Сучасні технології мобільного програмування

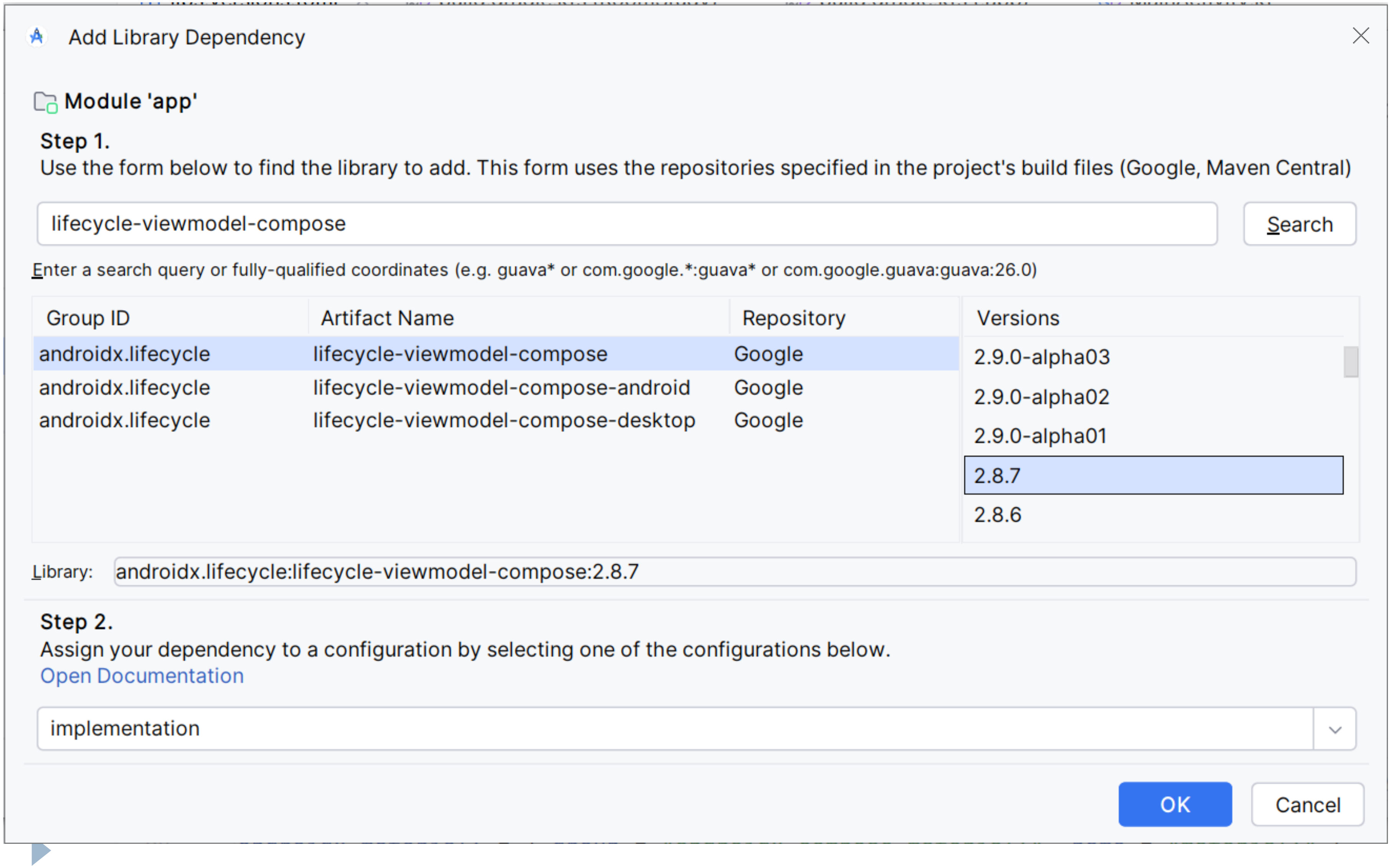
Room API in Compose

Слайди до лекцій

The Room library

- The **Room** library acts as an abstraction layer over **SQLite**, - embedding to the Android database - simplifying database management in Android applications. While direct usage of SQLite can lead to potential errors during SQL query execution, Room enhances safety and efficiency.
- It offers compile-time verification of SQL queries, significantly reducing the risk of errors.
- Also it offers convenience annotations that minimize repetitive and error-prone boilerplate code.
- It's generally recommended to utilize the Room library for local data storage in SQLite, unless there are specific reasons not to.

Add dependencies - lifecycle-viewmodel-compose



Add Library Dependency

Module 'app'

Step 1.
Use the form below to find the library to add. This form uses the repositories specified in the project's build files (Google, Maven Central)

lifecycle-viewmodel-compose

Enter a search query or fully-qualified coordinates (e.g. guava* or com.google.*:guava* or com.google.guava:guava:26.0)

Group ID	Artifact Name	Repository
androidx.lifecycle	lifecycle-viewmodel-compose	Google
androidx.lifecycle	lifecycle-viewmodel-compose-android	Google
androidx.lifecycle	lifecycle-viewmodel-compose-desktop	Google

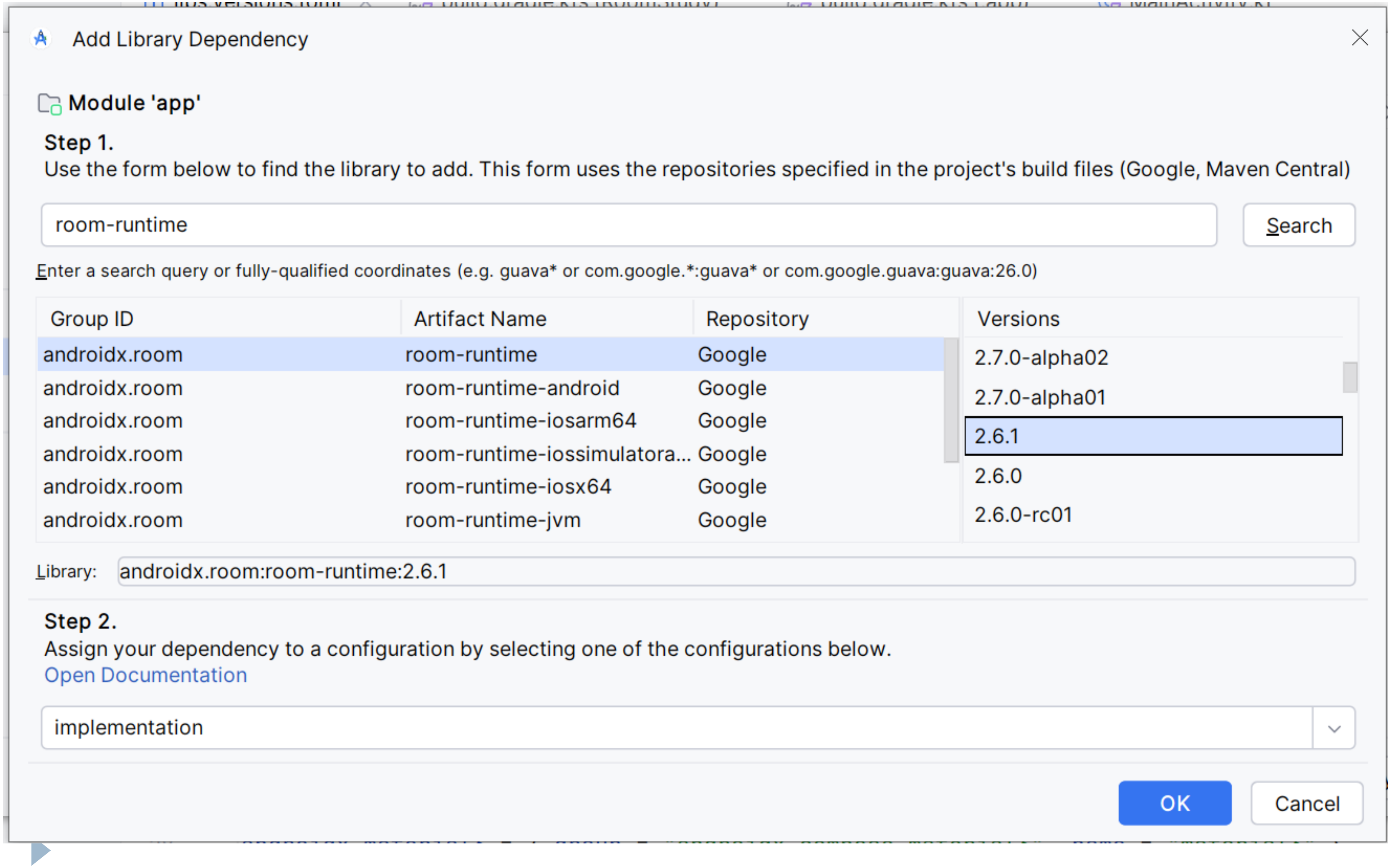
Versions
2.9.0-alpha03
2.9.0-alpha02
2.9.0-alpha01
2.8.7
2.8.6

Library: androidx.lifecycle:lifecycle-viewmodel-compose:2.8.7

Step 2.
Assign your dependency to a configuration by selecting one of the configurations below.
Open Documentation

implementation

Add dependencies - room-runtime



Add Library Dependency

Module 'app'

Step 1.
Use the form below to find the library to add. This form uses the repositories specified in the project's build files (Google, Maven Central)

room-runtime

Enter a search query or fully-qualified coordinates (e.g. guava* or com.google.*:guava* or com.google.guava:guava:26.0)

Group ID	Artifact Name	Repository
androidx.room	room-runtime	Google
androidx.room	room-runtime-android	Google
androidx.room	room-runtime-iosarm64	Google
androidx.room	room-runtime-iossimulatora...	Google
androidx.room	room-runtime-iosx64	Google
androidx.room	room-runtime-jvm	Google

Versions

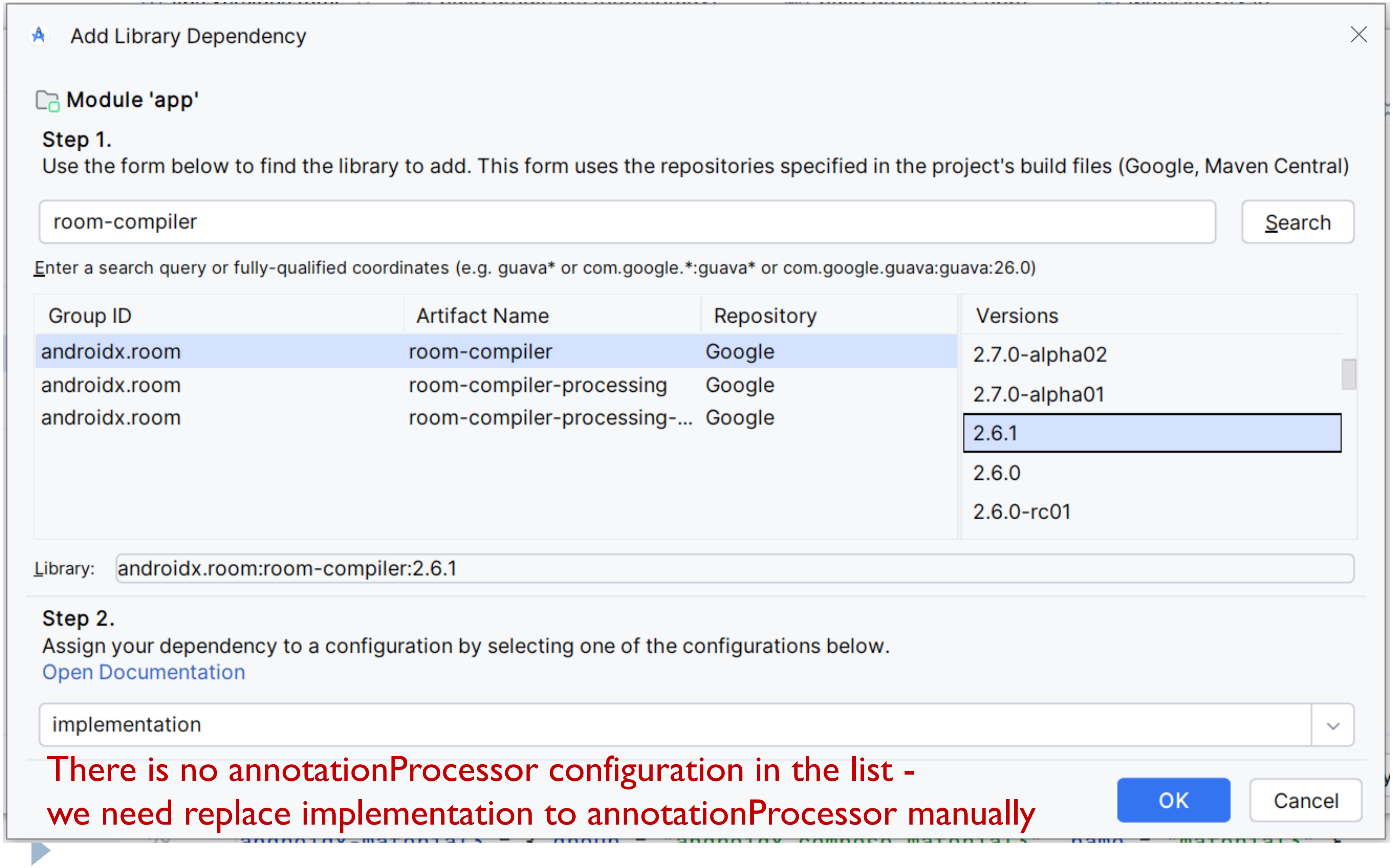
- 2.7.0-alpha02
- 2.7.0-alpha01
- 2.6.1
- 2.6.0
- 2.6.0-rc01

Library: androidx.room:room-runtime:2.6.1

Step 2.
Assign your dependency to a configuration by selecting one of the configurations below.
Open Documentation

implementation

Add dependencies - room-compiler



Add Library Dependency

Module 'app'

Step 1.
Use the form below to find the library to add. This form uses the repositories specified in the project's build files (Google, Maven Central)

room-compiler

Enter a search query or fully-qualified coordinates (e.g. guava* or com.google.*:guava* or com.google.guava:guava:26.0)

Group ID	Artifact Name	Repository
androidx.room	room-compiler	Google
androidx.room	room-compiler-processing	Google
androidx.room	room-compiler-processing-...	Google

Versions
2.7.0-alpha02
2.7.0-alpha01
2.6.1
2.6.0
2.6.0-rc01

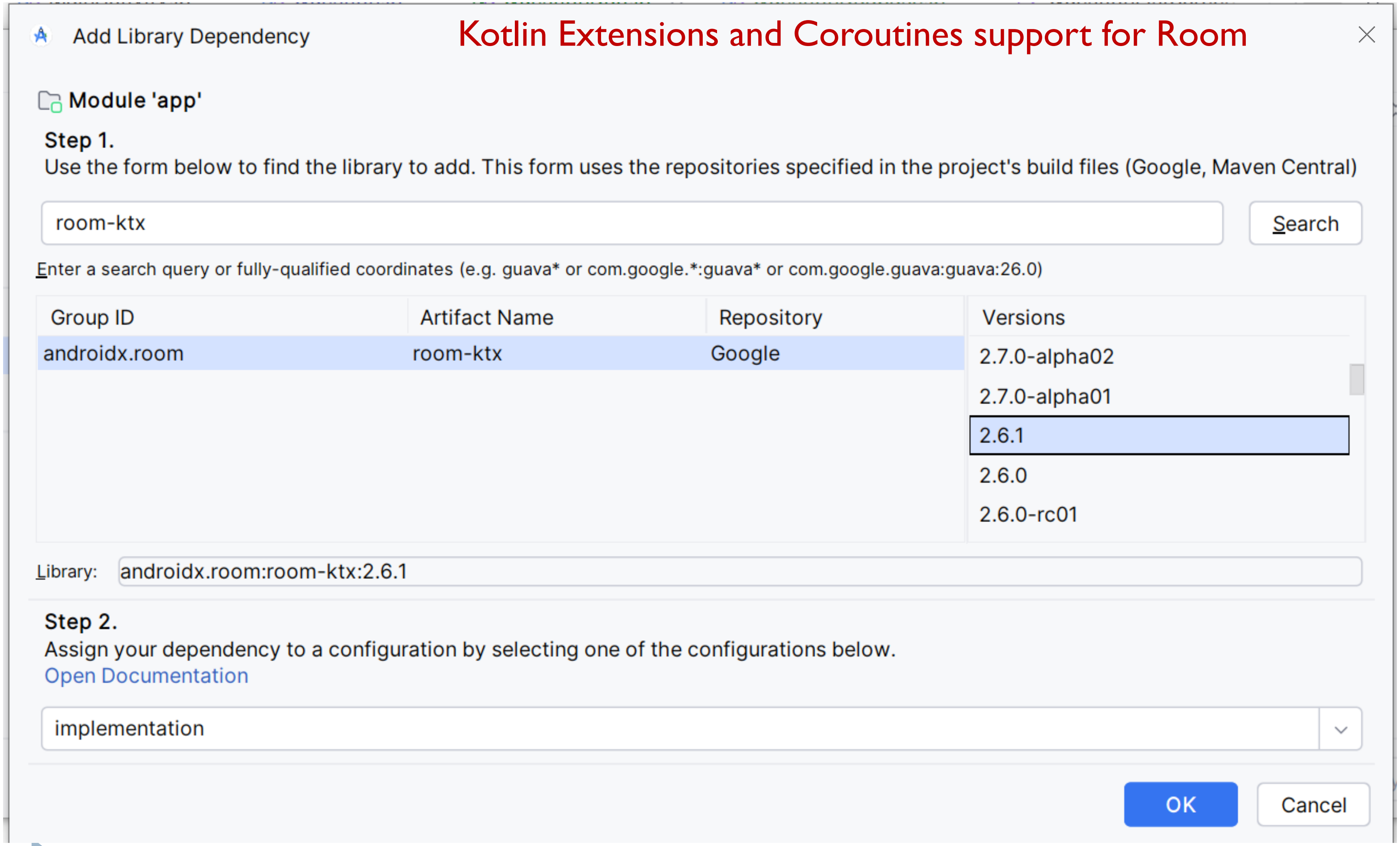
Library: androidx.room:room-compiler:2.6.1

Step 2.
Assign your dependency to a configuration by selecting one of the configurations below.
Open Documentation

implementation

There is no annotationProcessor configuration in the list -
we need replace implementation to annotationProcessor manually

Add dependencies - room-ktx



Add Library Dependency

Kotlin Extensions and Coroutines support for Room

Module 'app'

Step 1.
Use the form below to find the library to add. This form uses the repositories specified in the project's build files (Google, Maven Central)

room-ktx

Enter a search query or fully-qualified coordinates (e.g. guava* or com.google.*:guava* or com.google.guava:guava:26.0)

Group ID	Artifact Name	Repository
androidx.room	room-ktx	Google

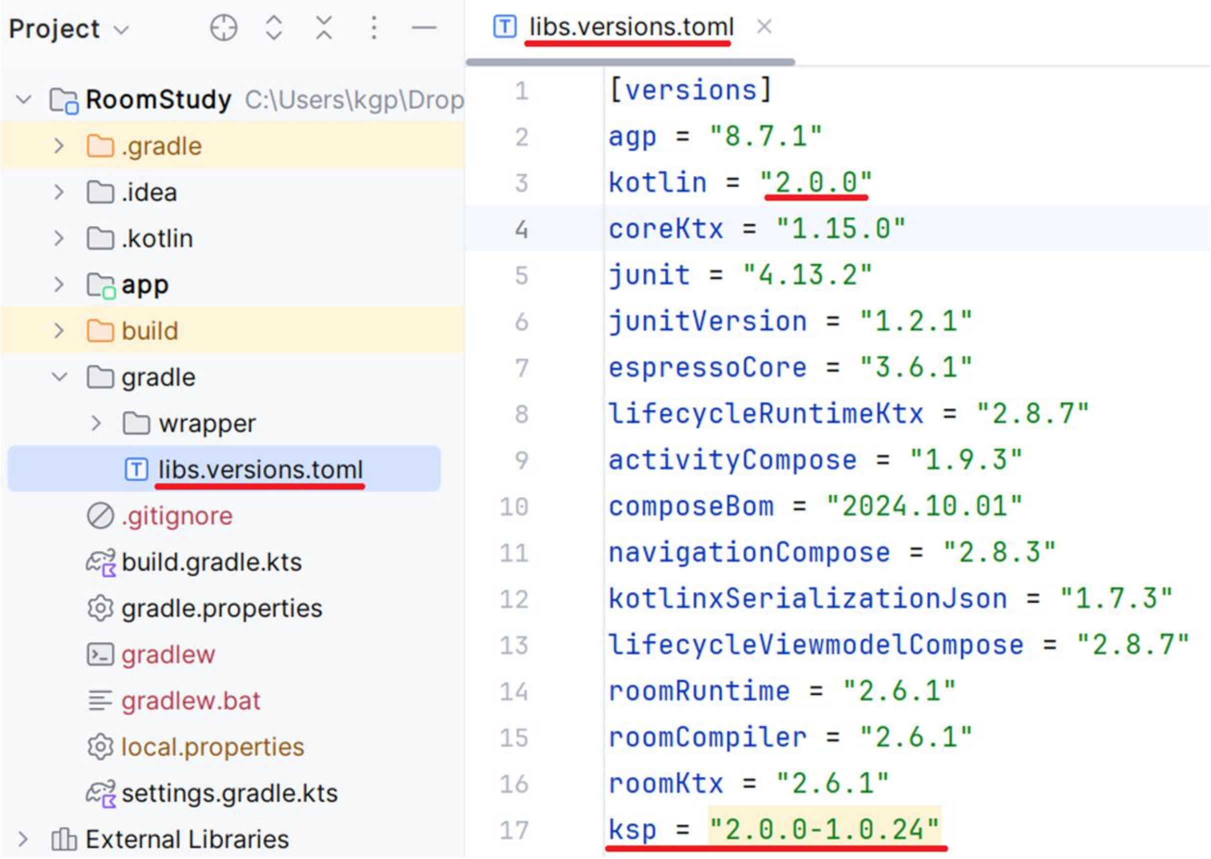
Versions:
- 2.7.0-alpha02
- 2.7.0-alpha01
- 2.6.1
- 2.6.0
- 2.6.0-rc01

Library: androidx.room:room-ktx:2.6.1

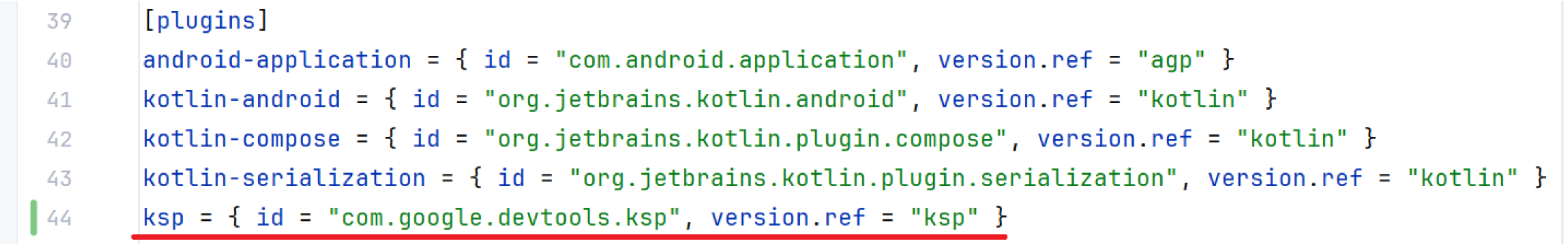
Step 2.
Assign your dependency to a configuration by selecting one of the configurations below.
Open Documentation

implementation

Add plugin com.google.devtools.ksp

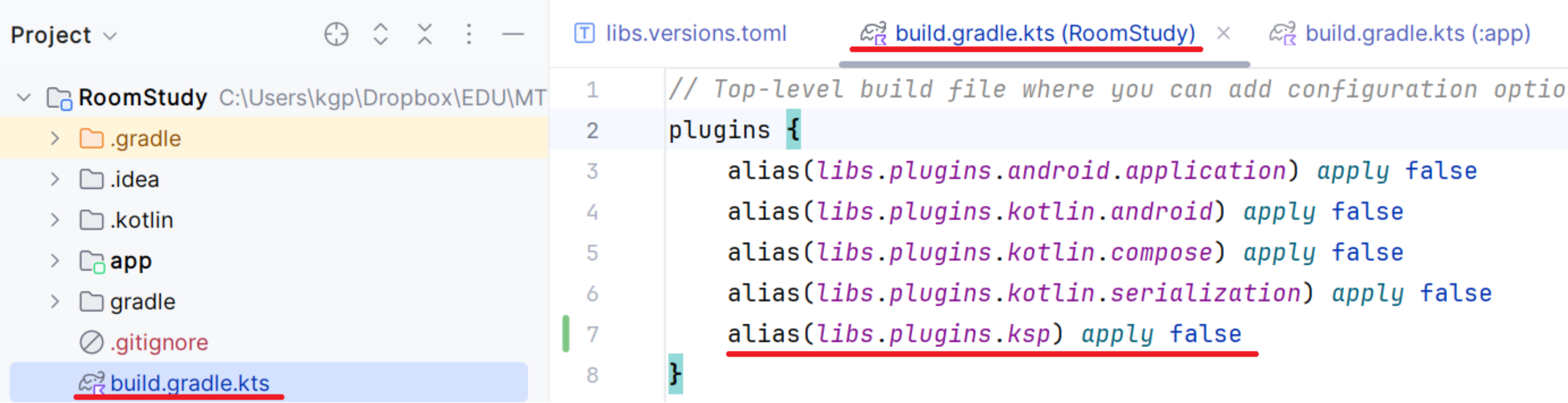


```toml
[versions]
agp = "8.7.1"
kotlin = "2.0.0"
coreKtx = "1.15.0"
junit = "4.13.2"
junitVersion = "1.2.1"
espressoCore = "3.6.1"
lifecycleRuntimeKtx = "2.8.7"
activityCompose = "1.9.3"
composeBom = "2024.10.01"
navigationCompose = "2.8.3"
kotlinxSerializationJson = "1.7.3"
lifecycleViewmodelCompose = "2.8.7"
roomRuntime = "2.6.1"
roomCompiler = "2.6.1"
roomKtx = "2.6.1"
ksp = "2.0.0-1.0.24"
```



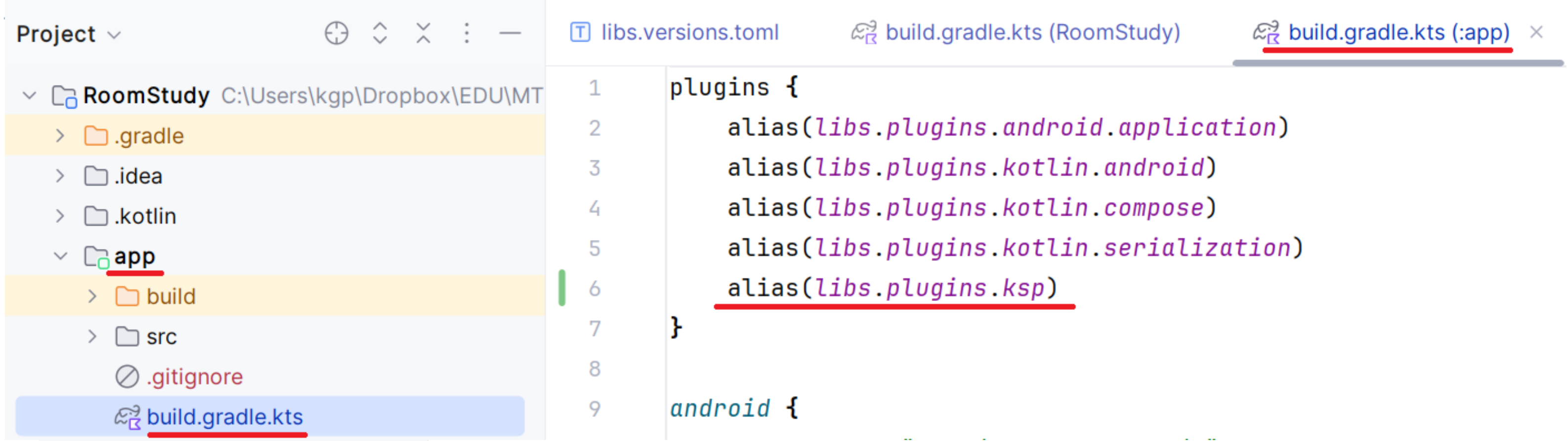
```toml
[plugins]
android-application = { id = "com.android.application", version.ref = "agp" }
kotlin-android = { id = "org.jetbrains.kotlin.android", version.ref = "kotlin" }
kotlin-compose = { id = "org.jetbrains.kotlin.plugin.compose", version.ref = "kotlin" }
kotlin-serialization = { id = "org.jetbrains.kotlin.plugin.serialization", version.ref = "kotlin" }
ksp = { id = "com.google.devtools.ksp", version.ref = "ksp" }
```

Add plugin com.google.devtools.ksp



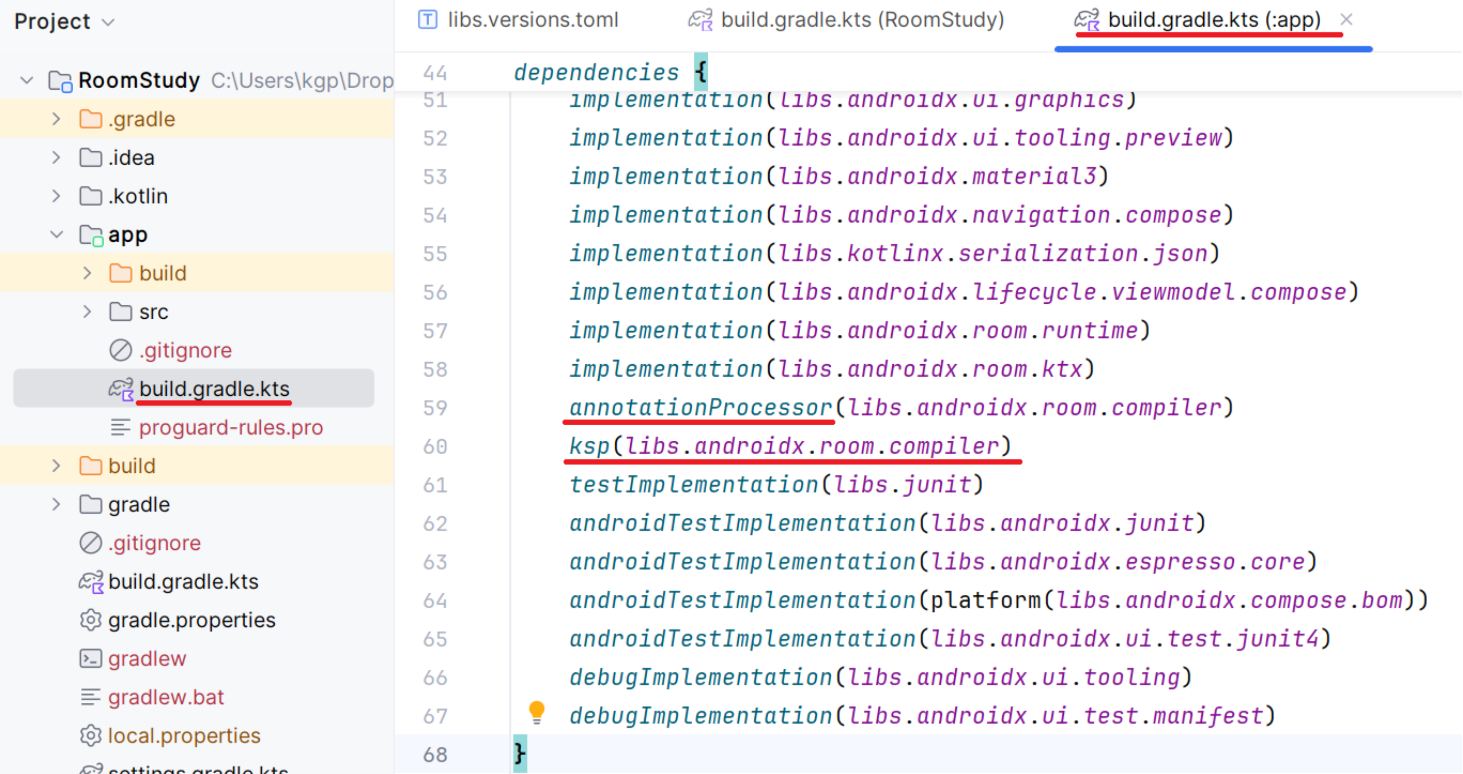
```kotlin
// Top-level build file where you can add configuration optio
plugins {
    alias(libs.plugins.android.application) apply false
    alias(libs.plugins.kotlin.android) apply false
    alias(libs.plugins.kotlin.compose) apply false
    alias(libs.plugins.kotlin.serialization) apply false
    alias(libs.plugins.ksp) apply false
}
```

Add plugin com.google.devtools.ksp



```
plugins {
    alias(libs.plugins.android.application)
    alias(libs.plugins.kotlin.android)
    alias(libs.plugins.kotlin.compose)
    alias(libs.plugins.kotlin.serialization)
    alias(libs.plugins.ksp)
}

android {
```



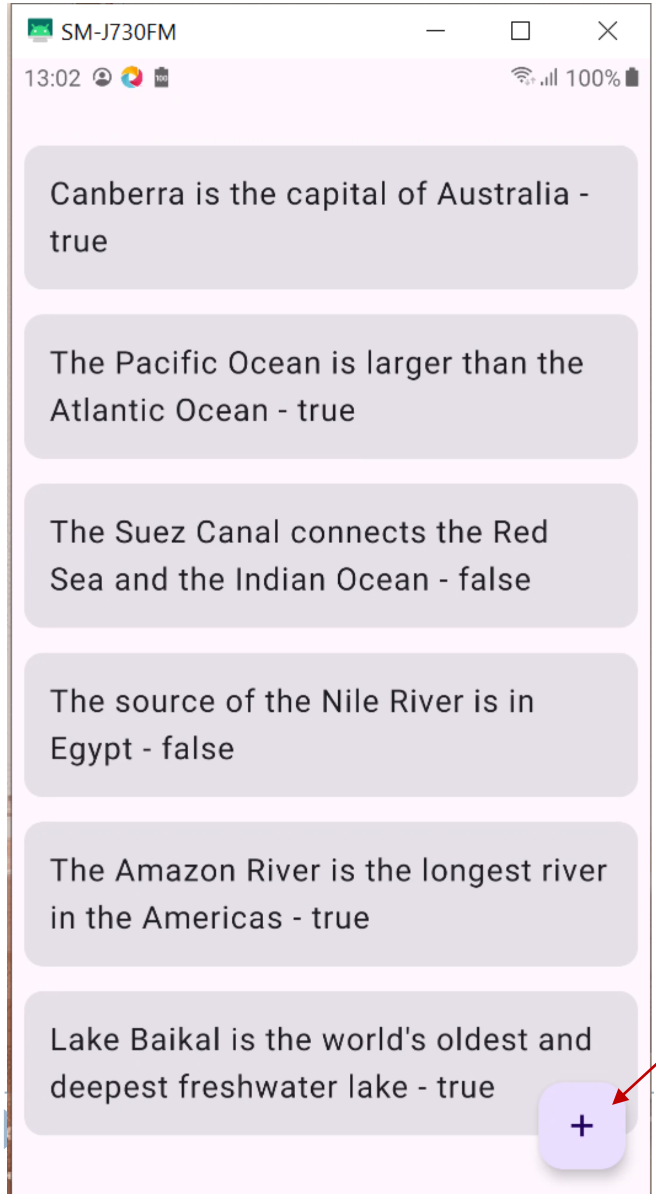
```
dependencies {
    implementation(libs.androidx.ui.graphics)
    implementation(libs.androidx.ui.tooling.preview)
    implementation(libs.androidx.material3)
    implementation(libs.androidx.navigation.compose)
    implementation(libs.kotlinx.serialization.json)
    implementation(libs.androidx.lifecycle.viewmodel.compose)
    implementation(libs.androidx.room.runtime)
    implementation(libs.androidx.room.ktx)
    annotationProcessor(libs.androidx.room.compiler)
    ksp(libs.androidx.room.compiler)
    testImplementation(libs.junit)
    androidTestImplementation(libs.androidx.junit)
    androidTestImplementation(libs.androidx.espresso.core)
    androidTestImplementation(platform(libs.androidx.compose.bom))
    androidTestImplementation(libs.androidx.ui.test.junit4)
    debugImplementation(libs.androidx.ui.tooling)
    debugImplementation(libs.androidx.ui.test.manifest)
}
```

App Overview

- The "list-detail" type app was choose. List screen contains questions from geographic area and has button to add a new question.
- Tap on the list question open detail screen, where user can edit question properties or delete the question.
- Question properties include question text and boolean type right answer.
- App uses type safe compose navigation to pass question from the list screen to the detail screen.

See data.Question.kt & data.QuestionRepository.kt

App Overview



Canberra is the capital of Australia - true

The Pacific Ocean is larger than the Atlantic Ocean - true

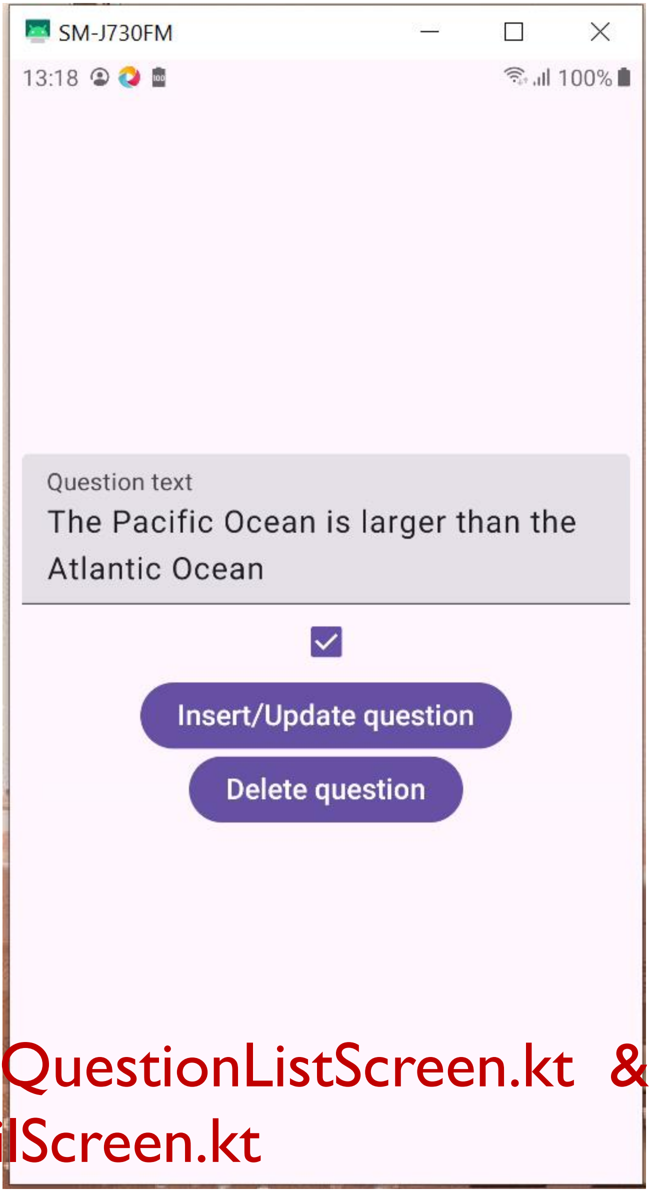
The Suez Canal connects the Red Sea and the Indian Ocean - false

The source of the Nile River is in Egypt - false

The Amazon River is the longest river in the Americas - true

Lake Baikal is the world's oldest and deepest freshwater lake - true

Pass question



Question text
The Pacific Ocean is larger than the Atlantic Ocean

☑

FAB

See ui.screens.QuestionListScreen.kt & QuestionDetailScreen.kt

Defining the Entity

- As we begin implementing the Room library, let's first set up our data structure by adding annotation to data.Question members:
- The @Entity annotation is used to denote a Room entity. This annotation requires a table name, which is set to "questions" in our example.
- The @PrimaryKey annotation marks a column as the primary key. Setting autoGenerate = true means that Room will automatically generate unique IDs for each entry.
- Use the @ColumnInfo annotation to specify a custom column name. Here, the column for storing question text is named "question."
- We can omit @ColumnInfo annotation if column name should be equal the class property name.
- We use not-null types as the class field type, so we must define values of the types for class instances.

See data.Question.kt

Defining the DAO

- The next step is to define the Data Access Object (DAO) - is a pattern you can use to separate the persistence layer from the rest of the application by providing an abstract interface. Through the DAO, we can simplify database operations.
- For the sample app, we require such fundamental operations (Room provides correspondent annotations for the Dao interface functions):
  1. Inserting new question
  2. Update existing question
  3. Deleting existing question
  4. Retrieving all questions
- Room API provides @Dao annotation.

See data.QuestionDao.kt

Defining the DAO - cont.

- The insert and update operations can be combined by upsert operation.

```
@Upsert
suspend fun upsertQuestion(question: Question)
```

- The upsert and delete functions are suspend - to ensure they're executed asynchronously, respecting coroutine best practices

```
@Delete
suspend fun deleteQuestion(question: Question)
```

- The getQuestions() method returns a Flow type to make list of question observable. This does not require the suspend modifier because it provides a continuous stream of data.

```
@Query("SELECT * FROM questions")
fun getQuestions(): Flow<List<Question>>
```

See data.QuestionDao.kt

Defining the Database

- The next step is to define the database class that uses Entity and DAO. Room API provides @Database annotation.
- We have to define this annotation parameters: **entities** that contains Entity classes, database **version** and **exportSchema** flag. Whenever you change the schema of the database table, you have to increase the version number. Set **exportSchema** to false so as not to keep schema version history backups .

```
@Database(entities = [Question::class], version = 1, exportSchema = false )
```

- Our database class extends RoomDatabase class and is abstract. Room takes care of its implementation:

```
abstract class QuestionDatabase : RoomDatabase()
```

- The questionDao() method exposes the Dao, enabling database operations through it (Room generates the implementation):

```
abstract fun questionDao(): QuestionDao
```

See data.QuestionDatabase.kt

Defining the Database - cont.

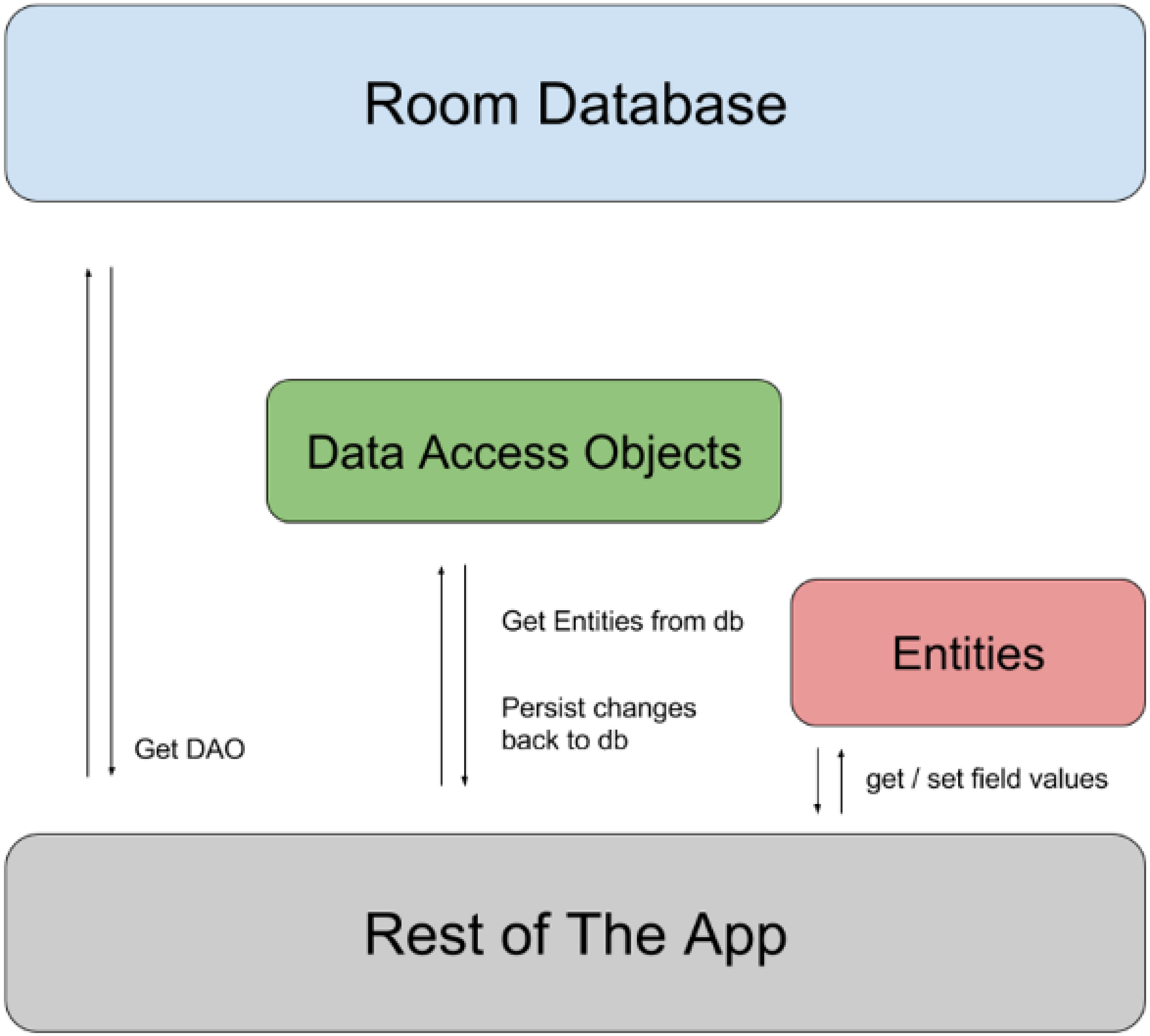
- The Instance variable, declared for the database within a companion object, ensures that QuestionDatabase adheres to the singleton pattern.
- Marking Instance with @Volatile guarantees that its value is always read from and written to the main memory, avoiding caching issues:

```
companion object {
    @Volatile
    private var Instance: QuestionDatabase? = null  ...
```

- Multiple threads can potentially ask for a database instance at the same time, which results in two databases instead of one (race condition). Wrapping the code to get the database inside a *synchronized* block prevents such issue.
- Use Room.databaseBuilder to create your ("question_database") database only if it doesn't exist. Otherwise, return the existing database.
- After build(), add an also block and assign Instance = **it** to keep a reference to the recently created database instance.

See data.QuestionDatabase.kt

The Room API components interaction



Defining the Repository

- The QuestionRepository class takes QuestionDao as parameter and implements Dao functions. This class will serve as an intermediary between our database operations defined in the DAO and the UI or business logic of our application.

  ```
  class QuestionRepository(private val questionDao: QuestionDao) {...}
  ```

- In the QuestionRepository, we define methods that correspond to the DAO's operations.
- Notice the us of suspend for upsertQuestion and deleteQuestion to support coroutines for asynchronous operations.

See data.QuestionRepository.kt

Defining the Container class

- To instantiate QuestionRepository, we require an instance of QuestionDao. This dependency chain necessitates a structured approach to ensure that all components are correctly instantiated. We'll address this by introducing a container class - QuestionContainer, which will manage the instantiation of QuestionRepository.
- The QuestionContainer class uses a *lazy* delegate to ensure that QuestionRepository is instantiated only when needed, using the appropriate Dao obtained from QuestionDatabase.

```
val questionRepository by lazy {
QuestionRepository(QuestionDatabase.getQuestionDatabase(context).questionDao()) }
```

See data.QuestionContainer.kt

Defining the Application class

- To supply the necessary context for our QuestionContainer, we'll create a custom QuestionApplication class in the root of the project:

  See QuestionApplication.kt

- To ensure our Application class is recognized, modify the AndroidManifest.xml (add android:name attribute):

```
<application
   android:name=".QuestionApplication"
...
```

  See manifests.AndroidManifest.xml

- This configuration ensures that our custom application class is used, allowing us to access QuestionContainer across our application.
- We are now ready to integrate the Room database within our app’s architecture.

Defining ViewModel

- ViewModels interact with the database via the DAO and provide data to the UI.
- QuestionViewModel takes a QuestionRepository as a parameter:

```
class QuestionViewModel(
    private val questionRepository: QuestionRepository) : ViewModel() {...}
```

- QuestionViewModel utilizes methods from QuestionRepository to perform data operations ().
- To solve the dependency issue, we provide a Factory instance within the QuestionViewModel to ensure it's instantiated with the necessary repository.

A CreationExtras.Key to query an application in which ViewModel is being created

```
companion object {
  val Factory: ViewModelProvider.Factory = viewModelFactory {
    initializer {
      val application = (this[APPLICATION_KEY] as QuestionApplication)
      QuestionViewModel(application.questionContainer.questionRepository)
  }   } }
```

See viewmodel.QuestionViewModel.kt

Defining QuestionListScreen Composable

- The QuestionListScreen takes QuestionViewModel and onNavigate To QuestionUpsert callback.

```
@Composable
fun QuestionListScreen(
   viewModel: QuestionViewModel =
                          viewModel(factory = QuestionViewModel.Factory),
                onNavigateToQuestionUpsert: (Question) -> Unit
) { .. }
```

- We get List<Question> questionBank from the viewModel in several stages:

1. QuestionViewModel's getQuestions() function returns a Flow<List<Question>>.
2. From the FLow instance we call collectAsState() function that returns a State<List<Question>>.
3. Operator by is used to get the value of the State object - List<Question>.

See ui.screens.QuestionListScreen.kt

Defining QuestionListScreen - cont.

- We use Material's Scaffold Composable to define Floating Action Button.
- The FAB's onClick handler creates an empty Question and pass it to Navigation Route to QuestionDetailScreen.

```
 Scaffold(
 modifier = Modifier.fillMaxSize(),
 floatingActionButton = {
    FloatingActionButton(
       onClick = {
          val question = Question(text = "", answer = false)
          onNavigateToQuestionUpsert(question)
       } ) { … }
{ … }
```

- In the LazyColumn Composable we iterate questionBank list items as Card component instances that display question text and correct answer.

See ui.screens.QuestionListScreen.kt

Defining QuestionDetailScreen

- The QuestionDetailScreen takes passed Question instance, QuestionViewModel and onQuestionUpdate callback.

```
@Composable
fun QuestionDetailScreen(
    question: Question,
    viewModel: QuestionViewModel,
    onQuestionUpdate: () -> Unit
) { … }
```

- We define vals for the passed question text and correct answer. We need to make them mutable to be able to update them. We use remember function to make the fields remember their state between recompositions.

```
val questionText = remember { mutableStateOf(question.text) }
val checkedState = remember { mutableStateOf(question.answer) }
```

- The vals states values changed by TextField's onValueChanged and Checkbox's onCheckedChange handlers.

See ui.screens.QuestionDetailScreen.kt

Defining QuestionDetailScreen - cont.

- In the Insert/Update question Button onClick handler we create an updated Question as a copy of the passed with changed text and answer values. Also QuestionViewModel is updated with inserted/updated Question and onQuestionUpdate callback function is invoked (used for navigation).

```
Button(onClick = {
    val updatedQuestion = question.copy(
        text = questionText.value,
        answer = checkedState.value,
    )
    viewModel.upsertQuestion(updatedQuestion)
    onQuestionUpdate()
}) { ... }
```

- In the Delete question Button onClick handler we call QuestionViewModel's delete function with passed question as argument and also invoke onQuestionUpdate callback function .

See ui.screens.QuestionDetailScreen.kt

MainActivity

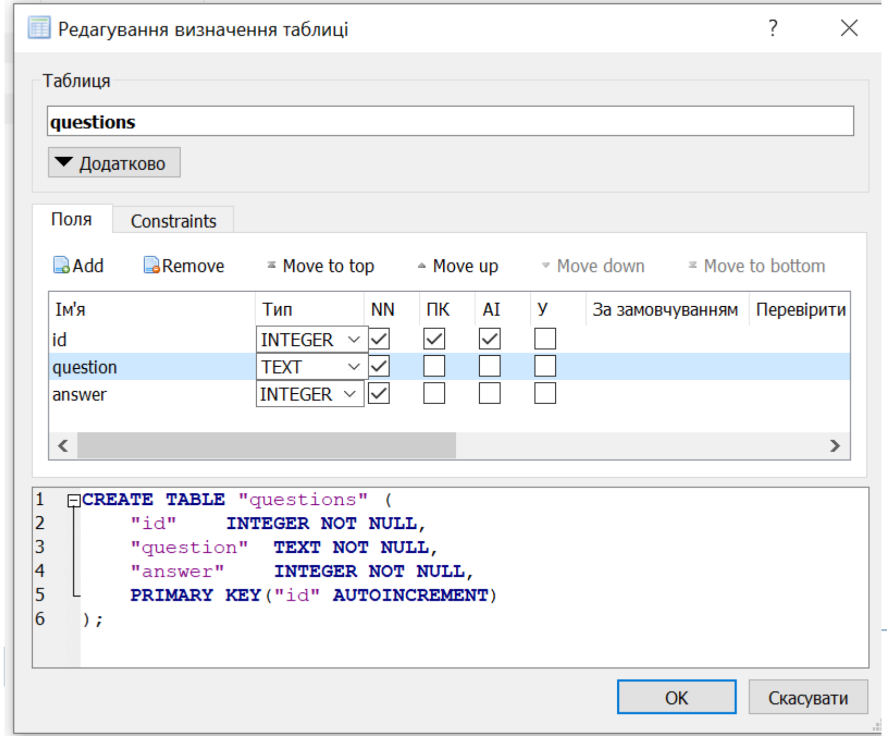
- MainActivity has defined QuestionViewModel in the Scaffold Composable, that passed to the QuestionListScreen and QuestionDetailScreen in the NavHost component.

See MainActivity.kt

Prepopulate database with Room

- Sometimes, you might want your app to start with a database that is already loaded with a specific set of data. This is called **prepopulating a database.** In Room 2.2.0 and higher, you can use API methods to prepopulate a Room database at initialization with contents from a prepackaged database file.

1. Create prepackaged database in **DB Browser for SQLite** and save it to file.



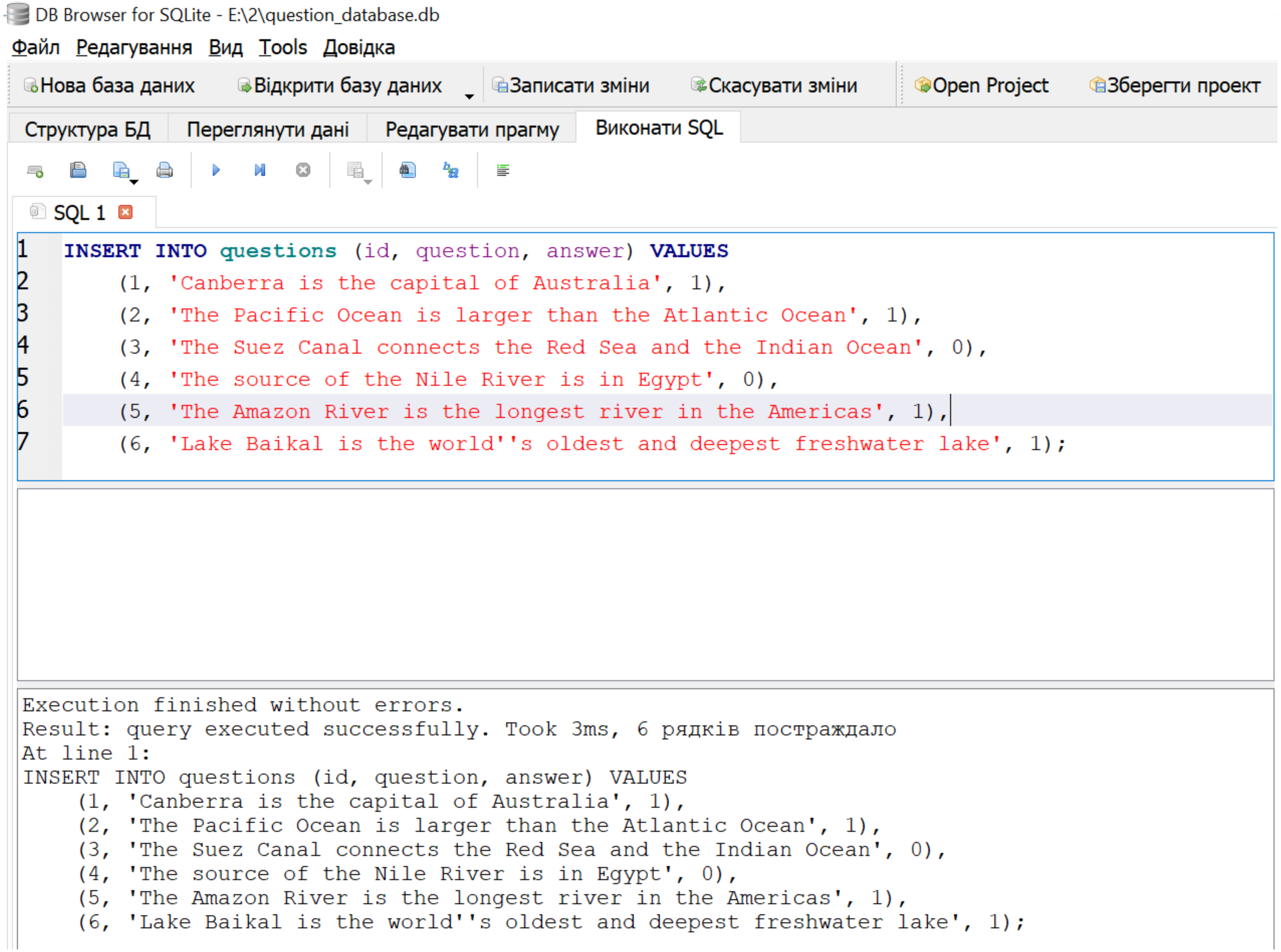
Таблиця: questions

Ім'я	Тип	NN	ПК	AI	У	За замовчуванням	Перевірити
id	INTEGER	☑	☑	☑	□		
question	TEXT	☑	□	□	□		
answer	INTEGER	☑	□	□	□		

```
CREATE TABLE "questions" (
	"id"	INTEGER NOT NULL,
	"question"	TEXT NOT NULL,
	"answer"	INTEGER NOT NULL,
	PRIMARY KEY("id" AUTOINCREMENT)
);
```

https://sqlitebrowser.org/

Prepopulate database with Room



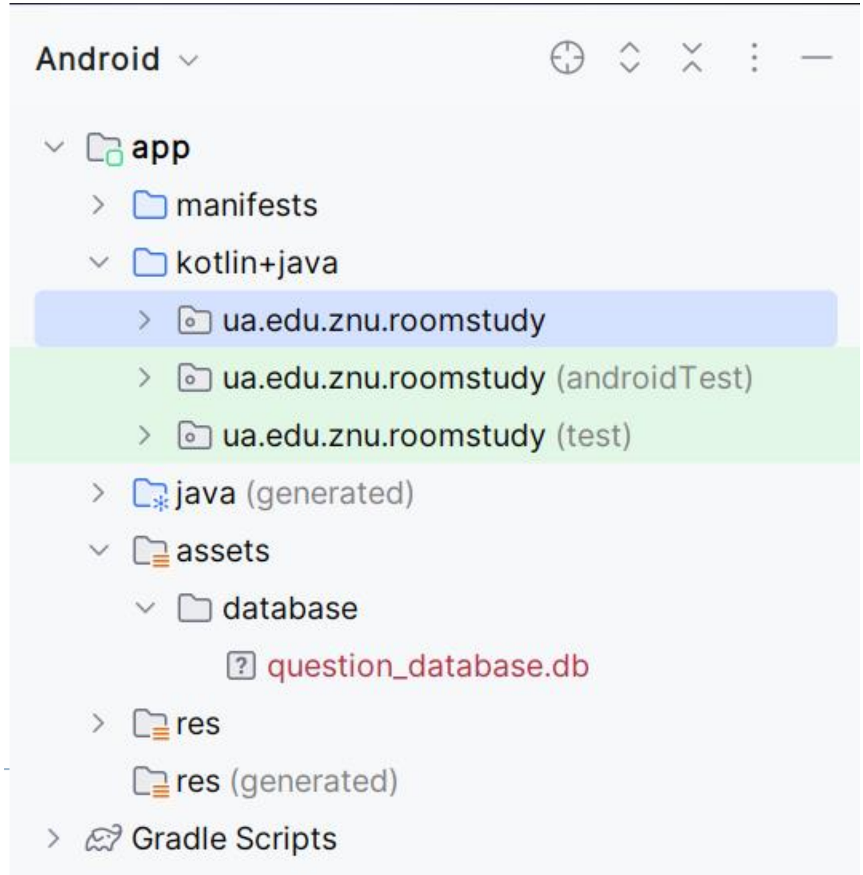
```sql
INSERT INTO questions (id, question, answer) VALUES
    (1, 'Canberra is the capital of Australia', 1),
    (2, 'The Pacific Ocean is larger than the Atlantic Ocean', 1),
    (3, 'The Suez Canal connects the Red Sea and the Indian Ocean', 0),
    (4, 'The source of the Nile River is in Egypt', 0),
    (5, 'The Amazon River is the longest river in the Americas', 1),
    (6, 'Lake Baikal is the world''s oldest and deepest freshwater lake', 1);
```

```
Execution finished without errors.
Result: query executed successfully. Took 3ms, 6 рядків постраждало
At line 1:
INSERT INTO questions (id, question, answer) VALUES
    (1, 'Canberra is the capital of Australia', 1),
    (2, 'The Pacific Ocean is larger than the Atlantic Ocean', 1),
    (3, 'The Suez Canal connects the Red Sea and the Indian Ocean', 0),
    (4, 'The source of the Nile River is in Egypt', 0),
    (5, 'The Amazon River is the longest river in the Americas', 1),
    (6, 'Lake Baikal is the world''s oldest and deepest freshwater lake', 1);
```

https://sqlitebrowser.org/

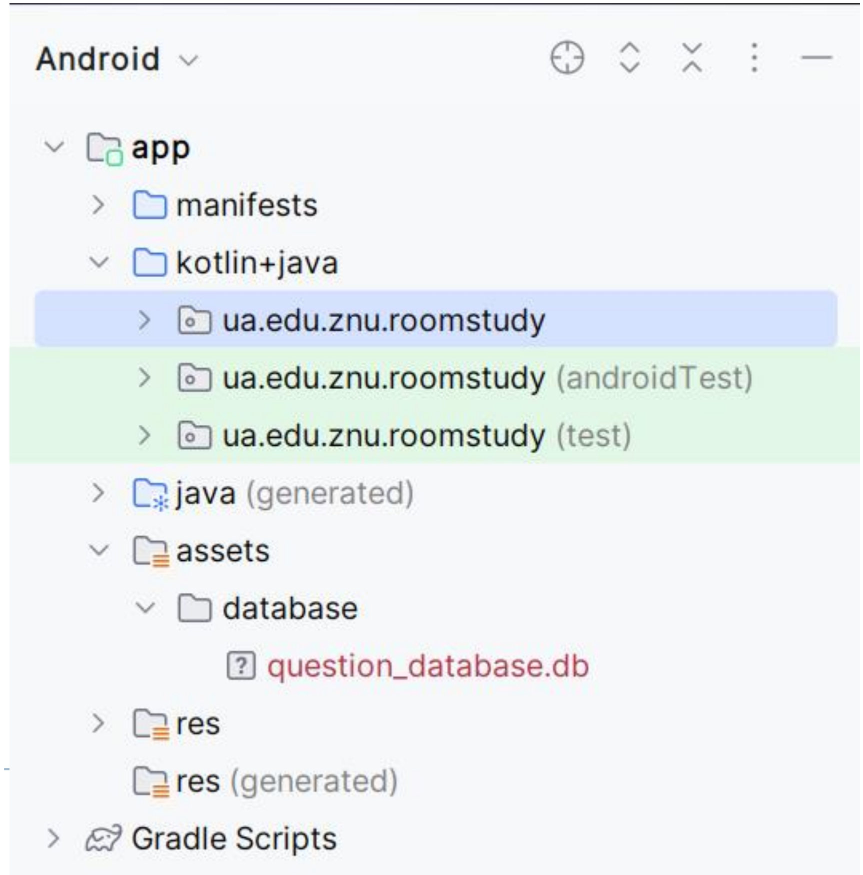
Prepopulate database with Room

2. Add assets folder to project File-New-Folder-Assets Folder with default settings.
3. Create database subdirectory in the assets folder.
4. Copy file with prepackaged database in this subdirectory.



Prepopulate database with Room

2. Add assets folder to project File-New-Folder-Assets Folder with default settings.
3. Create database subdirectory in the assets folder.
4. Copy file with prepackaged database in this subdirectory.



```
Android
app
  manifests
  kotlin+java
    ua.edu.znu.roomstudy
    ua.edu.znu.roomstudy (androidTest)
    ua.edu.znu.roomstudy (test)
  java (generated)
  assets
    database
      question_database.db
  res
  res (generated)
Gradle Scripts
```

Prepopulate database with Room

5. In the getQuestionDatabase(context: Context) function of the QuestionDatabase class add for RoomDatabase. Builder .createFromAsset("database/question_database.db") call:

```
fun getQuestionDatabase(context: Context): QuestionDatabase {
   return Instance ?: synchronized(this) {
      Room.databaseBuilder(
         context,
         QuestionDatabase::class.java,
         "question_database"
      )
         /* Uncomment this line to use a pre-populated database */
         .createFromAsset("database/question_database.db")
         .build()
         .also { Instance = it }
   }
}
```

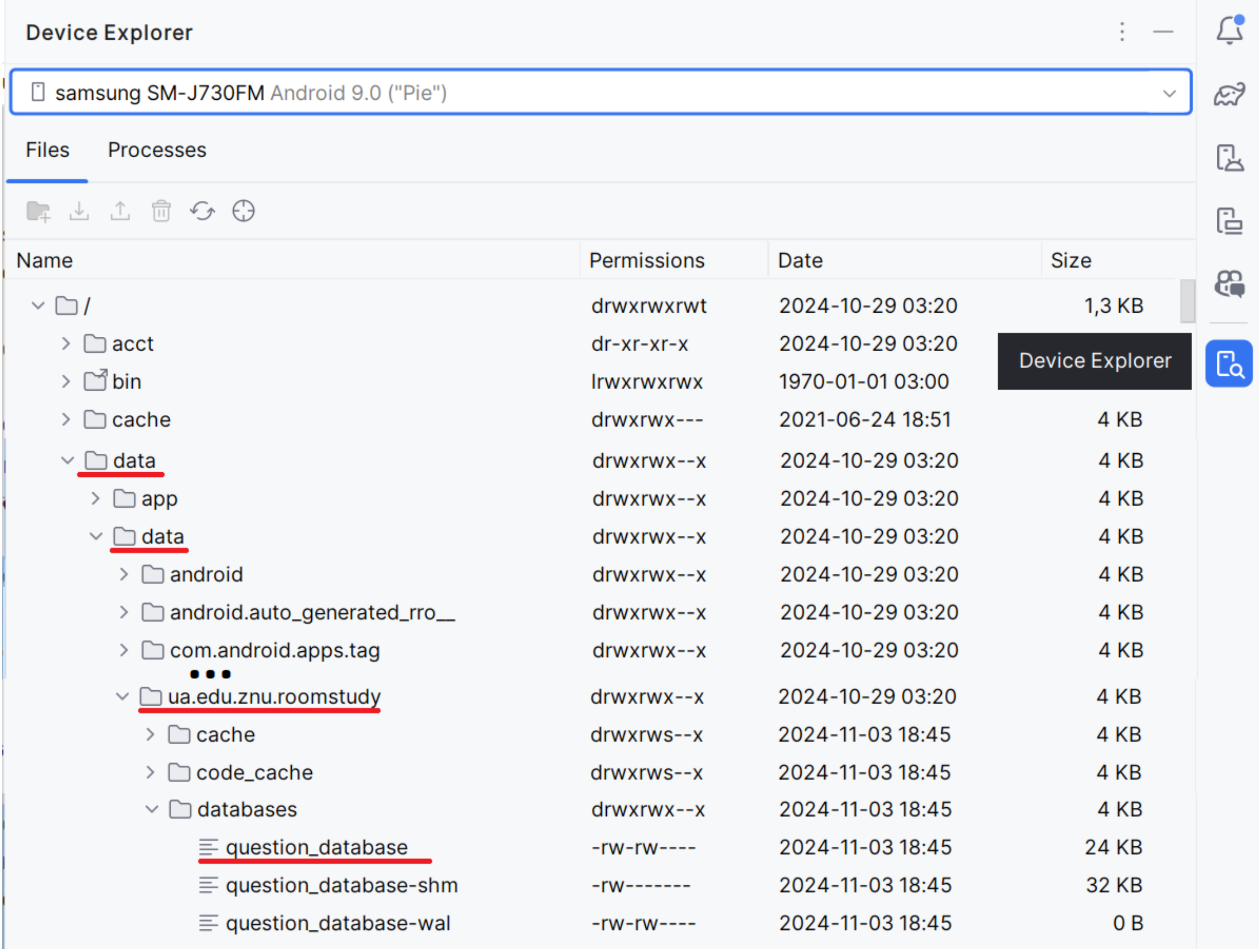
See data.QuestionDatabase.kt

Prepopulate database with Room

- To check this technology You should delete app data and remove app from the phone.
- You can check app database absence with Android Studio Device Explorer: the package with name of the app package must be missing in the device file structure in data/data folder.

Prepopulate database with Room

- Instead of creating prepackaged database in **DB Browser for SQLite** You can save populated database from the app package in data/data.



samsung SM-J730FM Android 9.0 ("Pie")

Name	Permissions	Date	Size
/	drwxrwxrwt	2024-10-29 03:20	1,3 KB
acct	dr-xr-xr-x	2024-10-29 03:20	
bin	lrwxrwxrwx	1970-01-01 03:00	
cache	drwxrwx---	2021-06-24 18:51	4 KB
data	drwxrwx--x	2024-10-29 03:20	4 KB
app	drwxrwx--x	2024-10-29 03:20	4 KB
data	drwxrwx--x	2024-10-29 03:20	4 KB
android	drwxrwx--x	2024-10-29 03:20	4 KB
android.auto_generated_rro__	drwxrwx--x	2024-10-29 03:20	4 KB
com.android.apps.tag	drwxrwx--x	2024-10-29 03:20	4 KB
•••			
ua.edu.znu.roomstudy	drwxrwx--x	2024-10-29 03:20	4 KB
cache	drwxrws--x	2024-11-03 18:45	4 KB
code_cache	drwxrws--x	2024-11-03 18:45	4 KB
databases	drwxrwx--x	2024-11-03 18:45	4 KB
question_database	-rw-rw----	2024-11-03 18:45	24 KB
question_database-shm	-rw-------	2024-11-03 18:45	32 KB
question_database-wal	-rw-rw----	2024-11-03 18:45	0 B