

TEMAT: Składowanie danych.

WPROWADZENIE



1. **Shared Preferences** – wykorzystywane do przechowywania niewielkich porcji danych w postaci pliku zawierającego pary {klucz, wartość}. Obsługuje typy proste, takie jak boolean, float, int, long, a także String. Zapewnia utrwalenie danych w trakcie sesji użytkownika (również w przypadku wyłączenia aplikacji). Standardowe zastosowanie obejmuje przechowywanie i współdzielenie między aktywnościami ustawień aplikacji (np. kolor tła aplikacji, wybrany tryb działania).
 - a. Preferencje aplikacji to zbiory danych przechowywanych w sposób **trwały**, co oznacza, że preferencje są zachowywane niezależnie od zdarzeń cyklu życia aplikacji → aplikacja lub urządzenie mogą zostać zamknięte/wyłączone i ponownie uruchomione, a nie spowoduje to utraty danych.
 - b. Wiele aplikacji potrzebuje tego prostego mechanizmu przechowywania danych do przechowywania swojego stanu, podstawowych informacji o użytkowniku, opcji konfiguracyjnych i innych.
 - c. Poszczególne aktywności mogą posiadać swoje własne, **prywatne**, preferencje. Są one przeznaczone wyłącznie dla danej aktywności (inne aktywności tworzące aplikację nie mogą z nich korzystać). Każda aktywność może posiadać tylko jedną grupę prywatnych preferencji, której nazwa odpowiada nazwie klasy danej aktywności.

```
SharedPreferences privatePreferences = getPreferences(MODE_PRIVATE);
```
 - d. Istnieje możliwość tworzenia wspólnych preferencji, używanych przez większą liczbę aktywności. Przy tworzeniu preferencji tego typu konieczne jest podanie nazwy zbioru preferencji – jest ona identyfikatorem preferencji, który umożliwia pobieranie ich z poziomu dowolnej aktywności aplikacji.

```
private static final String PREFERENCE_FILENAME = "CustomPref";

SharedPreferences sharedPreferences = getSharedPreferences(PREFERENCE_FILENAME, MODE_PRIVATE);
```
 - e. Modyfikowanie preferencji odbywa się za pomocą obiektu edytora. Wprowadzone zmiany powinny być zatwierdzone.

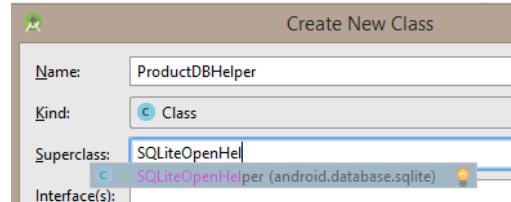
```
SharedPreferences privatePreferences = getPreferences(MODE_PRIVATE);
SharedPreferences.Editor prefEditor = privatePreferences.edit();
prefEditor.putBoolean("isUsingPreferencesSimple", true);
prefEditor.apply();
```
 - f. Reagowanie na zmiany w preferencjach może być zrealizowane poprzez implementację odpowiedniego odbiorcy zdarzeń i zarejestrowanie go w konkretnym obiekcie SharedPreferences, używając do tego celu metod *registerOnSharedPreferencesListener()* oraz *unregisterOnSharedPreferencesListener()*.
2. **Pamięć wewnętrzna i karta SD** – służą przede wszystkim do składowania plików w niezmienniczej formie (zdjęć, filmów itp.). Wyróżniamy pamięć wewnętrzną (zawsze dostępną) oraz zewnętrzną (trzeba mieć na uwadze, że może nie być dostępna, do jej użycia wymagane są specjalne uprawnienia).

- a. W wielu przypadkach wykorzystanie jedynie preferencji okazuje się niewystarczające – aplikacje często wymagają znacznie bardziej solidnego i elastycznego rozwiązania, pozwalającego na trwałe przechowywanie i dającego możliwość odczytu danych dowolnego typu.
 - b. Przykłady typów danych, których przechowywanie może być konieczne: treści multimedialne (obrazy, dźwięki, klipy wideo itp.), treści pobrane z Internetu, złożone treści generowane przez aplikację.
 - c. **Ważne:** każdy odczyt danych z dysku lub ich zapis na tym nośniku jest poważną operacją blokującą, która wykorzystuje cenne zasoby systemowe. Z tego względu, w większości przypadków, operacje związane z dostępem do plików **nie powinny być wykonywane w głównym wątku**. Zamiast tego należy je wykonywać asynchronicznie z wykorzystaniem wątków, obiektów *AsyncTask* itp.
 - d. Urządzenia z systemem Android mają ograniczoną przestrzeń do przechowywania danych. Stąd, aby ograniczać jej użycie, **należy przechowywać tylko to, co niezbędne**. Niepotrzebne dane powinny być niezwłocznie usuwane.
 - e. Zawsze warto **sprawdzać ilość dostępnych zasobów**, takich jak wielkość wolnej przestrzeni na dysku lub **możliwość wykorzystania zewnętrznego nośnika danych**, a dopiero potem używać tych zasobów.
 - f. Każda aplikacja dysponuje domyślnie własnym katalogiem oraz własnymi plikami. Pliki tworzone w katalogu aplikacji są domyślnie prywatne i dostępne wyłącznie dla danej aplikacji.
 - g. Aby aplikacja mogła korzystać ze swoich własnych plików, nie trzeba umieszczać w jej pliku manifestu żadnych szczególnych uprawnień. W systemie Android 4.4 (i nowszych) uzyskiwanie uprawnień *READ_EXTERNAL_STORAGE* i *WRITE_EXTERNAL_STORAGE* nie jest konieczne w przypadku odczytu i zapisu prywatnych plików aplikacji. Są one jednak niezbędne w razie odwoływania się do publicznego obszaru zewnętrznego nośnika danych.
 - h. Utworzenie pliku w domyślnym katalogu aplikacji i zapisanie w nim danych może odbywać się w następujący sposób:

```
String fileContent = "Some text that should be saved in file";
FileOutputStream fileOutputStream = openFileOutput( name: "filename.txt", MODE_PRIVATE);
fileOutputStream.write(fileContent.getBytes());
fileOutputStream.close();
```
 - i. Dopisywanie danych do istniejącego pliku odbywa się za pomocą trybu *MODE_APPEND*:

```
String fileContentAdditional = "Some text that should be added to file";
FileOutputStream fileOutputStream = openFileOutput( name: "filename.txt", MODE_APPEND);
fileOutputStream.write(fileContentAdditional.getBytes());
fileOutputStream.close();
```
3. **Lokalna baza danych SQLite** – zapewnia utrwalanie danych w ramach danej aplikacji. Konceptyjnie przypomina standardowe relacyjne bazy danych składające się z tabel, w których poszczególne rekordy opisane są wartościami atrybutów zwanych kolumnami. Baza ta jest standardowym elementem każdego urządzenia z systemem Android. Zapewnia obsługę transakcji, nie wymaga dodatkowej konfiguracji. Warto jednak pamiętać, że funkcjonalności tej bazy są ograniczone w porównaniu z typowymi silnikami bazodanowymi takimi jak Oracle czy MySQL.
- a. W przypadku konieczności przechowywania danych o określonej strukturze najlepiej sprawdzi się baza SQLite.

- b. Pierwotnie praca z bazą danych na platformie Android wymagała na początku zdefiniowania tabel za pomocą klasy *SQLiteOpenHelper*. Należało zdefiniować klasę dziedziczącą po tej klasie pomocniczej (*SQLiteOpenHelper*) i zadeklarować nazwę bazy danych, jej wersję, tabele i tworzące je kolumny. W kodzie tej klasy baza danych była tworzona i ewentualnie aktualizowana.
- c. **Ważne:** klasa *SQLiteOpenHelper* zakładała, że w przypadku aktualizacji numer wersji bazy danych będzie większy od poprzedniego.
- d. Dostęp do utworzonej bazy danych był możliwy za pomocą klasy *SQLiteDatabase*.



```
public class ProductDBHelper extends SQLiteOpenHelper {

    private static final String DB_NAME = "products.db";
    private static final int DB_VERSION = 1;

    public static final String TABLE_PRODUCTS = "PRODUCT";
    public static final String COLUMN_ID = "_ID";
    public static final String COLUMN_NAME = "NAME";
    public static final String COLUMN_PRICE = "PRICE";

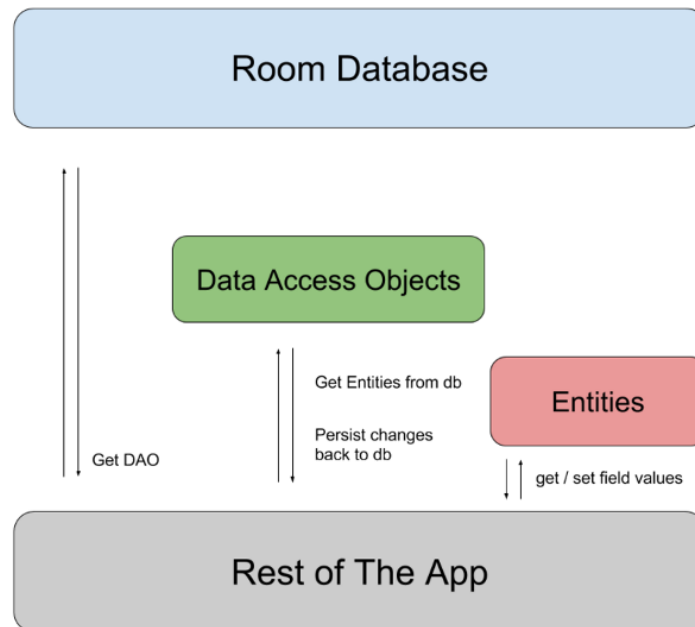
    private static final String TABLE_CREATE =
        "CREATE TABLE " + TABLE_PRODUCTS + " (" +
            COLUMN_ID + " INTEGER PRIMARY KEY AUTOINCREMENT, " +
            COLUMN_NAME + " TEXT, " +
            COLUMN_PRICE + " REAL" +
        ")";

    public ProductDBHelper(Context context) {
        super(context, DB_NAME, factory: null, DB_VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase sqLiteDatabase) {
        sqLiteDatabase.execSQL(TABLE_CREATE);
    }

    @Override
    public void onUpgrade(SQLiteDatabase sqLiteDatabase, int i, int i1) {
        sqLiteDatabase.execSQL("DROP TABLE IF EXISTS " + TABLE_PRODUCTS);
        onCreate(sqLiteDatabase);
    }
}
```

- e. Aktualnie preferowana metoda korzystania z bazy danych to biblioteka **Room**. Zapewnia ona dodatkową warstwę abstrakcji w dostępie do bazy SQLite. Nie ma więc konieczności implementacji własnych klas pomocniczych.



- f. Dlaczego nie jest rekomendowane korzystanie bezpośrednio z SQLite API? Oto informacja z oficjalnej strony Android Developers:

Caution: Although these APIs are powerful, they are fairly low-level and require a great deal of time and effort to use:

- There is no compile-time verification of raw SQL queries. As your data graph changes, you need to update the affected SQL queries manually. This process can be time consuming and error prone.
- You need to use lots of boilerplate code to convert between SQL queries and data objects.

For these reasons, we **highly recommended** using the [Room Persistence Library](#) as an abstraction layer for accessing information in your app's SQLite databases.

<https://developer.android.com/training/data-storage/sqlite>

- g. W celu uniknięcia problemów z wydajnością, biblioteka *Room* nie pozwala na wykonywanie zapytań do bazy w głównym wątku aplikacji. Reguła ta jest realizowana poprzez wykorzystanie *LiveData*, gdzie automatycznie zapytania uruchamiane są w tle (asynchronicznie).
- h. *LiveData* jest klasą pomocniczą, należącą do grupy komponentów odpowiadających za wykonywanie pewnych zadań w odpowiedzi na zmiany w cyklu życia innych komponentów, takich jak aktywności, czy fragmenty.
- i. Składnia zapytań jest sprawdzana na etapie kompilacji.

j. Biblioteka *Room* składa się z trzech głównych komponentów (klas):

- Database – odpowiada za połączenie z bazą danych, jej tworzenie i zainicjowanie. Wymagania:
 - klasa abstrakcyjna
 - dziedziczy po *RoomDatabase*
 - posiada adnotację *@Database*
 - zawiera listę encji
 - posiada bezparametrową metodę abstrakcyjną zwracającą instancję DAO (obiekt klasy oznaczonej adnotacją *@Dao*)

```
@Database(entities = {Product.class}, version = 1)
public abstract class ProductDatabase extends RoomDatabase {
    public abstract ProductDao productDao();
}
```

- Entity – reprezentuje tabelę w bazie danych. Posiada adnotację *@Entity*. Wymaga metod dostępowych do posiadanych pól (*getter* i *setter*). Zawiera przynajmniej jedno pole będące kluczem głównym (*@PrimaryKey*). Możliwe jest automatyczne generowanie wartości klucza głównego – należy skorzystać z właściwości *autoGenerate*. Domyślnie nazwa tabeli jest taka sama, jak nazwa klasy encji. W celu jej zmiany należy dodać właściwość *tableName* do adnotacji *@Entity*. W podobny sposób można nadawać inne nazwy poszczególnym kolumnom tabeli (właściwość *name* podawana wraz z adnotacją *@ColumnInfo*). Domyślnie wszystkie pola są przetwarzane na kolumny tabeli. Jeśli istnieje konieczność pominięcia jakiegoś pola klasy encji, należy poprzedzić jego deklarację adnotacją *@Ignore*.

```
@Entity
public class Product {
    @PrimaryKey(autoGenerate = true)
    private int id;
}
```

- DAO (Data Access Object) – posiada metody dostępne do bazy danych. Zapewnia możliwość korzystania z danych przechowywanych w bazie. Przechowuje zapytania. Wymaga adnotacji *@Dao*. Komponent taki powinien być interfejsem lub klasą abstrakcyjną. Podstawowe zapytania mogą być realizowane poprzez zastosowanie odpowiednich adnotacji, np. *@Insert*, *@Update*, *@Delete*. Własne zapytania oznaczane są adnotacją *@Query*. Implementacje metod realizujących zapytania generowane są automatycznie.

```
@Dao
public interface ProductDao {

    @Insert
    void insert(Product product);

    @Query("SELECT * FROM product ORDER BY name")
    LiveData<List<Product>> loadAllProducts();

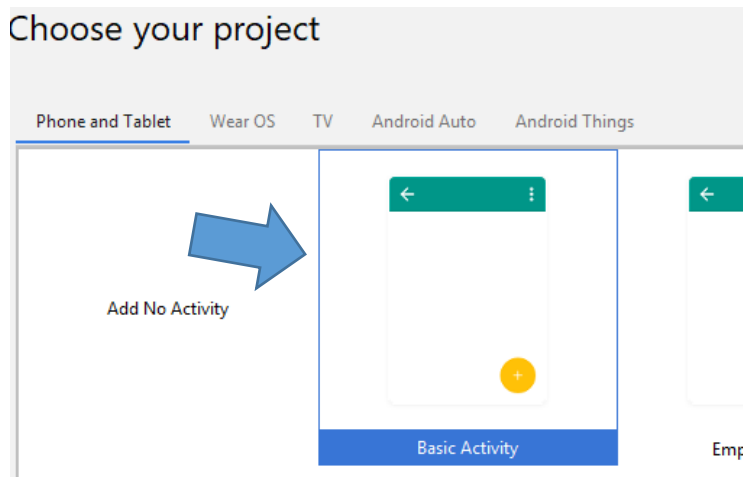
    @Query("SELECT * FROM product WHERE name LIKE :likeName")
    LiveData<List<Product>> findProductsWithName(String likeName);
}
```

TREŚĆ ZADANIA

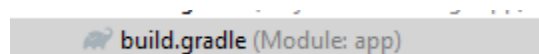
1. **Cel:** poznanie mechanizmów składowania danych z wykorzystaniem bazy danych SQLite oraz dedykowanego mechanizmu zarządzania wbudowaną bazą danych (framework Room). Zadanie obejmuje:



- a. Implementację warstwy danych z wykorzystaniem biblioteki Room
 - b. Implementację warstwy widoku listy z wykorzystaniem biblioteki *RecyclerView*
 - c. Implementację operacji CRUD na danych z bazy (create | read | update | delete)
 - d. Dodanie powiadomień typu *Snackbar*
 - e. Wykorzystanie przycisku typu *Floating Action Button*
2. Stwórz aplikację *LibraryApp* do przechowywania listy książek. Wybierz aplikację typu *Basic Activity*.



3. Dodaj niezbędne zależności, aby mieć możliwość korzystania z biblioteki *Room*:



W bloku *dependencies* dodaj wpisy potrzebne do zrealizowania zadania:

```
implementation "androidx.room:room-runtime:2.4.3"
annotationProcessor "androidx.room:room-compiler:2.4.3"
androidTestImplementation "androidx.room:room-testing:2.4.3"

implementation "androidx.lifecycle:lifecycle-common-java8:2.5.1"
implementation 'androidx.lifecycle:lifecycle-extensions:2.2.0'
implementation "com.google.android.material:material:1.7.0"
```

4. Dodaj encję książki. W tym celu utwórz klasę *Book* i dodaj odpowiednie adnotacje:

```
@Entity(tableName = "book")
public class Book {

    @PrimaryKey(autoGenerate = true)
    private int id;
    private String title;
    private String author;
```

Dodaj również *getter*y i *setter*y.

5. Utwórz interfejs *BookDao*:

```
@Dao
public interface BookDao {

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    void insert(Book book);

    @Update
    void update(Book book);

    @Delete
    void delete(Book book);

    @Query("DELETE FROM book")
    void deleteAll();

    @Query("SELECT * FROM book ORDER BY title")
    LiveData<List<Book>> findAll();

    @Query("SELECT * FROM book WHERE title LIKE :title")
    List<Book> findBookWithTitle(String title);
```

6. Utwórz bazę danych książek *BookDatabase*:

```
@Database(entities = {Book.class}, version = 1, exportSchema = false)
public abstract class BookDatabase extends RoomDatabase {

    private static BookDatabase databaseInstance;
    static final ExecutorService databaseWriteExecutor = Executors.newSingleThreadExecutor();

    public abstract BookDao bookDao();

    static BookDatabase getDatabase(final Context context) {
        if (databaseInstance == null) {
            databaseInstance = Room.databaseBuilder(context.getApplicationContext(),
                BookDatabase.class, name: "book_database")
                .build();
        }
        return databaseInstance;
    }
}
```

7. Utwórz klasę repozytorium: *BookRepository*. Dodaj w niej odpowiedniki metod z interfejsu DAO:

```
public class BookRepository {
    private final BookDao bookDao;
    private final LiveData<List<Book>> books;

    BookRepository(Application application) {
        BookDatabase database = BookDatabase.getDatabase(application);
        bookDao = database.bookDao();
        books = bookDao.findAll();
    }

    LiveData<List<Book>> findAllBooks() { return books; }

    void insert(Book book) {
        BookDatabase.databaseWriteExecutor.execute(() -> bookDao.insert(book));
    }

    void update(Book book) {
        BookDatabase.databaseWriteExecutor.execute(() -> bookDao.update(book));
    }

    void delete(Book book) {
        BookDatabase.databaseWriteExecutor.execute(() -> bookDao.delete(book));
    }
}
```

Zauważ, że metody typu CRUD interfejsu DAO są wywoływane w oddzielnych wątkach – wymusza to biblioteka *Room*, aby nie blokować głównego wątku aplikacji, w którym działa UI.

8. Dodaj klasę typu *ViewModel*:

```
public class BookViewModel extends AndroidViewModel {

    private final BookRepository bookRepository;
    private final LiveData<List<Book>> books;

    public BookViewModel(@NonNull Application application) {
        super(application);
        bookRepository = new BookRepository(application);
        books = bookRepository.findAllBooks();
    }

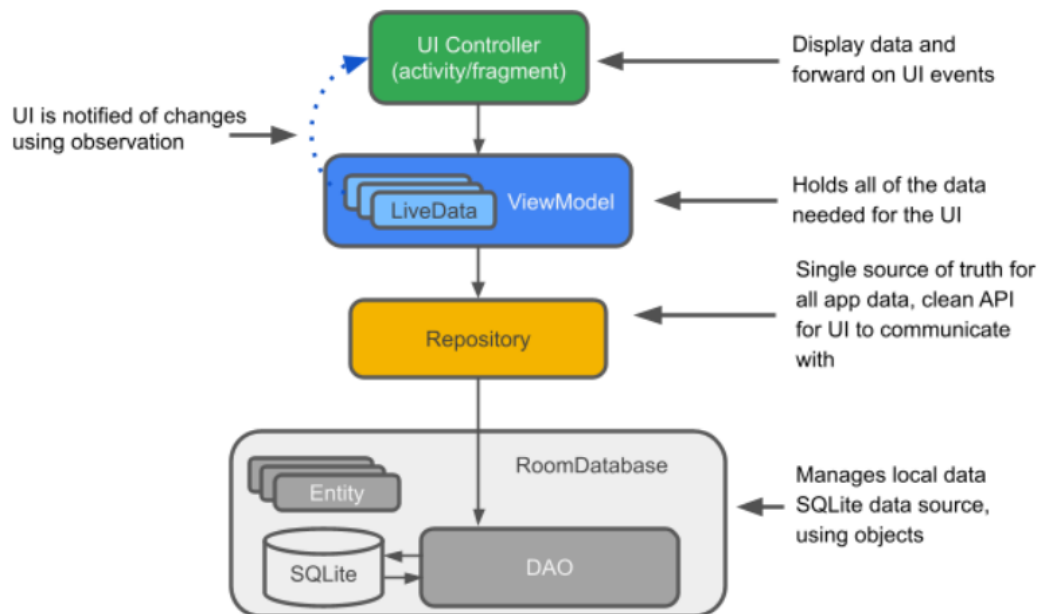
    public LiveData<List<Book>> findAll() { return books; }

    public void insert(Book book) { bookRepository.insert(book); }

    public void update(Book book) { bookRepository.update(book); }

    public void delete(Book book) { bookRepository.delete(book); }
}
```


ViewModel pośredniczy w komunikacji między repozytorium z danymi a interfejsem graficznym. Instancje *ViewModel* zostają zachowane w trakcie tworzenia od nowa aktywności lub fragmentu (np. przy obracaniu ekranu). Dzięki temu możliwe jest zachowanie stanu interfejsu graficznego.



9. Teraz kolej na przygotowanie widoku. Zaczniemy od dodania niezbędnych bibliotek w pliku *build.gradle* (Module app):

```
implementation "androidx.cardview:cardview:1.0.0"
implementation 'androidx.recyclerview:recyclerview:1.3.0-rc01'
```

10. Utwórz plik *book_list_item.xml*, a w nim dodaj elementy, które pozwolą na wyświetlenie na liście tytułu książki oraz autora.
11. Zmodyfikuj w następujący sposób plik *activity_main.xml*:

Usuń linię:

```
<include layout="@layout/content_main" />
```

Usuń wszystkie kontrolki oprócz *FloatingActionButton*.

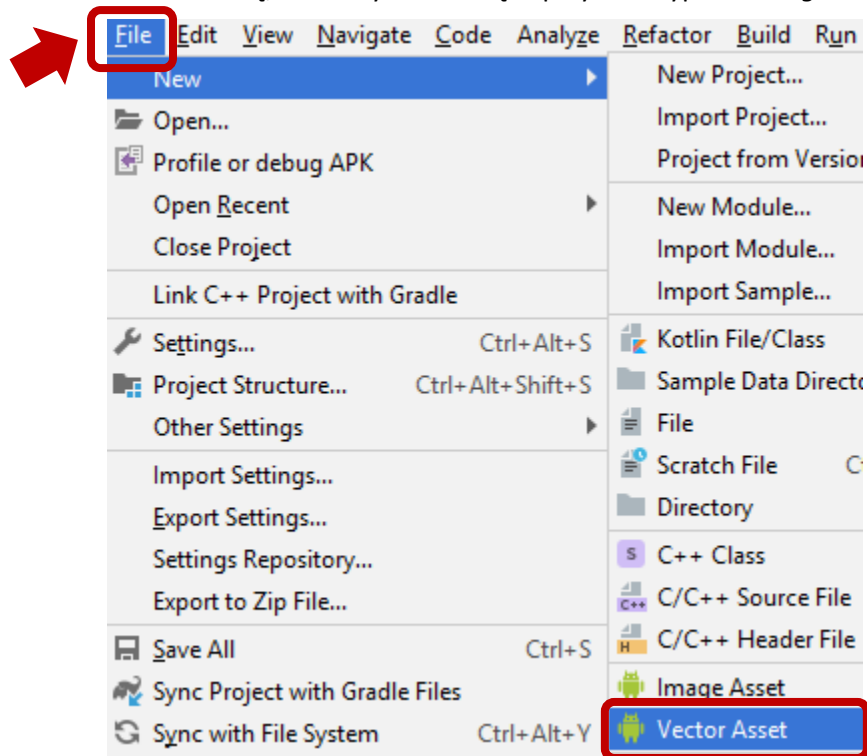
Zmień rodzaj głównego układu na *ConstraintLayout*.

```
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/main_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">
```

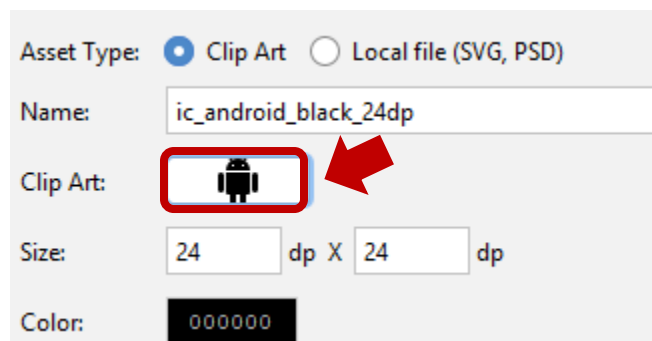
Wewnątrz układu *ConstraintLayout* dodaj element typu *RecyclerView*:

```
<androidx.recyclerview.widget.RecyclerView
    android:id="@+id/recyclerview"
    android:layout_width="0dp"
    android:layout_height="0dp"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    tools:listitem="@layout/book_list_item" />
```

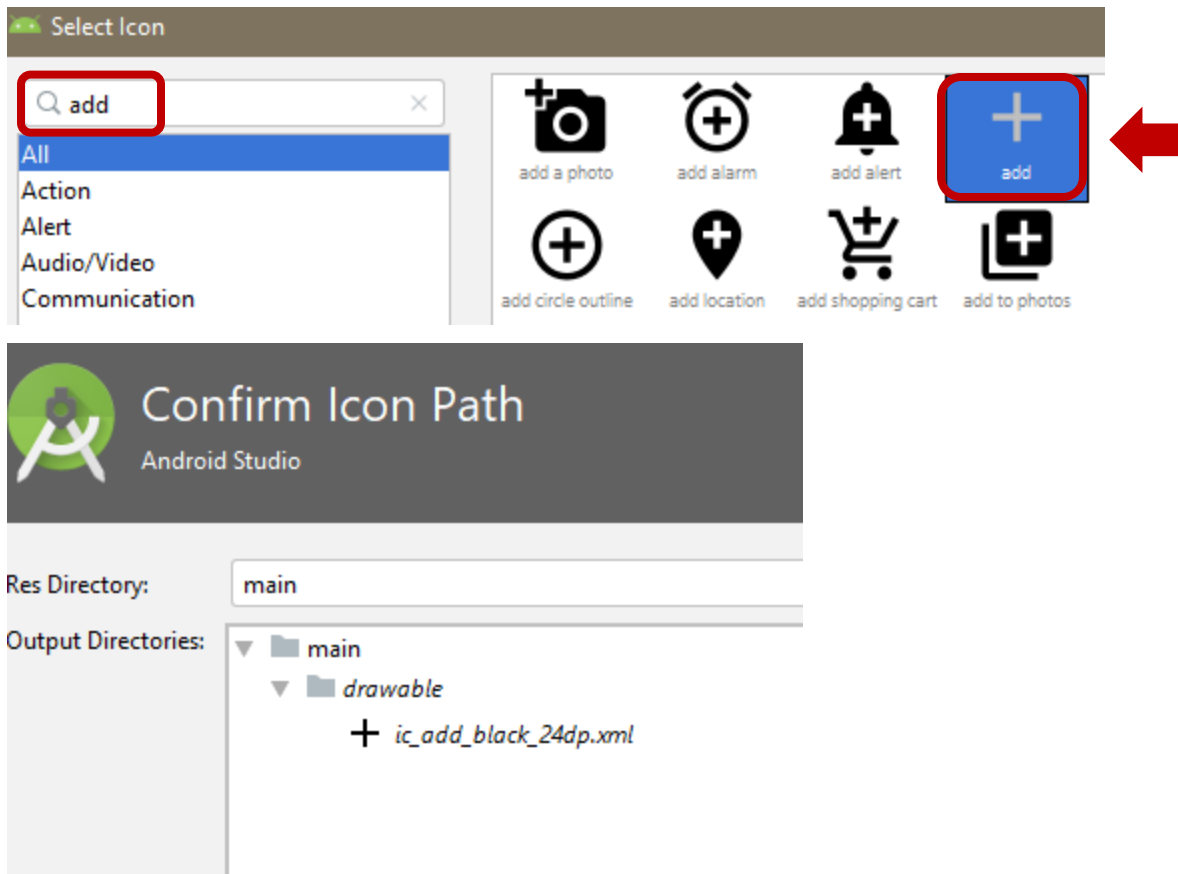
12. Zmień ikonkę, która wyświetla się w przycisku typu *Floating Action Button*:



Kliknij w ikonkę *Androida*:




Wybierz ikonę, która będzie pasowała do przycisku dodawania książek, np.:



Zatwierdź dodanie ikony klikając przycisk *Finish*.

Zaktualizuj kontrolkę przycisku w następujący sposób:

```
<com.google.android.material.floatingactionbutton.FloatingActionButton
    android:id="@+id/add_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    android:layout_margin="16dp"
    app:backgroundTint="@color/buttonColor"
    android:src="@drawable/ic_add_black_24dp" />
```



13. W *MainActivity* dodaj klasy adaptera i holdera:

```

private class BookHolder extends RecyclerView.ViewHolder {

    private TextView bookTitleTextView;
    private TextView bookAuthorTextView;

    public BookHolder(LayoutInflater inflater, ViewGroup parent) {
        super(inflater.inflate(R.layout.book_list_item, parent, attachToRoot: false));

        bookTitleTextView = itemView.findViewById(R.id.book_title);
        bookAuthorTextView = itemView.findViewById(R.id.book_author);
    }

    public void bind(Book book) {
        bookTitleTextView.setText(book.getTitle());
        bookAuthorTextView.setText(book.getAuthor());
    }
}

private class BookAdapter extends RecyclerView.Adapter<BookHolder> {
    private List<Book> books;

    @NonNull
    @Override
    public BookHolder onCreateViewHolder(@NonNull ViewGroup parent, int viewType) {
        return new BookHolder(getLayoutInflater(), parent);
    }

    @Override
    public void onBindViewHolder(@NonNull BookHolder holder, int position) {
        if (books != null) {
            Book book = books.get(position);
            holder.bind(book);
        } else {
            Log.d( tag: "MainActivity", msg: "No books");
        }
    }

    @Override
    public int getItemCount() {
        if (books != null) {
            return books.size();
        } else {
            return 0;
        }
    }

    void setBooks(List<Book> books) {
        this.books = books;
        notifyDataSetChanged();
    }
}

```

14. W klasie *MainActivity* Dodaj pole do przechowania obiektu *ViewModel*:

```
private BookViewModel bookViewModel;
```

15. Zaktualizuj metodę *onCreate()* w klasie *MainActivity* w celu połączenia bazy danych z widokiem przy pomocy mechanizmu *LiveData*:

```
RecyclerView recyclerView = findViewById(R.id.recyclerview);  
final BookAdapter adapter = new BookAdapter();  
recyclerView.setAdapter(adapter);  
recyclerView.setLayoutManager(new LinearLayoutManager(context, this));  
  
bookViewModel = new ViewModelProvider(owner: this).get(BookViewModel.class);  
bookViewModel.findAll().observe(owner: this, adapter::setBooks);
```

16. Wypełnij bazę danych przynajmniej kilkoma książkami. Dodaj następujący kod w klasie *BookDatabase*:

```
private static final RoomDatabase.Callback roomDatabaseCallback = new RoomDatabase.Callback() {  
    @Override  
    public void onCreate(@NonNull SupportSQLiteDatabase db) {  
        super.onCreate(db);  
        databaseWriteExecutor.execute(() -> {  
            BookDao dao = databaseInstance.bookDao();  
            Book book = new Book(title: "Clean Code", author: "Robert C. Martin");  
            dao.insert(book);  
        });  
    }  
};
```

W tej samej klasie zmodyfikuj tworzenie instancji bazy w metodzie *getDatabase()*, dodając następującą linię:

```
databaseInstance = Room.databaseBuilder(context.getApplicationContext(),  
    BookDatabase.class, name: "book_database")  
    .addCallback(roomDatabaseCallback)  
    .build();
```

Sprawdź jaka jest różnica między metodą *onCreate* i *onOpen* w klasie *Callback* bazy danych Room.

17. Dodaj potrzebne wpisy w pliku *strings.xml*:

```
<string name="hint_book_title">Title...</string>  
<string name="hint_book_author">Author...</string>  
<string name="button_save">Save</string>  
<string name="empty_not_saved">Book not saved because there are empty fields.</string>
```

oraz w pliku *colors.xml*:

```
<color name="buttonColor">#a1887f</color>
```

i w pliku *dimens.xml*:

```
<dimen name="small_padding">6dp</dimen>
<dimen name="big_padding">16dp</dimen>
```

18. Dodaj klasę aktywności typu *EmptyActivity* do tworzenia i aktualizowania elementów listy. Nazwij ją *EditBookActivity*.
19. Edytuj plik układu nowej klasy *activity_edit_book.xml*:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical" android:layout_width="match_parent"
    android:layout_height="match_parent">

    <EditText
        android:id="@+id/edit_book_title"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:fontFamily="sans-serif-light"
        android:hint="@string/hint_book_title"
        android:inputType="textAutoComplete"
        android:padding="@dimen/small_padding"
        android:layout_marginBottom="@dimen/big_padding"
        android:layout_marginTop="@dimen/big_padding"
        android:textSize="18sp" />

    <EditText
        android:id="@+id/edit_book_author"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:fontFamily="sans-serif-light"
        android:hint="@string/hint_book_author"
        android:inputType="textAutoComplete"
        android:padding="@dimen/small_padding"
        android:layout_marginBottom="@dimen/big_padding"
        android:layout_marginTop="@dimen/big_padding"
        android:textSize="18sp" />

    <Button
        android:id="@+id/button_save"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:background="@color/buttonColor"
        android:text="@string/button_save" />

</LinearLayout>
```

20. W klasie *EditBookActivity* dodaj następujący kod do obsługi dodawania nowej książki:

```

public class EditBookActivity extends AppCompatActivity {

    public static final String EXTRA_EDIT_BOOK_TITLE = "pb.edu.pl.EDIT_BOOK_TITLE";
    public static final String EXTRA_EDIT_BOOK_AUTHOR = "pb.edu.pl.EDIT_BOOK_AUTHOR";

    private EditText editTitleEditText;
    private EditText editAuthorEditText;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_edit_book);

        editTitleEditText = findViewById(R.id.edit_book_title);
        editAuthorEditText = findViewById(R.id.edit_book_author);

        final Button button = findViewById(R.id.button_save);
        button.setOnClickListener(view -> {
            Intent replyIntent = new Intent();
            if (TextUtils.isEmpty(editTitleEditText.getText())
                || TextUtils.isEmpty(editAuthorEditText.getText())) {
                setResult(RESULT_CANCELED, replyIntent);
            } else {
                String title = editTitleEditText.getText().toString();
                replyIntent.putExtra(EXTRA_EDIT_BOOK_TITLE, title);
                String author = editAuthorEditText.getText().toString();
                replyIntent.putExtra(EXTRA_EDIT_BOOK_AUTHOR, author);
                setResult(RESULT_OK, replyIntent);
            }
            finish();
        });
    }
}

```

21. W klasie *MainActivity* zaimplementuj metodę *onActivityResult()*.

Najpierw dodaj na początku definicji klasy stałą pomocniczą z kodem wywołania nowej aktywności:

```
public static final int NEW_BOOK_ACTIVITY_REQUEST_CODE = 1;
```

Następnie dodaj poniższy kod do metody *onActivityResult()*. Zwróć uwagę na wywołanie powiadomienia typu *Snackbar*.

```

@Override
public void onActivityResult(int requestCode, int resultCode, Intent data) {
    super.onActivityResult(requestCode, resultCode, data);

    if (requestCode == NEW_BOOK_ACTIVITY_REQUEST_CODE && resultCode == RESULT_OK) {
        Book book = new Book(data.getStringExtra(EditBookActivity.EXTRA_EDIT_BOOK_TITLE),
            data.getStringExtra(EditBookActivity.EXTRA_EDIT_BOOK_AUTHOR));
        bookViewModel.insert(book);
        Snackbar.make(findViewById(R.id.main_layout), getString(R.string.book_added),
            Snackbar.LENGTH_LONG).show();
    } else {
        Snackbar.make(findViewById(R.id.main_layout),
            getString(R.string.empty_not_saved),
            Snackbar.LENGTH_LONG)
            .show();
    }
}
}

```

22. Dodaj również obsługę przycisku typu *Floating Action Button* w metodzie *onCreate()* klasy *MainActivity*:

```

FloatingActionButton addBookButton = findViewById(R.id.add_button);
addBookButton.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        Intent intent = new Intent( packageContext: MainActivity.this, EditBookActivity.class);
        startActivityForResult(intent, NEW_BOOK_ACTIVITY_REQUEST_CODE);
    }
});

```

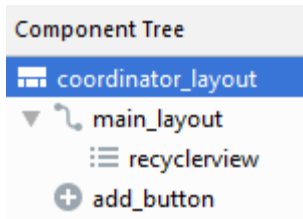
23. Uruchom aplikację. Co się dzieje z przyciskiem typu *Floating Action Button* w momencie wyświetlenia powiadomienia?
24. W celu poprawy działania przycisku w przypadku wyświetlania powiadomień, w pliku *activity_main.xml* dodaj *CoordinatorLayout* jako główny (nadrzędny) typ układu:

```

<androidx.coordinatorlayout.widget.CoordinatorLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:id="@+id/coordinator_layout">

```

Jednocześnie zadбай o to, że *Floating Action Button* będzie znajdował się bezpośrednio w układzie typu *CoordinatorLayout*, zaś *RecyclerView* w układzie zagnieżdżonym (*ConstraintLayout*):



Zmodyfikuj również definicję elementu *Floating Action Button*:

```
<com.google.android.material.floatingactionbutton.FloatingActionButton
    android:id="@+id/add_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    android:layout_margin="16dp"
    app:backgroundTint="@color/buttonColor"
    android:layout_gravity="bottom|end"
    android:src="@drawable/ic_add_black_24dp" />
```

Przy wywołaniu powiadomienia typu *Snackbar* zmień identyfikator widoku na *coordinator_layout*:

```
Snackbar.make(findViewById(R.id.coordinator_layout), getString(R.string.book_added),
    Snackbar.LENGTH_LONG).show();
```

25. Dodaj samodzielnie **obsługę edycji i usuwania książek** z listy:
- edycja** powinna odbywać się po **kliknięciu na element listy (click)**
 - usuwanie** powinno odbywać się po **dłuższym przyciśnięciu elementu listy (longclick)**
26. Dodaj do listy książek obsługę gestu „**swipe**” – wyświetl komunikat typu *Snackbar* z tekstem „Zarchiwizowano książkę”.

Podpowiedź:

- a. W celu dodania opcji edycji, uruchom aktywność *EditBookActivity* z innym *request code*, np.

```
public static final int EDIT_BOOK_ACTIVITY_REQUEST_CODE = 2;
```

- b. W drugiej aktywności (*EditBookActivity*) w metodzie *onCreate()* sprawdź, czy zostały przekazane jakieś wartości, np.:

```
if (getIntent().hasExtra(EXTRA_EDIT_BOOK_TITLE)) {
```

W ten sposób można będzie odróżnić, jakie wywołanie (dodanie książki, czy edycja) spowodowało wyświetlenie tej aktywności.

- c. Następnie przy pobieraniu zwracanego wyniku w metodzie *onActivityResult()* dodaj kod pobierający zaktualizowane dane książki, np.:

```
editedBook.setTitle(data.getStringExtra(EditBookActivity.EXTRA_EDIT_BOOK_TITLE));
editedBook.setAuthor(data.getStringExtra(EditBookActivity.EXTRA_EDIT_BOOK_AUTHOR));
```

Gdzie *editBook* jest zmienną pomocniczą, przechowującą instancję edytowanej książki.