Uploadity

Aplikacja mobilna służąca do publikacji postów w mediach społecznościowych

Autor:
Aleksandra Miloch

Spis treści

Rozdział 1

Użyte narzędzia

1.1 Android Studio

Android Studio to oficjalne zintegrowane środowisko programistyczne (ang. IDE - Integrated Development Environment) służące do tworzenia aplikacji na system Android. Jest oparte na popularnym narzędziu IntelliJ IDEA. Najważniejsze funkcje Android Studio to:

- szybki i bogaty w wiele funkcji emulator urządzeń,
- elastyczne narzędzie Gradle służące do m.in. automatyzacji procesu kompilacji kodu oraz pobierania zależności (dodatkowych bibliotek),
- ujednolicone środowisko umożliwiające tworzenie aplikacji na wszystkie urządzenia z systemem Android,
- integracja z systemem Git ułatwiające zarządzania wersjami kodu,
- inteligentna edycja kodu: autouzupełnianie, sprawdzanie poprawności oraz szybkie przeglądanie dokumentacji.

1.2 Kotlin

Kotlin[1] to nowoczesny, statycznie typowany język, który działa na maszynie wirtualnej Javy. Został stworzony przez firmę JetBrains i jest aktywnie rozwijany przez nią oraz społeczność, ponieważ działa na zasadzie open-source. Jest on w pełni interoperacyjny z Javą, co oznacza że oba języki mogą współpracować ze sobą w jednym projekcie oraz korzystać wzajemnie ze swoich bibliotek. W 2017 roku Google ogłosiło Kotlin oficjalnym językiem programowania aplikacji mobilnych na platformę Android powodując zwiększenie jego popularności.

1.3 XML

XML[2], czyli Extensible Markup Language, to uniwersalny język znaczników używany do reprezentacji strukturalnych danych w formie tekstowej. Ułatwia przechowywanie i przekazywanie informacji pomiędzy różnymi systemami. W Androidzie pełni funkcje związane z

[1] www.kotlinlang.org
[2] www.w3.org/XML

definiowaniem elementów interfejsu użytkownika (takich jak przyciski, pola tekstowe itp.) oraz konfiguracją aplikacji i jej zasobami graficznymi.

1.4 Retrofit

Retrofit[3] to popularna biblioteka służąca do tworzenia efektywnych klientów RESTful API. Jest rozwijana przez Square, a jej głównym celem jest ułatwienie komunikacji z serwerem HTTP poprzez tworzenie czystego i łatwego w użyciu interfejsu. Główne cechy biblioteki to:

- **Deklaratywny interfejs API**: Retrofit umożliwia definiowanie interfejsu API w sposób deklaratywny, co oznacza, że programista określa, jakie zapytania i jakie dane są wymagane bez konieczności bezpośredniej obsługi niskopoziomowych operacji HTTP.
- **Obsługa różnych formatów danych**: Retrofit automatycznie obsługuje różne formaty danych takie jak JSON czy XML. Dostępna jest również możliwość skonfigurowania własnych konwerterów danych.
- **Obsługa asynchroniczności**: Obsługuje operacje asynchroniczne, co jest istotne w kontekście komunikacji z serwerem. W przypadku Androida, Retrofit zwykle wykorzystuje `Coroutines` języka Kotlin lub interfejsy zwrotne (ang. callbacks).
- **Łatwa konfiguracja**: Retrofit jest łatwy w konfiguracji i zapewnia elastyczność w dostosowywaniu ustawień, takich jak timeouty, konwertery, obsługa błędów i tym podobne.
- **Obsługa elementów zapytań HTTP**: Pozwala na łatwą implementację nagłówków, adresów URL, parametrów, rodzajów zapytań, ciała zapytań itp., co jest niezbędne podczas komunikacji z serwerem.

[3] www.square.github.io/retrofit/

Rozdział 2

Koncepcja aplikacji i implementacja

Aplikacja Uploadity służy do publikacji postów w postaci zdjęcia i tekstu na platformach X, Tumblr oraz Linkedin. Użytkownik ma możliwość połączenia własnych kont z wymienionych platform przez zalogowanie się z poziomu aplikacji i przekazanie dostępu do publikacji treści na swoim profilu. Poniżej przedstawione są wymagania funkcjonalne i niefunkcjonalne aplikacji.

2.1 Wymagania funkcjonalne

- Użytkownik może dodać konto społecznościowe do aplikacji przez kliknięcie w przycisk, który przenosi do strony internetowej konkretnej platformy, gdzie należy się zalogować na konto i przekazać dostęp do publikacji treści na profilu.
- Użytkownik ma możliwość wyświetlania listy połączonych mediów społecznościowych oraz usuwania ich w przypadku, gdy chce unieważnić dostęp aplikacji do konkretnego konta.
- W celu stworzenia posta, należy wejść w ekran edycji publikacji, gdzie użytkownik ma możliwość nadania tytułu, opisu oraz wybrania zdjęcia. Po uzupełnieniu tych pól istnieją dwie opcje - dokonanie wyboru, na które konta ma zostać wstawiony post lub zapisanie zmian jako wersję roboczą. W każdej chwili użytkownik może porzucić zmiany przez naciśnięciu przycisku „Anuluj".
- Użytkownik ma możliwość przeglądania listy wersji roboczych oraz opublikowanych postów. Po wejściu w szczegóły istnieje opcja usunięcia wybranego elementu.

2.2 Wymaganie niefunkcjonalne

- Przejrzysty i intuicyjny interfejs użytkownika.
- Urządzenie mobilne powinno posiadać system Android w wersji 10 lub wyższej.
- Urządzenie mobilne powinno posiadać dostęp do internetu.
- Użytkownik posiada konto na przynajmniej jednym z mediów społecznościowych takich jak X, Tumblr lub Linkedin.

2.3 Zarządzanie danymi

System Android udostępnia deweloperom różne sposoby przechowywania danych aplikacji. Jest to kluczowy element, który pozwala na zachowywanie danych użytkownika na urządzeniu i unikanie ich utraty w przypadku zamknięcia aplikacji. W Uploadity wszystkie dane przechowywane są lokalnie, dlatego tworzenie serwera aplikacji nie jest potrzebne. Ważną kwestią jest również dbanie o odpowiednie zarządzanie pamięcią oraz korzystanie z odpowiednich metod przechowywania danych w zależności od ich rodzaju oraz poufności. Oto opcje przechowywania danych wykorzystane w aplikacji Uploadity:

2.3.1 Baza danych

Struktura bazy danych

Tabela **Accounts** przechowuje dane na temat kont użytkownika na poszczególnych platformach społecznościowych.

Atrybut	Typ	Opis
id	integer	klucz główny, identyfikator konta w bazie danych
accountId	integer	identyfikator profilu na platformie społecznościowej
name	string	nazwa profilu użytkownika
socialMediaServiceName	string	nazwa platformy społecznościowej

Tabela **Blogs** przechowuje dane na temat blogów na platformie Tumblr.

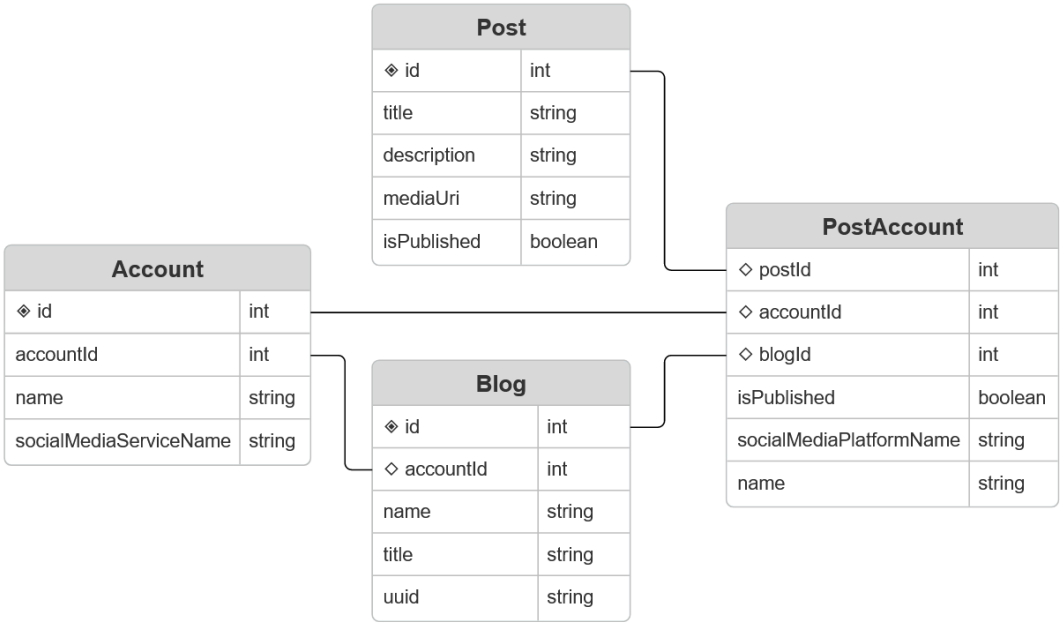
Atrybut	Typ	Opis
id	integer	klucz główny, identyfikator bloga w bazie danych
accountId	integer	klucz obcy odnoszący się do pola id w tabeli Accounts
name	string	nazwa bloga
title	string	tytuł bloga
uuid	string	identyfikator bloga w serwisie Tumblr

Tabela **Posts** przechowuje dane na temat postów i wersji roboczych utworzonych w aplikacji.

Atrybut	Typ	Opis
id	integer	klucz główny, identyfikator posta w bazie danych
title	string	tytuł posta
description	string	opis posta
mediaUri	string	ścieżka pliku obrazu dodanego do posta
isPublished	boolean	wartość wskazująca czy post został opublikowany, czy jest wersją roboczą

Tabela **PostAccounts** przechowuje dane na temat postów opublikowanych na każdej z wybranych platform. Klucz główny tworzą trzy atrybuty, które są kluczami obcymi z tabel Accounts, Posts oraz Blogs.

Atrybut	Typ	Opis
postId	integer	klucz obcy odnoszący się do pola id w tabeli Post
accountId	integer	klucz obcy odnoszący się do pola id w tabeli Accounts
blogId	integer	klucz obcy odnoszący się do pola id w tabeli Blogs
isPublished	boolean	wartość wskazująca czy post został opublikowany na konkretnej platformie
socialMediaPlatformName	string	nazwa platformy społecznościowej
name	string	nazwa konta lub bloga



Rysunek 2.1: Diagram bazy danych

Biblioteka Room

W celu umożliwienia płynnego dostępu do lokalnej bazy danych Android stworzył bibliotekę Room[1], u podstawy której znajduje się popularny system zarządzania relacyjną bazą danych SQLite. Room dostarcza wygodny i efektywny interfejs do operacji bazodanowych. Oto korzyści wynikające z używania tej biblioteki:

- weryfikacja zapytań SQL w czasie kompilacji,

[1] www.developer.android.com/jetpack/androidx/releases/room

- adnotacje, które minimalizują prawdopodobieństwo powtarzalnego i podatnego na błędy kodu,
- możliwość wygodnej integracji bazy z innymi elementami architektury Androida.

Do podstawowych elementów biblioteki Room należą:

- klasa bazy danych, która jest głównym punktem dostępu do połączenia z danymi aplikacji,
- encje, które reprezentują tabele w bazie danych,
- obiekty dostępu do danych DAO (Data Access Objects), które dostarczają funkcje tworzenia zapytań bazodanowych SQL.

Aby wykorzystać bibliotekę Room w aplikacji, należy dodać następujące zależności do pliku `build.gradle`:

```
implementation 'androidx.room:room-runtime:2.6.0'
annotationProcessor 'androidx.room:room-compiler:2.6.0'
```

Następnie należy zdefiniować encje, które będą reprezentować strukturę w bazie danych. Encje to klasy, które odpowiadają tabelom w bazie danych. Przykładowo implementacja encji w tabeli Accounts wygląda następująco:

```
@Entity(tableName = "accounts")
data class Account(
    @PrimaryKey(autoGenerate = true) val id: Int,
    val accountId: String,
    val name: String,
    val socialMediaServiceName: String
)
```

Kolejnym krokiem jest implementacja DAO (Data Access Object), który definiuje operacje, które można wykonywać na bazie danych. Zawiera metody do wstawiania, pobierania, aktualizowania i usuwania danych.

```
@Dao
interface AccountDao {
    @Query("SELECT * FROM accounts")
    fun getAllAccounts(): List<Account>

    @Query("SELECT * FROM accounts WHERE id = :id")
    fun getAccount(id: Int): Account?

    @Query("SELECT * FROM accounts WHERE socialMediaServiceName =
            :socialMediaName")
    fun getAccountBySocialMediaName(socialMediaName: String): Account?

    @Insert
```

```
    fun insert(account: Account): Long

    @Update
    fun update(account: Account)

    @Delete
    fun delete(account: Account)
}
```

Bazę danych reprezentuje klasa, która rozszerza `RoomDatabase` i zawiera metody abstrakcyjne zwracające DAO:

```
@Database(entities = [Post::class, Account::class, Blog::class,
            PostAccount::class])
abstract class AppDatabase : RoomDatabase()  {
    companion object {
        private var instance: AppDatabase? = null

        fun getInstance(context: Context): AppDatabase {
            if (instance == null) {
                instance = Room.databaseBuilder(context,
                AppDatabase::class.java,"the_database.db")
                    .allowMainThreadQueries()
                    .fallbackToDestructiveMigration()
                    .build()
            }

            return instance as AppDatabase
        }
    }

    abstract fun postDao(): PostDao
    abstract fun accountDao(): AccountDao
    abstract fun blogDao(): BlogDao
    abstract fun postAccountDao(): PostAccountDao
}
```

W klasie `AppDatabase` przez funkcję `getInstance()` zaimplementowany został wzorzec projektowy singleton, który zapewnia, że w aplikacji zawsze będzie istniała tylko jedna instancja bazy danych, co zapobiega wyciekom pamięci.
Teraz można użyć DAO do wykonywania operacji na bazie danych w całej aplikacji. Poniższy kod przedstawia przykład pobrania listy wszystkich kont użytkownika:

```
val appDao = AppDatabase.getInstance(getApplicationContext())
val accounts = appDao.accountDao().getAllAccounts()
```

2.3.2 User DataStore

To biblioteka służąca do przechowywania prostych danych (np. ustawienia użytkownika czy stan sesji) klucz-wartość w formie mapy. Pozwala na asynchroniczne, spójne i transakcyjne zarządzanie danymi. Działa na zasadzie przesyłania strumieniowego, dzięki czemu nie trzeba wczytywać wszystkich danych do pamięci jednocześnie, co może być korzystne w przypadku dużych zbiorów danych. W aplikacji rozwiązanie to zostało użyte do przechowywanie tokenów profili mediów społecznościowych, które zostały połączone przez użytkownika.

Aby wykorzystać bibliotekę DataStore w aplikacji, należy dodać następujące zależności do pliku `build.gradle`:

```
implementation 'androidx.datastore:datastore-preferences:1.0.0'
```

W klasie `UserDataStore` zaimplementowane zostało tworzenie instancji `DataStore<Preferences>` oraz funkcje `saveStringPreference()` i `getStringPreference()`, które służą do zapisywania i odczytywania danych typu string.

```
class UserDataStore(private val context: Context?) {
    companion object {
        private val Context.dataStore: DataStore<Preferences>
            by preferencesDataStore("datastore_name")
    }

    suspend fun getStringPreference(key: String): String {
        val stringPreferenceKey = stringPreferencesKey(key)
        return context!!.dataStore.data.map { preferences ->
            preferences[stringPreferenceKey] ?: ""
        }.first()
    }

    suspend fun saveStringPreference(key: String, value: String) {
        val stringPreferenceKey = stringPreferencesKey(key)
        context!!.dataStore.edit { preferences ->
            preferences[stringPreferenceKey] = value
        }
    }
}
```

2.3.3 Pamięć współdzielona

Jest to przestrzeń, w której dane mogą być współdzielone między różnymi aplikacjami i użytkownikami. W aplikacji Uploadity konieczny jest dostęp do mediów, aby móc załadować obraz do posta. Do tej funkcji wykorzystany został `MediaStore`, czyli dostępny w Androidzie system, który śledzi i zarządza różnymi rodzajami mediów, takimi jak obrazy, filmy, dźwięki itp. Udostępnia interfejs do dostępu do plików multimedialnych, a także umożliwia dostęp do nich przez różne aplikacje. Aby uzyskać dostęp do mediów w

Androidzie, należy dodać odpowiednie uprawnienie do odczytu mediów w pliku AndroidManifest.xml:

```
<uses-permission android:name="android.permission.READ_MEDIA_IMAGES" />
```

Następnie należy zarejestrować uruchomienie wybierania pojedynczego obrazu z galerii:

```
val pickMedia = registerForActivityResult(
                ActivityResultContracts.PickVisualMedia()) { uri ->
    if (uri != null) {
        chooseMedia(uri)
    }
}
```

Po wczytaniu obrazu, można nim zarządzać przez jego `URI` (ang. Uniform Resource Identifier), czyli identyfikator do obsługiwania zasobów w Androidzie.

2.3.4 Pamięć wewnętrzna aplikacji

Każda aplikacja Androida posiada dedykowaną lokalizację do przechowywania plików. Dane zapisane wewnętrznie są dostępne tylko dla danej aplikacji i nie są widoczne w innych miejscach. W momencie usunięcia aplikacji wszystkie pliki zostają usunięte. W Uploadity rozwiązanie to zostało wykorzystane do przechowywania obrazów, które zostały dodane do postów oraz wersji roboczych. Plik należy utworzyć w następujący sposób:

```
val file = File(context.filesDir, "post_\${post.id}.png")
```

Plik powstaje w lokalizacji zdefiniowanej przez `filesDir`. Nazwa każdego zapisanego obrazu zawiera id posta, aby zapobiec nadpisywaniu przez siebie plików. Następnie wczytywane jest zdjęcie na podstawie jego `URI` przez użycie `ImageDecoder`. Bitmapa zostaje zapisana do pliku, a strumień zapisu zamknięty.

```
val source = ImageDecoder.createSource(contentResolver, mediaUri)
val bitmap = ImageDecoder.decodeBitmap(source)
val fileOutputStream = FileOutputStream(file)
bitmap.compress(Bitmap.CompressFormat.PNG, 100, fileOutputStream)
fileOutputStream.flush()
fileOutputStream.close()
```

2.4 Komponenty aplikacji

2.4.1 AndroidManifest.xml

Każdy projekt Androida posiada plik `AndroidManifest.xml`. Zawiera on informacje na temat aplikacji, jej komponentów oraz wymagań systemowych i sprzętowych. Dodatkowo jest niezbędny do rozpoznania i skonfigurowania aplikacji podczas jej instalacji i uruchamiania. Główne elementy pliku `AndroidManifest.xml` to:

- **Informacje o aplikacji:** Zdefiniowanie nazwy projektu i aplikacji, jej ikony, motywu i innych w znaczniku `<application>`:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android">
    <application
        android:name=".UploadityApplication"
        android:label="@string/app_name"
        android:icon="@mipmap/ic_launcher"
        android:theme="@style/Theme.Uploadity">
        <!-- Składniki aplikacji -->
    </application>
</manifest>
```

- **Aktywności:** Elementy reprezentujące ekrany z interfejsem użytkownika, takie jak ekran listy kont, ekran tworzenia nowego posta itp. Oto przykładowe elementy `Activity` w pliku `AndroidManifest.xml`:

```
<activity android:name=".AccountActivity"
    android:exported="false" />
<activity android:name=".MainActivity"
    android:exported="true">
    <intent-filter>
        <action android:name="android.intent.action.MAIN"/>
        <category android:name="android.intent.category.LAUNCHER"/>
    </intent-filter>
</activity>
```

  Warto zauważyć, że `MainActivity` jest główną aktywnością, co oznacza, że po otwarciu aplikacji pokazuje się użytkownikowi jako pierwsza. Dzieje się to dzięki dodaniu odpowiedniej akcji i kategorii wewnątrz znacznika `<intent-filter>`.

- **Uprawnienia:** Określają, jakie uprawnienia do zasobów systemowych są wymagane przez aplikację. W przypadku Uploadity jest to dostęp do internetu i wczytywania obrazów z galerii:

```
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission
    android:name="android.permission.READ_MEDIA_IMAGES"/>
```

2.4.2 Gradle

Gradle jest systemem budowania wykorzystywanym w aplikacjach Androida. Konfiguracja projektu znajduje się w plikach `build.gradle`, które głównie zawierają informacje na temat kompilacji, zarządzania zależnościami i zasobami. Oto skrócony przegląd zawartości pliku `build.gradle` w projekcie Uploadity:

```
plugins {
    id 'com.android.application'
    ...
}
android {
    applicationId 'com.uploadity'
    compileSdk 34
    versionCode 1
    versionName '1.0'
    ...
}
dependencies {
    implementation 'androidx.core:core-ktx:1.12.0'
    implementation 'com.squareup.retrofit2:retrofit:2.9.0'
    implementation 'androidx.datastore:datastore-preferences:1.0.0'
    ...
}
```

W sekcji `plugins` znajduje się wtyczka Gradle przeznaczona dla Androida. W bloku `android` skonfigurowane są wszystkie opcje budowania aplikacji, takie jak jej wersja, przestrzeń nazw (namespace) czy wersja Androida, na którą domyślnie jest kompilowana. Blok `dependencies` definiuje wszystkie zależności (czyli dodatkowe biblioteki) dodane do projektu.

2.4.3 Activity

W Androidzie `Activity` to jedna z podstawowych jednostek aplikacji. Jest to komponent, który reprezentuje pojedynczy ekran aplikacji. Każda aktywność jest zazwyczaj odpowiedzialna za przedstawienie UI (interfejsu użytkownika) i interakcję z użytkownikiem w ramach jednej określonej funkcji aplikacji. W przeciwieństwie do paradygmatów programowania, w których aplikacje są uruchamiane za pomocą metody `main()`, system Android inicjuje kod w instancji `Activity`, wywołując go w metodach odpowiadających konkretnym etapom jej cyklu życia. Oto kilka kluczowych cech `Activity` w Androidzie:

- **Interfejs użytkownika**: Aktywność jest często związana z interfejsem użytkownika i może zawierać różne elementy wizualne takie jak przyciski, pola tekstowe, listy, itp. Widok ekranu jest definiowany w pliku XML.
- **Intencje** (`Intents`): Aktywności komunikują się ze sobą i innymi komponentami systemu za pomocą intencji. To zazwyczaj żądanie do systemu Androida, aby wykonał określoną akcję taką jak uruchomienie nowej aktywności, usługi i tym podobne.
- **Obsługa zdarzeń**: Aktywność może reagować na różne zdarzenia generowane przez użytkownika, takie jak dotknięcie ekranu. Odbywa się ona poprzez implementację różnych metod, na przykład `onClick()` w przypadku kliknięcia przycisku.

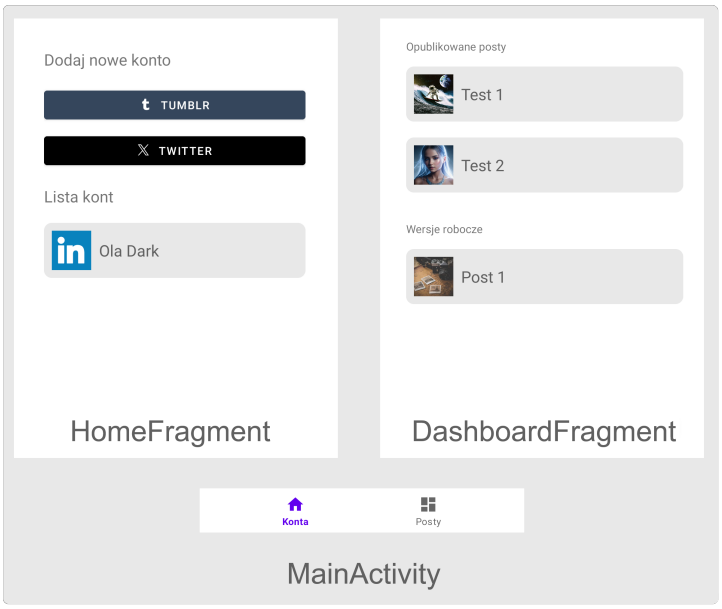
Poniższy kod przedstawia podstawowe elementy `MainActivity`, czyli głównej aktywności w aplikacji:

```
class MainActivity : AppCompatActivity() {
    private lateinit var binding: ActivityMainBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)

        //Inicjalizacja interfejsu użytkownika i obsługa zdarzeń
    }
    // Dodatkowe metody i logika aktywności
}
```

Każda klasa aktywności dziedziczy po klasie `Activity` lub jednym z jej wariantów, w powyższym przykładzie jest to `AppCompatActivity`. Nadpisując metodę `onCreate()` możemy dodać akcje, które zostaną wykonane w trakcie stworzenia aktywności. W tym przypadku jest to zainicjowanie zmiennej `binding`, która odnosi się do interfejsu użytkownika zapisanego w pliku XML.

2.4.4 Fragment

`Fragment` to komponent reprezentujący część interfejsu użytkownika lub zachowanie w jednym oknie aplikacji, który jest umieszczony w aktywności (`Activity`). Fragmenty są często używane w celu zorganizowania interfejsu użytkownika na większych ekranach (takich jak tablety) lub do ułatwienia ponownego użycia i modularyzacji kodu. Co więcej, mogą być dodawane, usuwane lub zastępowane dynamicznie podczas działania aplikacji. W aplikacji Uploadity fragmenty zostały użyte w zakładkach listy kont i postów, które są umieszczone w `MainActivity`. W tej aktywności obsługiwany jest również dolny pasek nawigacji umożliwiający przełączanie pomiędzy zakładkami.



Rysunek 2.2: Umieszczenie fragmentów listy kont oraz postów w MainActivity

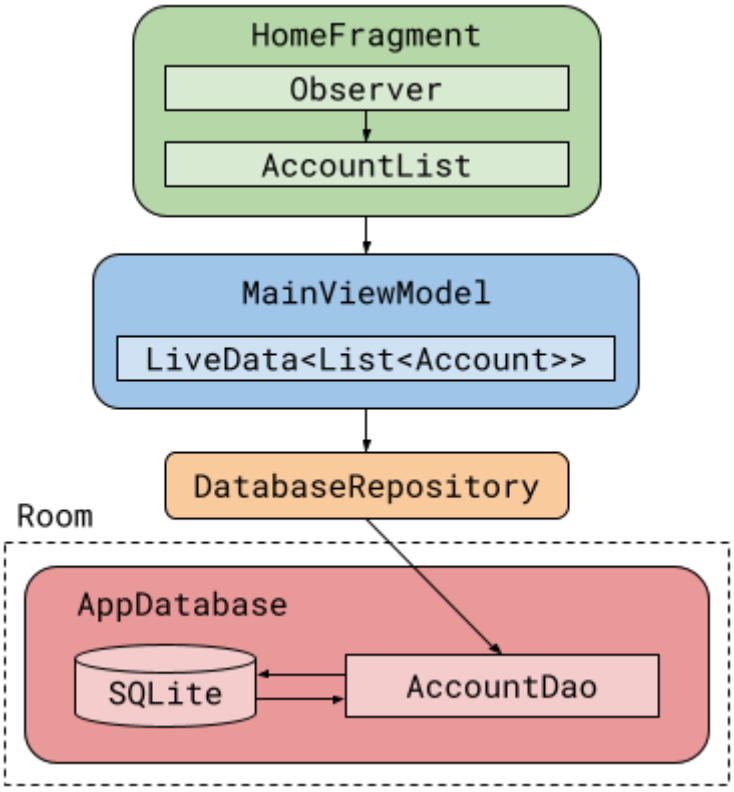
Poniższy kod przedstawia podstawowe elementy `DashboardFragment`, gdzie znajduje się lista kont:

```
class HomeFragment : Fragment() {
    private var binding: FragmentHomeBinding

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        binding = FragmentHomeBinding.inflate(inflater, container, false)

        //Inicjalizacja interfejsu użytkownika i obsługa zdarzeń

        return binding.root
    }
    // Dodatkowe metody i logika fragmentu
}
```

`OnCreateView()` to jedna z metod cyklu życia fragmentu w Androidzie. Jest wywoływana, gdy fragment inicjuje swoje interfejsy użytkownika przez załadowanie widoku z pliku XML.

2.4.5 ViewModel

ViewModel to część architektury, która pomaga w separacji logiki biznesowej od interfejsu użytkownika oraz w przetrzymywaniu danych w sposób trwały podczas zmian konfiguracji (takich jak obrót urządzenia). W aplikacji Uploadity został wykorzystany w celu przejrzystego i asynchronicznego dostępu do danych. Poniższy diagram przedstawia zarys implementacji tej architektury na przykładzie ekranu listy kont umieszczonego w `HomeFragment` i zarządzanego przez `MainViewModel`:



Rysunek 2.3: Diagram implementacji ViewModel

Oto opis poszczególnych kroków wymaganych do implementacji tej architektury:

1. Na początku należy zmodyfikować `AccountDao` i zmienić typ danych zwracanych przez funkcję pobierającą wszystkie konta użytkownika na `Flow`. Jest to klasa umożliwiająca przesyłanie danych w sposób asynchroniczny. Zaktualizowana metoda wygląda następująco:

```
@Query("SELECT * FROM accounts")
fun getAllAccountsFlow(): Flow<List<Account>>
```

2. Następnie należy stworzyć klasę repozytorium, która zajmuje się operacjami na danych i zapewnia przejrzysty interfejs API dla pozostałej części aplikacji. Oto przegląd najważniejszych części implementacji tej klasy:

```
class AppDatabaseRepository(
    private val accountDao: AccountDao
) {
    val allAccounts: Flow<List<Account>> =
            accountDao.getAllAccountsFlow()
    ...
}
```

3. Następnym krokiem jest implementacja klasy `MainViewModel`. Będzie ona zapewniała listę kont użytkownika i przesyłała ją do widoku `HomeFragment` w asynchroniczny sposób. Aby tego dokonać, należy wykorzystać `LiveData`, czyli obiekt prze-

chowujący dane obserwowane przez interfejs użytkownika. Pozwoli to na automatyczne aktualizowanie widoku w odpowiedzi na zmiany danych w efektywny sposób. Na początku należy dodać do pliku `build.gradle` następujące zależności:

```
implementation 'androidx.lifecycle:lifecycle-livedata-ktx:2.6.2'
implementation 'androidx.lifecycle:lifecycle-viewmodel-ktx:2.6.2'
```

Poniższy kod przedstawia, jak została zaimplementowana klasa `MainViewModel`:

```
class MainViewModel(
    private val appDatabase: AppDatabaseRepository
): ViewModel() {
    private val allAccounts: LiveData<List<Account>> =
        appDatabase.allAccounts.asLiveData()

    fun getAllAccounts(): LiveData<List<Account>> {
        return allAccountsLiveData
    }
    ...
}
```

4. Teraz w klasie `HomeFragment` możemy stworzyć obserwatora za pomocą metody `observe()`, który będzie automatycznie aktualizował listę połączonych kont użytkownika:

```
mainViewModel.getAllAccounts().observe(viewLifecycleOwner)
{ accounts ->
    accounts.let { accountItemListAdapter.submitList(it) }
}
```

Zmienna `accountItemListAdapter` jest częścią interfejsu użytkownika, która wyświetla listę kont na ekranie. Obserwator z każdą aktualizacją danych przekazuje je za pomocą funkcji `submitList()`.

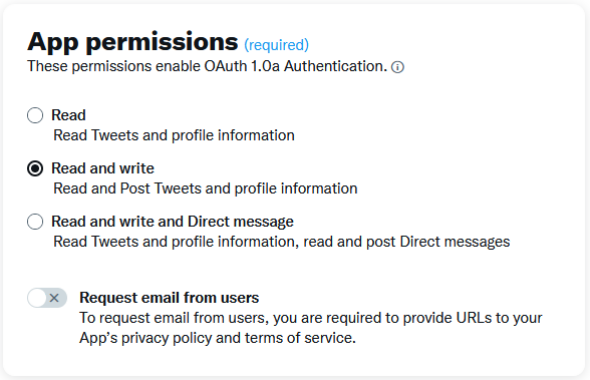
2.5 Łączenie z API mediów społecznościowych

Aby wykonywać działania w imieniu profilu użytkownika, należy uzyskać jego token dostępu (ang. access token). W aplikacji Uploadity proces uwierzytelniania rozpoczyna się po wciśnięciu jednego z przycisków na ekranie listy kont. Łączenie się z API mediów społecznościowych obejmuje kilka kluczowych kroków, które zostaną opisane w tym rozdziale.

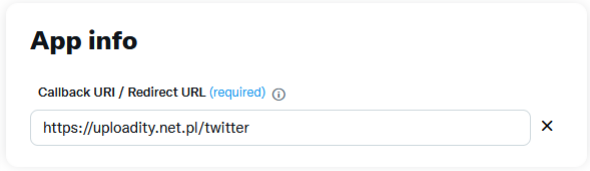
2.5.1 Pobieranie klucza API

W celu uzyskania dostępu do API mediów społecznościowych, programista musi uzyskać odpowiednie klucze API i tokeny dostępu w panelu dewelopera. Dodatkowo, aby móc udostępniać posty w imieniu użytkownika, należy ustawić wymagania umożliwiające tę akcję za pomocą aplikacji. Oprócz tego wymagane jest podanie strony callback URL,

czyli strony internetowej, na którą użytkownik zostanie przeniesiony podczas procesu autoryzacji. W przypadku platformy X wygląda to następująco:



Rysunek 2.4: Ustawienia uprawnień aplikacji w panelu dewelopera. Źródło: opracowanie własne.



Rysunek 2.5: Podanie adresu callback URL w panelu dewelopera. Źródło: opracowanie własne.

Należy unikać przechowywania kluczy API bezpośrednio w kodzie źródłowym, ponieważ mogą one zostać wyodrębnione podczas dekompilacji, dlatego w aplikacji Uploadity są przechowywane w następujący sposób:

1. Na początku stworzony został plik `local.properties`, gdzie umieszczone są klucze API:

```
TWITTER_CLIENT_ID="XXXXXXXXXXXXXX"
TWITTER_CLIENT_SECRET="XXXXXXXXXXXXXX"
```

2. Następnie w pliku `build.gradle` klucze zostały zdefiniowane jako właściwości:

```
android {
    defaultConfig {
        Properties properties = new Properties()
        properties.load(project.rootProject.file("local.properties")
            .newDataInputStream())
        buildConfigField "String", "TWITTER_CLIENT_ID",
            properties.getProperty("TWITTER_CLIENT_ID")
        buildConfigField "String", "TWITTER_CLIENT_SECRET",
            properties.getProperty("TWITTER_CLIENT_SECRET")
    }
}
```

3. Teraz klucze API są dostępne w klasie `BuildConfig` w kodzie źródłowym. Oto przykład użycia:

```
val linkedinClientId = BuildConfig.TWITTER_CLIENT_ID
```

2.5.2 Logowanie

Logowanie do API mediów społecznościowych opiera się na otwartym standardzie uwierzytelniania internetowego OAuth, który umożliwia bezpieczny dostęp do zasobów użytkownika bez konieczności ujawniania jego hasła. Oto jego dwie wersje użyte w aplikacji Uploadity:

- **OAuth 1.0** polega na tym, że każde żądanie wymagające dostępu do zasobów użytkownika jest podpisywane przy użyciu klucza konsumenta (ang. consumer key) i klucza dostępu (ang. access token). To zabezpiecza przed modyfikacją żądania podczas transmisji. W aplikacji Uploadity protokół został zaimplementowany w celu połączenia się z API platformy X.
- **OAuth 2.0** został zbudowany w sposób bardziej elastyczny i prosty niż poprzednia wersja. Jego autoryzacja jest oparta na tokenach dostępu. Użytkownik autoryzuje aplikację, a w zamian otrzymuje token, który pozwala na wykonywanie działań w jego imieniu. OAuth 2.0 został zaimplementowany w celu połączenia się z API platform Linkedin i Tumblr.

Autoryzacja użytkownika za pomocą OAuth 1.0

Oto kroki procesu uwierzytelnienia konta użytkownika platformy X:

1. Uzyskanie tymczasowego tokenu żądania (ang. request token) za pomocą zapytania HTTP `POST oauth/request_token` do API platformy społecznościowej. Oprócz wymaganych parametrów wymaganych w nagłówku autoryzacyjnym protokołu OAuth 1.0, należy dodatkowo umieścić tam wartość `oauth_callback`, czyli adres URL, do którego zostanie przekierowany użytkownik po udzieleniu aplikacji dostępu do wykonywania zapytań w jego imieniu, a także parametr `oauth_consumer_key`, czyli klucz API aplikacji. Poniższy kod przedstawia funkcję `requestToken()` w interfejsie `TwitterApiInterface` implementującą to zapytanie za pomocą biblioteki Retrofit:

```
@POST("oauth/request_token")
fun requestToken(
    @Header("Authorization") authorizationHeader: String,
    @Query("oauth_callback") callbackUrl: String
): Call<ResponseBody>
```

   Jeśli powyższe żądanie powiodło się, powinniśmy otrzymać następującą odpowiedź:

```
oauth_token=XXX&oauth_token_secret=XXX&oauth_callback_confirmed=true
```

Parametr `oauth_token` i `oauth_token_secret` to tymczasowe tokeny dostępu, a `oauth_callback_confirmed` to dodatkowe potwierdzenie, że zapytanie przebiegło pomyślnie.

2. Po otrzymaniu tokenów z poprzedniego żądania możemy wykonać zapytanie `GET oauth/authorize`, w którym przekazujemy parametr `oauth_token` uzyskany w poprzednim kroku. Oto przykładowy adres URL, na który przenosimy użytkownika:

```
https://api.twitter.com/oauth/authorize?oauth_token=XXX
```

Poniższy kod z klasy `MainActivity` przedstawia sposób, w jaki ta akcja jest wykonywana:

```
val intent = Intent(Intent.ACTION_VIEW,
Uri.parse("https://api.twitter.com/oauth/authorize?oauth_token=XXX")
startActivity(intent)
```

Po wykonaniu tego żądania użytkownik zostaje przekierowany do przeglądarki na stronę platformy, gdzie zostaje poproszony o udzielenie dostępu aplikacji do udostępniania postów w jego imieniu:

Rysunek 2.6: Strona udzielenia uprawnień użytkownika przez aplikację. Źródło: opracowanie własne.

Po zaakceptowaniu prośby platforma X przekierowuje użytkownika na podaną wcześniej stronę jako `oauth_callback` z parametrami umieszczonymi w tym adresie URL. W aplikacji Uploadity calback URL dla platformy X to `https://uploadity.net.pl/twitter`. Przykładowo w przypadku, gdy użytkownik autoryzował aplikację, zostanie przeniesiony do przeglądarki na następującą stronę:

```
https://uploadity.net.pl/twitter?oauth_token=XXXXXXX
&oauth_verifier=XXXXXXX
```

Teraz należy przekierować użytkownika z powrotem do aplikacji. Aby umożliwić tę czynność, na początku należy umieścić w `AndroidManifest.xml` kod, który zezwoli na otwieranie aplikacji z podanego adresu:

```
<activity
    android:name=".MainActivity">
    <intent-filter android:autoVerify="true">
        <action android:name="android.intent.action.VIEW"/>
        <category android:name="android.intent.category.DEFAULT"/>
        <category android:name="android.intent.category.BROWSABLE"/>
        <data android:scheme="https"/>
        <data android:host="uploadity.net.pl"/>
        <data android:pathPrefix="/twitter"/>
    </intent-filter>
</activity>
```

Atrybut `android:autoVerify="true"` sygnalizuje systemowi, że podany adres URL może otwierać automatycznie aplikację. Dodatkowo należy zadeklarować połączenie pomiędzy stroną a `<intent-filter>` przez plik JSON umieszczony w lokalizacji `https://uploadity.net.pl/.well-known/assetlinks.json`, który zostaje automatycznie wygenerowany w Android Studio po podaniu odpowiednich informacji. Zawartość pliku wygląda następująco:

```
[{
  "relation": ["delegate_permission/common.handle_all_urls"],
  "target": {
    "namespace": "android_app",
    "package_name": "com.uploadity",
    "sha256_cert_fingerprints":
    ["1E:42:89:08:E1:AB:43:56:17:54:01:DA:64:52:77:28:AD:3C:60:9F:F7
    :64:67:54:4E:B9:26:DD:DE:D2:48:D4"]
  }
}]
```

Istotne elementy w tym pliku to wartość `package_name`, który definiuje ID aplikacji zadeklarowane w pliku `build.gradle` oraz `sha256_cert_fingerprints`, czyli klucz podpisywania aplikacji.
Po automatycznym przekierowaniu użytkownika do aplikacji, ląduje on na ekranie `MainActivity` dzięki wcześniejszej deklaracji tej aktywności w pliku `AndroidManifest.xml`. Teraz możemy odczytać adres strony URL, z którego zostaliśmy przeniesieni i przetworzyć parametry przekazane w tym adresie umieszczając następujący kod w klasie `MainActivity`:

```
val appLinkIntent: Intent = intent
val appLinkData: Uri? = appLinkIntent.data
if (appLinkData != null) {
    when (appLinkData.lastPathSegment) {
        "twitter" -> {
            val oauthToken = appLinkData
                .getQueryParameter("oauth_token")
            val oauthVerifier = appLinkData
                .getQueryParameter("oauth_verifier")
            val oauthCallbackConfirmed = appLinkData
                .getQueryParameter("oauth_callback_confirmed")
            if (oauthCallbackConfirmed == true) {
                getTwitterAccessToken(oauthToken, oauthVerifier)
            }
        }
        ...
    }
}
```

3. Ostatnim etapem autoryzacji jest wykonanie zapytania `POST oauth/access_token` z parametrami `oauth_token` i `oauth_verifier` otrzymanymi w poprzednim kroku. Poniższy kod przedstawia funkcję `accessToken()` w interfejsie `TwitterApiInterface` implementującą to zapytanie za pomocą biblioteki Retrofit:

```
@POST("oauth/access_token")
fun accessToken(
    @Query("oauth_token") oauthToken: String,
    @Query("oauth_verifier") oauthVerifier: String
): Call<ResponseBody>
```

4. Teraz możemy zapisać otrzymane w odpowiedzi tokeny dostępu w DataStore aplikacji w następujący sposób:

```
userDataStore.saveStringPreference(
    userDataStore.twitterAccessTokenKey, userOauthToken)
userDataStore.saveStringPreference(
    userDataStore.twitterAccessTokenSecretKey, userOauthTokenSecret)
```

Autoryzacja użytkownika za pomocą OAuth 2.0

Oto kroki procesu uwierzytelnienia konta użytkownika w protokole OAuth 2.0 na przykładzie platformy Linkedin:

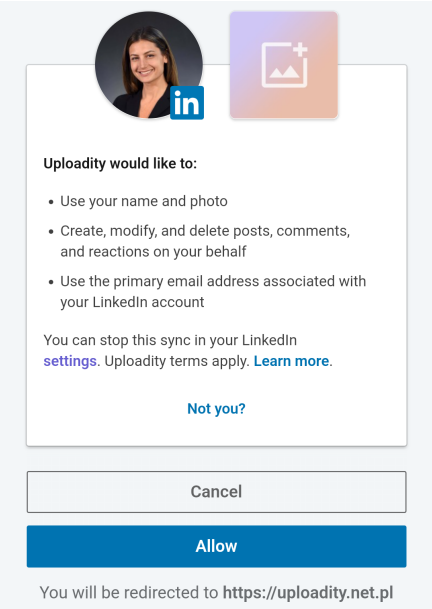
1. Aplikacja przekierowuje użytkownika na stronę autoryzacji `https://www.linkedin.com/oauth/v2/authorization`, gdzie użytkownik uprawnia aplikację do wykonywania akcji w jego imieniu. W adresie URL powinny się znajdować następujące parametry:

- `response_type` - jego wartość to zawsze `code`,
- `client_id`, czyli klucz API aplikacji wygenerowany w panelu dewelopera,
- `redirect_uri`, czyli adres URL, do którego użytkownik zostanie przekierowany po przejściu autoryzacji. Wartość ta powinna być taka sama jak podany wcześniej adres w panelu dewelopera,
- `scope`, czyli lista pozwoleń wymagana przez aplikację. Wartość ta powinna być zakodowana w systemie UTF-8.

Przykładowy adres URL, na jaki przekierowujemy użytkownika będzie wyglądał następująco:

```
https://www.linkedin.com/oauth/v2/authorization?response_type=code
&client_id=XXXXX&redirect_uri=https://uploadity.net.pl/linkedin
&scope=scope=w_member_social%20openid%20profile%20email
```

Po wykonaniu tego żądania użytkownik zostaje przekierowany do przeglądarki na stronę platformy, gdzie zostaje poproszony o udzielenie aplikacji pozwolenia na udostępnianie postów w jego imieniu oraz przekazanie podstawowych informacji:



Rysunek 2.7: Strona udzielenia uprawnień użytkownika przez aplikację. Źródło: opracowanie własne.

2. W przypadku, gdy użytkownik wyrazi zgodę, zostaje przekierowany na stronę przekazaną wcześniej jako parametr `redirect_uri` wraz z wartością `code`, która jest wymagany w następnym kroku autoryzacji. Przykładowy adres URL, na jaki zostanie przeniesiony użytkownik to:

```
https://uploadity.net.pl/linkedin?code=XXXXXX
```

Powyższy adres przekierowuje użytkownika z powrotem do aplikacji na tej samej zasadzie, jak w przypadku logowania na platformę X.

3. Ostatnim krokiem logowania jest wymiana kodu autoryzacyjnego uzyskanego w poprzednim kroku na token dostępowy za pomocą zapytania `POST oauth/v2/accessToken`. Ciało zapytania (ang. request body) powinno zawierać następujące parametry:

   - `grant_type` - jego wartość to zawsze `authorization_code`,
   - `code`, czyli klucz kod autoryzacyjny otrzymany w poprzednim kroku,
   - `redirect_uri`, czyli callback URL podany w poprzednim zapytaniu,
   - `client_id`, czyli klucz API aplikacji wygenerowany w panelu dewelopera,
   - `client_secret`, czyli czyli tajny klucz API aplikacji wygenerowany w panelu dewelopera.

   Wynikiem pomyślnie zakończonego zapytania jest wartość `access_token`, czyli token dostępowy do autoryzowania działań w imieniu użytkownika. Powinien być on przesyłany w nagłówku przy każdym żądaniu wykonywanym przez aplikację w następującym formacie:

```
Authorization: Bearer {access_token}
```

2.5.3 Przygotowanie obrazu

W celu udostępnienia posta z dołączonym do niego obrazem należy najpierw odpowiednio przygotować grafikę według wytycznych mediów społecznościowych. Ten proces zostanie omówiony poniżej z uwzględnieniem każdej z platform z osobna.

X

Aby przesłać zdjęcie, należy wykonać zapytanie `POST` na adres `https://upload.twitter.com/1.1/media/upload.json`. W żądaniu powinien znajdować się parametr `media_category`, czyli rodzaj przesyłanych mediów, w przypadku obrazu jest to `TWEET_IMAGE`. Należy również dodać dwa nagłówki: autoryzacyjny oraz `Content-Type: multipart/form-data` - pozwala on na jednoczesne przesyłanie wielu rodzajów danych w jednym zapytaniu. W żądaniu umieszczamy ciało zapytania w formacie `octet-stream`, który pozwala na przesyłanie pliku binarnego. Poniższy kod przedstawia implementację tego zapytania:

```
val file = createImageFile()
val client = OkHttpClient()
val requestBody = MultipartBody.Builder()
    .setType(MultipartBody.FORM)
    .addFormDataPart("media", file.name,
        file.asRequestBody("application/octet-stream".toMediaType()))
    .build()
val request = Request.Builder()
    .url("https://upload.twitter.com/1.1/media/upload.json
        ?media_category=tweet_image")
    .post(requestBody)
    .addHeader("Authorization", authorizationHeader)
    .build()
```

Jeśli żądanie przebiegło pomyślnie, otrzymujemy odpowiedź z danymi na temat obrazu, gdzie najważniejszym parametrem jest `media_id`, który później zostanie wykorzystany do załączenia mediów do posta.

Linkedin

W celu wysłania obrazu, należy na początku wykonać zapytanie `POST https://api.linkedin.com/rest/images?action=initializeUpload` inicjalizujące ten proces. Przekazujemy parametr `owner`, który jest id użytkownika na platformie Linkedin otrzymanym w procesie autoryzacji. Odpowiedź pomyślnie zakończonego zapytania zawiera adres URL, na który powinniśmy przesłać obraz oraz id grafiki. Następnie należy wysłać żądanie `PUT` na wskazany wcześniej adres i przekazać plik w ciele zapytania. Poniższy kod przedstawia implementację tego kroku:

```
val request = Request.Builder()
    .header("Authorization", "Bearer {access_token}")
    .url("https://www.linkedin.com/dms-uploads/sp/D4E10AQEHbyAxUyRRkA")
    .put(file.asRequestBody("image/png".toMediaTypeOrNull()))
    .build()
```

Jeśli zapytanie zwróci kod `201 Created`, oznacza to że przebiegło ono pomyślnie i otrzymany wcześniej kod id obrazu może zostać wykorzystany później do stworzenia posta.

Tumblr

Przygotowanie obrazu do przesłania w poście na platformę Tumblr polega na zakodowaniu pliku graficznego do ciągu znaków w formie base64. Na początku za pomocą URI (identyfikatora mediów) zostaje odczytana bitmapa obrazu, która zostaje skompresowana i przekonwertowana na postać binarną, a następnie zakodowana. Uzyskany ciąg znaków może zostać poźniej wykorzystany do przesłania grafiki w postaci tekstowej. Poniższy kod przedstawia, jak zostało to zaimplementowane w aplikacji:

```
val source = ImageDecoder.createSource(contentResolver, mediaUri)
val bitmap = ImageDecoder.decodeBitmap(source)
val byteArrayOutputStream = ByteArrayOutputStream()
bitmap.compress(Bitmap.CompressFormat.PNG, 100, byteArrayOutputStream)
val byteArray = byteArrayOutputStream.toByteArray()
val string = Base64.encodeToString(byteArray, Base64.DEFAULT)
```

2.5.4 Publikacja posta

Po przygotowaniu obrazu możemy przystąpić do publikacji posta. Proces udostępniania zostanie omówiony poniżej z uwzględnieniem każdej z platform z osobna.

X

Aby udostępnić post, należy wykonać zapytanie `POST /2/tweets` do API platformy. Poniższy kod przedstawia implementację tego żądania za pomocą biblioteki Retrofit:

```
@POST("2/tweets")
@Headers("Content-Type: application/json")
fun createTwitterPost(
    @Header("Authorization") authorizationHeader: String,
    @Body requestBody: RequestBody
): Call<ResponseBody>
```

W ciele zapytania powinien znajdować się obiekt JSON zawierający parametry `text`, czyli opis posta oraz `media`, czyli media dołączone do publikacji. Oto przykład takiego obiektu:

```
{
    "text": "Hello world!",
    "media": {"media_ids": ["1455952740635586573"]}
}
```

Parametr `media_ids` to tablica id mediów załączonych w poście. Tam umieszczamy wartość `media_id` otrzymaną w procesie przesyłania obrazu. Jeśli zapytanie przebiegło pomyślnie i post został udostępniony, w odpowiedzi otrzymujemy identyfikator publikacji.

Linkedin

Aby udostępnić post na platformie Linkedin, należy wykonać zapytanie `POST /rest/posts` do API. Poniższy kod przedstawia implementację tego żądania za pomocą biblioteki Retrofit:

```
@POST("rest/posts")
@Headers("LinkedIn-Version: 202308",
    "X-Restli-Protocol-Version: 2.0.0",
    "Content-Type: application/json")
fun createPost(
    @Query("oauth2_access_token") accessToken: String,
    @Body requestBody: RequestBody
): Call<ResponseBody>
```

W ciele zapytania powinien znajdować się obiekt JSON zawierający parametry `text`, czyli opis posta, `author`, czyli id profilu użytkownika oraz `id`, czyli identyfikator obrazu uzyskany wcześniej w procesie przesyłania grafiki. Oto przykład takiego obiektu:

```
{
    "author": "urn:li:person:5515715",
    "commentary": "Post",
    "visibility": "PUBLIC",
    "distribution": {
        "feedDistribution": "MAIN_FEED",
        "targetEntities": [],
        "thirdPartyDistributionChannels": []
    },
    "content": {
```

```
        "media": {
            "title":"Hello World!",
            "id": "urn:li:image:C5F10AQGKQg_6y2a4sQ"
        }
    },
    "lifecycleState": "PUBLISHED",
    "isReshareDisabledByAuthor": false
}
```

Pomyślnie wykonane zapytanie powinno zwrócić identyfikator udostępnionego posta.

Tumblr

Aby udostępnić publikację na platformie Tumblr, należy wykonać zapytanie `POST` do API. W ścieżce URL znajduje się parametr `blog-identifier`, czyli wybrany identyfikator bloga uzyskany w procesie autoryzacji użytkownika. Poniższy kod przedstawia implementację tego żądania za pomocą biblioteki Retrofit:

```
@POST("v2/blog/{blog-identifier}/post")
@Headers("Content-Type: application/json")
fun createPost(
    @Header("Authorization") authorization: String,
    @Path("blog-identifier") blogIdentifier: String,
    @Body requestBody: RequestBody
): Call<ResponseBody>
```

W ciele zapytania powinien znajdować się obiekt JSON zawierający parametry `type`, którego wartość to `photo`, `caption`, czyli opis posta oraz `data_64`, czyli zakodowany wcześniej ciąg znaków w formacie base64 reprezentujący przesyłaną grafikę. Oto przykład takiego obiektu:

```
{
    "type": "photo",
    "caption": "Hello World!",
    "data_64": "MIIHNjCCBh6gAwIBAgIQCVe4E0h49mzI0NcSqMy1+jANBgkqhkiG9w0B
                ..."
}
```

Jeśli zapytanie przebiegło pomyślnie i post został udostępniony, w odpowiedzi otrzymujemy identyfikator publikacji.

2.5.5 Usuwanie posta

API mediów społecznościowych oprócz funkcji udostępniania postów dostarczają możliwość usunięcia publikacji. Ten proces zostanie opisany poniżej z uwzględnieniej każdej z platform z osobna.

X

W celu usunięciu posta należy wykonać zapytanie `DELETE /2/tweets/{id}`, gdzie parametr `id` jest identyfikatorem otrzymanym po utworzeniu publikacji. Poniższy kod przedstawia implementację tego żądania za pomocą biblioteki Retrofit:

```
@DELETE("2/tweets/{id}")
fun deletePost(
    @Header("Authorization") authorizationHeader: String,
    @Path("id") id: String
): Call<ResponseBody>
```

Linkedin

Aby usunąć post udostępniony na platformie Linkedin, należy wykonać zapytanie `DELETE /rest/posts/{id}`, gdzie parametr `id` jest identyfikatorem otrzymanym po utworzeniu publikacji. Poniższy kod przedstawia implementację tego żądania z użyciem biblioteki Retrofit:

```
@DELETE("/rest/posts/{id}")
@Headers("LinkedIn-Version: 202308")
fun deletePost(
    @Header("Authorization") authorizationHeader: String,
    @Path("id") id: String
): Call<ResponseBody>
```

Tumblr

W celu usunięciu publikacji na platformie Tumblr, należy wykonać zapytanie `DELETE /v2/blog/{blog-identifier}/post/delete`, gdzie parametr `blog-identifier` jest identyfikatorem bloga, na którym post został udostępniony. W ciele zapytania powinien zostać umieszczony parametr `id`, czyli identyfikator publikacji. Poniższy kod przedstawia implementację tego żądania za pomocą biblioteki Retrofit:

```
@POST("/v2/blog/{blog-identifier}/post/delete")
fun deletePost(
    @Header("Authorization") authorization: String,
    @Path("blog-identifier") blogIdentifier: String,
    @Body requestBody: RequestBody
): Call<ResponseBody>
```

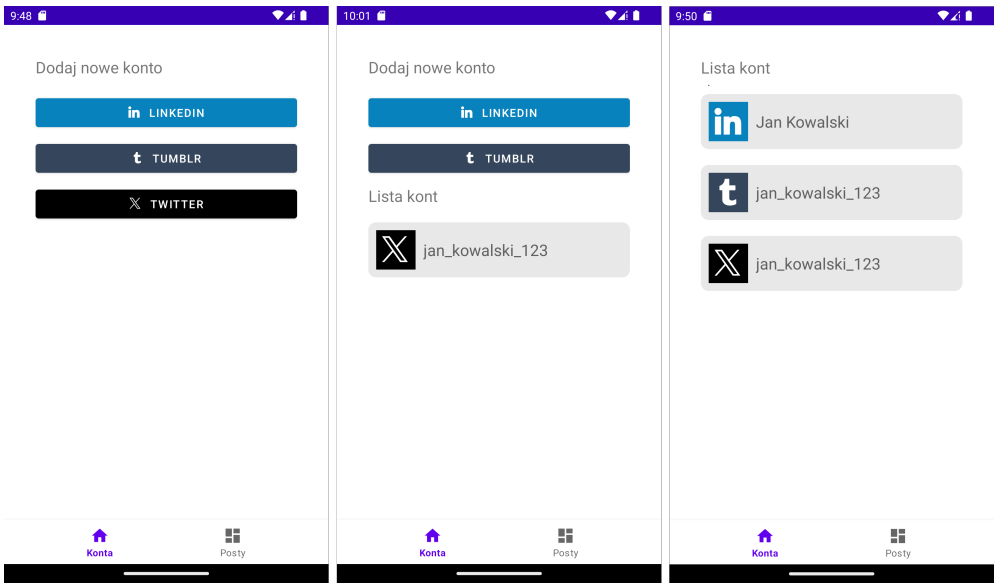
Rozdział 3

Aplikacja Uploadity

W tym rozdziale zostanie przedstawiona końcowa wersja użytkownika wraz z jej zrzutami ekranu, opisem wszystkich funkcji i przykładami użycia.

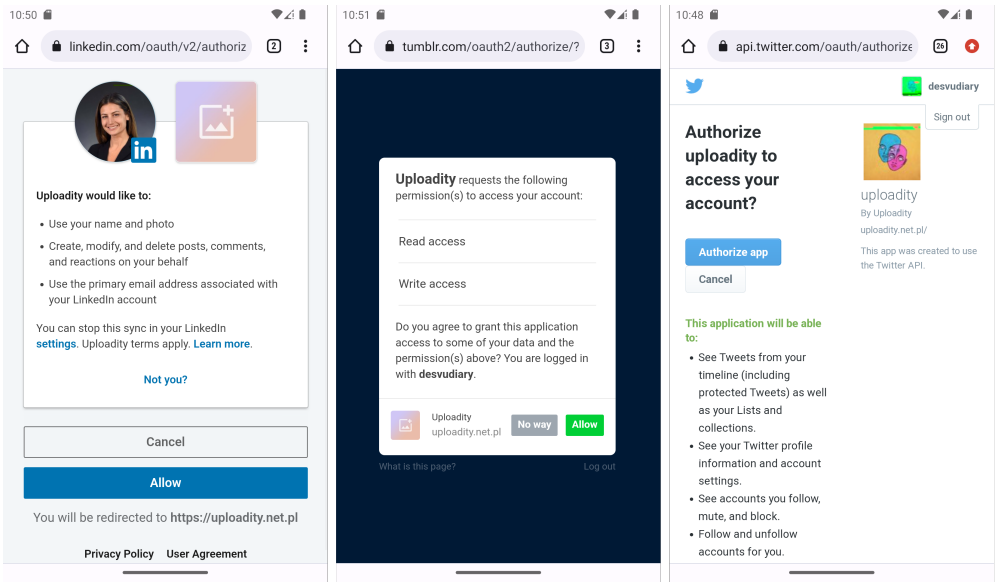
3.1 Zakładka konta

Pierwszym ekranem aplikacji jest zakładka Konta, na której użytkownik może połączyć Uploadity ze swoimi profilami na platformach społecznościowych przez wciśnięcie przycisku z listy, gdzie każde z mediów społecznościowych odpowiada jednemu z nich. Po zsynchronizowaniu konta pojawia się ono na liście z nazwą użytkownika i ikoną danej platformy.



Rysunek 3.1: Widok ekranu Konta przed i po połączeniu kont społecznościowych

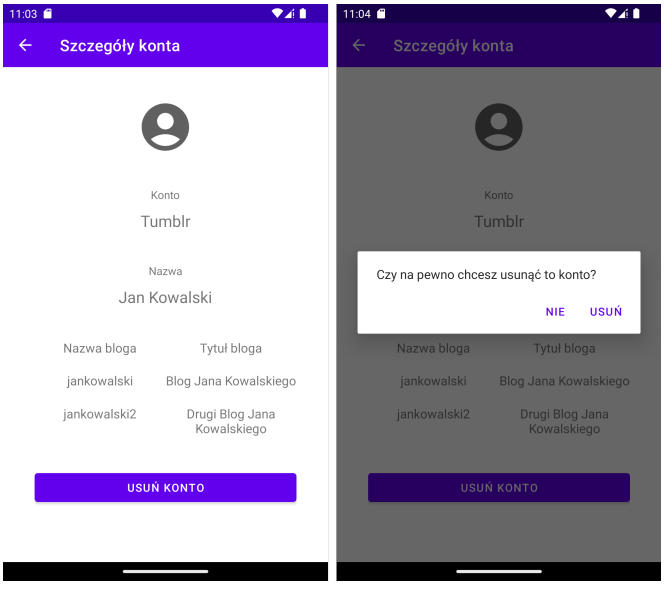
Po wciśnięciu jednego z przycisków użytkownik zostaje przeniesiony na stronę internetową wybranej platformy, gdzie po zalogowaniu do swojego konta pojawia się ekran udzielenia dostępu do publikacji postów z poziomu aplikacji Uploadity. Są na nim umieszczone uprawnienia, o jakie prosi aplikacja oraz przyciski z pozwoleniem lub odmówieniem dostępu. W obu przypadkach użytkownik zostaje przeniesiony spowrotem do aplikacji, która otrzymuje informację o dokonanym wyborze.



Rysunek 3.2: Ekrany udzielenia dostępu aplikacji do publikacji postów

3.2 Szczegóły konta

Po wciśnięciu jednego z elementów listy kont użytkownik zostaje przeniesiony na ekran szczegółów konta, gdzie znajdują się informacje o nazwie platformy, nazwie użytkownika oraz w przypadku aplikacji Tumblr lista blogów połączonych z profilem. Znajduje się tam również przycisk „Usuń konto", które pozwala użytkownikowi na usunięcie profilu z aplikacji.

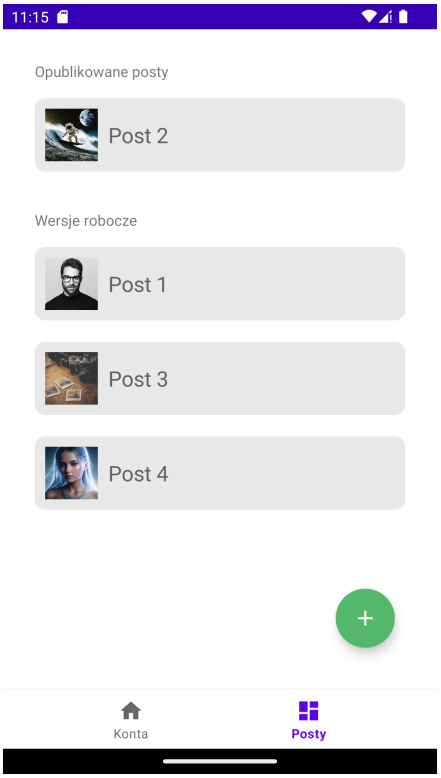


Rysunek 3.3: Ekrany szczegółów konta

3.3 Zakładka posty

Zakładka Posty przedstawia listę wersji roboczych oraz opublikowanych postów. Każdy z elementów listy posiada miniaturę zdjęcia oraz tytuł posta. W prawym dolnym rogu

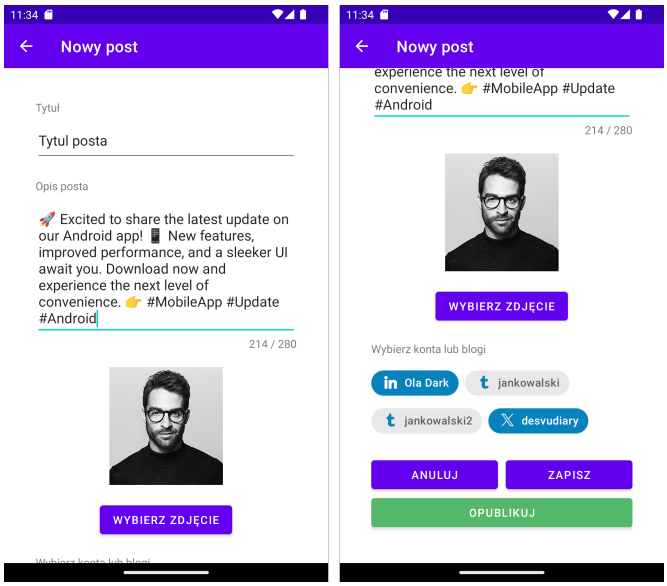
znajduje się przycisk, który przenosi użytkownika do ekranu tworzenia nowego posta.



Rysunek 3.4: Zakładka posty

3.4 Nowy post

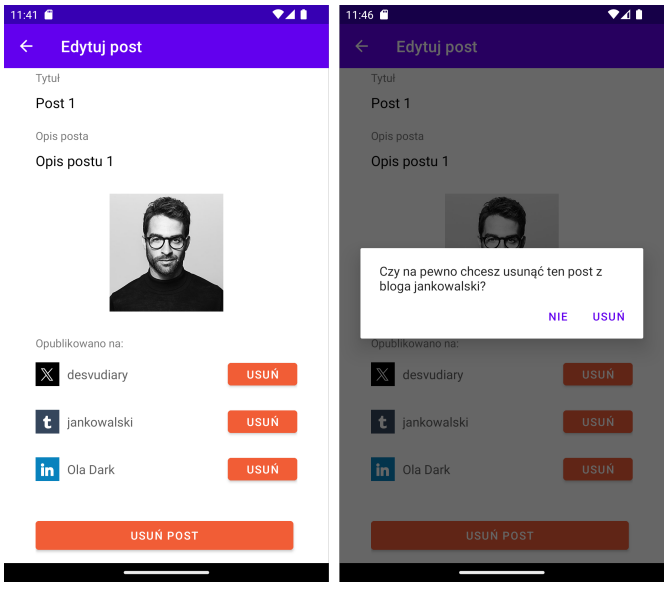
Ekran ten służy do stworzenia nowej wersji roboczej lub publikacji posta. Znajdują się tu dwa pola tekstowe, gdzie można nadać tytuł oraz opis. Maksymalna ilość znaków, jakie może posiadać treść posta wynosi 280 - z tego powodu pole opisu również ma takie ograniczenie, a aktualna długość tekstu wyświetlona jest w prawym dolnym rogu elementu. Kolejnym elementem jest przycisk wyboru zdjęcia, który otwiera galerię zdjęć telefonu. Po wybraniu obrazu pojawia się ono na ekranie. Użytkownik ma również możliwość wybrania kont, na których ma zostać opublikowany post.



Rysunek 3.5: Ekran tworzenia nowego posta

3.5 Edytuj post

Na tym ekranie znajdują się szczegóły opublikowanego posta - jego tytuł, opis oraz lista kont, na które został wysłany. Użytkownik ma możliwość usunięcia publikacji z wybranej platformy lub z listy postów w aplikacji.



Rysunek 3.6: Ekran edycji opublikowanego posta