AKADEMIA NAUK STOSOWANYCH
W NOWYM SĄCZU

Wydział Nauk Inżynieryjnych
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

## 1.1. Ogólny zarys wymagań

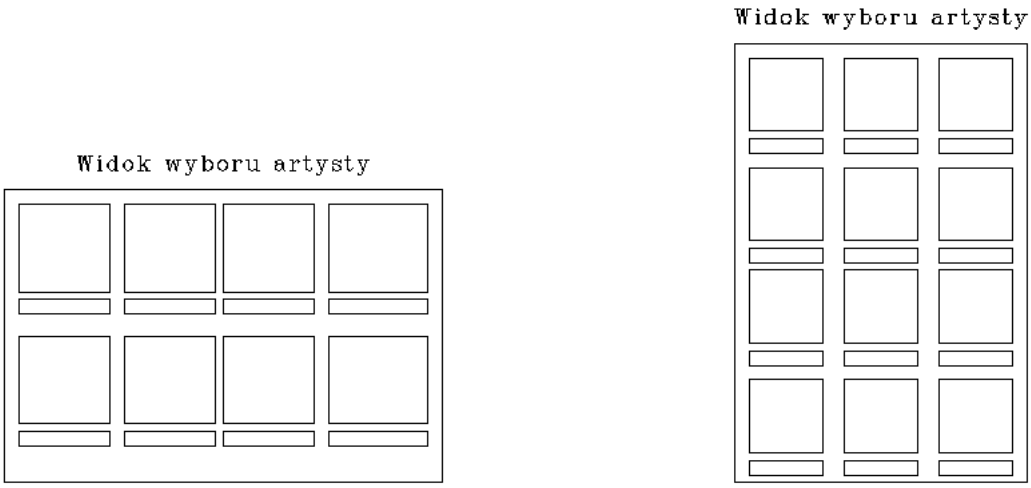
Celem programu jest pełnienie funkcji odtwarzacza muzyki oraz dodatkowo ma on pełnić rolę dyktafonu. Program będzie mógł skanować dany folder, a w nim tagi zawartych plików muzycznych i tworzyć na jego podstawie graficzną reprezentację biblioteki.

## 1.2. Wykorzystane czujniki

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

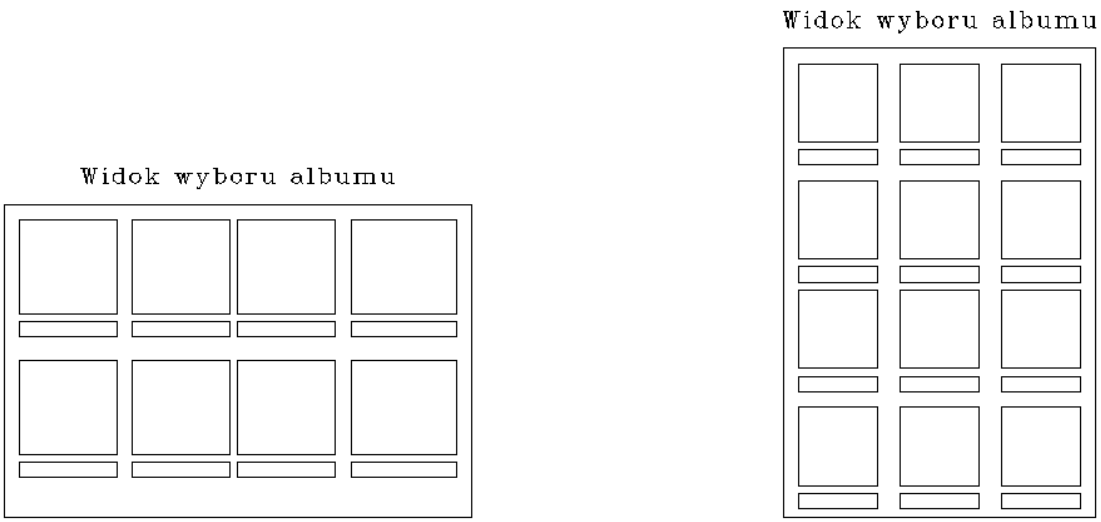
- Żyroskop - Interfejs programu będzie się zmieniał w zależności od orientacji urządzenia.
- Mikrofon - Program będzie posiadał funkcję nagrywania dźwięku. Nagrane pliki będzie można odtwarzać w odtwarzaczu
- Czujnik światła - Interfejs programu będzie mógł zmieniać swoje kolory w zależności od wykrytego poziomu światła na czujniku

## 1.3. Zarys interfejsu



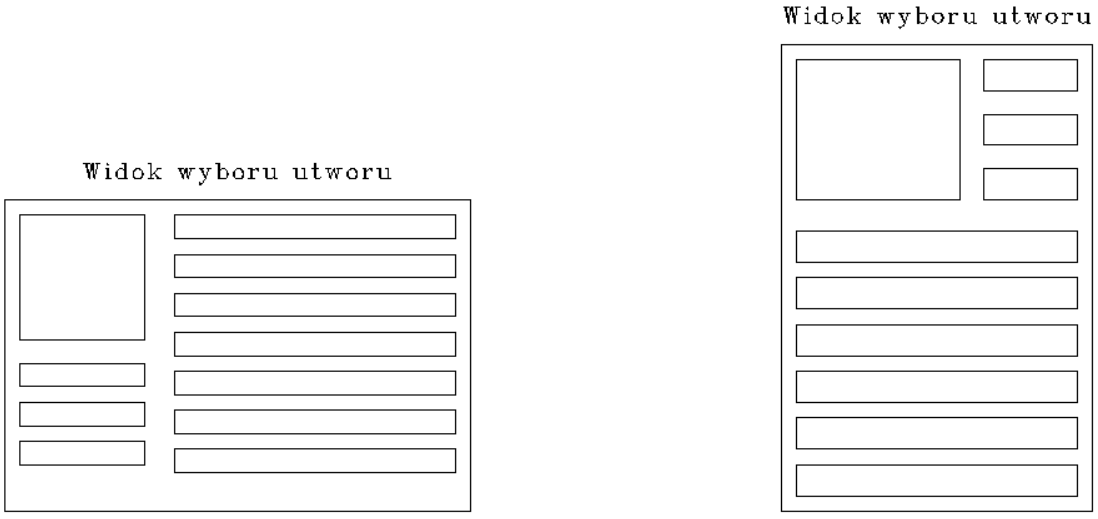

**Rys. 1.1.** Mockup widoku biblioteki - listing wykonawców

Widok wykonawców, jest przedstawiony na rysunku 1.1. Ten widok będzie ekranem startowym aplikacji. "Kafelki"będa zdjęciami wykonawców. Klikanie na jeden z nich przejdzie do widoku albumów danego wykonawcy




**Rys. 1.2.** Mockup widoku albumów danego wykonawcy

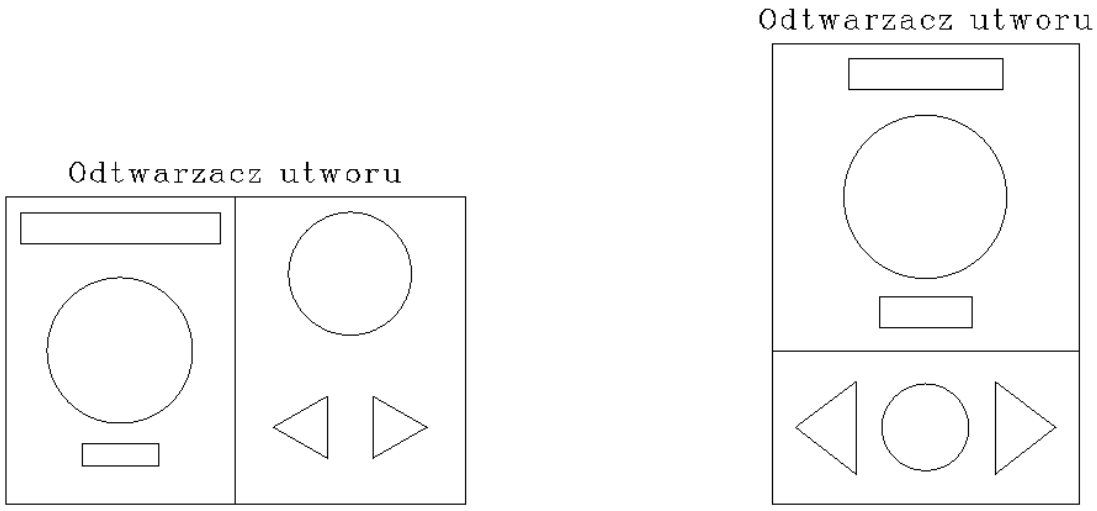
Widok albumów jest przedstawiony na rysunku 1.2. Widok będzie identyczny jak widok wykonawców. Jedyna różnica polega tym, że zdjęcia na kafelkach będą zdjęciami albumów.




**Rys. 1.3.** Mockup widoku wyboru utworu

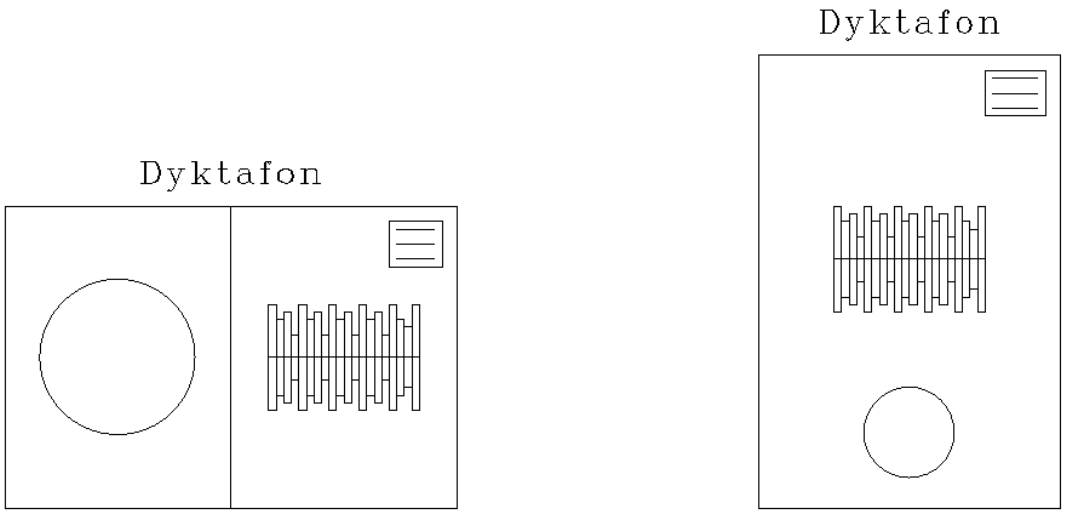
Rysunek 1.3 przedstawia okno pokazujące się po wybraniu albumu. Po wejściu na jakiś album zaprezentowane zostaną zawarte w nim utwory. W lewym górnym jest zdjęcie danego albumu, a obok niego jest kilka informacji o albumie jak wykonawca,

data, tytuł. Dłuższe paski to lista tytułów piosenek, które można kliknąć, aby daną piosenkę włączyć.




**Rys. 1.4.** Mockup odtwarzacza

Wygląd interfejsu odtwarzacza został zaprezentowany na rysunku 1.4. Odtwarzacz będzie działał następująco: duże koło będzie stylizowane na płytę, gdzie wypełniona ona będzie obrazem albumu. Płyta ta będzie się kręcić w czasie gdy gra piosenka. Kąt płyty (od 0°, do 360°) będzie określał jak duża część piosenki została odtworzona. Kąt ten będzie określony jeszcze niezdefiniowanym efektem graficznym. Prostokąty wokół płyty to tytuł piosenki, a na dole czas grania. Kółko i wokół niego trójkąty to przyciski odtwarzania - graj/pauza, następny, poprzedni.




**Rys. 1.5.** Mockup dyktafonu

Na rysunku 1.5 zaprezentowany został interfejs dyktafonu. Do dyktafonu będzie można się dostać przesuwając palcem w prawo na ekranie wykonawców. Dyktafon jest aktywowany wielkim okrągłym przyciskiem. Te kreski obok niego to wizualizacja dźwięku z mikrofonu. Menu w rogu będzie pozwalało na m.in. skonfigurowanie folderu zapisu nagrań.

# 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`[**doc˙compose**] Używając wbudowanych bibliotek w SDK Androida, będzie mogła odczytywać pliki ze wskazanego folderu. Odczytywanie tagów z plików odbędzie się za pomocą biblioteki `Media3`[**doc˙media3**]. 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`[**doc˙audioproc**]. Odtwarzaniem pliku będzie zajmowała się biblioteka `ExoPlayer`[**doc˙exoplayer**], 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`[**doc˙room**]. 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 InteliJ 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 Stdio, 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 Kotlina jako rekomendowany język do tworzenia aplikacji na Android.

Składnia Kotlina wygląda następująco:

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

**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.

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

**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[**doc˙sensorapi**], 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[**doc˙mediarecorder**]

- 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[**doc˙sensorapi**]

## 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. Opis przygotowania narzędzi

## 3.2. Założenie programu

W tym rodziale przedstawiona zostanie ogólna zasada działania programu.

Głównym celem programu jest odtwarzanie muzyki.

## 3.3. Przedstawienie menu

Program składa się z trzech okien.

- Okno wyboru autora - AuthorsView()
- Okno wyboru albumu - AuthorView()
- Okno wyboru utworów - SongView()

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

Po wybraniu autora włączane jest menu wyboru albumu.

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

## 3.4. Odczyt i przetwarzanie plików

## 3.5. Strukutura bazy danych

### 3.5.1. 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. 3.5.3. 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. 3.5.2.

### 3.5.2. Opis pól tabel

#### 3.5.2.1. Autorzy

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

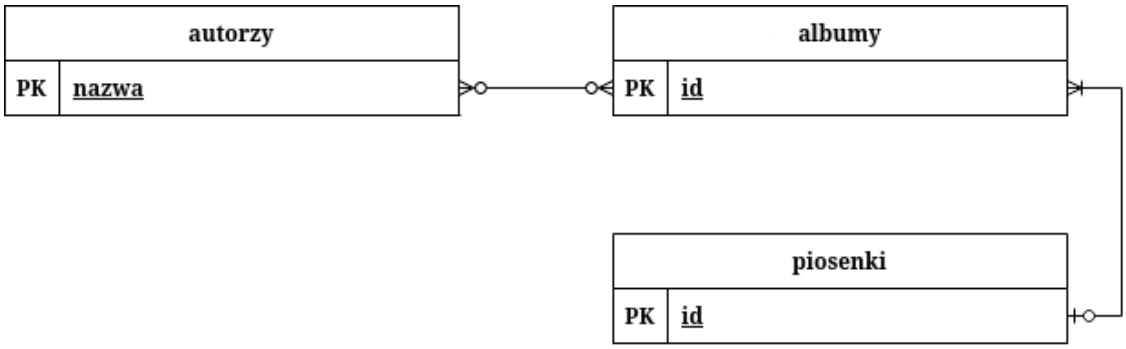
#### 3.5.2.2. Albumy

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

#### 3.5.2.3. 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

### 3.5.3. 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 niebezpośrednio łączą się z autorami. Jeżeli piosenka chce uzyskać swojego autora, musi zrobić to poprzez album.

# 4. Implementacja

## 4.1. Zarządzanie bazą danych

### 4.1.1. Klasa `DatabaseManager`

Za zarządzanie bazą danych odpowiedzialna jest klasa `DatabaseManager`, której kod jest zamieszony na listingu nr. 3. Klasa jest wrapperem do bazy danych `Room`[**doc˙room**] i do niej akcesorów.

```
@Singleton
class DatabaseManager @Inject constructor(
    @ApplicationContext context: Context
) {
    private val database: LibraryDb = Room.databaseBuilder(
        context,
        LibraryDb::class.java, "Library"
    ).build()

    fun collectAuthorsFlow(): Flow<List<Author>> = database.
    uiDao().getAllAuthorsFlow()

    fun collectAlbumsByAuthorFlow(authorName: String): Flow<
    List<Album>> {
        return database.uiDao().getAuthorWithAlbums(authorName)
            .map { it.albums }
    }

    fun collectSongsByAlbumFlow(albumId: Long): Flow<List<Song
    >> {
        return database.uiDao().getAlbumWithSongs(albumId)
            .map { it.songs }
    }

...

    fun populateDatabase(songs: List<TagExtractor.SongInfo>) {
        assert(Thread.currentThread().name != "main")

        val dao = database.logicDao()

        fun addAuthors() {
            songs.fastForEach { song ->
                //TODO: there should be a distinction between
    albumartists and regular artists
```

```
32                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: xddddddddd
43            val distinctAlbumArtistsList = songs
44                .map { Triple(it.album, it.albumArtists, it.
    coverUri) }
45                .distinct()
46            Log.d(javaClass.simpleName, "Distinct artists set:
    $distinctAlbumArtistsList")
47
48            distinctAlbumArtistsList.fastForEach {
49                val albumTitle = it.first.toString()
50                val artists = it.second
51                val coverUri = it.third
52
53                val albumId = dao.insertAlbum(Album(
54                    title = albumTitle,
55                    coverUri = coverUri.toString(),
56                ))
57
58                artists?.fastForEach {
59                    dao.insertAlbumAuthorCrossRef(
    AlbumAuthorCrossRef(
60                        albumId = albumId,
61                        name = it.toString()
62                    ))
63                }
64            }
65        }
66
67        fun addSongs() {
68            songs.fastForEach { song ->
69                Log.d(javaClass.simpleName, "NEW SONG\n")
70                Log.d(javaClass.simpleName, "Album artists: ${
    song.albumArtists}")
71
72                val albumWithAuthorCandidates = dao
```

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

**Listing 3.** Strukutura klasy `DatabaseManager`

Na pierwszej linijce można zauważyć adnotację `@Singleton`. Pochodzi ona z bilbioteki `Hilt`[**doc˙hilt**]. 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 4.1.2

Dalej, do linijki nr. 22 pokazane są metody zwracające rózne elementy bazy. Wiekszosc 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 4.1.3. 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, mozna je pominąć.

Metoda `populateDatabase()` zadeklarowana na linijce nr. 24, jest odpowiedzialna za ładowanie wyjętych z 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()`. Wywoływane są one po kolei w metodzie głównej.

Funkcja `addAuthors()`, zadeklarowana na linijce nr. 29, jest prosta w swoim działaniu. Lista z `SongInfo` jest iterowana i po kolei 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 sprawdzane 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 lini 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,
      AlbumAuthorCrossRef::class], version
2 = 1)
3 abstract class LibraryDb : RoomDatabase() {
4     abstract fun logicDao(): LogicDao
5     abstract fun uiDao(): UIDao
6 }
```

**Listing 4.** Deklaracja bazy LibraryDb

Jak widać na listingu nr. 4, 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. 4.1.4. 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. 4.1.3.

### 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 adnotacja `@Dao`. Są one implementowane przez `Room`. Baza danych w projekcie

wykorzystuje dwa interfejsy `dao - UIdao`, którego kod zamieszczony jest na listingu nr. 5 oraz `LogicDao`, którego kod zamieszczony jest na listingu nr. 6.

```
@Dao
interface UIDao {
    @Query("SELECT * FROM Song")
    fun getAllSongs(): Flow<List<Song>>

    @Query("SELECT * FROM Song WHERE songId = :songId")
    fun collectSongFromId(songId: Long): Flow<Song>

    // Get an album with its songs
    @Transaction
    @Query("SELECT * FROM album WHERE albumId = :albumId")
    fun getAlbumWithSongs(albumId: Long): Flow<AlbumWithSongs>

    @Query("SELECT * FROM album WHERE albumId = :albumId")
    fun getAlbumById(albumId: Long?): Flow<Album>

    // Get an album with its authors
    @Transaction
    @Query("SELECT * FROM album WHERE albumId = :albumId")
    fun getAlbumWithAuthors(albumId: Long?): Flow<
   AlbumWithAuthors?>

    @Query("SELECT * FROM Author")
    fun getAllAuthorsFlow(): Flow<List<Author>>

    // Get an author with their albums
    @Transaction
    @Query("SELECT * FROM author WHERE name = :name")
    fun getAuthorWithAlbums(name: String): Flow<
   AuthorWithAlbums>
}
```

**Listing 5.** Deklaracja interfejsu `UIdao`

```
@Dao
interface LogicDao {
    @Insert
    fun insertSong(song: Song)

    @Insert
    fun insertAlbum(album: Album): Long

    @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. `UIdao` natomiast, służy ekskluzywnie do zwracania `Flow`ów. Większość elementów związanych z interfejsem w reszcie kodu aplikacji już korzysta z `Flow`ów, 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. 6. Funkcja `insertSong()`, zadnoto-

wana 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 jakiejś 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. 7, określa autorów. Tabela jest prosta, jedynym polem jest `name`, który jest kluczem.

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

**Listing 8.** Deklaracja tabeli Album

Tabela `Album`, zawarta na listingu nr. 8, 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.

```

@Entity(
    foreignKeys = [
        ForeignKey(
            entity = Album::class,
            parentColumns = ["albumId"],
            childColumns = ["albumId"],
            onDelete = ForeignKey.CASCADE
        )
    ],

    indices = [Index(value = ["albumId"])]
)
data class Song(
    @PrimaryKey(autoGenerate = true) val songId: Long = 0,
    val title: String?,
    val albumId: Long?,
    val fileUri: String?,
)
```

**Listing 9.** Deklaracja tabeli Song

Klasa ta, zawarta na listingu nr. 9, 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 wychodziła, do tej adnotacji.

#### 4.1.5.1. AlbumWithSongs

Klasa `AlbumWithSongs` na listingu nr. 10, określa relację albumów i piosenek.

```
data class AlbumWithSongs(
    @Embedded val album: Album,
    @Relation(
        parentColumn = "albumId",
        entityColumn = "albumId"
    )
    val songs: List<Song>
)
```

**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.

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

**Listing 11.** Deklaracja tabeli relacji `AlbumAuthorCrossRef`

Tabela na listingu nr. 11 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.

```
data class AlbumWithAuthors(
    @Embedded val album: Album,
    @Relation(
        parentColumn = "albumId",
        entityColumn = "name",
        associateBy = Junction(AlbumAuthorCrossRef::class)
    )
    val authors: List<Author>
)
```

**Listing 12.** Deklaracja relacji `AlbumWithAuthors`

```
data class AuthorWithAlbums(
    @Embedded val author: Author,
    @Relation(
        parentColumn = "name",
        entityColumn = "albumId",
        associateBy = Junction(AlbumAuthorCrossRef::class)
    )
    val albums: List<Album>
)
```

**Listing 13.** Deklaracja relacji `AuthorWithAlbums`

Na listingu nr. 12 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. 4.1.5.2. 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. 13. Tym razem to autor jest rodzicem i oczekujemy od relacji listy albumów danego autora.

## 4.2. Czujnik światła

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

```
@AndroidEntryPoint
class MainActivity : FragmentActivity(), SensorEventListener {
  private lateinit var sensorManager: SensorManager
  private var lightSensor: Sensor? = null
  private val _isDarkTheme = mutableStateOf(false)
  private val isDarkTheme: State<Boolean> = _isDarkTheme

  private val _isAuthenticated = mutableStateOf(false)
  private val isAuthenticated: State<Boolean> =
 _isAuthenticated

  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    val biometricAuthenticator = BiometricAuthenticator(this)

    sensorManager = getSystemService(Context.SENSOR_SERVICE)
 as SensorManager
    lightSensor = sensorManager.getDefaultSensor(Sensor.
 TYPE_LIGHT)

    enableEdgeToEdge()

    setContent {
      val darkTheme by isDarkTheme
      val authenticated by isAuthenticated
      RaptorTheme(darkTheme = darkTheme) {
        Surface(
        modifier = Modifier.fillMaxSize(),
        color = MaterialTheme.colorScheme.background
        ) {
          if (authenticated) {
            MainScreen()
          } else {
            AuthenticationScreen(
            onAuthenticate = {
              promptBiometricAuthentication(
 biometricAuthenticator)
            }
            )
          }
        }
      }
    }
  }
```

```
41
42    private fun promptBiometricAuthentication(
      biometricAuthenticator: BiometricAuthenticator) {
43      biometricAuthenticator.PromptBiometricAuth(
44      title = "Authentication Required",
45      subtitle = "Please authenticate to proceed",
46      negativeButtonText = "Cancel",
47      fragmentActivity = this,
48      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.SENSOR_DELAY_NORMAL)
64      }
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 światłą w `MainActivity.kt`

Na listingu 14 przedstawiona jest funkcja MainScreen. W wierszu 2 mamy SensorEventListener, który jest frameworkiem 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 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 oświetlenią(jednosta 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.

## 4.3. Autoryzacja odciskiem palca

## 4.4. Sound wave

# 5. Testowanie

# 6. Podręcznik użytkownika

# Spis rysunków

# Spis tabel

# Spis listingów