POLITECHNIKA RZESZOWSKA
KATEDRA INFORMATYKI I AUTOMATYKI
DR INŻ. MATEUSZ POMIANEK

ĆWICZENIE LABORATORYJNE - KOTLIN 3

Room Database i Kotlin Coroutines

Programowanie Aplikacji Mobilnych (Android / Kotlin / Jetpack Compose)

Spis treści

1. Kotlin Coroutines....1
2. Room Database - architektura....4
3. Konfiguracja Room w projekcie....5
4. Implementacja warstwy danych....6
5. ViewModel z Repository....9
6. Migracje schematu bazy danych....11
7. Testy jednostkowe DAO....12
8. Zadania do wykonania....13
10. Najczęstsze błędy i rozwiązania....15
11. Narzędzie diagnostyczne Database Inspector....15
12. Sprawozdanie i repozytorium....16

1. Kotlin Coroutines

W aplikacjach mobilnych konieczne jest przechowywanie danych lokalnie. System Android wykorzystuje bazę SQLite jako podstawowy mechanizm trwałego zapisu danych. Praca bezpośrednio z SQLite wymaga jednak pisania dużej ilości kodu pomocniczego oraz ręcznego mapowania obiektów na rekordy bazy danych. Biblioteka Jetpack Room upraszcza ten proces poprzez wprowadzenie warstwy abstrakcji nad SQLite.

Room integruje się z Kotlin Coroutines oraz Flow, co pozwala budować reaktywne aplikacje mobilne zgodne z architekturą MVVM.

Kotlin Coroutines to natywny mechanizm asynchroniczności w Kotlinie. Zastępuje Callbacki, RxJava i AsyncTask. Pozwalają pisać asynchroniczny kod sekwencyjnie - bez zagnieżdżonych callbacków.

1.1. Dlaczego Coroutines?

Problem	Operacje sieciowe i bazodanowe blokują wątek UI. Aplikacja się zawiesza (ANR, *Application Not Responding*). Wątek główny (UI/Main) powinien wykonywać operacje krótsze niż 16 ms.
Stare podejście	AsyncTask (deprecated), Handler, Thread to dużo powtarzalnego kodu (boilerplate), trudne w testowaniu, podatne na wycieki pamięci (*memory leaks*).

Coroutines	Lekkie współbieżne zadania (możliwe setki tysięcy jednocześnie), nieblokujące, anulowalne (*cancellable*), łatwe do testowania. Kompilator przekształca kod do postaci maszyny stanów (*state machine*) – brak kosztów tworzenia wątków systemowych.
suspend fun	Funkcja oznaczona jako suspend może zostać wstrzymana bez blokowania wątku. Kompilator Kotlin generuje dla niej maszynę stanów (*state machine*). Może być wywoływana tylko z poziomu innej funkcji suspend lub w obrębie coroutine.
Scope	Definiuje cykl życia coroutine. Coroutine żyje tak długo jak przypisany jej scope (np. ViewModelScope). Automatyczne anulowanie (*auto-cancel*) przy zniszczeniu ViewModel czyli brak wycieków pamięci.
Dispatcher	Określa, na jakim wątku wykonywana jest coroutine: • Dispatchers.Main – wątek UI (np. aktualizacja widoku), • Dispatchers.IO – operacje I/O (sieć, baza danych, pliki), • Dispatchers.Default – operacje CPU (np. sortowanie, parsowanie).

1.2. Przykłady Coroutines

Coroutine uruchomiona w viewModelScope jest powiązana z cyklem życia ViewModel.

Kotlin - Coroutines basics

```
viewModelScope.launch {

    val tasks = repository.getTasks()

    _uiState.value = tasks
}
```

1.3. Reaktywny strumień danych Flow

Flow jest reaktywnym odpowiednikiem List,- emituje wartości w czasie. Flow jest asynchronicznym strumieniem danych emitowanych w czasie. Room potrafi zwracać wyniki zapytań w postaci Flow. Room zwraca Flow<List<Task>>, co oznacza że każda zmiana w bazie aktualizuje UI.

Kotlin – podstawy Flow

```
// suspend fun zwraca jedną wartość
suspend fun getUser(): User = ...

// Flow emituje wiele wartości w czasie
fun getTasks(): Flow<List<Task>> = ...

// Operatory Flow
val tasksFlow: Flow<List<Task>> = repository.getTasks()

// map – transformacja każdej emitowanej wartości
val sortedTasksFlow: Flow<List<Task>> =
    tasksFlow.map { tasks ->
        tasks.sortedBy { it.text }
    }

// filter (przez map) – filtrowanie elementów wewnątrz emisji
val activeTasksFlow: Flow<List<Task>> =
    tasksFlow.map { tasks ->
        tasks.filter { !it.isDone }
    }
```

```
// combine – łączenie dwóch Flow
val filteredTasksFlow: Flow<List<Task>> =
    combine(tasksFlow, filterFlow) { tasks, filter ->
        when (filter) {
            Filter.ALL    -> tasks
            Filter.ACTIVE -> tasks.filter { !it.isDone }
            Filter.DONE   -> tasks.filter { it.isDone }
        }
    }

// ViewModel – StateFlow z bazy danych
class TaskViewModel(
    private val repository: TaskRepository
) : ViewModel() {

    // stateIn – konwersja Flow → StateFlow
    // zawsze posiada wartość i jest thread-safe
    val tasksState: StateFlow<List<Task>> =
        repository.getAllTasks()
            .stateIn(
                scope = viewModelScope,
                started = SharingStarted.WhileSubscribed(5_000),
                initialValue = emptyList()
            )
}

// Composable (Jetpack Compose)
// Automatyczna aktualizacja UI przy zmianach danych (np. Room)
@Composable
fun TaskScreen(viewModel: TaskViewModel) {
    val tasks by viewModel.tasksState.collectAsStateWithLifecycle()
    // UI wykorzystujący tasks
}
```

2. Room Database - architektura

Room to biblioteka Jetpack zapewniająca warstwę abstrakcji nad SQLite. Eliminuje boilerplate SQL, weryfikuje zapytania w czasie kompilacji i integruje się natywnie z Flow i Coroutines.

2.1. Trzy składniki Room

@Entity - tabela bazy danych

```
@Entity(tableName = "tasks")                    ← anotacja klasy
data class TaskEntity(                          ← Kotlin data class
    @PrimaryKey val id: String,                 ← klucz główny
    @ColumnInfo(name = "task_text") val text: String,
    val isDone: Boolean = false,                ← nazwa kolumny = nazwa pola
    val priority: Int = 1,                      ← enum → Int (Room nie obsługuje enum natywnie)
    val createdAt: Long = System.currentTimeMillis()
)
```

@Dao - Data Access Object (interfejs)

```
@Dao                                                    ← anotacja interfejsu
interface TaskDao {
    @Query("SELECT * FROM tasks ORDER BY createdAt DESC")
    fun getAllTasks(): Flow<List<TaskEntity>>           ← Flow = reaktywne zapytanie
```

```
    @Query("SELECT * FROM tasks WHERE id = :taskId")
    suspend fun getTaskById(taskId: String): TaskEntity?

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertTask(task: TaskEntity)

    @Update  suspend fun updateTask(task: TaskEntity)
    @Delete  suspend fun deleteTask(task: TaskEntity)

    @Query("DELETE FROM tasks WHERE isDone = 1")
    suspend fun deleteCompletedTasks()
}
```

@Database - abstrakcyjna klasa bazy danych

```
@Database(
    entities = [TaskEntity::class],                ← lista tabel
    version = 1,                                   ← wersja schematu
    exportSchema = true                            ← zapisuje schemat do JSON (wymagane!)
)
abstract class AppDatabase : RoomDatabase() {
    abstract fun taskDao(): TaskDao                ← Room generuje implementację

    companion object {
        @Volatile private var INSTANCE: AppDatabase? = null
        fun getInstance(context: Context): AppDatabase {
            return INSTANCE ?: synchronized(this) {
                Room.databaseBuilder(context, AppDatabase::class.java, "task_database")
                    .build().also { INSTANCE = it }
            }
        }
    }
}
```

Jak Room przetwarza @Query?

Kompilator Room (kapt/ksp) czyta anotacje @Query w czasie kompilacji i generuje implementację w Javie. Jeśli SQL zawiera błąd (literówka w nazwie tabeli/kolumny). BUILD FAILED, a nie crash w runtime. To jest główna przewaga Room nad czystym SQLite.

Zapytanie zwracające `Flow<List<T>>` jest 'żywym' zapytaniem. Room automatycznie ponownie wykonuje je gdy dane w tabeli się zmienią i emituje nowy wynik do wszystkich obserwatorów.

2.1. Porównanie SQLite vs Room

Cecha	SQLite	Room
Poziom abstrakcji	Niski, bezpośrednie zapytania SQL	Wyższy, ORM nad SQLite
Mapowanie obiektów	Ręczne	Automatyczne
Weryfikacja zapytań	Runtime	Compile time
Integracja z Coroutines	Brak natywnej	Pełna integracja
Integracja z Flow	Brak	Tak
Migracje	Ręczne	Obsługiwane przez Room

Room w rzeczywistości nadal korzysta z SQLite, ale upraszcza dostęp do danych i pracę z bazą SQLite, dlatego, że integruje się z nowoczesną architekturą aplikacji Android i jego narzędziami, takimi jak Coroutines, Flow czy Paging.

3. Konfiguracja Room w projekcie

3.1. dodaj Room w libs.versions.toml

TOML - libs.versions.toml

```
[versions] # Wersje
room = "2.6.1"
ksp  = "2.0.21-1.0.28"

[libraries] # Biblioteki
room-runtime  = { group = "androidx.room", name = "room-runtime",  version.ref = "room" }
room-ktx      = { group = "androidx.room", name = "room-ktx",      version.ref = "room" }
room-compiler = { group = "androidx.room", name = "room-compiler", version.ref = "room" }
room-testing  = { group = "androidx.room", name = "room-testing",  version.ref = "room" }

[plugins] # Pluginy
ksp = { id = "com.google.devtools.ksp", version.ref = "ksp" }
```

3.2. KSP plugin i zależności build.gradle.kts

Kotlin DSL - build.gradle.kts (fragment)

```
plugins {
   // istniejące pluginy...
   alias(libs.plugins.ksp)                                  // KSP dla Room
}
ksp {                                                          // Konfiguracja KSP (schemat bazy)
   arg("room.schemaLocation", "$projectDir/schemas")
}
dependencies {                                                 // Zależności
   implementation(libs.room.runtime)                        // Room
   implementation(libs.room.ktx)
   ksp(libs.room.compiler)                                  // KSP – generator kodu DAO
   testImplementation(libs.room.testing)                    // Testy Room
}
```

Wersja KSP musi pasować do Kotlin!

KSP ma format: <kotlin-version>-<ksp-version>. Dla Kotlin 2.0.21 użyj KSP 2.0.21-1.0.28.
Sprawdź aktualne wersje na: https://github.com/google/ksp/releases.
Niezgodność wersji objawia się błędem: *"KSP and Kotlin versions don't match"* podczas Gradle Sync.

4. Implementacja warstwy danych

Implementacja podzielona jest na 4 pliki w pakiecie data/. To separacja warstw zgodna z Clean Architecture. ViewModel nie wie nic o Room, Repository abstrahuje źródło danych. Entity reprezentuje tabelę w bazie danych.

Kotlin - Struktura projektu

```
app/src/main/java/pl/edu/uczelnia/taskapp/
├── data/
│   ├── local/
│   │   ├── TaskEntity.kt          ← @Entity - definicja tabeli
│   │   ├── TaskDao.kt             ← @Dao - zapytania SQL
│   │   └── AppDatabase.kt         ← @Database - singleton bazy
│   └── repository/
│       ├── TaskRepository.kt      ← interfejs (abstrakcja)
│       └── TaskRepositoryImpl.kt  ← implementacja z Room
├── domain/
│   └── model/
│       └── Task.kt                ← domain model (używany w ViewModel/UI)
├── ui/ ...                        ← ekrany z Lab 2 (bez zmian)
└── TaskViewModel.kt               ← rozszerzona o Repository
```

4.1. Definicja tabeli TaskEntity

Kotlin - data/local//TaskEntity.kt

```
package pl.edu.prz.taskapp.data.local
import androidx.room.ColumnInfo
import androidx.room.Entity
import androidx.room.Index
import androidx.room.PrimaryKey

@Entity(
   tableName = "tasks",
   indices = [
      Index(value = ["created_at"]) // indeks przyspiesza sortowanie
   ]
)
data class TaskEntity(
   @PrimaryKey
   val id: String, // UUID jako String

   @ColumnInfo(name = "task_text")
   val text: String,

   @ColumnInfo(name = "is_done", defaultValue = "0")
   val isDone: Boolean = false,

   @ColumnInfo(name = "priority", defaultValue = "1")
```

```
    val priority: Int = 1, // 0=LOW, 1=NORMAL, 2=HIGH

    @ColumnInfo(name = "created_at")
    val createdAt: Long = System.currentTimeMillis(),

    @ColumnInfo(name = "updated_at")
    val updatedAt: Long = System.currentTimeMillis()
)
// TypeConverter – jeśli potrzebujesz enum w bazie
// Room domyślnie nie obsługuje enumów.
```

4.2. TaskDao - Data Access Object

Kotlin - TaskDao.kt

```
@Dao
interface TaskDao {

    @Query("SELECT * FROM tasks ORDER BY createdAt DESC")
    fun getAll(): Flow<List<TaskEntity>>

    @Query("SELECT * FROM tasks WHERE id = :id")
    suspend fun getById(id: String): TaskEntity?

    @Insert
    suspend fun insert(task: TaskEntity)

    @Update
    suspend fun update(task: TaskEntity)

    @Query("DELETE FROM tasks WHERE id = :id")
    suspend fun delete(id: String)
}
```

4.3. AppDatabase - singleton bazy danych

Kotlin - data/local/AppDatabase.kt

```
package pl.edu.uczelnia.taskapp.data.local

import android.content.Context
import androidx.room.*
import androidx.room.migration.Migration
import androidx.sqlite.db.SupportSQLiteDatabase

@Database(
    entities = [TaskEntity::class],
    version = 1,
    exportSchema = true
)
abstract class AppDatabase : RoomDatabase() {
```

```
    abstract fun taskDao(): TaskDao

    companion object {
        @Volatile
        private var INSTANCE: AppDatabase? = null

        fun getInstance(context: Context): AppDatabase =
            INSTANCE ?: synchronized(this) {
                INSTANCE ?: buildDatabase(context).also { INSTANCE = it }
            }

        private fun buildDatabase(context: Context): AppDatabase =
            Room.databaseBuilder(
                context.applicationContext,
                AppDatabase::class.java,
                "task_database.db"
            )
                // .addMigrations(MIGRATION_1_2)
                // .fallbackToDestructiveMigration() // tylko DEV
                .build()

        // Przykład migracji 1 → 2
        val MIGRATION_1_2 = object : Migration(1, 2) {
            override fun migrate(db: SupportSQLiteDatabase) {
                db.execSQL(
                    "ALTER TABLE tasks ADD COLUMN category TEXT NOT NULL DEFAULT 'general'"
                )
            }
        }
    }
}
```

4.4. Domain model: Task

Kotlin - domain/model/Task.kt (domain model)

```
package pl.edu.uczelnia.taskapp.domain.model
import java.util.UUID

enum class Priority(val value: Int) {
    LOW(0),
    NORMAL(1),
    HIGH(2)
}

data class Task(
    val id: String = UUID.randomUUID().toString(),
```

```
    val text: String,
    val isDone: Boolean = false,
    val priority: Priority = Priority.NORMAL,
    val createdAt: Long = System.currentTimeMillis()
)

// ── Mapowanie Entity <-> Domain model (w Repository) ──────────────────────

fun TaskEntity.toDomainModel(): Task = Task(
    id        = this.id,
    text      = this.text,
    isDone    = this.isDone,
    priority  = Priority.entries.find { it.value == this.priority } ?: Priority.NORMAL,
    createdAt = this.createdAt
)

fun Task.toEntity(): TaskEntity = TaskEntity(
    id        = this.id,
    text      = this.text,
    isDone    = this.isDone,
    priority  = this.priority.value,
    createdAt = this.createdAt,
    updatedAt = System.currentTimeMillis()
)
```

4.5. Warstwa Repository: abstrakcja źródła danych

Kotlin – data/repository/TaskRepository.kt

```
class TaskRepository(
    private val dao: TaskDao
) {
    fun getTasks(): Flow<List<Task>> =
        dao.getAll().map { list ->
            list.map { it.toDomain() }
        }
    suspend fun save(task: Task) {
        dao.insert(task.toEntity())
    }
    suspend fun delete(id: String) {
        dao.delete(id)
    }
}
```

4.6. Operacja Upsert

Upsert wstawia rekord do bazy lub aktualizuje go jeśli już istnieje.

```
@Dao
interface TaskDao {
```

```
@Upsert
suspend fun upsert(task: TaskEntity)
```

4.7. Paging: obsługa dużych zbiorów danych

Paging umożliwia ładowanie danych partiami.

```
@Query("SELECT * FROM tasks ORDER BY createdAt DESC")
fun pagingSource(): PagingSource<Int, TaskEntity>
```

5. ViewModel z Repository

5.1. TaskViewModel - wersja z Room

Kotlin - TaskViewModel.kt (z Room)

```
package pl.edu.uczelnia.taskapp

import androidx.lifecycle.ViewModel
import androidx.lifecycle.ViewModelProvider
import androidx.lifecycle.viewModelScope
import kotlinx.coroutines.ExperimentalCoroutinesApi
import kotlinx.coroutines.FlowPreview
import kotlinx.coroutines.flow.*
import kotlinx.coroutines.launch
import pl.edu.uczelnia.taskapp.data.repository.TaskRepository
import pl.edu.uczelnia.taskapp.domain.model.Task
import pl.edu.uczelnia.taskapp.domain.model.Priority

enum class SortOrder { DATE_DESC, DATE_ASC, PRIORITY_DESC }

data class TaskUiState(
    val tasks: List<Task> = emptyList(),
    val filter: Filter = Filter.ALL,
    val searchQuery: String = "",
    val sortOrder: SortOrder = SortOrder.DATE_DESC,
    val isLoading: Boolean = false,
    val activeCount: Int = 0,
    val totalCount: Int = 0,
    val snackbarMessage: String? = null
)

@OptIn(ExperimentalCoroutinesApi::class, FlowPreview::class)
class TaskViewModel(private val repository: TaskRepository) : ViewModel() {

    private val _filter = MutableStateFlow(Filter.ALL)
    private val _searchQuery = MutableStateFlow("")
    private val _sortOrder = MutableStateFlow(SortOrder.DATE_DESC)

    val uiState: StateFlow<TaskUiState> = combine(
        repository.getAllTasks(),
        repository.getActiveTaskCount(),
        _filter,
        _searchQuery.debounce(300),
        _sortOrder
```

```
    ) { tasks, activeCount, filter, query, sortOrder ->
        val filtered = tasks
            .filter { task ->
                val matchesFilter = when (filter) {
                    Filter.ALL -> true
                    Filter.ACTIVE -> !task.isDone
                    Filter.DONE -> task.isDone
                }
                matchesFilter && (query.isBlank() || task.text.contains(query, ignoreCase = true))
            }
            .let { list ->
                when (sortOrder) {
                    SortOrder.DATE_DESC -> list.sortedByDescending { it.createdAt }
                    SortOrder.DATE_ASC -> list.sortedBy { it.createdAt }
                    SortOrder.PRIORITY_DESC -> list.sortedByDescending { it.priority.value }
                }
            }
        TaskUiState(
            tasks = filtered,
            filter = filter,
            searchQuery = query,
            sortOrder = sortOrder,
            activeCount = activeCount,
            totalCount = tasks.size
        )
    }.stateIn(
        scope = viewModelScope,
        started = SharingStarted.WhileSubscribed(5_000),
        initialValue = TaskUiState(isLoading = true)
    )

    // ── Akcje ──────────────────────────────────────
    fun addTask(text: String, priority: Priority = Priority.NORMAL) {
        if (text.isBlank()) return
        viewModelScope.launch {
            repository.saveTask(Task(text = text.trim(), priority = priority))
        }
    }

    fun toggleTask(taskId: String) = viewModelScope.launch { repository.toggleTask(taskId) }
    fun deleteTask(taskId: String) = viewModelScope.launch { repository.deleteTask(taskId) }
    fun deleteCompleted() = viewModelScope.launch { repository.deleteCompletedTasks() }

    fun setFilter(filter: Filter) { _filter.value = filter }
    fun setSearchQuery(query: String) { _searchQuery.value = query }
    fun setSortOrder(order: SortOrder) { _sortOrder.value = order }

    // ── Factory bez Hilt ──────────────────────────
    companion object {
        fun factory(repository: TaskRepository): ViewModelProvider.Factory =
            object : ViewModelProvider.Factory {
                @Suppress("UNCHECKED_CAST")
                override fun <T : ViewModel> create(modelClass: Class<T>): T =
                    TaskViewModel(repository) as T
            }
    }
}
```

5.2. Inicjalizacja w MainActivity

Kotlin - MainActivity.kt + AppNavigation.kt (fragment)

```kotlin
// MainActivity.kt
package pl.edu.uczelnia.taskapp

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.activity.enableEdgeToEdge
import androidx.lifecycle.ViewModelProvider
import pl.edu.uczelnia.taskapp.data.local.AppDatabase
import pl.edu.uczelnia.taskapp.data.repository.TaskRepositoryImpl
import pl.edu.uczelnia.taskapp.ui.theme.TaskAppTheme

class MainActivity : ComponentActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()

        // 1. Inicjalizacja bazy danych
        val database = AppDatabase.getInstance(applicationContext)
        val dao = database.taskDao()
        val repository = TaskRepositoryImpl(dao)

        // 2. Fabryka ViewModel z repository
        val viewModelFactory = TaskViewModel.factory(repository)

        setContent {
            TaskAppTheme {
                // 3. Przekazanie factory do AppNavigation
                AppNavigation(viewModelFactory = viewModelFactory)
            }
        }
    }
}
```

Hilt vs ręczne DI

W tym ćwiczeniu używamy ręcznego wstrzykiwania zależności (manual DI) - przekazujemy Repository przez konstruktor/factory. To prostszy sposób bez dodatkowych bibliotek.

W projektach produkcyjnych zalecane jest Hilt (Dagger) lub Koin. Hilt integruje się z ViewModel przez `@HiltViewModel` + `@Inject constructor(repository: TaskRepository)` - eliminuje ViewModelFactory. Hilt będzie tematem Ćwiczenia 5.

6. Migracje schematu bazy danych

Gdy zmieniasz strukturę tabeli (dodajesz kolumnę, zmieniasz typ), musisz napisać migrację. Bez migracji Room rzuca IllegalStateException i aplikacja crashuje przy starcie.

6.1. Kiedy wymagana jest migracja?

Dodanie kolumny	ALTER TABLE tasks ADD COLUMN category TEXT NOT NULL DEFAULT 'general'
Zmiana nazwy kolumny	Room nie obsługuje RENAME COLUMN w starej SQLite. Utwórz nową tabelę, skopiuj dane, usuń starą.
Usunięcie kolumny	Brak DROP COLUMN (SQLite < 3.35) → utwórz nową tabelę bez kolumny, skopiuj dane, usuń starą
Dodanie indeksu	CREATE INDEX idx_tasks_created ON tasks(createdAt) (bez rekonstrukcji tabeli)
Nowa tabela	CREATE TABLE tasks_new (id TEXT PRIMARY KEY NOT NULL, text TEXT NOT NULL, isDone INTEGER NOT NULL, priority INTEGER NOT NULL, createdAt INTEGER NOT NULL)

Kotlin - Migracja Room 1→2

```
// 1. Zaktualizuj wersję w AppDatabase
@Database(entities = [TaskEntity::class], version = 2, exportSchema = true)
abstract class AppDatabase : RoomDatabase() { /* ... */ }

// 2. Dodaj pole do TaskEntity
@ColumnInfo(name = "category", defaultValue = "general")
val category: String = "general"

// 3. Napisz migrację 1 → 2
val MIGRATION_1_2 = object : Migration(1, 2) {
    override fun migrate(database: SupportSQLiteDatabase) {
        database.execSQL(
            "ALTER TABLE tasks ADD COLUMN category TEXT NOT NULL DEFAULT 'general'"
        )
    }
}

// 4. Zarejestruj migrację przy tworzeniu bazy
Room.databaseBuilder(
    context.applicationContext,
    AppDatabase::class.java,
    "task_database.db"
)
    .addMigrations(MIGRATION_1_2)
    .build()
```

7. Testy jednostkowe DAO

Room umożliwia testowanie DAO w izolacji przy użyciu bazy danych in-memory. Testy in-memory są szybkie, nie wymagają emulatora (uruchamiane jako zwykłe unit testy JVM) i nie zostawiają danych na urządzeniu.

7.1. Konfiguracja testu

Kotlin - test/java/.../data/local/TaskDaoTest.kt

```
package pl.edu.uczelnia.taskapp.data.local

import androidx.room.Room
import androidx.test.core.app.ApplicationProvider
import androidx.test.ext.junit.runners.AndroidJUnit4
import app.cash.turbine.test
import kotlinx.coroutines.test.runTest
import org.junit.After
import org.junit.Before
import org.junit.Test
import org.junit.runner.RunWith
import kotlin.test.assertEquals
import kotlin.test.assertNotNull
import kotlin.test.assertNull

@RunWith(AndroidJUnit4::class)
class TaskDaoTest {

    private lateinit var database: AppDatabase
    private lateinit var dao: TaskDao

    @Before
    fun setUp() {
        database = Room.inMemoryDatabaseBuilder(
            ApplicationProvider.getApplicationContext(),
            AppDatabase::class.java
        )
            .allowMainThreadQueries() // tylko w testach
            .build()
        dao = database.taskDao()
    }

    @After
    fun tearDown() {
        database.close()
    }
    // ── Testy ──────────────────────────────────────
    @Test
    fun insertTask_getById_returnsCorrectTask() = runTest {
        val task = TaskEntity(id = "1", text = "Test task", isDone = false, priority = 1)
        dao.insertTask(task)

        val result = dao.getTaskById("1")
        assertNotNull(result)
        assertEquals("Test task", result.text)
        assertEquals(false, result.isDone)
    }
    @Test
    fun toggleTask_updatesIsDoneStatus() = runTest {
        val task = TaskEntity(id = "2", text = "Toggle me", isDone = false, priority = 1)
        dao.insertTask(task)
        dao.updateTaskStatus(id = "2", isDone = true)

        val result = dao.getTaskById("2")
        assertEquals(true, result?.isDone)
    }
    @Test
    fun deleteTask_removesFromDatabase() = runTest {
        val task = TaskEntity(id = "3", text = "Delete me", isDone = false, priority = 1)
        dao.insertTask(task)
        dao.deleteTaskById("3")

        val result = dao.getTaskById("3")
        assertNull(result)
    }
    @Test
    fun deleteCompleted_removesOnlyDoneTasks() = runTest {
        dao.insertTasks(
```

```
            listOf(
                TaskEntity(id = "a", text = "Active", isDone = false, priority = 1),
                TaskEntity(id = "b", text = "Done 1", isDone = true, priority = 1),
                TaskEntity(id = "c", text = "Done 2", isDone = true, priority = 1)
            )
        )
        val deletedCount = dao.deleteCompletedTasks()
        assertEquals(2, deletedCount)

        val remaining = dao.getTaskById("a")
        assertNotNull(remaining)
        assertNull(dao.getTaskById("b"))
    }
    @Test
    fun getAllTasks_returnsTasksInDescendingOrder() = runTest {
        dao.insertTask(TaskEntity(id = "x1", text = "First", isDone = false, priority = 1, createdAt = 1000))
        dao.insertTask(TaskEntity(id = "x2", text = "Second", isDone = false, priority = 1, createdAt =
2000))
        dao.getAllTasks().test {
            val tasks = awaitItem()
            assertEquals(2, tasks.size)
            assertEquals("Second", tasks[0].text) // DESC: nowsze pierwsze
            cancelAndIgnoreRemainingEvents()
        }
    }
}
```

Biblioteka Turbine do testowania Flow

Turbine (https://github.com/cashapp/turbine) to popularna biblioteka do testowania Flow w Kotlinie. Dodaj zależność: testImplementation("app.cash.turbine:turbine:1.2.0") Bez Turbine możesz użyć: flow.first() (jednorazowe pobranie) lub flow.take(1).toList().

8. Zadania do wykonania

Kontynuujesz projekt TaskApp z Ćwiczenia 2. Zastępujesz stan in-memory Room Database. Aplikacja po zamknięciu i ponownym otwarciu powinna zachować wszystkie zadania.

Zadanie 1 - Konfiguracja Room

Wymagania

1.1. Dodaj Room + KSP do libs.versions.toml i build.gradle.kts. Pokaż prowadzącemu BUILD SUCCESSFUL po Sync Gradle.

1.2. Utwórz TaskEntity.kt z @Entity(tableName = "tasks") i polami: id (PrimaryKey), text, isDone, priority (Int), createdAt (Long). Minimum 5 pól.

1.3. Utwórz TaskDao.kt z metodami: getAllTasks() (Flow), insertTask(), updateTask() lub updateTaskStatus(), deleteTaskById().

1.4. Utwórz AppDatabase.kt z @Database(entities = [TaskEntity::class], version = 1, exportSchema = true). Singleton getInstance().

1.5. Sprawdź w Android Studio: View → Tool Windows → App Inspection → Database Inspector - baza powinna być widoczna po uruchomieniu aplikacji.

Zadanie 2 - Repository i ViewModel

Wymagania

2.1 Utwórz interfejs TaskRepository z metodami odpowiadającymi operacjom DAO + mapowaniem na domain model Task.

2.2 Zaimplementuj TaskRepositoryImpl(taskDao: TaskDao) : TaskRepository. Mapuj TaskEntity ↔ Task przez funkcje rozszerzające toDomainModel() i toEntity().

2.3 Zaktualizuj TaskViewModel - wstrzyknij TaskRepository przez konstruktor. Zastąp MutableStateFlow(lista) przez repository.getAllTasks().stateIn(...).

2.4 Upewnij się że addTask, toggleTask, deleteTask wywołują viewModelScope.launch { repository.xxx() }.

2.5 Zaktualizuj MainActivity - utwórz AppDatabase, TaskRepositoryImpl, ViewModelFactory i przekaż do AppNavigation.

Zadanie 3 - Integracja z UI i trwałość danych

Wymagania

3.1 Uruchom aplikację, dodaj kilka zadań. Zamknij aplikację (swipe up w recent apps - nie tylko wciśnij Home). Otwórz ponownie - zadania muszą być zachowane.

3.2 Sprawdź Database Inspector: View → Tool Windows → App Inspection → Database Inspector. Kliknij tabelę tasks - pokaż prowadzącemu dane wstawione przez aplikację.

3.3 Wgraj dane testowe przy pierwszym uruchomieniu: jeśli baza jest pusta, wstaw 3 przykładowe zadania. Użyj RoomDatabase.Callback (onCreate) lub sprawdź count w ViewModel init {}.

3.4 Dodaj pole SearchBar (OutlinedTextField lub SearchBar z M3) do HomeScreen. Wpisywanie tekstu filtruje zadania w czasie rzeczywistym (debounce 300ms przez _searchQuery w ViewModel).

3.5 Obroć emulator (Ctrl+F11 lub Device Manager → Rotate). Zweryfikuj że lista zadań się nie zresetowała - ViewModel przeżył rotację dzięki Room.

Zadanie 4 - Migracja i testy

Wymagania

4.1 Dodaj kolumnę category (TEXT NOT NULL DEFAULT 'general') do TaskEntity. Zaktualizuj version = 2 w @Database. Napisz MIGRATION_1_2 z ALTER TABLE.

4.2 Zarejestruj migrację w AppDatabase.buildDatabase() przez .addMigrations(MIGRATION_1_2). Uruchom aplikację - dane z poprzedniej sesji muszą być zachowane (nie wolno używać fallbackToDestructiveMigration!).

4.3 Napisz minimum 3 testy jednostkowe DAO w TaskDaoTest.kt używając Room.inMemoryDatabaseBuilder. Testy: insertTask, deleteTask, deleteCompletedTasks.

4.4 Uruchom testy: kliknij prawym na klasę TaskDaoTest → Run 'TaskDaoTest'. Pokaż prowadzącemu zielony pasek w test runner.

10. Najczęstsze błędy i rozwiązania

Brak getterów / błędna definicja encji	Upewnij się, że **TaskEntity** jest data class z właściwościami w konstruktorze (np. val, nie pola w stylu Java). Room wymaga publicznych **getterów** – w Kotlin są generowane automatycznie dla właściwości.
Room nie może zweryfikować schematu	Problem wynika z różnicy między aktualnym kodem a zapisanym exportSchema. Rozwiązania: usuń katalog schemas/ i przebuduj projekt, lub tymczasowo użyj: fallbackToDestructiveMigration() (tylko DEV, bo usuwa wszystkie dane!)
Błędne użycie DAO	Próba uruchomienia DAO bez coroutine lub bez Flow. Sprawdź:

(wywołanie na wątku głównym)	◦ czy metoda DAO ma suspend lub zwraca Flow, ◦ używaj viewModelScope.launch { ... } dla suspend, ◦ lub obserwuj Flow w UI.
Niezgodna wersja KSP	W **libs.versions.toml** wersja KSP musi być zgodna z wersją Kotlin. W razie problemów: **File → Invalidate Caches → Invalidate and Restart →** potem **Gradle Sync**.
Migracja wykonana, ale brak kolumny	Najczęściej błąd w SQL w **Migration.migrate().** Sprawdź zapytanie (np. ALTER TABLE). Dodatkowo: ◦ testuj na czystej instalacji (usuń dane aplikacji/ przeinstaluj APK), ◦ emulator może mieć starą wersję bazy.
Zły typ zwracany w DAO (brak reaktywności)	Użycie **List<T>** zamiast **Flow<List<T>>**: ◦ **List<T>** zwraca jednorazowy wynik (brak aktualizacji), ◦ **Flow<List<T>>** automatycznie emituje zmiany. Poprawnie: ◦ fun getTasks(): Flow<List<TaskEntity>> (bez suspend)

11. Narzędzie diagnostyczne Database Inspector

Database Inspector to wbudowane narzędzie Android Studio pozwalające przeglądać, edytować i wykonywać zapytania SQL na działającej bazie danych uruchomionej aplikacji, bez zatrzymywania procesu.

#	Akcja	Korzystanie z Database Inspector
1	**Otwórz inspektor**	View → Tool Windows → **App Inspection** → zakładka **Database Inspector**. Aplikacja musi być uruchomiona na emulatorze lub urządzeniu.
2	**Wybierz bazę**	Na liście po lewej wybierz task_database. Pojawi się lista tabel: tasks.
3	**Przeglądaj dane**	Kliknij tabelę tasks → widok wierszy. Możesz sortować po kliknięciu nagłówka kolumny. Checkbox 'Live updates' automatycznie odświeża widok gdy aplikacja modyfikuje bazę.
4	**SQL Query**	Kliknij ikonę New Query Session (lub Open new query tab). Wpisz zapytanie: SELECT * FROM tasks WHERE is_done = 0 ORDER BY priority DESC → Execute.
5	**Edycja na żywo**	Kliknij dwukrotnie komórkę - możesz ją edytować bezpośrednio. Naciśnij Enter - zmiana jest natychmiast zapisana do bazy. Aplikacja odczyta nową wartość przez Flow.
6	**Export bazy**	Prawy klik na bazę → Export Database. Pobiera plik .db - możesz go otworzyć w DB Browser for SQLite (dbsqlitebrowser.org) na komputerze.

12. Sprawozdanie i repozytorium

Sprawozdanie nie jest wymagane. *Link do repo na Moodle. Nazwa: pam-lab3-<inicjały>.*

#	Akcja	Checklist przed wysłaniem
1	**Kompilacja**	Build → Make Project → 0 errors. Aplikacja startuje i wyświetla listę zadań.
2	**Trwałość danych**	Dodaj zadanie → zamknij app (Recent Apps → swipe up) → otwórz ponownie → zadanie widoczne.

#	Akcja	Checklist przed wysłaniem
3	**Testy przechodzą**	Run TaskDaoTest → wszystkie testy zielone. Screenshot testu jako dowód (dołącz do README).
4	**Schemat Room**	Sprawdź czy plik app/schemas/pl.edu.../AppDatabase/1.json istnieje i jest commitowany do repo.
5	**README.md**	Opis projektu + instrukcja uruchomienia + screenshot aplikacji + screenshot Database Inspector.
6	**Link na Moodle**	Wklej URL repo w pole zadania na Moodle. Dodaj prowadzącego jako collaboratora.

Instrukcja Kotlin 3: Room Database i Kotlin Coroutines
Programowanie Aplikacji Mobilnych | Katedra Informatyki i Automatyki
MobileHub
Następne ćwiczenie: Kotlin 4 - REST API, Retrofit i Coil