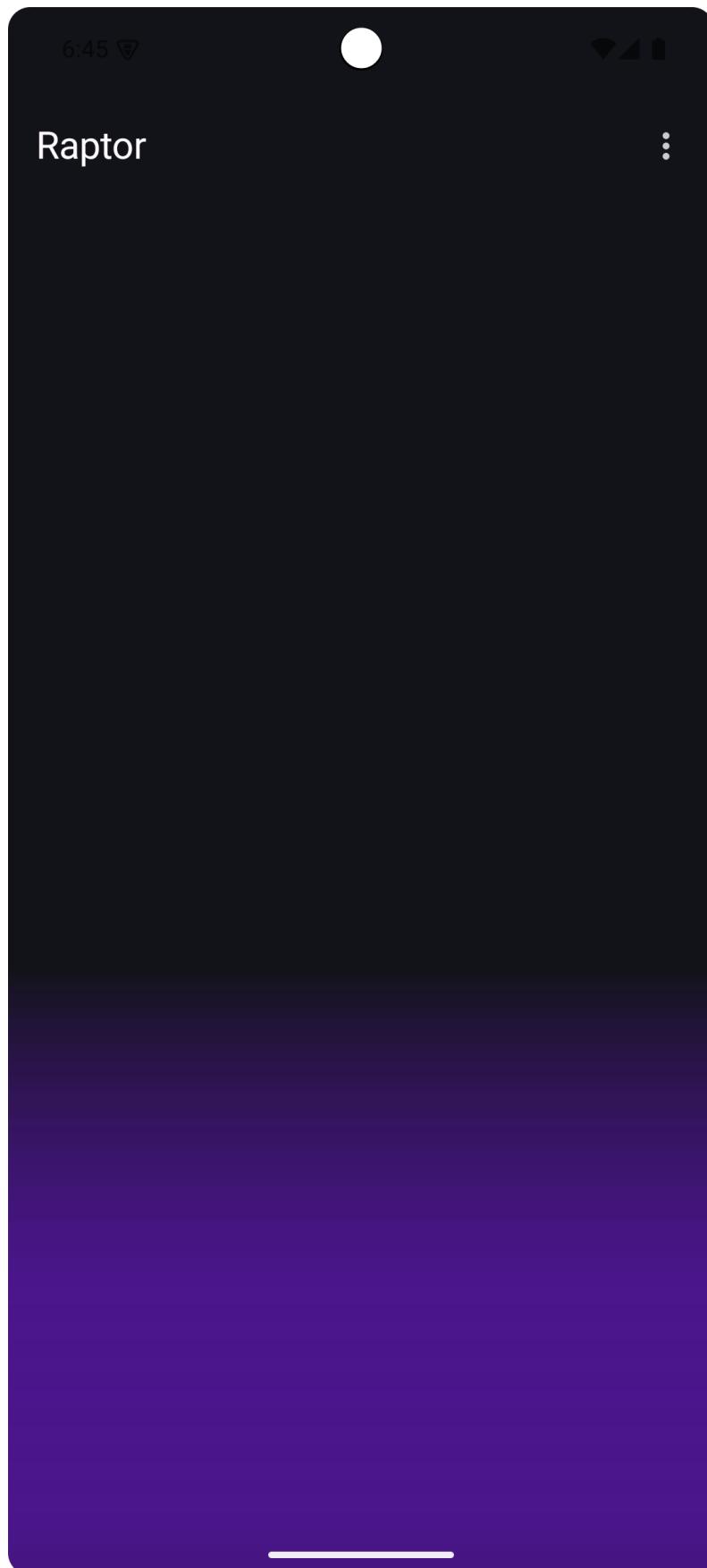
7. Podręcznik użytkownika

Przed włączeniem aplikacji należy upewnić się, że w telefonie użytkownika zapisany jest odcisk palca. Jest on potrzebny do dostania się do aplikacji.



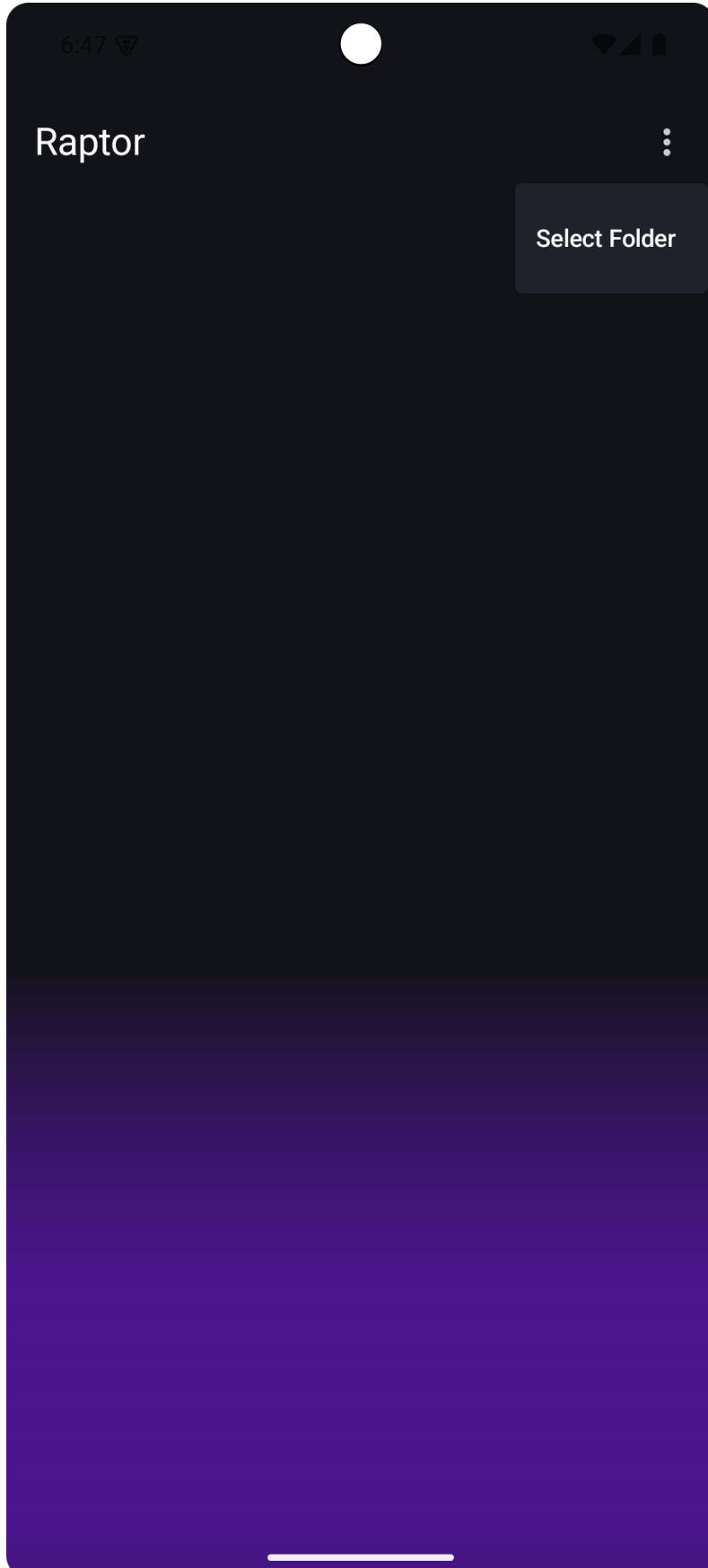
Rys. 7.1. Prośba o podanie odcisku palca.

Po włączeniu aplikacji użytkownik zostanie poproszony o podanie odcisku palca jak widać na rysunku nr. 7.1. Po zweryfikowaniu, aplikacja przechodzi na główny ekran. Domyślnie, wygląda on jak na rysunku nr. 7.2



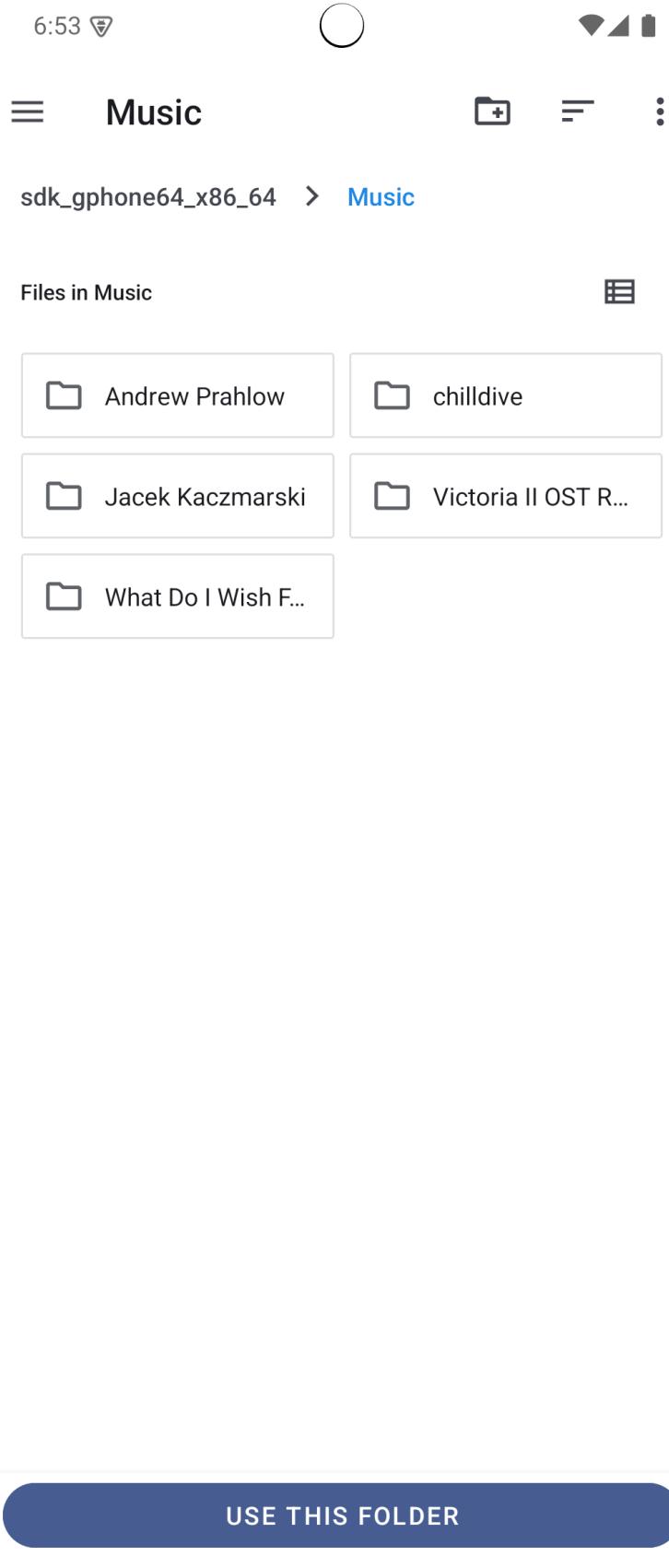
Rys. 7.2. Widok po zweryfikowaniu odcisku.

Jeżeli użytkownik chce dodać swoje piosenki, powinien nacisnąć przycisk w postaci trzech kropek w prawym górnym rogu i wybrać opcję select folder, jak widać na rysunku nr. 7.3.



Rys. 7.3. Włączenie wyboru folderu.

Otworzy się następnie systemowy dialog wyboru folderu. Użytkownik powinien wybrać jakiś, jak na rys. nr. 7.4. Nie musi mieć on bezpośrednio w sobie plików z muzyką, może zawierać podfoldery.



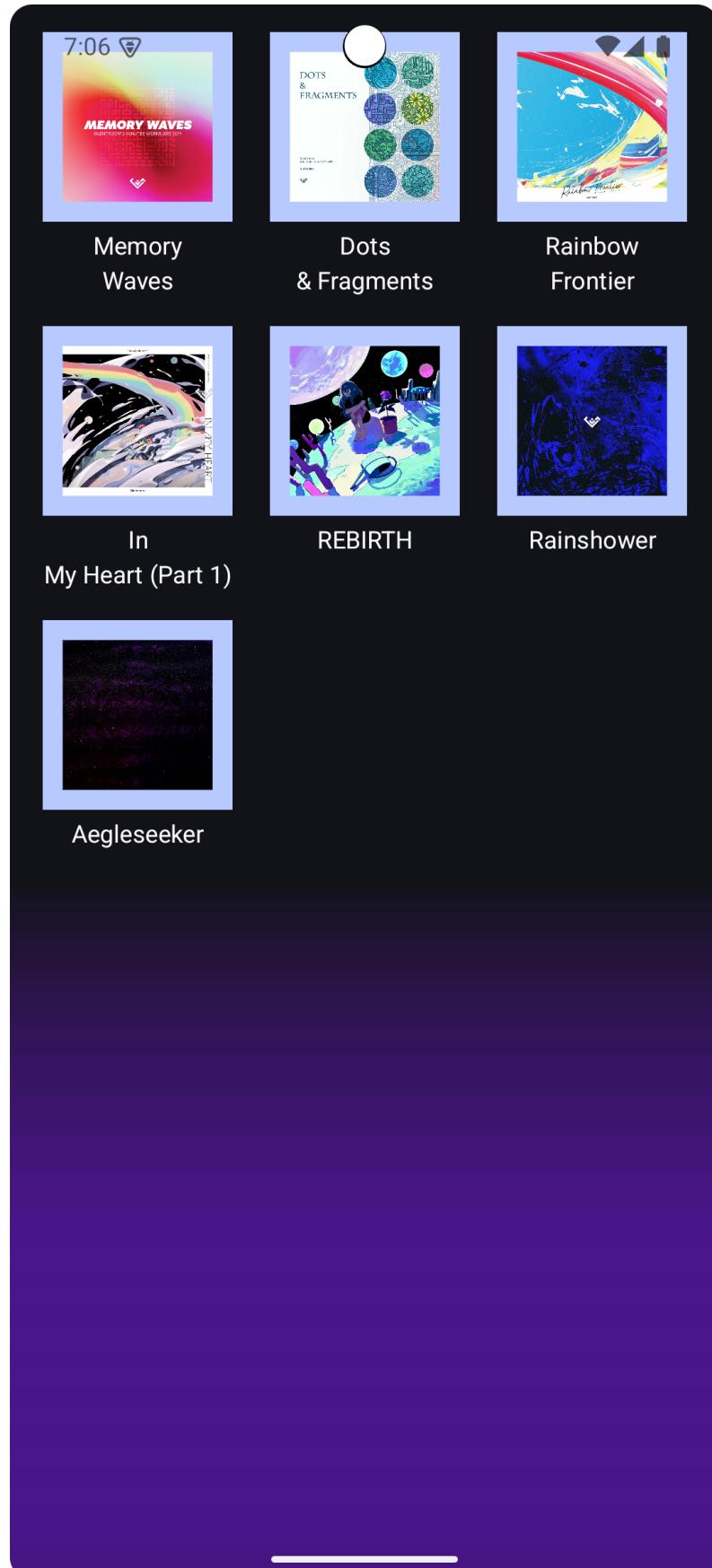
Rys. 7.4. Wybieranie folderu z muzyką.

Po wybraniu folderu, aplikacja wraca użytkownika do ekranu głównego. Po poczekaniu chwili, pokaże się lista autorów, jakie aplikacja wykryła. Jeżeli jakieś pliki nie mają w swoich tagach autora, zostają one przypisane do sekcji **Unknown**. Nowy widok jest przedstawiony na rysunku nr. 7.5.



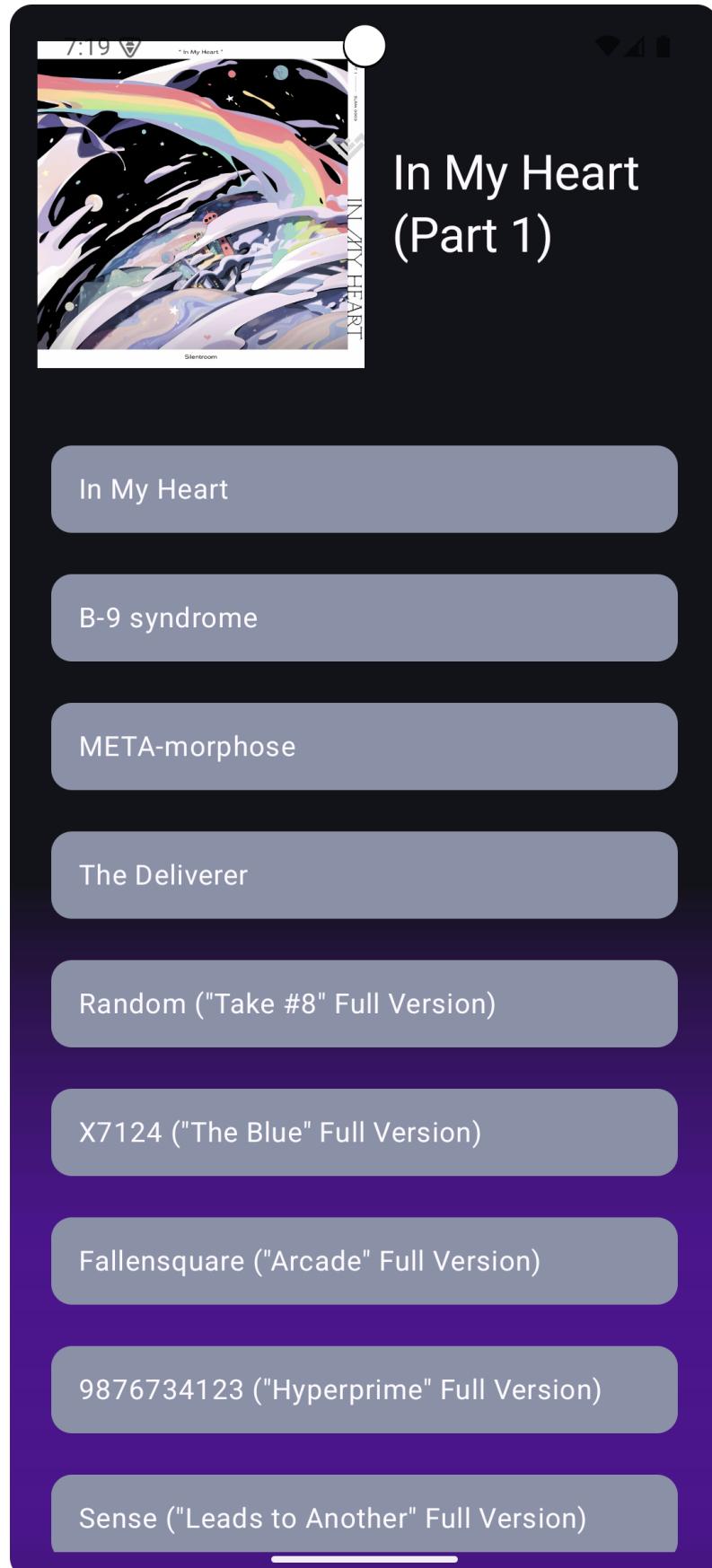
Rys. 7.5. Widok po załadowaniu plików.

Użytkownik może teraz wejść w katalog albumów dowolnego autora, klikając na kafelek z jego nazwą. Przeniesie to użytkownika do widoku albumów, wyglądający jak na rysunku nr. 7.6.



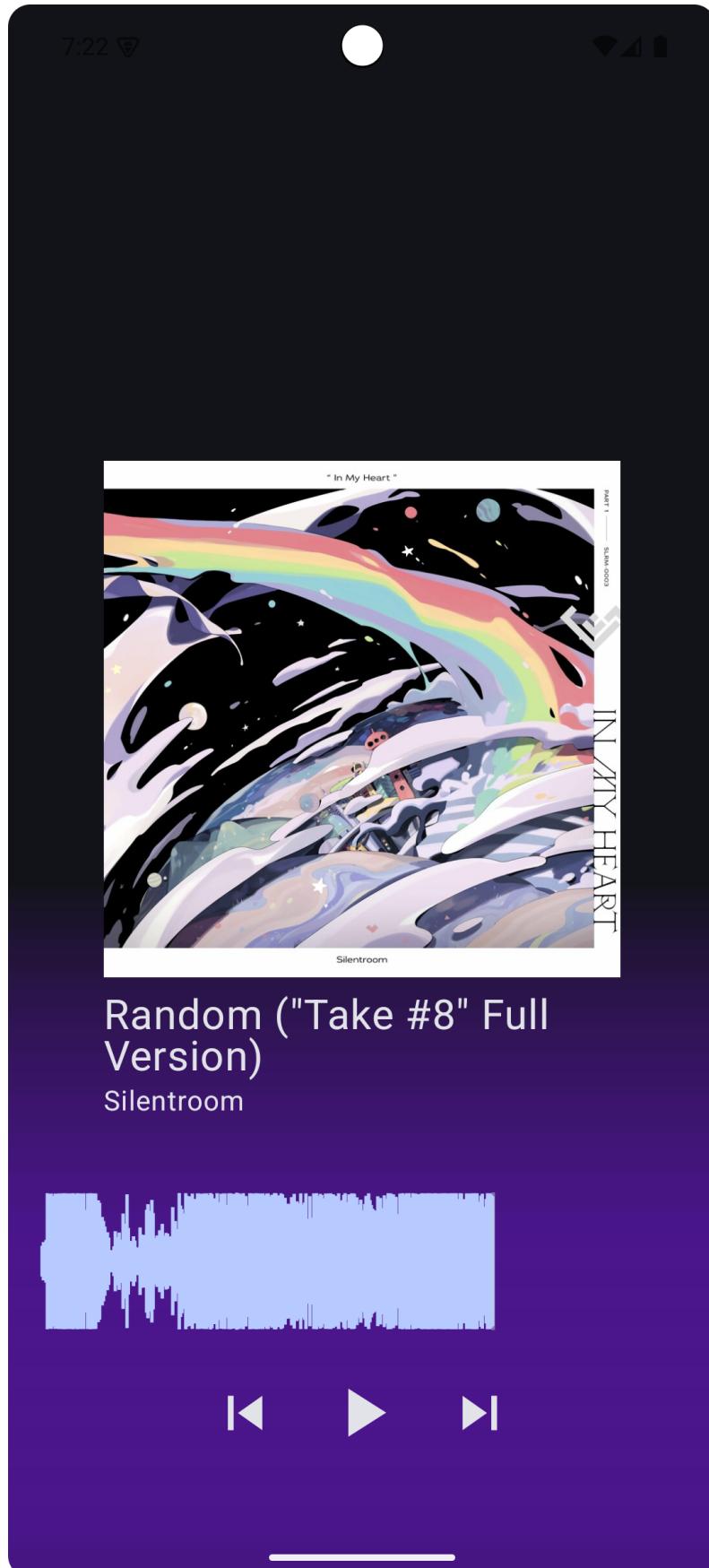
Rys. 7.6. Widok po wejściu w autora.

Analogicznie, należy kliknąć na kafelek korespondujący do odpowiedniego albumu. Aplikacja wtedy przejdzie do widoku piosenek, gdzie można faktycznie wybrać piosenkę do odtworzenia, co pokazana no rys. nr. 7.7.



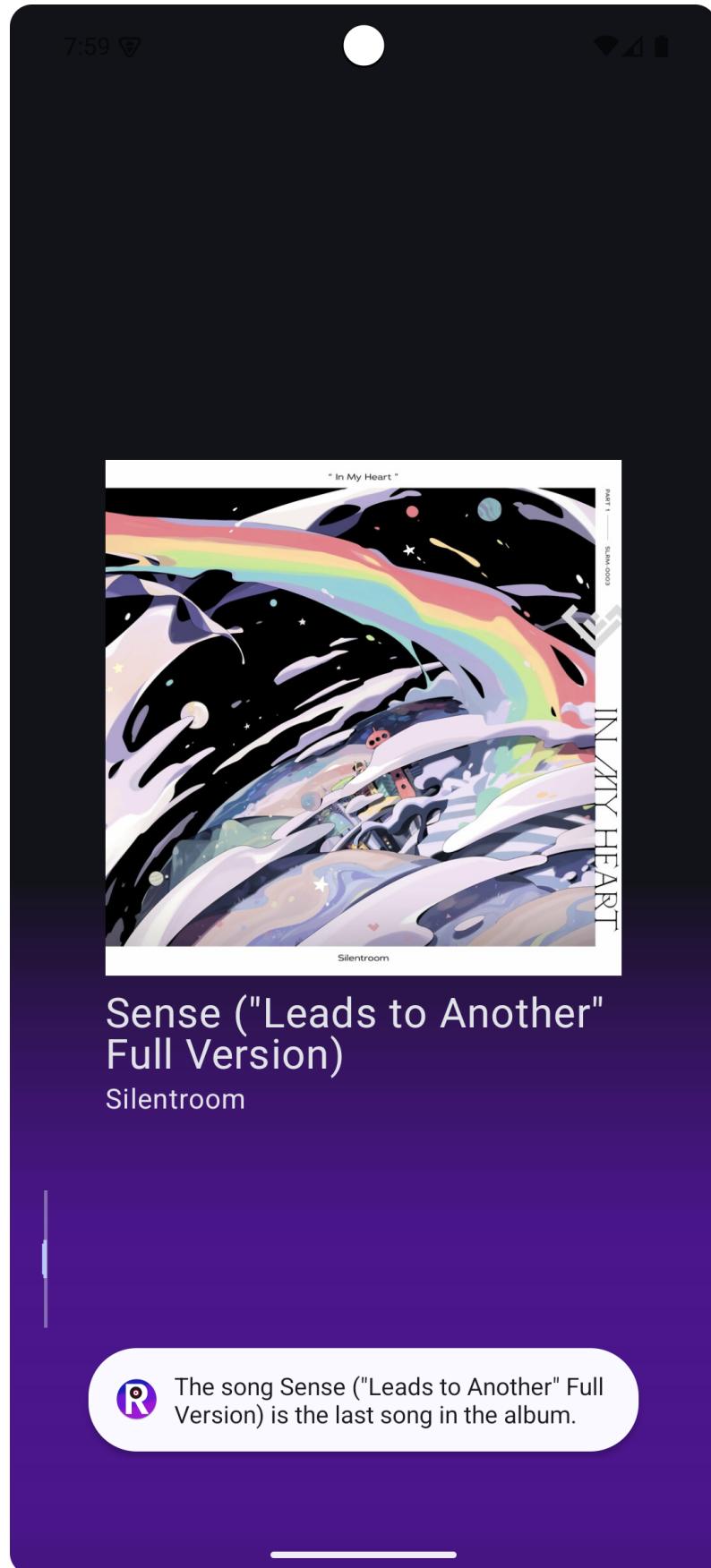
Rys. 7.7. Widok wyboru piosenki.

Po wybraniu piosenki, aplikacja otwiera ekran odtwarzacza, pokazany na rys. nr. 7.8



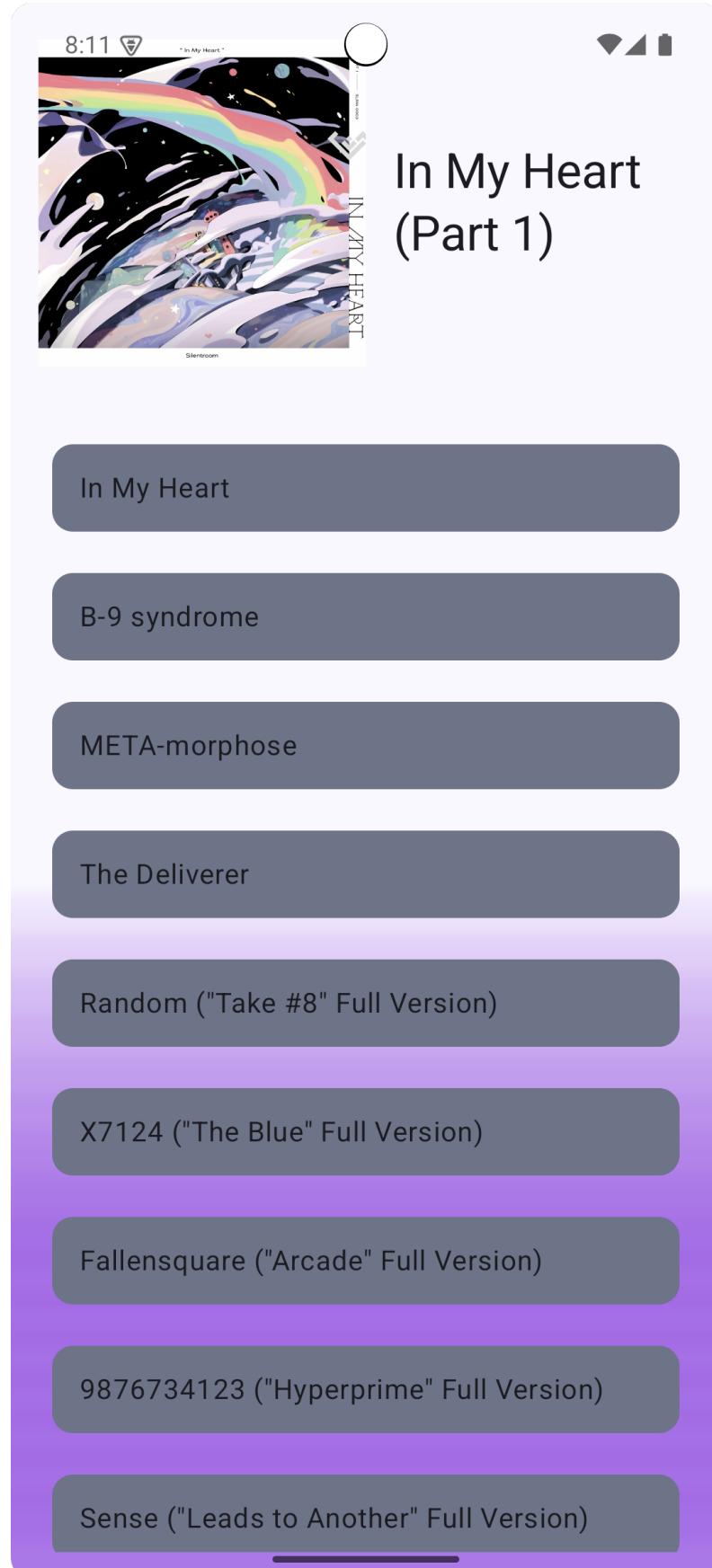
Rys. 7.8. Widok odtwarzacza.

Waveform na dole odtwarzacza pokazuje postęp odtwarzania. Przycisk na środku, pod nim pełni funkcje odtwarzania, pauzowania i restartowania piosenki, zależnie od tego w jakim stanie jest odtwarzanie. Przyciski po lewej i prawej jego stronie odtwarzają poprzednie i następne utwory. Kolejność odtwarzania determinowana jest poprzez numer w albumie piosenki, jeżeli jest on w tagach. Czyli, dla piosenki o numerze 5, klikając w lewy przycisk, odtwarzanie przejdzie do piosenki o numerze 4 w albumie. Przy prawej strzałce, odtwarzanie przejdzie do piosenki nr. 6. Jeżeli użytkownik spróbuje przejść do piosenki, której numer nie istnieje w albumie, to aplikacja powiadomi go o tym, co widać na rys. nr. 7.9.



Rys. 7.9. Próba zagrania następnej piosenki, grając ostatnią.

Można też zwrócić uwagę na inne funkcje aplikacji. Aplikacja obsługuje standar-dową nawigację w Androidzie, można wyjść z odtwarzacza wbudowanym przyciskiem w tył / przesunięciem palca, łącznie z innymi ekranami. Ponadto zademonstrować można inne czujniki obsługiwane przez aplikację. Jeżeli zwiększy się ilość światła, to aplikacja przejdzie w jasny tryb (bądź ciemny jeżeli światło się zmniejszy), jak pokazano na rys. nr. 7.10.



Rys. 7.10. Aplikacja zmieniła kolorystykę na jasną.

Ekran można też obrócić o 90°, co zmieni nieco układ aplikacji. Pokazano to na rys. nr. 7.11



Rys. 7.11. Obrócenie ekranu.

Bibliografia

- [1] *Dokumentacja Jetpack Compose.* URL: <https://developer.android.com/compose>.
- [2] *Dokumentacja Biblioteki Media3.* URL: <https://developer.android.com/media/media3>.
- [3] *Dokumentacja modułu AudioProcessor.* URL: <https://developer.android.com/reference/androidx/media3/common/audio/AudioProcessor>.
- [4] *Dokumentacja Modułu Exoplayer w Media3.* URL: <https://developer.android.com/media/media3/exoplayer>.
- [5] *Dokumentacja Biblioteki Room.* URL: <https://developer.android.com/jetpack/androidx/releases/room>.
- [6] *Dokumentacja Sensor API.* URL: https://developer.android.com/develop/sensors-and-location/sensors/sensors%5C_overview.
- [7] *Dokumentacja modułu MediaRecorder.* URL: <https://developer.android.com/media/platform/mediarecorder>.
- [8] *Dokumentacja Biblioteki Hilt.* URL: <https://developer.android.com/training/dependency-injection/hilt-android>.

Spis rysunków

1.1.	Mockup widoku biblioteki - listing wykonawców	5
1.2.	Mockup widoku albumów danego wykonawcy	6
1.3.	Mockup widoku wyboru utworu	6
1.4.	Mockup widoku wyboru utworu	7
3.1.	Model relacji bazy	12
7.1.	Prośba o podanie odcisku palca.	76
7.2.	Widok po zweryfikowaniu odcisku.	78
7.3.	Włączenie wyboru folderu.	80
7.4.	Wybieranie folderu z muzyką.	82
7.5.	Widok po załadowaniu plików.	84
7.6.	Widok po wejściu w autora.	86
7.7.	Widok wyboru piosenki.	88
7.8.	Widok odtwarzacza.	90
7.9.	Próba zagrania następnej piosenki, grając ostatnią.	92
7.10.	Aplikacja zmieniła kolorystykę na jasną.	94
7.11.	Obrócenie ekranu.	95

Spis tabel

Spis listingów

1.	kotlin001 - Funkcje	9
2.	kotlin002 - Zmienne	9
3.	Strukutura klasy DatabaseManager	15
4.	Deklaracja bazy LibraryDb	19
5.	Deklaracja interfejsu UIdao	19
6.	Deklaracja interfejsu LogicDao	20
7.	Deklaracja tabeli Author	22
8.	Deklaracja tabeli Album	22
9.	Deklaracja tabeli Song	23
10.	Deklaracja relacji AlbumWithSongs	24
11.	Deklaracja tabeli relacji AlbumAuthorCrossRef	24
12.	Deklaracja relacji AlbumWithAuthors	24
13.	Deklaracja relacji AuthorWithAlbums	25
14.	Implementacja czujnika światłą w MainActivity.kt	25
15.	Sprawdzenie dostępności uwierzytelnienia biometrycznego	28
16.	Sprawdzenie monitu biometrycznego	29
17.	Obsługa wyniku monitu	31
18.	Zawartosć onCreate	31
19.	Deklaracja tagów	32
20.	Metoda buildSongInfo()	32
21.	Metoda TagExtractor()	36
22.	Kod MusicFileLoader	37
23.	Kod Navhost AlbumsScreen	41
24.	Kod Navhost PortraitView	43
25.	Kod Navhost LandscapeView	44
26.	Strukutura klasy DatabaseManager	46
27.	Deklaracja bazy LibraryDb	50
28.	Deklaracja interfejsu UIdao	50
29.	Deklaracja interfejsu LogicDao	51
30.	Deklaracja tabeli Author	53
31.	Deklaracja tabeli Album	53

32. Deklaracja tabeli Song	54
33. Deklaracja relacji <code>AlbumWithSongs</code>	55
34. Deklaracja tabeli relacji <code>AlbumAuthorCrossRef</code>	55
35. Deklaracja relacji <code>AlbumWithAuthors</code>	55
36. Deklaracja relacji <code>AuthorWithAlbums</code>	56
37. Implementacja czujnika światłą w <code>MainActivity.kt</code>	56
38. Sprawdzenie dostępności uwierzytelnienia biometrycznego	59
39. Sprawdzenie monitu biometrycznego	60
40. Obsługa wyniku monitu	62
41. Zawartość <code>onCreate</code>	62
42. Deklaracja tagów	63
43. Metoda <code>buildSongInfo()</code>	63
44. Metoda <code>TagExtractor()</code>	67
45. Kod <code>MusicFileLoader</code>	68
46. Struktura klasy <code>ImageManager</code>	72