Сучасні технології мобільного програмування

Збереження даних застосунку засобами Room API

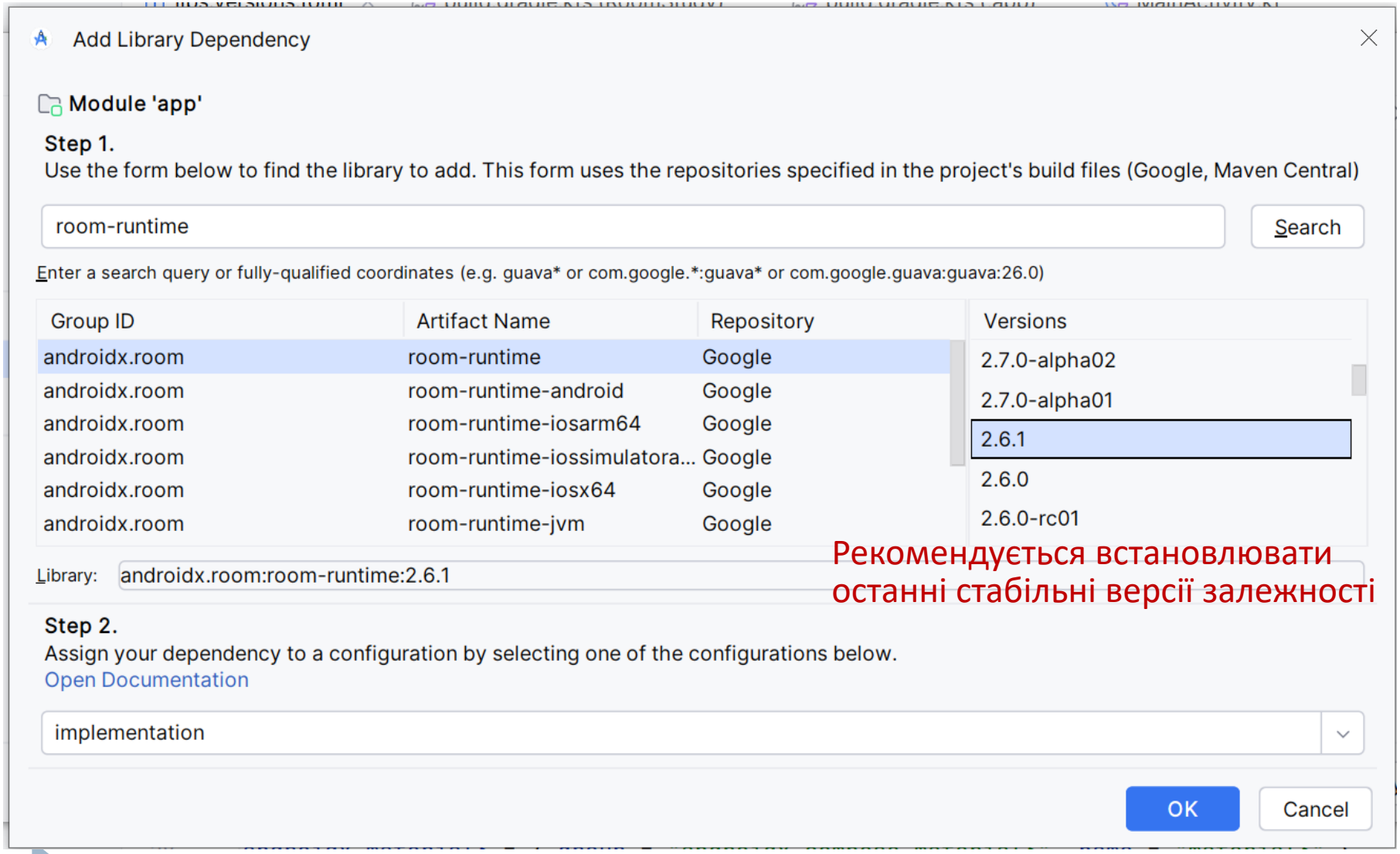
Слайди до лекцій

Програмна бібліотека Room

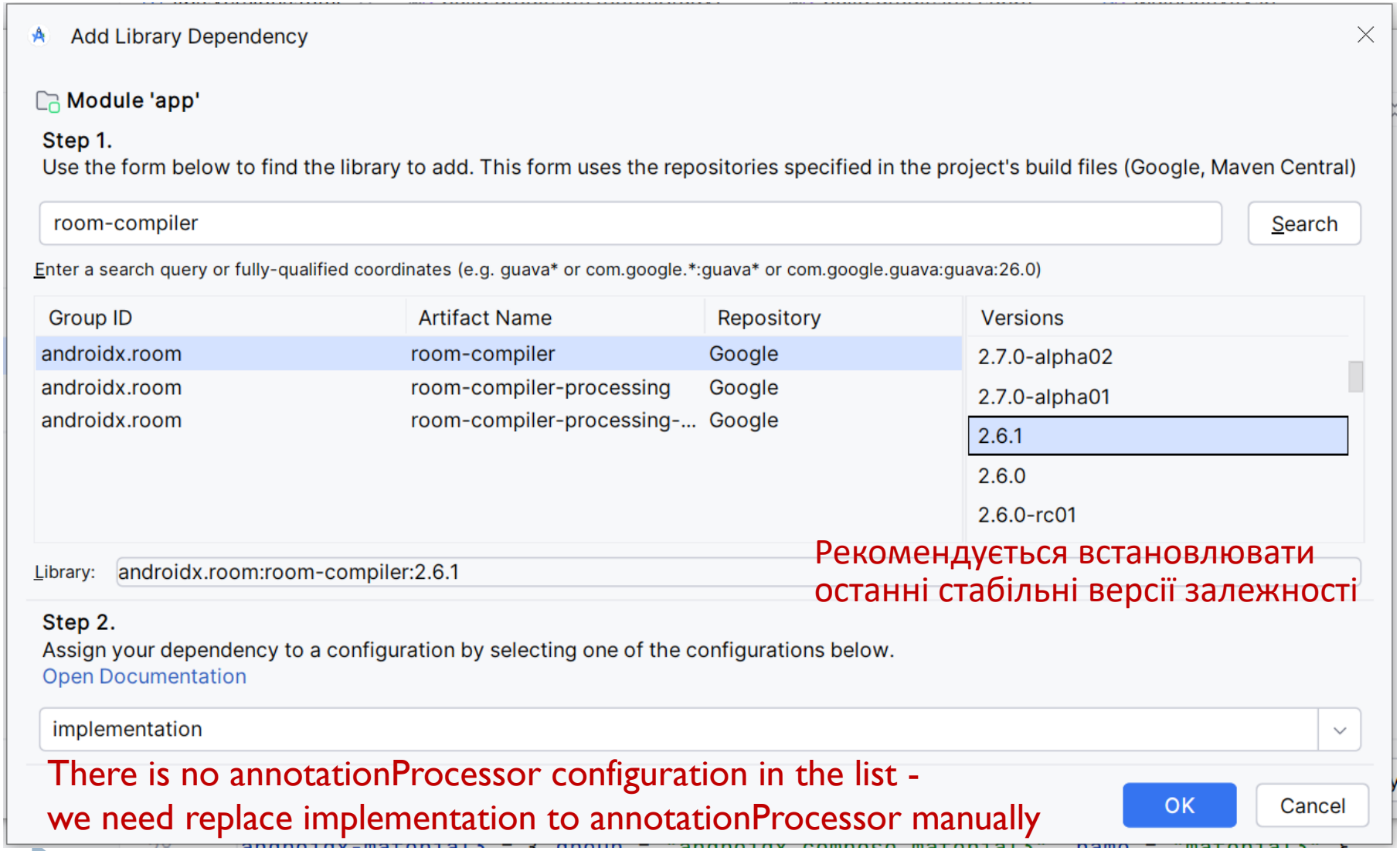
- Бібліотека Room діє як рівень абстракції над вбудованою до платформи Android СУБД SQLite, який спрощує керування базами даних у застосунках Android.
- Безпосереднє використання SQLite за допомогою SQL-запитів може призвести до потенційних помилок часу виконання, використання Room підвищує безпеку та ефективність за рахунок перевірки SQL-запитів під час компіляції.
- Також бібліотека пропонує зручні анотації, які мінімізують повторюваний та схильний до помилок шаблонний код.
- Зазвичай рекомендується використовувати бібліотеку Room для локального зберігання даних у SQLite, якщо немає конкретних причин її не використовувати.



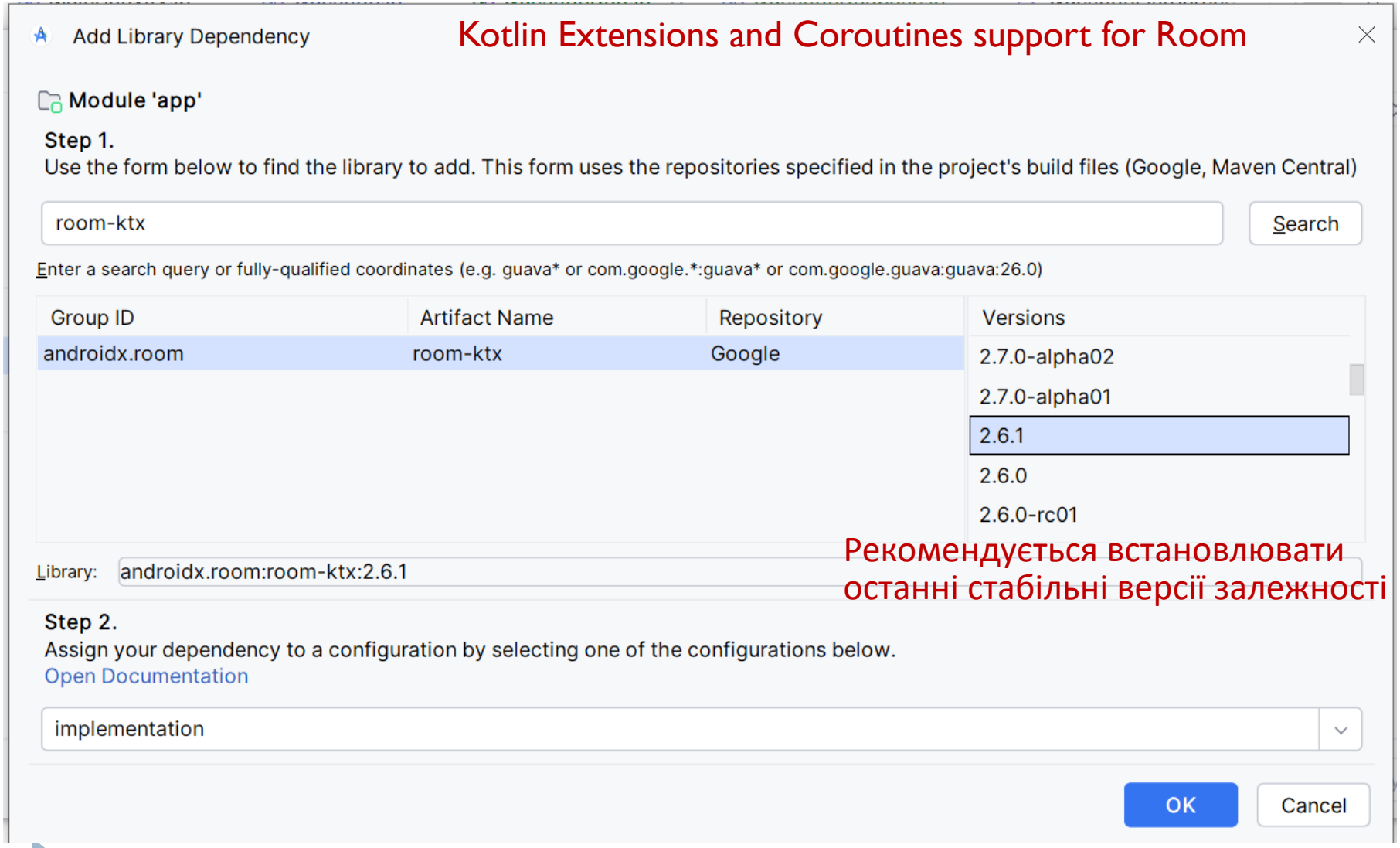
Додання залежностей - room-runtime



Додання залежностей - room-compiler

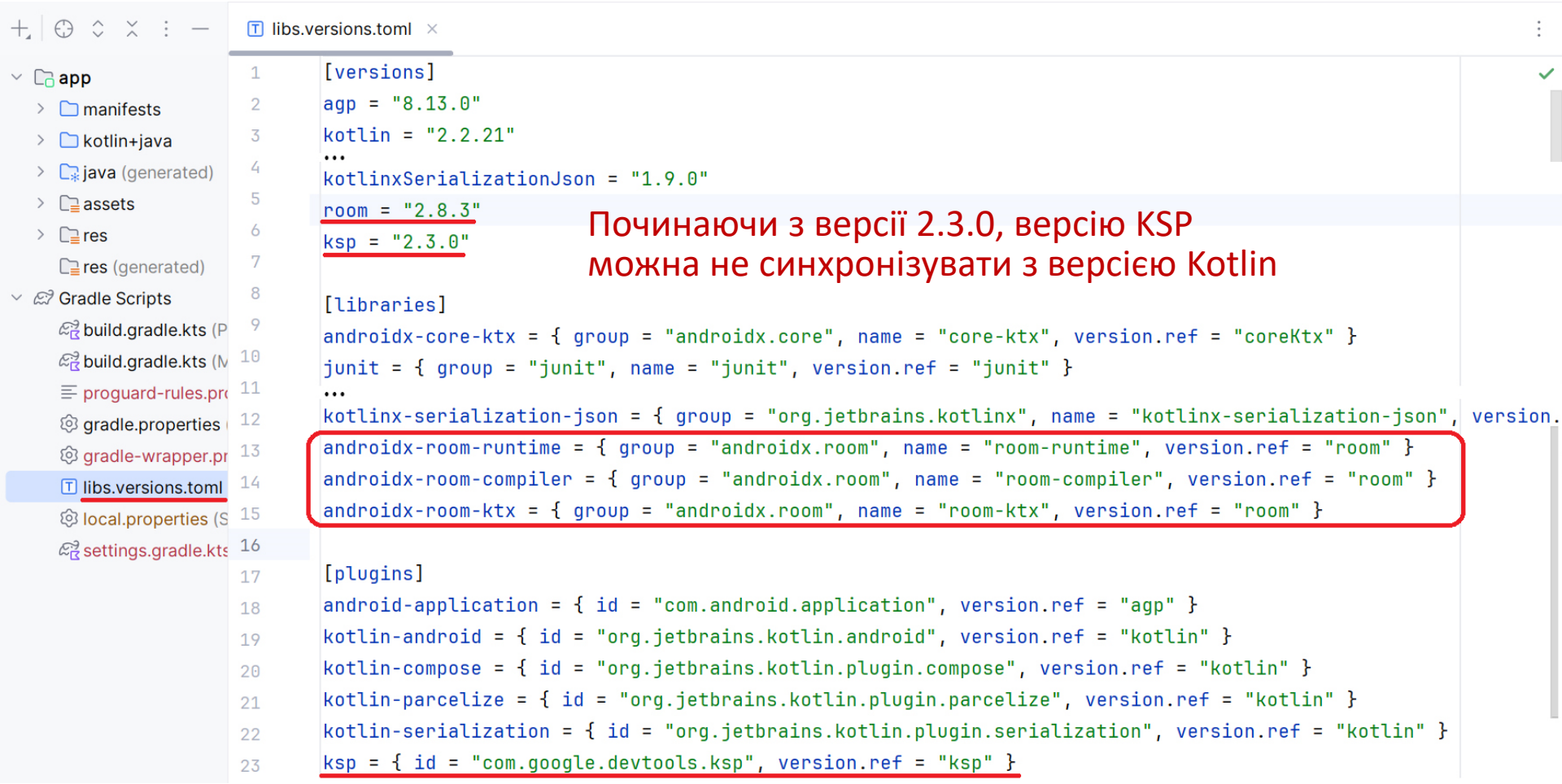


Додання залежностей - room-ktx



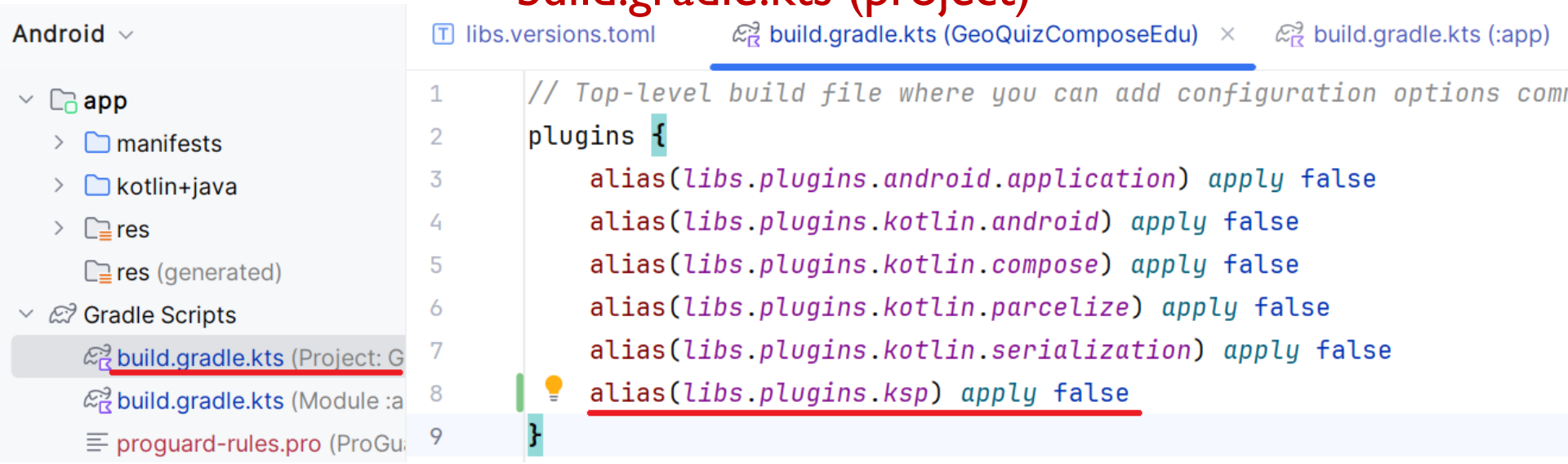
Додання плагіну KSP (Kotlin Symbol Processing)

libs.versions.toml



Додання плагіну KSP (Kotlin Symbl Processing)

build.gradle.kts (project)



build.gradle.kts (module)



Додання плагіну KSP (Kotlin Symbl Processing)

build.gradle.kts (module)

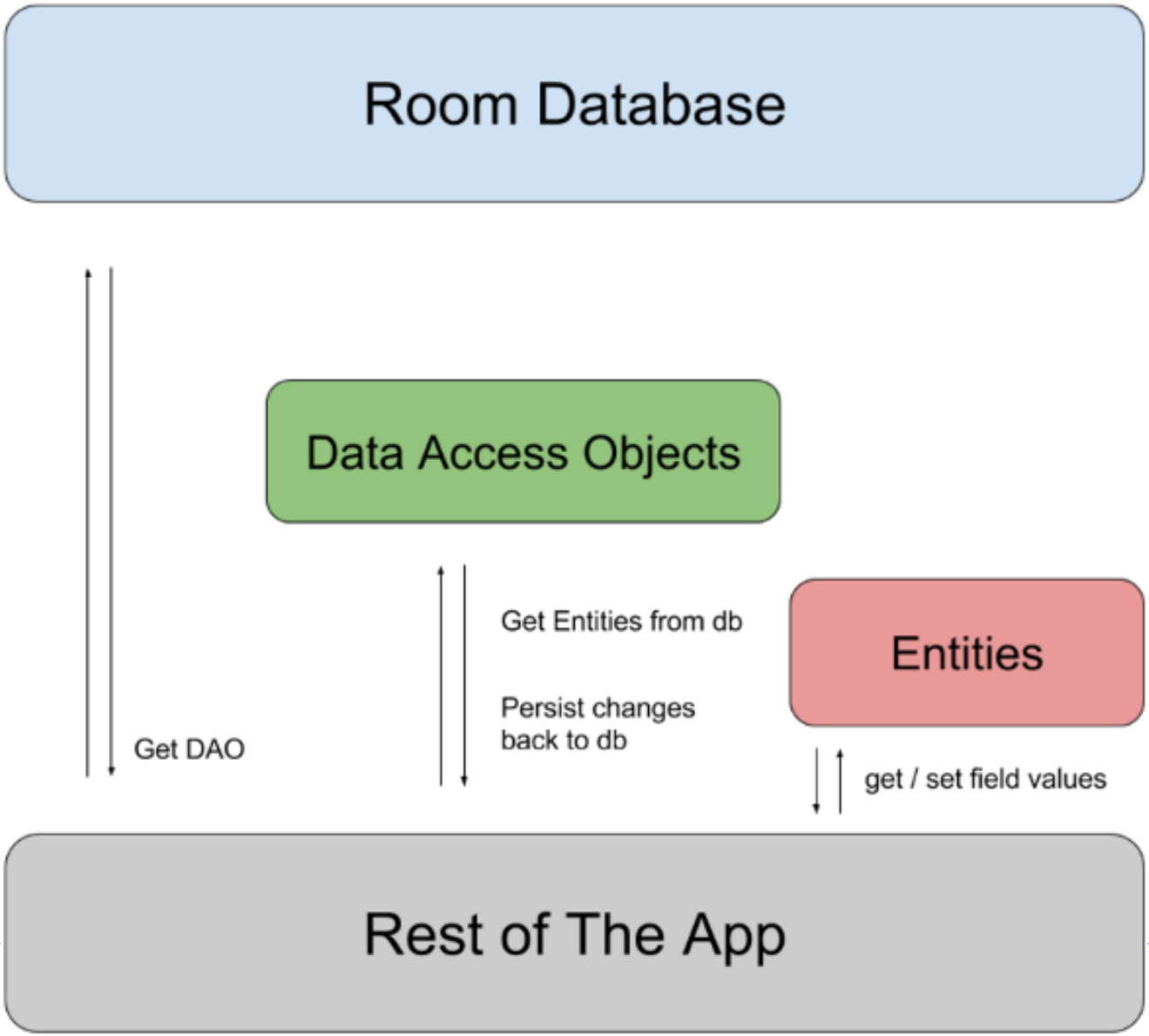


Основні компоненти Room

- **Сутності (Entities) -** класи даних, що відповідають таблицям в базі даних застосунку. Використовуються для отримання, оновлення даних, що зберігаються в рядках таблиць, та для створення і видалення нових рядків (CRUD - Create, Read, Update, Delete).
- **Об'єкти доступу до даних (Data Access Objects - DAO)** - інтерфейси, у яких оголошені абстрактні методи для отримання, оновлення, вставки та видалення даних у базі даних.
- **Клас Room Database** – це абстрактний клас, який успадковує бібліотечний клас `RoomDatabase` та є точкою входу для доступу до даних у вашій базі даних. Цей клас надає застосунку екземпляри DAO, пов'язані з базою даних.



Основні компоненти Room



Визначення класів-сутностей (Entity)

- Анотація **@Entity** використовується для позначення сутності Room. Властивість анотації **tableName** дозволяє визначати назву таблиці бази даних, що буде зберігати дані екземплярів класу-сутності (без її використання таблиця буде мати назву - ім'я класу).
- Анотація **@PrimaryKey** позначає стовпець з даними анотованого поля як первинний ключ таблиці. Властивість анотації **autoGenerate = true** конфігурує автоматичну генерацію унікальних первинних ключів для кожного запису.
- Анотація **@ColumnInfo** дозволяє визначати назву стовпця таблиці (без її використання назва стовпця буде співпадати з назвою поля класу).
- Анотація @Entity має властивість indicies, яка дозволяє визначати індекси для одного або групи полів та зазначати вимогу їх унікальності:

```
@Entity(indices = [Index(value = ["question"],
                         unique = true)])
```

Визначення класів-сутностей (Entity)

```
package ua.edu.znu.geoquizcomposeedu.data

import ...

@Entity(
    tableName = "questions",
    indices = [Index(value = ["question"], unique = true)]
)
@Serializable
@Parcelize
data class Question(
    @PrimaryKey(autoGenerate = true)
    val id: Int = 0,
    @ColumnInfo(name = "question")
    val questionText: String,
    val answer: Boolean
) : Parcelable
```

Now use database Int primary key for id instead of random string

@Serializable and @Parcelize with Parcelable are use by Navigation for Question passing



Оголошення DAO

- Об’єкт доступу до даних (Data Access Object - DAO) – інтерфейс, що містить абстрактні методи доступу до бази даних застосунка. Під час компіляції Room автоматично генерує реалізації визначених абстрактних методів.
- Існує два типи методів DAO, що визначають взаємодію з базою даних:
  - методи, які дозволяють вставляти, оновлювати та видаляти рядки у базі даних без написання будь-якого SQL-коду;
  - методи, які дозволяють писати SQL-запити для взаємодії з базою даних.
- API Room надає анотацію @Dao для маркування інтерфейсу з абстрактними методами та анотації @Insert, @Update, @Delete, @Query та анотацію @Upsert, яка є об'єднанням @Insert та @Update.
- Зазначимо, що Room перевіряє SQL-запити під час компіляції.



Оголошення DAO - анотації

- Анотація @Insert маркує методи, які вставляють свої аргументи у відповідну таблицю бази даних. Кожен аргумент методу має бути або екземпляром класу сутності, або колекцією екземплярів класу сутностей даних.
  Якщо маркований @Insert метод отримує один аргумент, він опціонально може повертати Long ідентифікатор (rowId) доданого рядка. Якщо аргумент є масивом або колекцією, метод може повертати масив або колекцію rowId доданих записів.
- Анотація @Update маркує методи, які оновлюють рядки у таблиці бази даних. Як і методи @Insert, методи @Update приймають аргументами екземпляри сутностей.
  Room використовує первинний ключ для зіставлення переданих екземплярів сутностей із рядками в базі даних. Якщо рядок з первинним ключем, що дорівнює первинному ключу аргумента у базі даних відсутній, Room не вносить змін.
  Метод @Update може опціонально повертати цілочисельне значення, що вказує на кількість успішно оновлених рядків.



Оголошення DAO - анотації

- Анотація @Delete маркує методи, які видаляють рядки, що відповідають аргументам методу, з таблиці бази даних. Як і методи @Insert, методи @Delete приймають екземпляри сутностей даних як аргументи. Room використовує первинний ключ для зіставлення екземплярів сутностей із рядками в базі даних. Якщо рядок з таким самим первинним ключем відсутній, Room не вносить змін. Метод @Delete може опціонально повертати цілочисельне значення, що вказує на кількість успішно видалених рядків.
- Анотація @Upsert, яка є об'єднанням @Insert та @Update
- Анотація @Query має властивість value, що приймає рядок з довільним SQL-запитом, який виконується при виклику методу, що маркується цією анотацією. Зазвичай ця анотація використовується для читання даних та для складних запитів вставки, оновлення та видалення. Рядок з SQL-запитом може мати параметр метода, який зазначається як :ім'я_параметра.

Оголошення DAO

```
package ua.edu.znu.geoquizcomposeedu.data

import ...

@Dao
interface QuestionDao {
    @Insert(onConflict = OnConflictStrategy.IGNORE)
    suspend fun addQuestion(question: Question)

    @Update
    suspend fun updateQuestion(question: Question)

    @Delete
    suspend fun removeQuestion(question: Question)

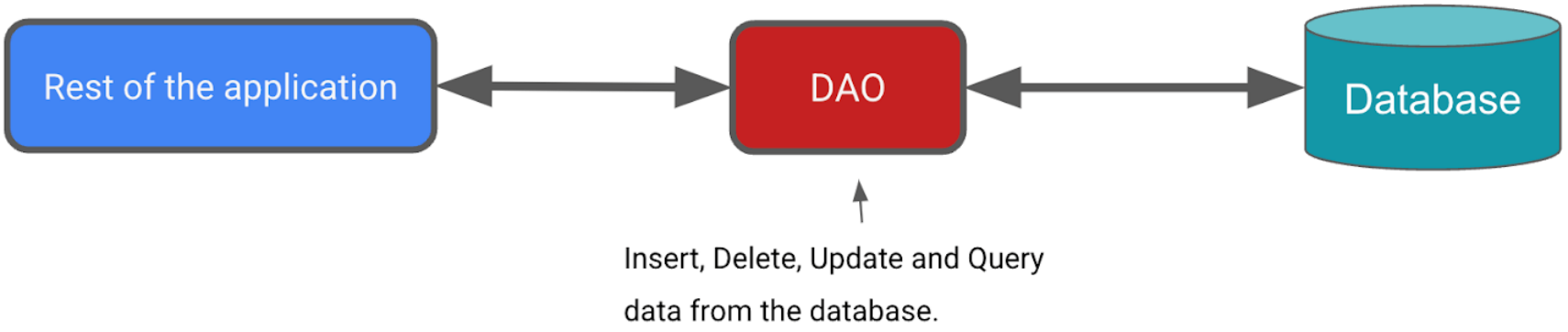
    @Query("SELECT * FROM questions")
    fun getQuestions(): Flow<List<Question>>

    @Query("SELECT * FROM questions WHERE id = :index LIMIT 1")
    fun getQuestionByIndex(index: Int): Flow<Question>
}
```

Функції insert, update та delete оголошуються suspend, щоб забезпечити їх асинхронне виконання, дотримуючись найкращих практик корутин.

Функції getQuestions та getQuestionByIndex повертають Flow, реалізуючи вбудовану у Room функцію автоматичного сповіщення про зміни у базі даних. Модифікатор suspend не потрібен, оскільки Flow забезпечує безперервний потік даних.

Оголошення DAO



За шляхом:
app\build\generated\ksp\debug\kolin\пакет_застосунку\data
можна подивитися код згенерованого класу QuestionDao_Impl



Визначення Database

- Для визначення класу бази даних, який використовує Entity та DAO Room API надає анотацію @Database. Необхідно визначити параметри цієї анотації: **entities** - містять класи Entity, **version** - містить версію бази даних та прапорець **exportSchema**, який рекомендується встановлювати у false, щоб не зберігати резервні копії історії версій схеми. Зауважте, щоразу, коли ви змінюєте схему таблиці бази даних, вам потрібно збільшити номер версії.
  @Database(entities = [Question::class], version = 1, exportSchema = false)
- Клас бази даних розширює клас RoomDatabase і є абстрактним. Room піклується про його реалізацію:
  abstract class QuestionDatabase : RoomDatabase()
- Абстрактний метод questionDao() надає доступ до Dao, дозволяючи операції з базою даних через нього (Room генерує реалізацію):

  abstract fun questionDao(): QuestionDao



Визначення Database

```
package ua.edu.znu.geoquizcomposeedu.data

import ...

@Database(entities = [Question::class], version = 1, exportSchema = false)
abstract class QuestionDatabase : RoomDatabase() {

    abstract fun questionDao(): QuestionDao

    companion object {

        @Volatile
        private var Instance: QuestionDatabase? = null

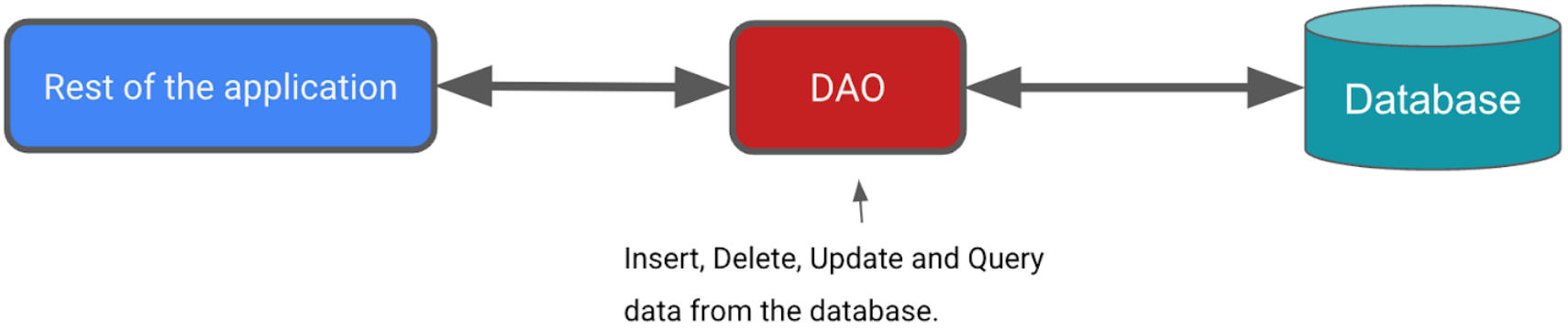
        fun getInstance(context: Context): QuestionDatabase {
            return Instance ?: synchronized(this) {
                Room.databaseBuilder(
                    context,
                    QuestionDatabase::class.java,
                    "question_database"
                )
                    .build()
                    .also { Instance = it }
            }
        }
    }
}
```

Безпечний для роботи з потоками сінглетон, який повертає єдиний екземпляр бази даних, побудований Bilder-ом.

гарантує, що змінна завжди зчитується з основної пам'яті та записується в неї

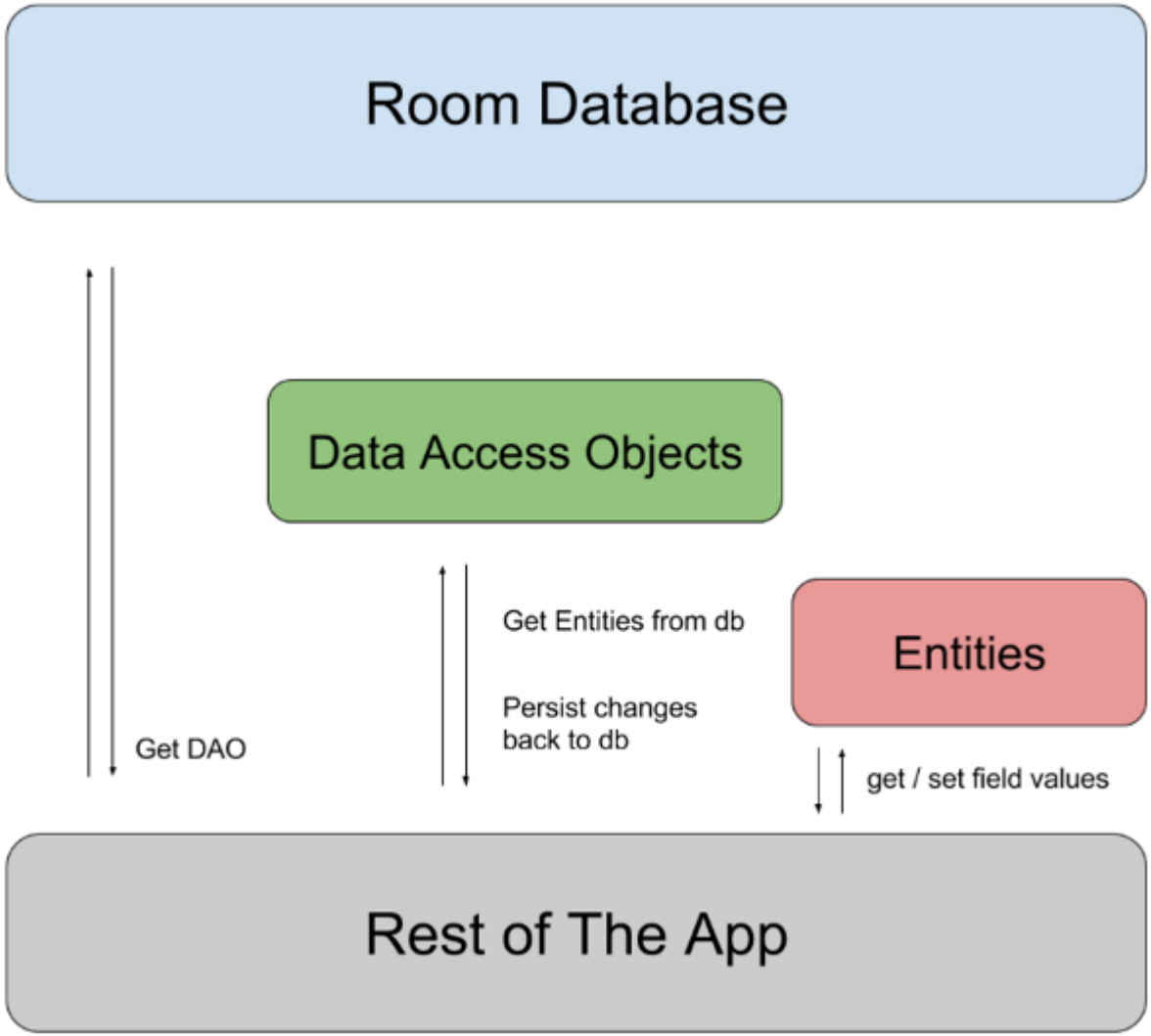
Зберігає посилання на створений екземпляр бази даних у змінній сінглетона.

Визначення Database



За шляхом:
app\build\generated\ksp\debug\kolin\пакет_застосунку\data можна подивитися код згенерованого класу QuestionDatabase_Impl

The Room API components interaction



Оголошення Repository

- Інтерфейс QuestionRepository залишається майже без змін. Тільки додаються модифікатори suspend для методів, що змінюють базу даних. Це необхідно для забезпечення підтримуваних Room асинхронних операцій за допомогою корутин Kotlin:

```
package ua.edu.znu.geoquizcomposeedu.data

import kotlinx.coroutines.flow.StateFlow

interface QuestionRepository {
    fun getQuestionByIndex(index: Int): Question
    fun getQuestionBankSize(): Int
    fun getQuestionListState(): StateFlow<List<Question>>
    suspend fun addQuestion(question: Question)
    suspend fun updateQuestion(updatedQuestion: Question)
    suspend fun removeQuestion(question: Question)
}
```



Реалізація Repository

- Основні зміни у класі QuestionRepositoryImpl пов'язані з видаленням MutableStateFlow, який відповідав за оновлення інтерфейсу при використання in-memory списку питань. Тепер Room через події Flow інформує про зміни у базі даних.
- Перетворення з потоку Flow<List<Question>> Room до потоку стану StateFlow<List<Question>>, що використовується у ViewModel виконується функцією:

```
public fun <T> Flow<T>.stateIn(
    scope: CoroutineScope,
    started: SharingStarted,
    initialValue: T
): StateFlow<T>
```



Реалізація Repository

```
package ua.edu.znu.geoquizcomposeedu.data

import ...

class QuestionRepositoryImpl private constructor(private val questionDataSource:
                                                 QuestionDao) : QuestionRepository {
    companion object {
        @Volatile
        private var instance: QuestionRepositoryImpl? = null

        private val applicationScope = CoroutineScope(Dispatchers.IO + SupervisorJob())

        fun getInstance(questionDao: QuestionDao): QuestionRepositoryImpl {
            return instance ?: synchronized(this) {
                    instance ?: QuestionRepositoryImpl(questionDao).also { INSTANCE = it }
            }       }   }

    override fun getQuestionByIndex(index: Int): Question {
        val list = getQuestionListState().value
        if (list.isEmpty()) {
            return Question(-1,
                            "Жодного питання не було створено", true)
        }
        return getQuestionListState().value[index]    } ...
```

Безпечний для роботи з потоками сінглетон, який повертає єдиний екземпляр репозиторію.

Екземпляр CoroutineScope з областю видимості застосунку, що передається до функції stateIn. Розташування у сінглетоні забезпечує активність колекції Flow на протязі життєвого циклу застосунку.

… Реалізація Repository

```
override fun getQuestionBankSize() = getQuestionListState().value.size

private val questionsStateFlow: StateFlow<List<Question>> =
    questionDataSource.getQuestions()
        .stateIn(
            scope = applicationScope,
            started = SharingStarted.Eagerly,
            initialValue = emptyList()
        )

override fun getQuestionListState(): StateFlow<List<Question>> {
    return questionsStateFlow
}

override suspend fun addQuestion(question: Question) {
    questionDataSource.addQuestion(question)
}

override suspend fun updateQuestion(updatedQuestion: Question) {
    questionDataSource.updateQuestion(updatedQuestion)
}

override suspend fun removeQuestion(question: Question) {
    questionDataSource.removeQuestion(question)    } }
```

Функція stateIn перетворює Room Flow на StateFlow, який містить останній список питань і використовується UI.

Визначення класу Application

- Для створення єдиного екземпляру репозиторію, що буде використовуватися різними частинами застосунку, він може бути внесений до контексту застосунку. Для цього створюється клас GeoQuizApplication, що успадковується від android.app.Application, у якому визначається поле-посилання на репозиторій з його ініціалізацією як сінглетона:

```
package ua.edu.znu.geoquizcomposeedu

import ...

class GeoQuizApplication : Application() {
    val questionRepository: QuestionRepository by lazy {
        QuestionRepositoryImpl
            .getInstance(
                QuestionDatabase.getInstance(this)
                    .questionDao()
            )
    }
}
```

Визначення класу Application

- Для створення екземпляру класу GeoQuizApplication при запуску застосунку (ще до запуску активності) необхідно зазначити шлях до цього класу від пакету застосунку в AndroidManifest.xml (необхідно додати атрибут android:name до тегу application):

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

- Ця конфігурація гарантує використання GeoQuizApplication, що дозволяє отримувати доступ до репозиторію у всій програмі.
- Тепер можна інтегрувати базу даних Room в архітектуру проєкту.



Класи ViewModel

- Класи MainViewModel та QuestionListViewModel не мали змін при інтеграції Room, повторюємо тут їх код для зручності:

```
package ua.edu.znu.geoquizcomposeedu.viewmodel

import ...

class MainViewModel(private val questionRepository: QuestionRepository) : ViewModel() {

    private val _mainScreenState = MutableStateFlow(MainScreenState())
    val mainScreenState: StateFlow<MainScreenState> = _mainScreenState.asStateFlow()

    var currentIndex = _mainScreenState.value.currentIndex

    fun getQuestionByIndex(index: Int): Question {
        return questionRepository.getQuestionByIndex(index)
    }

    fun onAnswerButtonClick(isTrue: Boolean): Int {
        val currentQuestion = questionRepository.getQuestionByIndex(currentIndex)
        val messageId = if (isTrue == currentQuestion.answer) {
            R.string.correct_toast
        } else { R.string.incorrect_toast }
        return messageId
    } ...
```

Класи ViewModel

```
...
    fun onNextQuestionButtonClick() {
        _mainScreenState.value = _mainScreenState.value.copy(
            currentIndex = (currentIndex + 1) % questionRepository.getQuestionBankSize()
        )
        currentIndex = _mainScreenState.value.currentIndex
        Log.d(TAG, "MainViewModel.onNextQuestionButtonClick currentIndex: $currentIndex")
    }
}
```

```
package ua.edu.znu.geoquizcomposeedu.viewmodel

import ...

class QuestionListViewModel(questionRepository: QuestionRepository) : ViewModel() {
    // Expose the question list as a StateFlow
    val questionListFlow = questionRepository.getQuestionListState()
}
```



Класи ViewModel

- У класі QuestionViewModel зміни виконані тільки для викликів suspend функцій:

```
package ua.edu.znu.geoquizcomposeedu.viewmodel

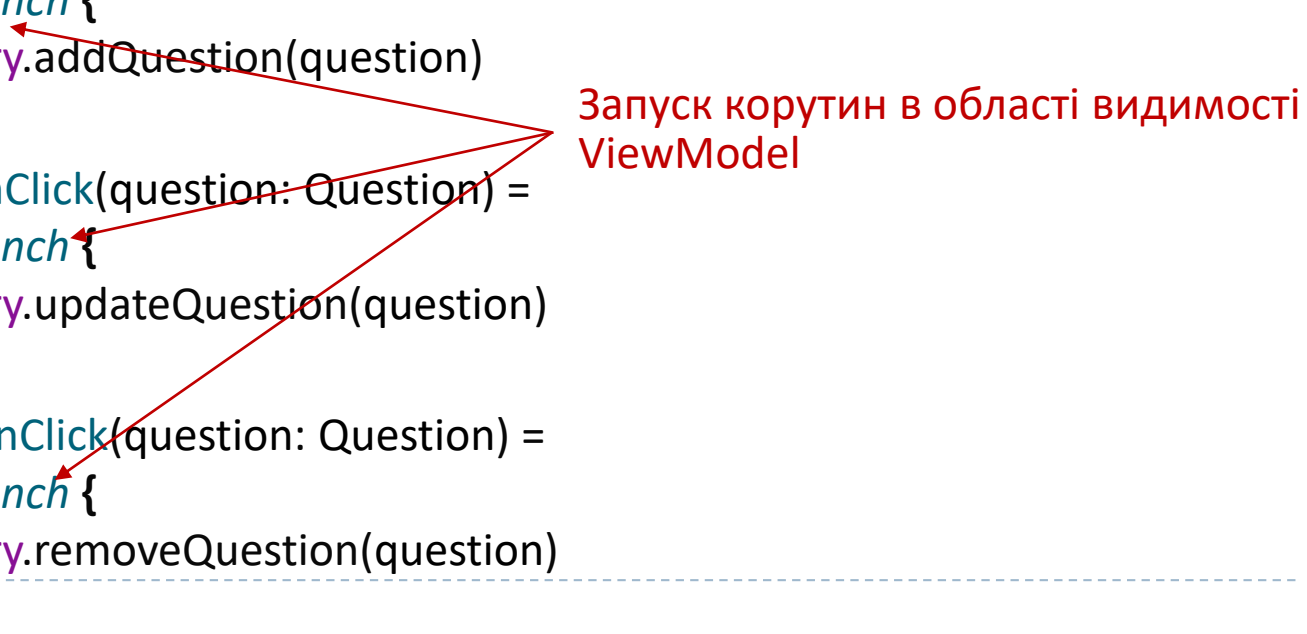
import ...

class QuestionViewModel(private val questionRepository: QuestionRepository) :
                                                                    ViewModel() {

    fun onAddQuestionClick(question: Question) =
        viewModelScope.launch {
            questionRepository.addQuestion(question)
        }

    fun onUpdateQuestionClick(question: Question) =
        viewModelScope.launch {
            questionRepository.updateQuestion(question)
        }

    fun onRemoveQuestionClick(question: Question) =
        viewModelScope.launch {
            questionRepository.removeQuestion(question)
        }
}
```



Composable функції екранів

- Composable функції екранів MainScreen, QuestionListScreen та QuestionScreen змінилися тільки способом отримання репозиторію - тепер він отримується з контексту застосунку:

```
package ua.edu.znu.geoquizcomposeedu.ui.screens

import ...

data class MainScreenState(
    val currentIndex: Int = 0,
    val questionList: List<Question> = emptyList()
)

@Composable
fun MainScreen(
    snackbarHostState: SnackbarHostState,
) {
    val questionRepository =
        (LocalContext.current.applicationContext as GeoQuizApplication).questionRepository

    val mainViewModel: MainViewModel = viewModel(
        factory = ViewModelFactory(MainViewModel::class.java) {
            MainViewModel(questionRepository)
        }   ) ...
```

Composable функції екранів

...

```
    val mainScreenState by mainViewModel.mainScreenState.collectAsStateWithLifecycle()
    val scrollState = rememberScrollState()
    val scope = rememberCoroutineScope()
    val context = LocalContext.current
    Column( ... далі без змін)
```

```
package ua.edu.znu.geoquizcomposeedu.ui.screens

import ...

@OptIn(ExperimentalFoundationApi::class)
@Composable
fun QuestionListScreen(
    navController: NavController,
    onEditQuestionClick: (Question) -> Unit = {},
) {
    val questionRepository = (LocalContext.current.applicationContext as
                                                    GeoQuizApplication).questionRepository

val questionListViewModel: QuestionListViewModel = viewModel(
    factory = ViewModelFactory(QuestionListViewModel::class.java) {
        QuestionListViewModel(questionRepository)       }    ) ...
```

Composable функції екранів

...

```
val questionList: List<Question> by
                        questionListViewModel.questionListFlow.collectAsStateWithLifecycle()
Box(modifier = Modifier.fillMaxSize()) {
    LazyColumn { ... далі без змін )
```

```
package ua.edu.znu.geoquizcomposeedu.ui.screens

import ...

@Composable
fun QuestionScreen(
    initialQuestion: Question?,
    buttonTextRes: Int,
    onNavigateBack: () -> Unit = {}
) {
    val questionRepository = (LocalContext.current.applicationContext as
                                                    GeoQuizApplication).questionRepository

    val questionViewModel: QuestionViewModel = viewModel(
        factory = ViewModelFactory(QuestionViewModel::class.java) {
            QuestionViewModel(questionRepository)
        }
    ) ...
```

Composable функції екранів

```
...
var textValue by rememberSaveable { mutableStateOf(initialQuestion?.questionText ?: "") }
var state by rememberSaveable { mutableStateOf(initialQuestion?.answer ?: false) }
val localFocusManager = LocalFocusManager.current
Column( ... далі без змін)
```



Навігаційний компонент

- Оскільки застосунок запускається з порожньою базою даних, у визначенні NavHost аргумент startDestination був змінений на маршрут до екрану зі списком питань:

```
package ua.edu.znu.geoquizcomposeedu.nav

import ...

@Composable
fun NavPassQuestion(
    innerPadding: PaddingValues,
    snackbarHostState: SnackbarHostState,
    navController: NavHostController
) {
    NavHost(
        navController = navController,
        startDestination = Routes.QuestionList,
        modifier = Modifier.padding(innerPadding)
    ) { ... далі без змін)
```



MainActivity

- Клас MainActivity залишився без змін - він викликає функцію NavPassQuestion:

```
package ua.edu.znu.geoquizcomposeedu

import ...

class MainActivity : ComponentActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {
            GeoQuizComposeEduTheme {
                val navController = rememberNavController()
                val snackbarHostState = remember { SnackbarHostState() }
                Scaffold(
                    topBar = {
                        GeoQuizTopAppBar()
                    },
                    bottomBar = {
                        BottomAppBar(navController)
                    }, ...
```

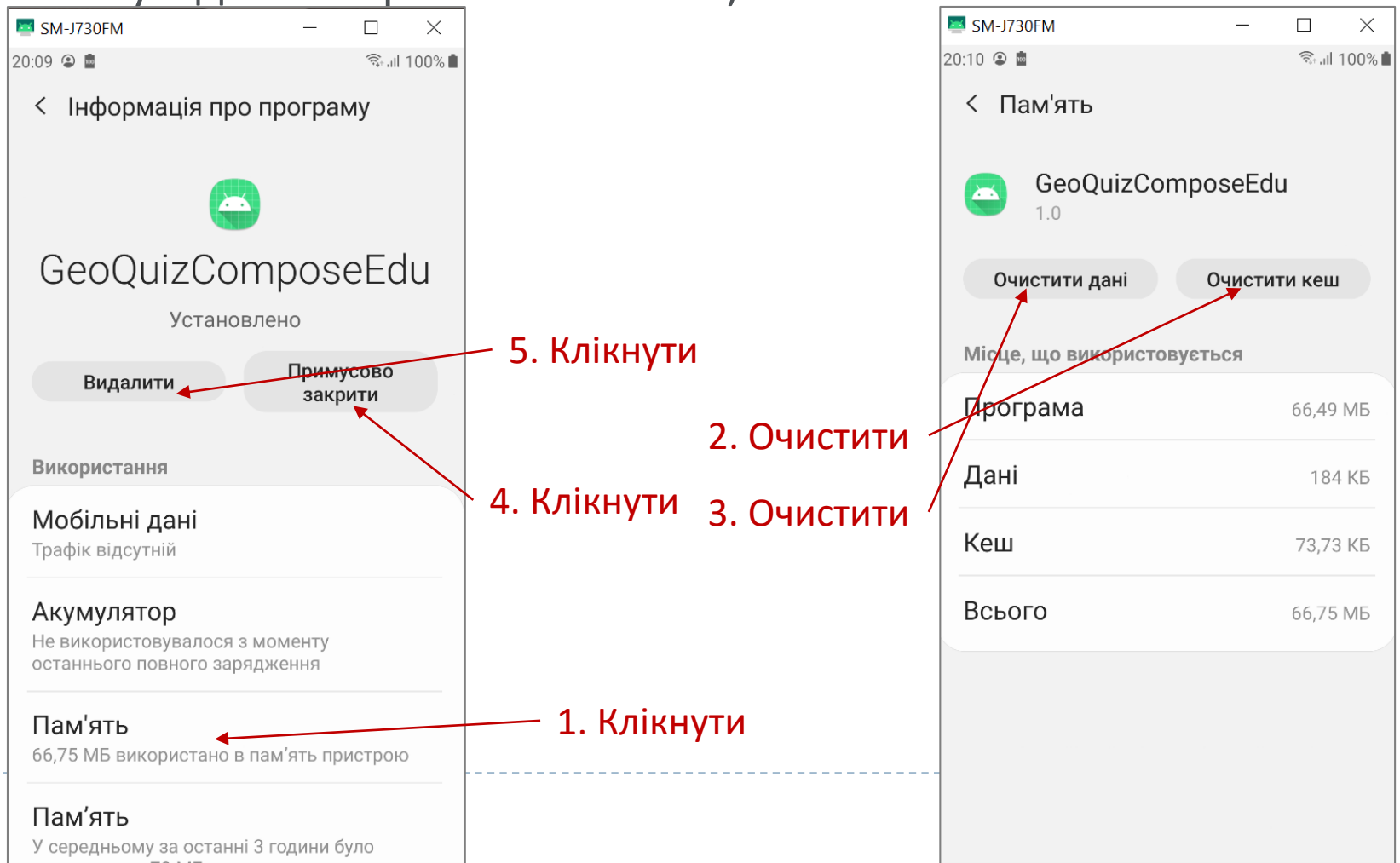
MainActivity

```
...
                snackbarHost = {
                    AppSnackbar(snackbarHostState = snackbarHostState)
                },
            ) { innerPadding ->
                // main content
                NavPassQuestion(innerPadding, snackbarHostState, navController)
            }
        }
    }
}
```

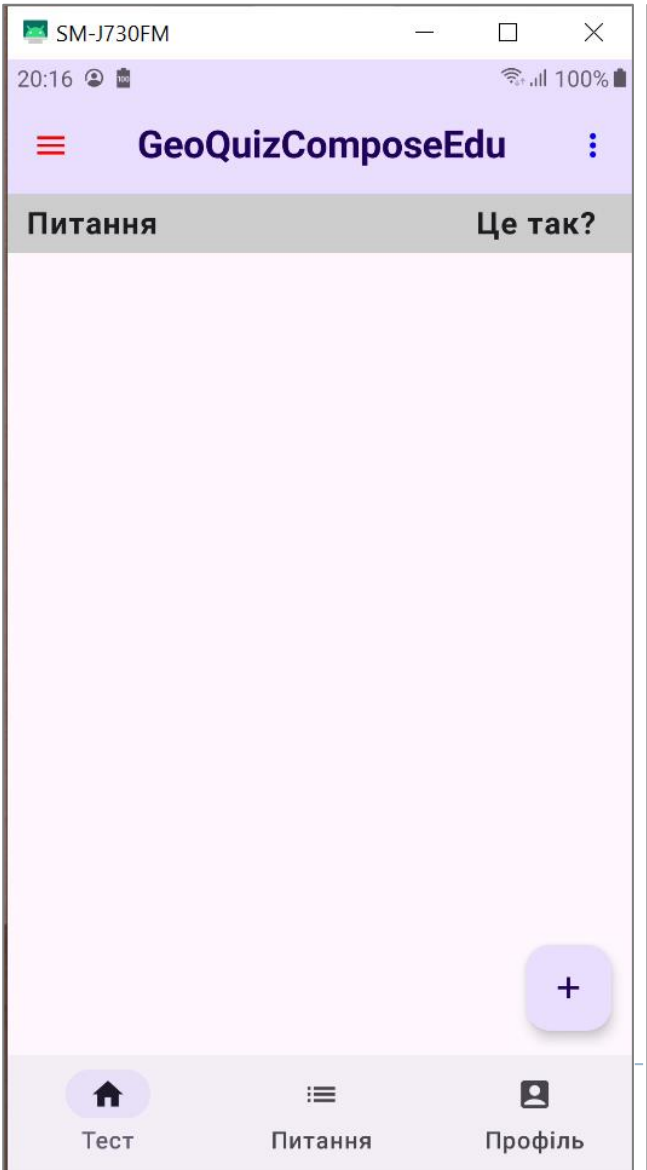


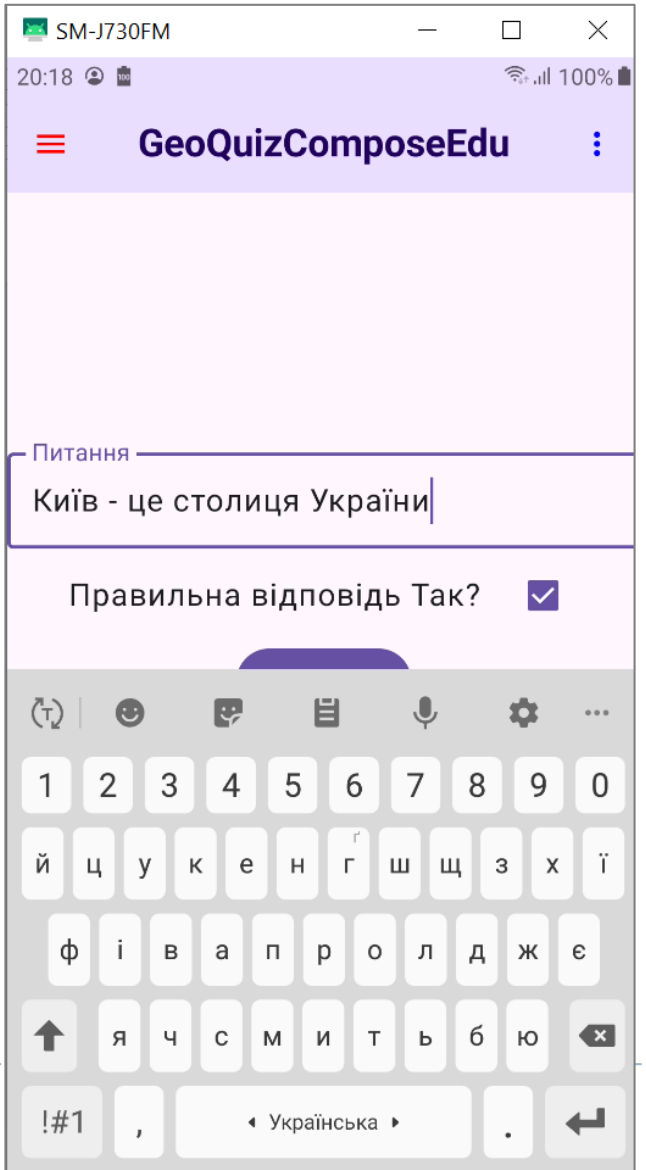
Апробація роботи застосунку

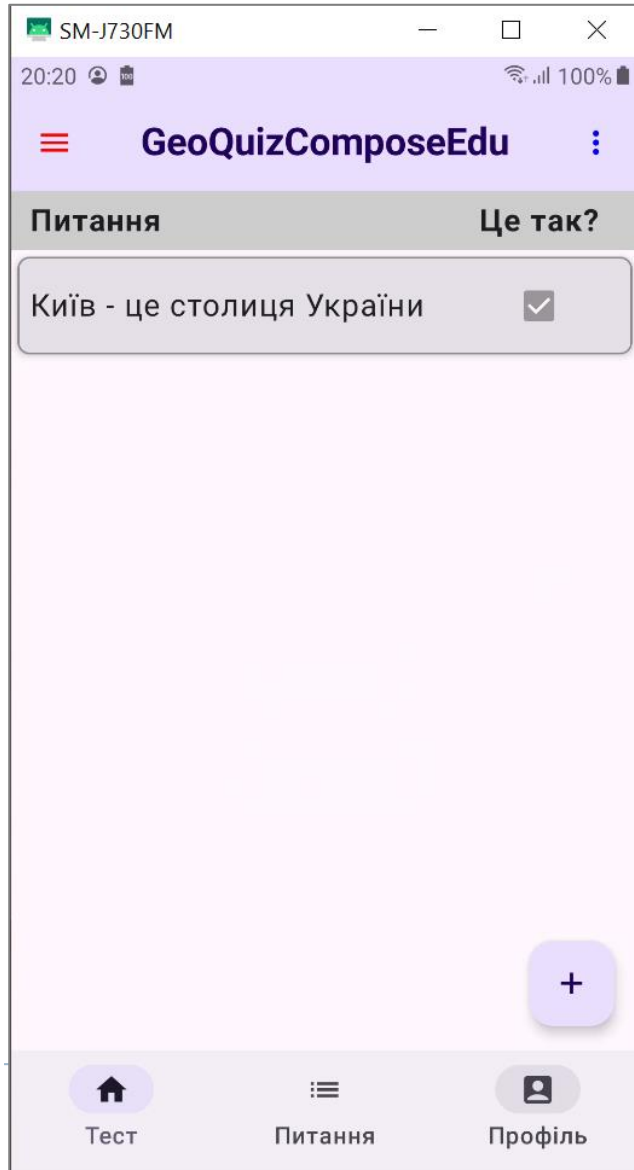
- Перед запуском застосунку рекомендується його видалення штатними засобами ОС Android в налаштуваннях пристрою (також необхідно видалити усі дані. збережені в пам'яті):



Апробація роботи застосунку - запуск та додання питань



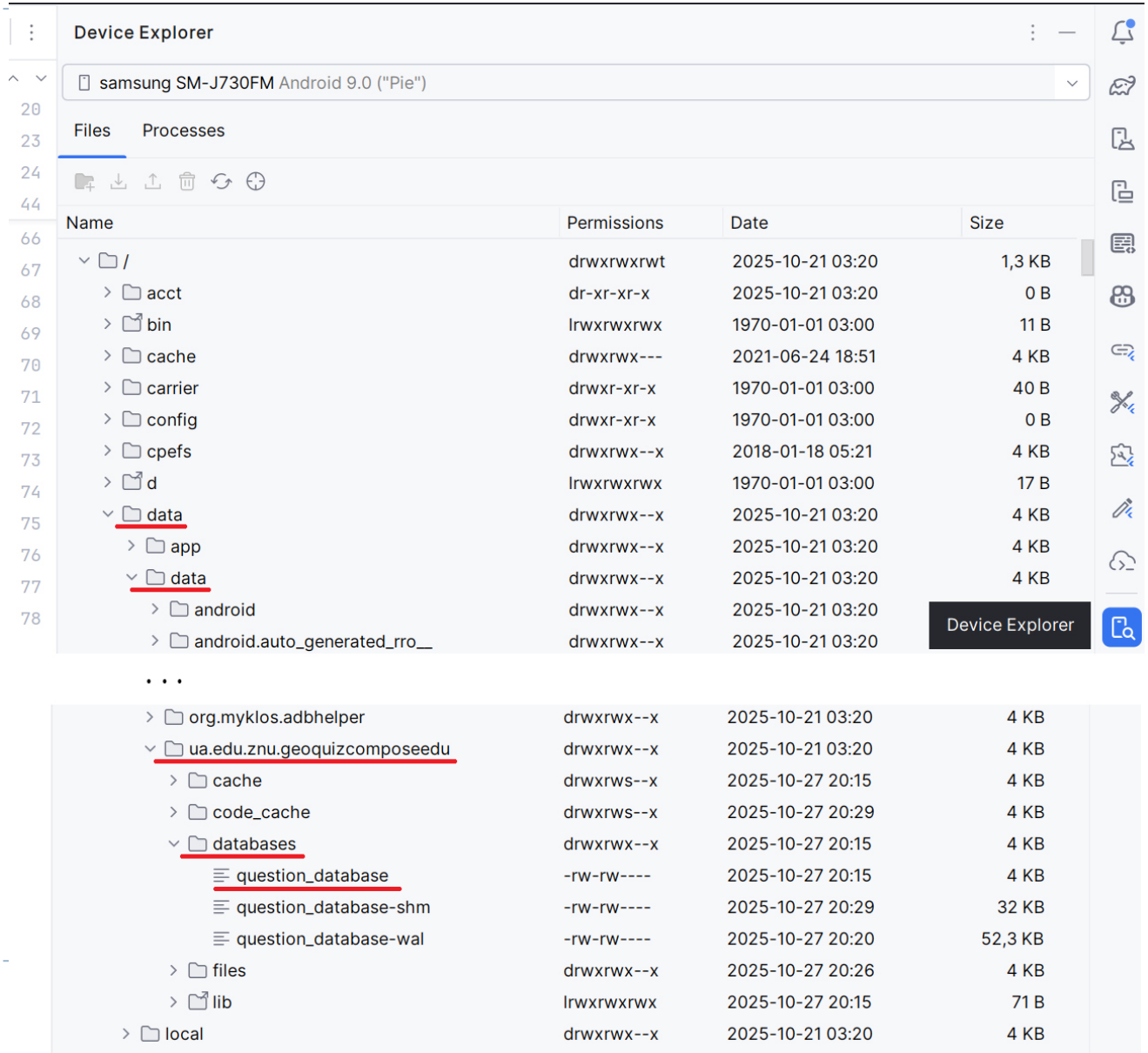




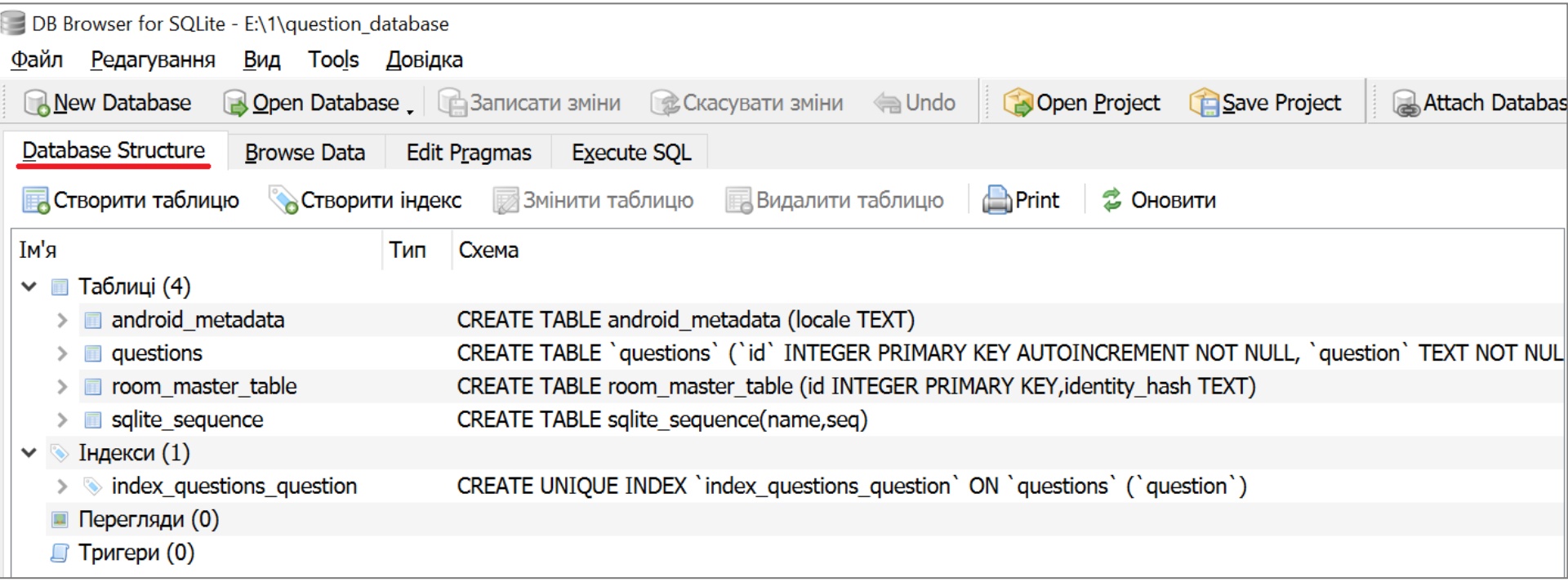
Апробація роботи застосунку - перегляд бази даних (DB Browser for SQLite)

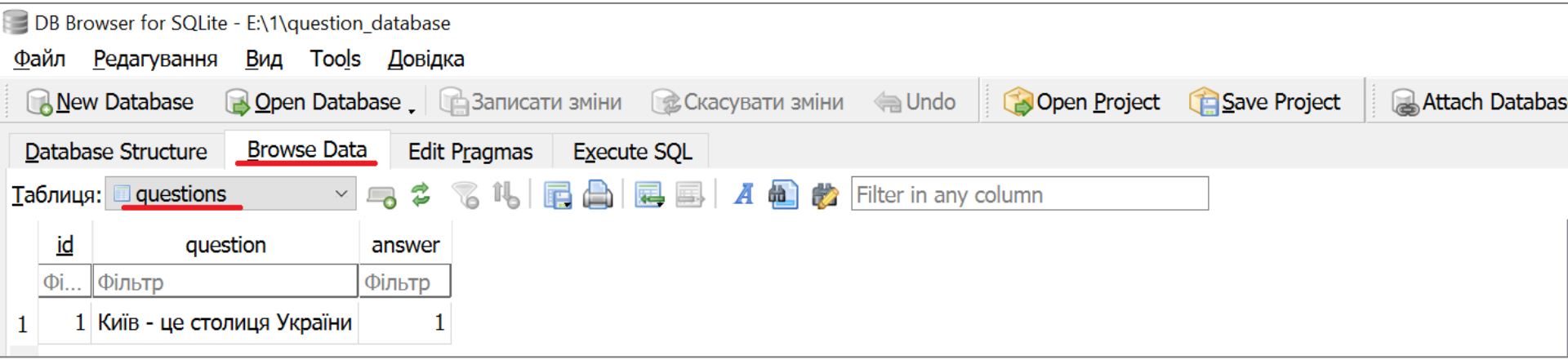
- Android Studio має вбудований інструмент App Inspection (View - Tool Windows - App Inspection) для перегляду та роботи з базою даних. Але враховуючи його нестабільну роботу (і неможливість запуска на пристроях з Android < 8.0), розглянемо використання вільно розповсюджуваної кросплатформної програми DB Browser for SQLite.
- Для отримання файлу бази даних необхідно відкрити Device Explorer (View -Tool Windows - Device Explorer) та у каталозі data\data знайти пакет застосунку, у якому відкрити підкаталог databases, з якого скопіювати командою контекстного меню Save As ... файли **question_database** (ім'я бази даних задавалося у білдері бази даних у сінглетоні класу QuestionDatabase), **question_database-shm** та **question_database-wal** у каталог на комп'ютері.
- Після чого файл **question_database** можна відкрити в DB Browser for SQLite.

Апробація роботи застосунку - перегляд бази даних (DB Browser for SQLite)



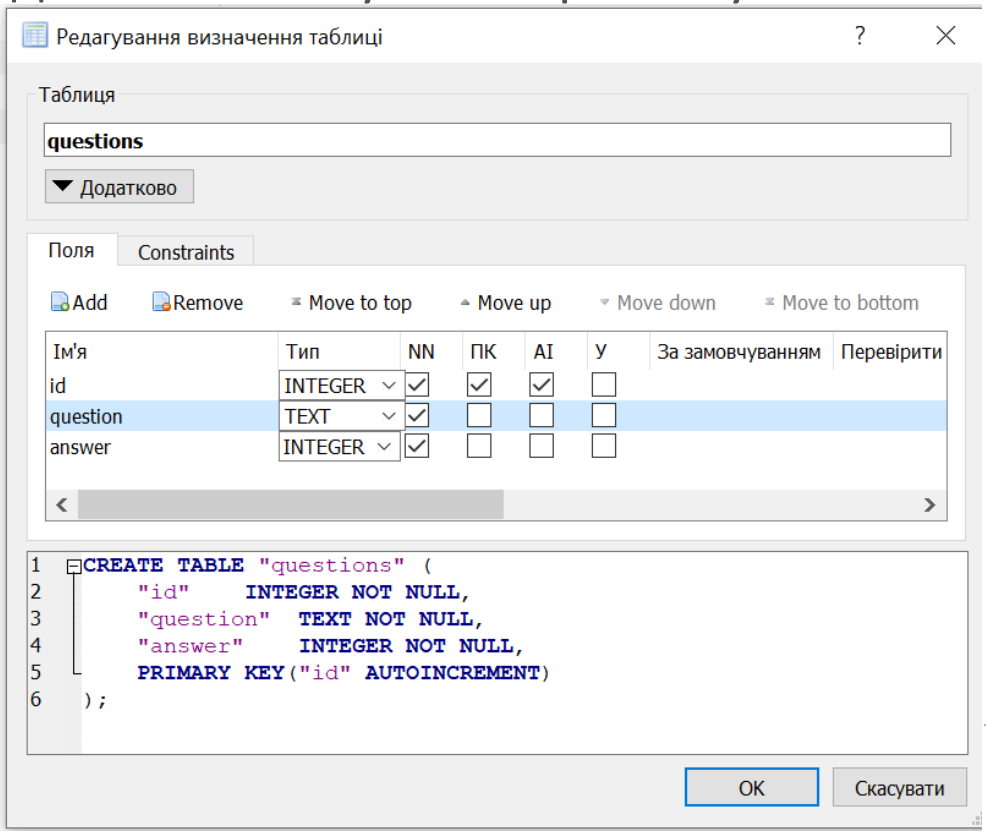
Апробація роботи застосунку - перегляд бази даних (DB Browser for SQLite)



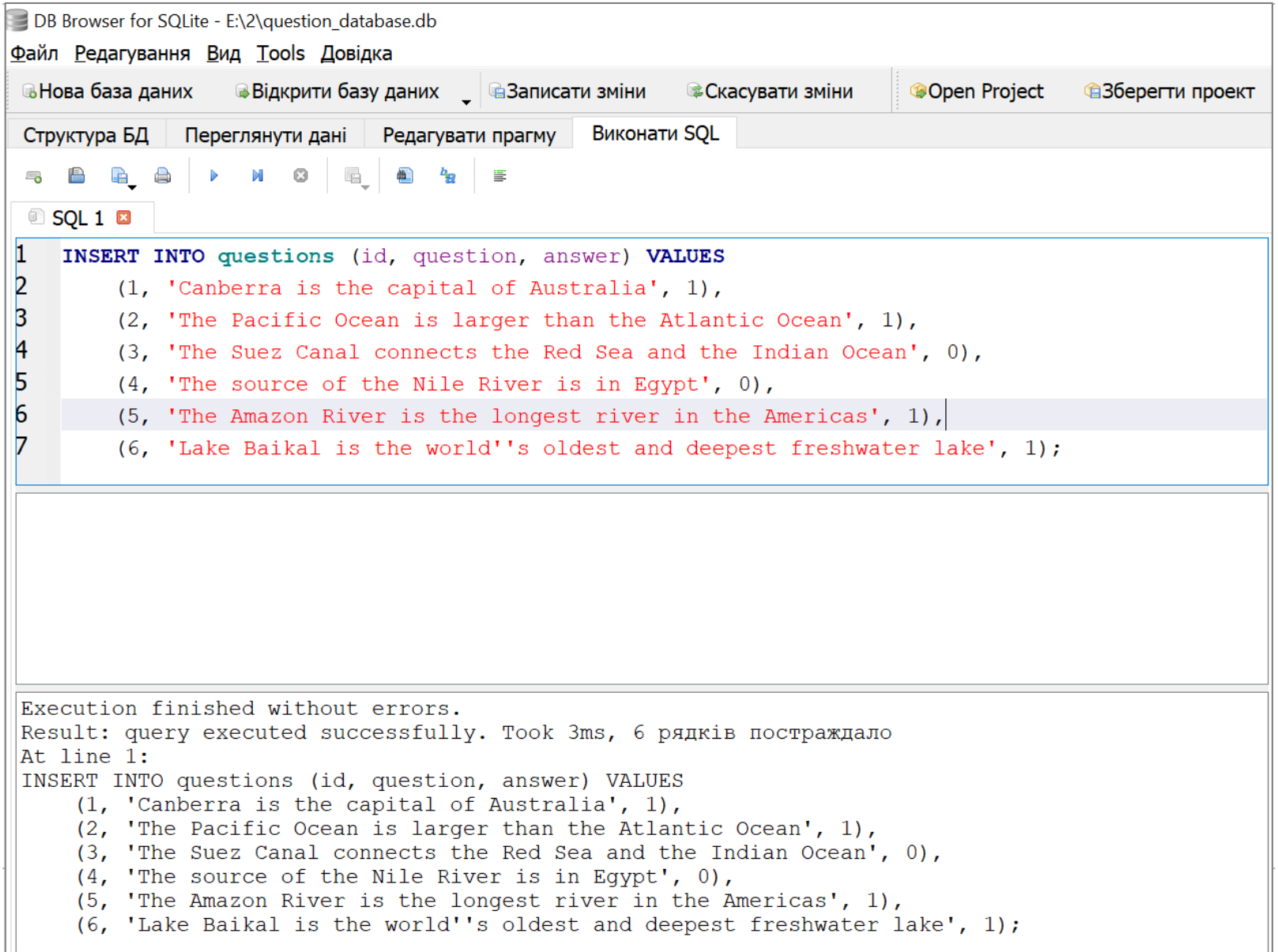


Попереднє заповнення бази даних за допомогою Room

- Часто необхідно, щоб застосунок стартував вже з наповненою базою даних - з попереднім заповненням бази даних. У Room 2.2.0 та вище існують методи для попереднього заповнення бази даних Room під час ініціалізації вмістом із попередньо створеного файлу бази даних.
- Такий файл бази даних може бути створений у DB Browser для SQLite.

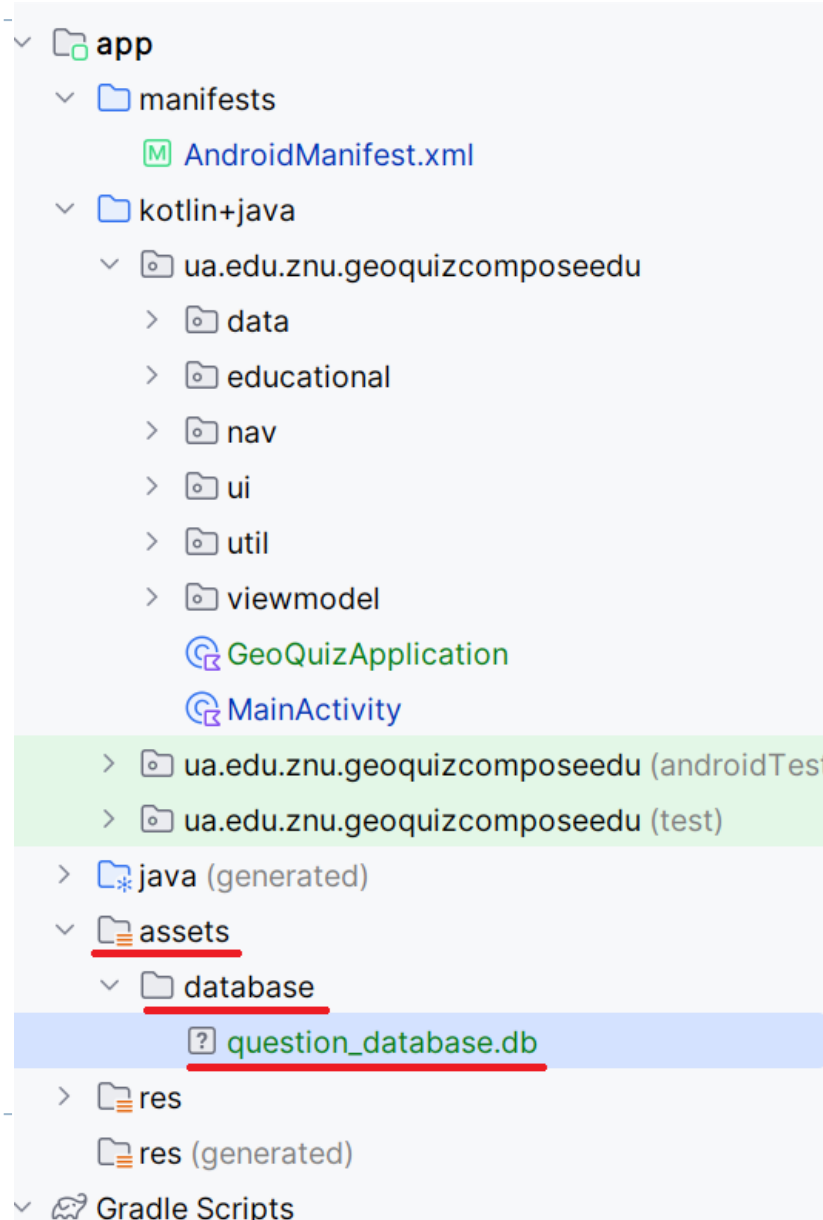


Попереднє заповнення бази даних за допомогою Room



Попереднє заповнення бази даних за допомогою Room

2. Додати каталог **assets** до модуля проєкту командою контекстного меню: File - New - Folder - Assets Folder з налаштуваннями за замовчуванням.
3. Створити підкаталог **databases** у папці assets.
4. Скопіювати файл із попередньо упакованою базою даних у цей підкаталог.



Попереднє заповнення бази даних за допомогою Room

5. В функції getQuestionDatabase(context: Context) класу QuestionDatabase для RoomDatabase. Builder необхідно додати виклик функції .createFromAsset("database/question_database.db") :

```
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 }
    }
}
```

Попереднє заповнення бази даних за допомогою Room

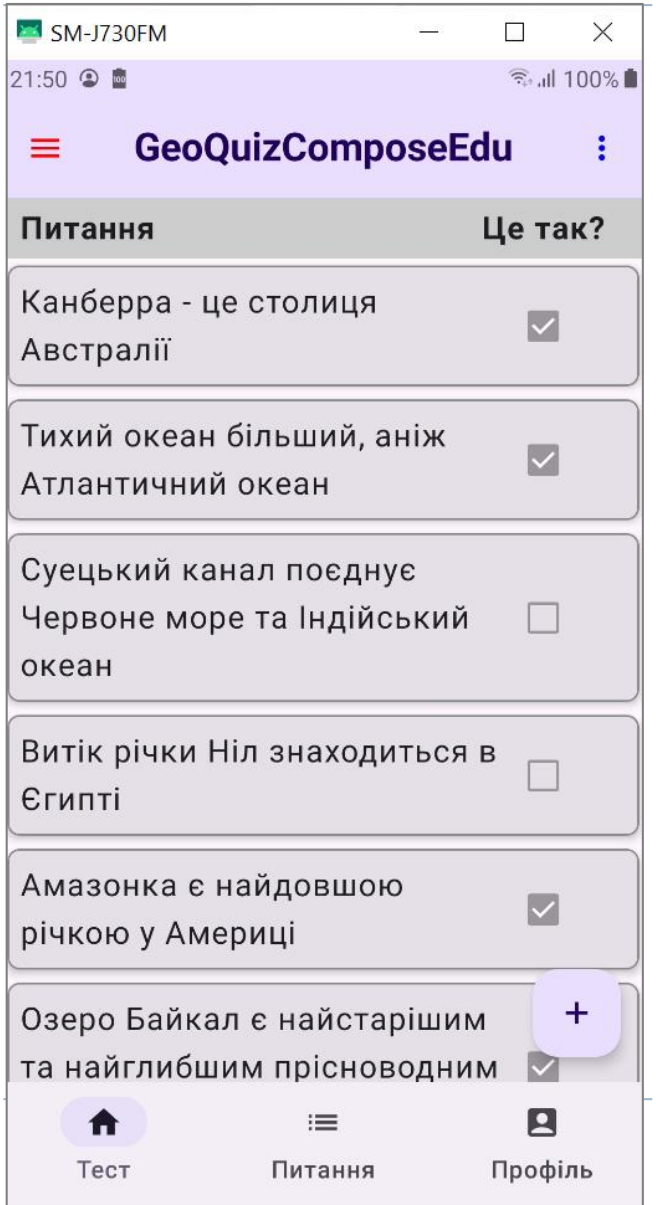
- Для перевірки попереднього заповнення бази даних за допомогою Room необхідно видалити дані програми та видалити саму програму з телефону так, як це було показано раніше.
- Можна також перевірити відсутність бази даних програми за допомогою Android Studio Device Explorer: пакет з назвою пакета програми має бути відсутнім у структурі файлів пристрою в папці data/data.
- Після запуску застосунку список питань буде заповненим.
- Можна повернути налаштування екрану тесту як стартового у NavHost:

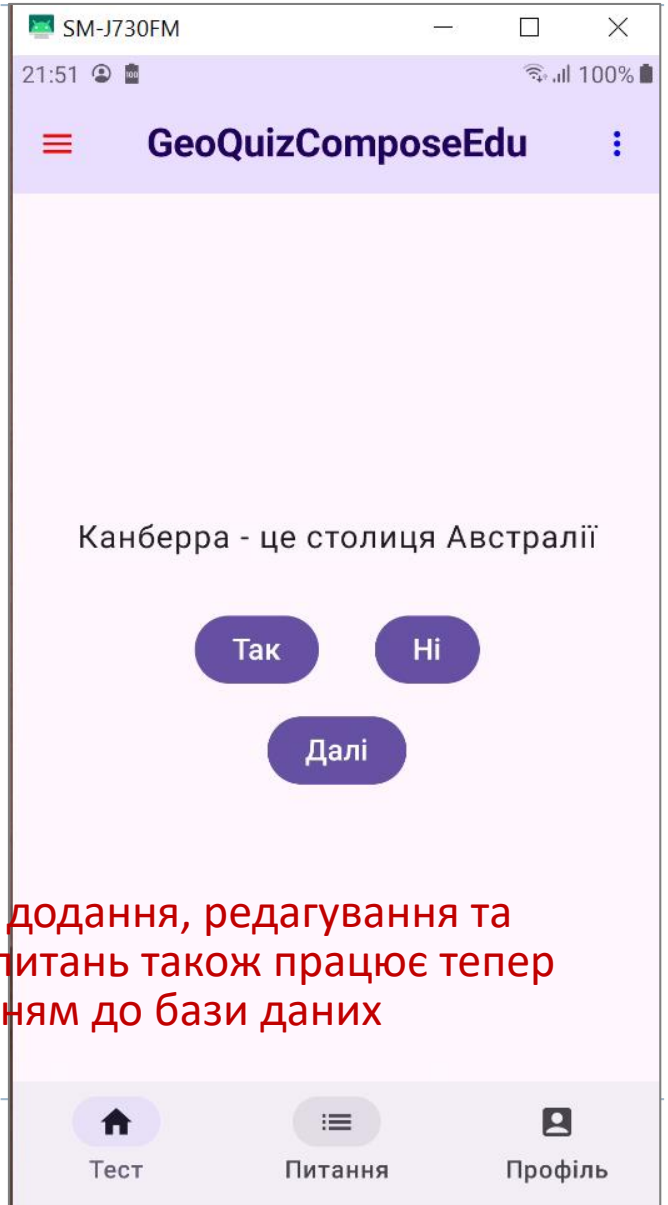
```
package ua.edu.znu.geoquizcomposeedu.nav

import ...

@Composable
fun NavPassQuestion( ...) {
   NavHost(
      navController = navController,
      startDestination = Routes.Home,
      modifier = Modifier.padding(innerPadding)     ) { ... }
```

Попереднє заповнення бази даних за допомогою Room





Функціонал додання, редагування та видалення питань також працює тепер зі збереженням до бази даних

Організація зв'язків між сутностями - One-to-Many

- Припустимо, що у проєкті є дві сутності користувач (User) та плейлист (Playlist), один користувач має багато плейлистів, а кожен плейлист належить тільки одному користувачу:

```
package ....room.data.onetomany

import ...

@Entity(tableName = "users")
data class User(
    @PrimaryKey(autoGenerate = true)
    val userId: Long = 0,
    val name: String,
    val age: Int
)
```

```
package ...room.data.onetomany

import ...

@Entity(
    tableName = "playlists",
    foreignKeys = [ForeignKey(
            entity = User::class,
            parentColumns = ["userId"],
            childColumns = ["creatorId"],
            onDelete = CASCADE)    ] )
data class Playlist(
    @PrimaryKey(autoGenerate = true)
    val playlistId: Long = 0,
    val name: String,
    val creatorId: Long
)
```

Зовнішній ключ - посилається на User.userId

Організація зв'язків між сутностями - One-to-Many

- Щоб запитувати список користувачів та відповідні списки відтворення, спочатку потрібно змоделювати зв'язок «один до багатьох» між двома сутностями - для цього створюється клас даних:

```
package ua.edu.znu.geoquizcomposeedu.educational.room.data.onetomany

import ...

data class UserWithPlaylists(
    @Embedded
    val user: User,
    @Relation(
        parentColumn = "userId",
        entityColumn = "creatorId"
    )
    val playlists: List<Playlist>
)
```

@Embedded повідомляє Room про необхідність звести поля вкладеного об'єкта до сутності, що його містить, щоб Room зіставляв ці поля зі стовпцями в результаті запиту, а не обробляв вкладений об'єкт як окрему таблицю.

@Relation повідомляє Room як заповнити поля в UserWithPlaylists:
parentColumn - ім'я стовпця первинного ключа батьківської сутності, а entityColumn - ім'я стовпця дочірньої сутності, який посилається на первинний ключ батьківської сутності.

Організація зв'язків між сутностями - One-to-Many

- Інтерфейс Dao визначає функції для роботи з базою даних:

```
package ua.edu.znu.geoquizcomposeedu.educational.room.data.onetomany

import ...

@Dao
interface UserDao {
    @Insert
    suspend fun insertUser(user: User)

    @Insert
    suspend fun insertPlaylist(playlist: Playlist)

    @Transaction
    @Query("SELECT * FROM users WHERE userId = :userId")
    fun getUserWithPlaylists(userId: Long): Flow<UserWithPlaylists>
}
```



Організація зв'язків між сутностями - One-to-Many

- Абстрактний клас Database визначає базу даних:

```
package ua.edu.znu.geoquizcomposeedu.educational.room.data.onetomany

import androidx.room.Database
import androidx.room.RoomDatabase

@Database(
    entities = [User::class, Playlist::class],
    version = 1
)
abstract class UserDatabase : RoomDatabase() {
    abstract fun userDao(): UserDao
}
```



Організація зв'язків між сутностями - Many-to-Many

- Припустимо, що у проєкті є дві сутності студент (Student) та дисципліна (Course), один студент вивчає багато дисциплін, а кожна дисципліна вивчається багатьма студентами:

```
package ...room.data.manytomany.data

import androidx.room.Entity
import androidx.room.PrimaryKey

@Entity(tableName = "students")
data class Student(
    @PrimaryKey(autoGenerate = true)
    val studentId: Long = 0,
    val name: String
)
```

```
package ...room.data.manytomany.data

import androidx.room.Entity
import androidx.room.PrimaryKey

@Entity(tableName = "courses")
data class Course(
    @PrimaryKey(autoGenerate = true)
    val courseId: Long = 0,
    val title: String
)
```



Організація зв'язків між сутностями - Many-to-Many

- Для зв'язку "багато до багатьох" визначається таблиця перетину:

```
package ua.edu.znu.geoquizcomposeedu.educational.room.data.manytomany.data

import androidx.room.Entity

/**
 * Cross-reference entity to represent the many-to-many relationship
 * between Students and Courses.
 */
@Entity(tableName = "student_courses", primaryKeys = ["studentId", "courseId"])
data class StudentCourseCrossRef(
    val studentId: Long,
    val courseId: Long
)
```



Організація зв'язків між сутностями - Many-to-Many

- Щоб запитувати студентів та відповідні списки їх дисциплін, потрібно змоделювати зв'язок «багато до багатьох» зі сторони студента:

```
package ua.edu.znu.geoquizcomposeedu.educational.room.data.manytomany.data

import ...

data class StudentWithCourses(
    @Embedded
    val student: Student,
    @Relation(
        parentColumn = "studentId",
        entityColumn = "courseId",
        entity = Course::class,
        associateBy = Junction(
            StudentCourseCrossRef::class
        )
    )
    val courses: List<Course>
)
```

@Embedded повідомляє Room про необхідність звести поля вкладеного об'єкта до сутності, що його містить, щоб Room зіставляв ці поля зі стовпцями в результаті запиту, а не обробляв вкладений об'єкт як окрему таблицю.

@Relation повідомляє Room як заповнити поля в UserWithPlaylists:
parentColumn - ім'я стовпця первинного ключа батьківської сутності, а entityColumn - ім'я стовпця дочірньої сутності, який посилається на первинний ключ батьківської сутності.

Організація зв'язків між сутностями - Many-to-Many

- Щоб запитувати дисципліну та відповідні списки студентів, що її вивчають, потрібно змоделювати зв'язок «багато до багатьох» зі сторони дисципліни:

```
package ua.edu.znu.geoquizcomposeedu.educational.room.data.manytomany.data

import ...

data class CourseWithStudents(
    @Embedded
    val course: Course,
    @Relation(
        parentColumn = "courseId",
        entityColumn = "studentId",
        entity = Student::class,
        associateBy = Junction(
            StudentCourseCrossRef::class
        )
    )
    val students: List<Student>
)
```

@Embedded повідомляє Room про необхідність звести поля вкладеного об'єкта до сутності, що його містить, щоб Room зіставляв ці поля зі стовпцями в результаті запиту, а не обробляв вкладений об'єкт як окрему таблицю.

@Relation повідомляє Room як заповнити поля в UserWithPlaylists:
parentColumn - ім'я стовпця первинного ключа батьківської сутності, а entityColumn - ім'я стовпця дочірньої сутності, який посилається на первинний ключ батьківської сутності.

Організація зв'язків між сутностями - Many-to-Many

- Інтерфейс Dao визначає функції для роботи з базою даних:

```
package ua.edu.znu.geoquizcomposeedu.educational.room.data.manytomany.data

import...

@Dao
interface StudentDao {
    @Insert
    suspend fun insertStudent(student: Student)

    @Insert
    suspend fun insertCourse(course: Course)

    @Insert
    suspend fun insertStudentCourseCrossRef(crossRef: StudentCourseCrossRef)

    @Transaction
    @Query("SELECT * FROM students WHERE studentId = :studentId")
    fun getStudentWithCourses(studentId: Long): Flow<StudentWithCourses>

    @Transaction
    @Query("SELECT * FROM courses WHERE courseId = :courseId")
    fun getCourseWithStudents(courseId: Long): Flow<CourseWithStudents>
}
```

Організація зв'язків між сутностями - Many-to-Many

- Абстрактний клас Database визначає базу даних:

```
package ua.edu.znu.geoquizcomposeedu.educational.room.data.manytomany.data

import androidx.room.Database
import androidx.room.RoomDatabase

@Database(
    entities = [Student::class, Course::class, StudentCourseCrossRef::class],
    version = 1)
abstract class StudentDatabase : RoomDatabase() {
    abstract fun studentDao(): StudentDao
}
```



Організація зв'язків між сутностями - One-to-One

- Припустимо, що у проєкті є дві сутності користувач (User) та профіль (Profile), один користувач має тільки один профіль і кожен профіль належить тільки одному користувачу:

```
package ....room.data.onetoone

import ...

@Entity(tableName = "users")
data class User(
    @PrimaryKey(autoGenerate = true)
    val userId: Long = 0,
    val name: String,
    val age: Int
)
```

```
package ...room.data.onetoone

import ...

@Entity(tableName = "profiles")
data class Profile(
    @PrimaryKey(autoGenerate = true)
    val profileId: Long = 0,
    val bio: String,
    val userOwnerId: Long
)
```

Зовнішній ключ - посилається на User.userId



Організація зв'язків між сутностями - One-to-One

- Щоб запитувати список користувачів та відповідний профіль, спочатку потрібно змоделювати зв'язок «один до одного» між двома сутностями - для цього створюється клас даних:

```
package ua.edu.znu.geoquizcomposeedu.educational.room.data.onetoone

import ...

data class UserWithProfile(
    @Embedded
    val user: User,
    @Relation(
        parentColumn = "userId",
        entityColumn = "userOwnerId"
    )
    val profile: Profile
)
```

@Embedded повідомляє Room про необхідність звести поля вкладеного об'єкта до сутності, що його містить, щоб Room зіставляв ці поля зі стовпцями в результаті запиту, а не обробляв вкладений об'єкт як окрему таблицю.

@Relation повідомляє Room як заповнити поля в UserWithProfile:
parentColumn - ім'я стовпця первинного ключа батьківської сутності, а entityColumn - ім'я стовпця дочірньої сутності, який посилається на первинний ключ батьківської сутності.



Організація зв'язків між сутностями - One-to-One

- Інтерфейс Dao визначає функції для роботи з базою даних:

```
package ua.edu.znu.geoquizcomposeedu.educational.room.data.onetoone

import ...

@Dao
interface UserDao {
    @Insert
    suspend fun insertUser(user: User)

    @Insert
    suspend fun insertProfile(profile: Profile)

    @Transaction
    @Query("SELECT * FROM users")
    fun getUsersWithProfiles(): Flow<UserWithProfile>
}
```



Організація зв'язків між сутностями - One-to-One

- Абстрактний клас Database визначає базу даних:

```
package ua.edu.znu.geoquizcomposeedu.educational.room.data.onetoone

import androidx.room.Database
import androidx.room.RoomDatabase

@Database(
    entities = [User::class, Profile::class],
    version = 1
)
abstract class UserDatabase : RoomDatabase() {
    abstract fun userDao(): UserDao
}
```



QuestionListScreen State Hoisting

```
package ua.edu.znu.geoquizcomposeedu.ui.screens

import ...

@OptIn(ExperimentalFoundationApi::class)
@Composable
fun QuestionListScreen(
    questionList: List<Question>,
    onEditQuestionClick: (Question) -> Unit = {},
    onAdd: () -> Unit = {}
) {
    Box(modifier = Modifier.fillMaxSize()) {
        LazyColumn {
            listHeader()
            items(
                items = questionList,
                key = { question -> question.id }
            ) { question ->
                QuestionCard(
                    question,
                    onEditQuestionClick = {
                        onEditQuestionClick(question)
                    }            )        }      } ...
```

QuestionListScreen State Hoisting

```
…
    FloatingActionButton(
        onClick = {
            onAdd()
        },
        modifier = Modifier
            .align(Alignment.BottomEnd)
            .padding(16.dp)
    ) {
        Icon(Icons.Filled.Add, contentDescription = stringResource(R.string.add_question))
    }   }}
@OptIn(ExperimentalFoundationApi::class)
fun LazyListScope.listHeader() {
    stickyHeader {
        Row(
            modifier = Modifier
                .fillMaxWidth()
                .background(Color.LightGray)
                .padding(4.dp),
            horizontalArrangement = Arrangement.Start
        ) { …
```

QuestionListScreen State Hoisting

```
...
        Text(
            text = stringResource(R.string.question),
            fontWeight = FontWeight.Bold,
            modifier = Modifier
                .fillMaxWidth(0.75f)
                .padding(start = 8.dp)
        )
        Text(
            text = stringResource(R.string.is_true),
            fontWeight = FontWeight.Bold,
        )
    }
  }
}
@Composable
fun QuestionCard(
    question: Question,
    onEditQuestionClick: () -> Unit = {},
) { ...
```



QuestionListScreen State Hoisting

```
...
    Card(
        shape = RoundedCornerShape(8.dp),
        border = BorderStroke(1.dp, Color.Gray),
        elevation = CardDefaults.cardElevation(4.dp),
        modifier = Modifier
            .fillMaxWidth()
            .padding(4.dp)
            .clickable(
                onClick = onEditQuestionClick
            )
    ) {
        Row(
            verticalAlignment = Alignment.CenterVertically,
            modifier = Modifier
                .padding(8.dp)
        ) {
            Text(
                text = question.questionText,
                modifier = Modifier.fillMaxWidth(0.8f)
            )
...
```

QuestionListScreen State Hoisting

```
        Checkbox(
            checked = question.answer,
            enabled = false,
            onCheckedChange = {},
        )      }   }}
@Preview(showBackground = true)
@Composable
fun QuestionCardPreview() {
    QuestionCard(
        question = Question(
            questionText = "Канберра - це столиця Австралії",
            answer = true
        )   )}
@Preview(showSystemUi = true)
@Composable
fun QuestionListScreenPreview() {
    QuestionListScreen(
        questionList = listOf(
            Question(questionText = stringResource(R.string.question_australia), answer = true),
            Question(questionText = stringResource(R.string.question_oceans), answer = true),
            Question(questionText = stringResource(R.string.question_mideast), answer = false),))}
```

QuestionScreen State Hoisting

- У разі підняття стану з QuestionScreen, ця Composable-фйнкція буде мати такі параметри:

```
QuestionScreen(
    textValue = textValue,
    onTextChange = { textValue = it },
    isAnswerTrue = isAnswerTrue,
    onCheckedChange = { isAnswerTrue = it },
    buttonTextRes = buttonTextRes,
    onSave = onSave,
    onRemove = onRemove,
)
```

- QuestionScreen приймає параметри textValue, isAnswerTrue та пов'язані з ними callbacks: onTextChange, onCheckedChange, тому доцільно винести створення станів text та isAnswerTrue до доданої Composable функції QuestionScreenHost:



QuestionScreen State Hoisting

```
package ua.edu.znu.geoquizcomposeedu.ui.screens

import ...

@Composable
fun QuestionScreenHost(
    initialQuestion: Question?,
    buttonTextRes: Int,
    questionViewModel: QuestionViewModel,
    onDone: () -> Unit,
    onRemoveDone: (() -> Unit)? = null
) {
    var textValue by rememberSaveable { mutableStateOf(initialQuestion?.questionText ?: "") }
    var isAnswerTrue by rememberSaveable { mutableStateOf(initialQuestion?.answer ?: false)}
    val onSave = {
        val question = initialQuestion?.copy(questionText = textValue, answer = isAnswerTrue)
            ?: Question(questionText = textValue, answer = isAnswerTrue)
        if (initialQuestion == null) questionViewModel.onAddQuestionClick(question)
        else questionViewModel.onUpdateQuestionClick(question)
        onDone()
    } ...
```

QuestionScreen State Hoisting

```
...
val onRemove = initialQuestion?.let { q ->
    {
        questionViewModel.onRemoveQuestionClick(q)
        onRemoveDone?.invoke()
        Unit
    }
}

    QuestionScreen(
        textValue = textValue,
        onTextChange = { textValue = it },
        isAnswerTrue = isAnswerTrue,
        onCheckedChange = { isAnswerTrue = it },
        buttonTextRes = buttonTextRes,
        onSave = onSave,
        onRemove = onRemove,
    )
}
```

- В результаті використання QyestionScreen у нпафі навігації буде простішим:

NavGeoQuiz - State Hoisting

```
package ua.edu.znu.geoquizcomposeedu.nav

import ...

@Composable
fun NavPassQuestion(
    innerPadding: PaddingValues,
    snackbarHostState: SnackbarHostState,
    navController: NavHostController
) {
    val questionRepository =
        (LocalContext.current.applicationContext as GeoQuizApplication).questionRepository
    val onNavigateBack: () -> Unit = { navController.popBackStack() }

    NavHost(
        navController = navController,
        startDestination = Routes.Home,
        modifier = Modifier.padding(innerPadding)
    ) { ...
```



NavGeoQuiz - State Hoisting

```
...
    composable<Routes.Home> {
        val mainViewModel: MainViewModel = viewModel(
            factory = ViewModelFactory(MainViewModel::class.java) {
                MainViewModel(
                    questionRepository
                )
            }
        )
        val mainScreenState by
                                mainViewModel.mainScreenState.collectAsStateWithLifecycle()
        MainScreen(
            snackbarHostState,
            mainScreenState,
            getQuestionText = { index ->
                mainViewModel.getQuestionByIndex(index).questionText },
            onAnswerButtonClick = { answer -> mainViewModel.onAnswerButtonClick(answer) },
            onNextQuestionButtonClick = { mainViewModel.onNextQuestionButtonClick() }
        )
    }
...
```

Стан у викликаючій функції

NavGeoQuiz - State Hoisting

```
...
    composable<Routes.QuestionList> {
        val questionListViewModel: QuestionListViewModel = viewModel(
            factory = ViewModelFactory(QuestionListViewModel::class.java) {
                QuestionListViewModel(questionRepository)
            }
        )
        val questionList: List<Question> by
                                questionListViewModel.questionListFlow.collectAsStateWithLifecycle()
        QuestionListScreen(
            questionList = questionList,
            onEditQuestionClick = { question ->
                navController.navigate(Routes.QuestionEdit(question))
            },
            onAdd = {
                navController.navigate(Routes.QuestionAdd)
            }
        )
    } ...
```

Стан у викликаючій функції



NavGeoQuiz - State Hoisting

```
...
    composable<Routes.QuestionEdit>(
        typeMap = mapOf(
            typeOf<Question>() to QuestionNavType.questionType,
        )
    ) { backStackEntry ->
        val route = backStackEntry.toRoute<Routes.QuestionEdit>()
        val initialQuestion: Question? = route.question
        val questionViewModel: QuestionViewModel = viewModel(
            factory = ViewModelFactory(QuestionViewModel::class.java) {
                QuestionViewModel(questionRepository)
            }
        )
        QuestionScreenHost(
            initialQuestion = initialQuestion,
            buttonTextRes = R.string.update_question,
            questionViewModel = questionViewModel,
            onDone = onNavigateBack,
            onRemoveDone = onNavigateBack
        )
}...
```

Передача ViewModel зі станом

NavGeoQuiz - State Hoisting

```
...
        composable<Routes.QuestionAdd> {
            val questionViewModel: QuestionViewModel = viewModel(
                factory = ViewModelFactory(QuestionViewModel::class.java) {
                    QuestionViewModel(questionRepository)
                }
            )
            QuestionScreenHost(
                initialQuestion = null,
                buttonTextRes = R.string.add_question,
                questionViewModel = questionViewModel,
                onDone = onNavigateBack
            )
        }
    }
}
```

Передача ViewModel зі станом



MainActivity - State Hoisting

```
package ua.edu.znu.geoquizcomposeedu

import ...

class MainActivity : ComponentActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {
            GeoQuizComposeEduTheme {
                val navController = rememberNavController()
                val snackbarHostState = remember { SnackbarHostState() }
                Scaffold(
                    topBar = {
                        GeoQuizTopAppBar()
                    },
                    bottomBar = {
                        BottomAppBar(navController)
                    },
                    snackbarHost = {
                        AppSnackbar(snackbarHostState = snackbarHostState)
                    }, ...
```



MainActivity - State Hoisting

```
...
            ) { innerPadding ->
                // main content
                    NavPassQuestion(innerPadding, snackbarHostState, navController)
            }
        }
    }
}
```

