Room

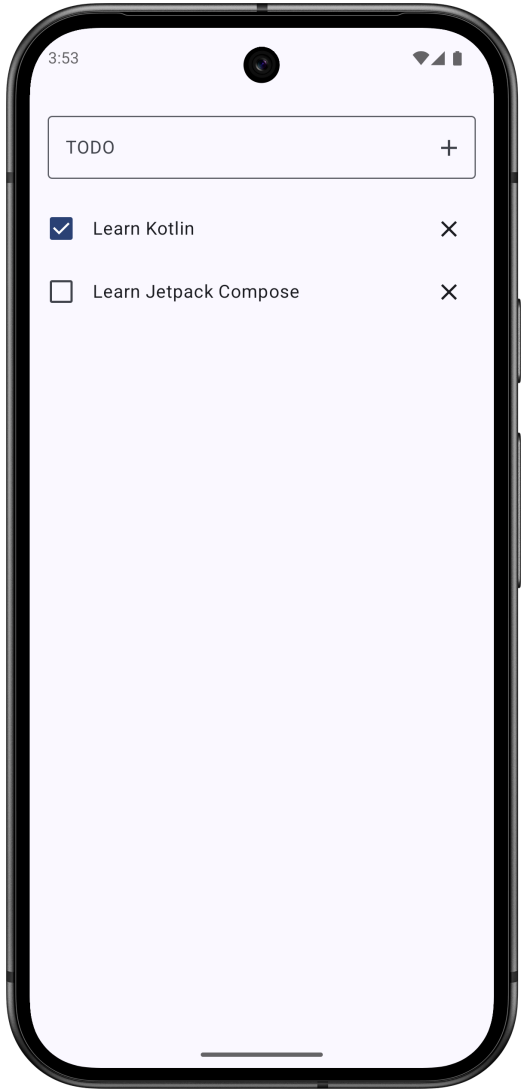
Persistenza dati in un database locale

Laboratorio di oggi

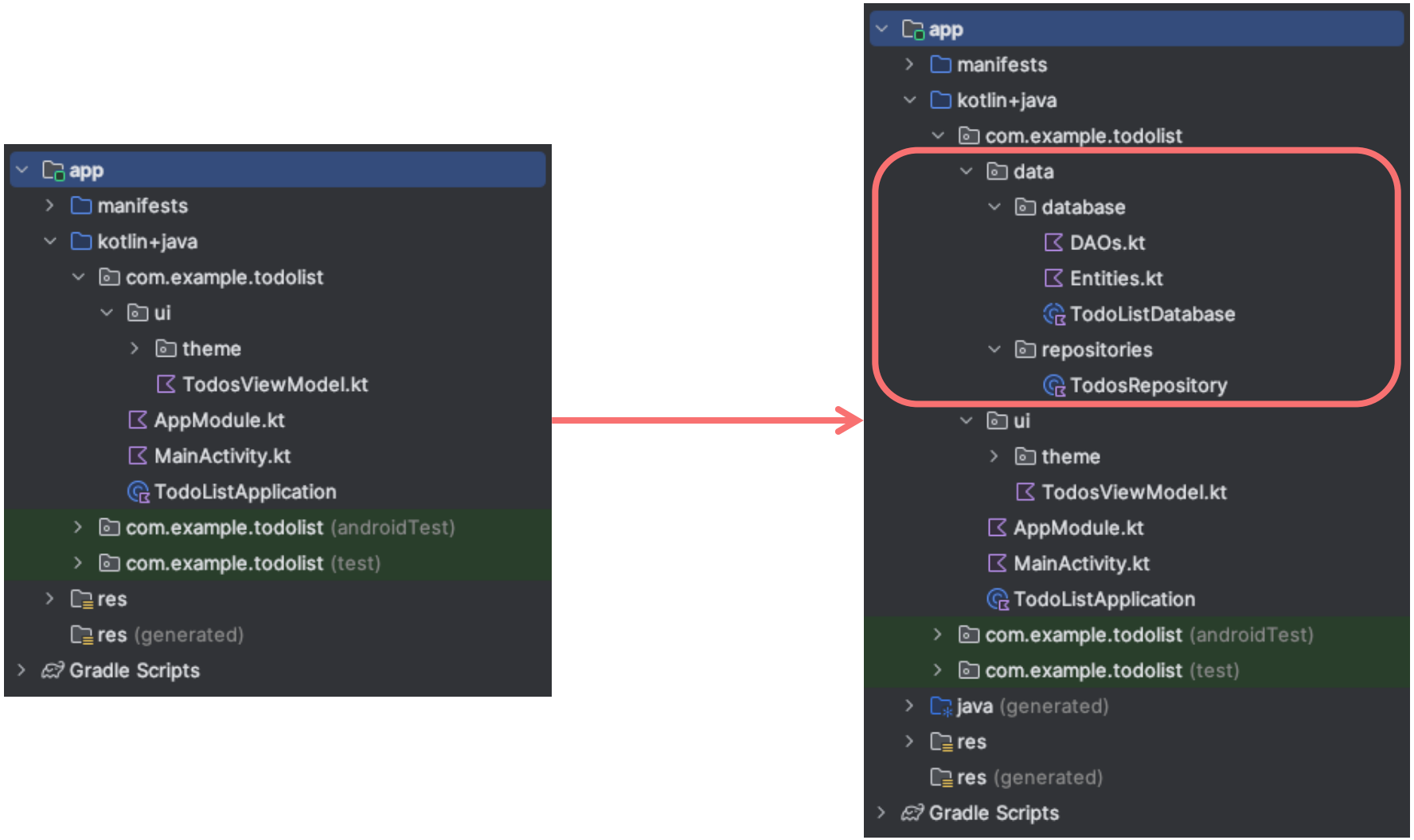
1. Esercitazione guidata – TodoList
   - Punto di partenza: semplice app per gestire una lista di todo, con ViewModel e dependency injection già configurati
   - Esercizio: salvataggio dei dati su un database SQLite locale con Room
2. Esercitazione libera – TravelDiary
   - Salvataggio su Room dei luoghi visitati

1. TodoList

- A partire da un’app per la gestione di una lista di todo, aggiungiamo Room per la persistenza dei dati
- Progetto di partenza su Virtuale: **TodoList.zip**
- Step
  1. Configurazione progetto
  2. Creazione database (entities, DAOs)
  3. Creazione repository
  4. Aggiunta repository al ViewModel



1. Struttura progetto



1.1. Configurazione progetto

- Aggiungiamo KSP e Room al nostro progetto
- **Kotlin Symbol Processing** (**KSP**) è un set di API per sviluppare plugin del compilatore Kotlin
- Room ne ha bisogno per generare codice a partire dalle interfacce e annotazioni che andremo a utilizzare
- Dato che KSP è un plugin Gradle e che Room deve essere configurato per utilizzarlo, l'installazione è diversa dalle librerie viste in precedenza

1.1. Configurazione progetto

libs.versions.toml

```
[versions]
# ...
ksp = "2.0.0-1.0.24"
room = "2.6.1"

[libraries]
# ...
androidx-room-compiler = { module = "androidx.room:room-compiler", version.ref = "room" }
androidx-room-ktx = { module = "androidx.room:room-ktx", version.ref = "room" }
androidx-room-runtime = { module = "androidx.room:room-runtime", version.ref = "room" }

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

1.1. Configurazione progetto

build.gradle.kts (Project: TodoList)

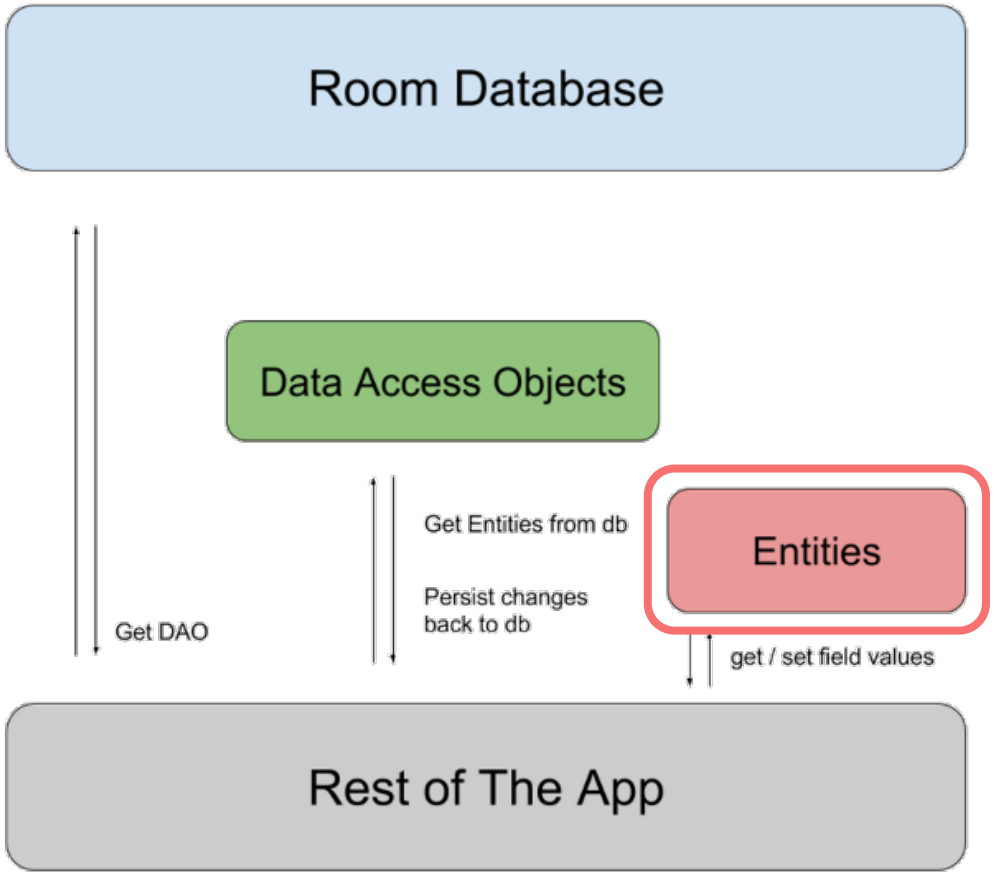
```
plugins {
    // ...
    alias(libs.plugins.ksp) apply false
}
```

build.gradle.kts (Module :app)

```
plugins {
    // ...
    alias(libs.plugins.ksp)
}

// ...

dependencies {
    // ...
    implementation(libs.androidx.room.runtime)
    implementation(libs.androidx.room.ktx)
    annotationProcessor(libs.androidx.room.compiler)
    ksp(libs.androidx.room.compiler)

}
```

1.2. Creazione database – Entities

- Il primo passo è decidere di quante **entities**, o tabelle, abbiamo bisogno
- Nel nostro caso, una sola tabella **Todo** è sufficiente



1.2. Creazione database – Entities

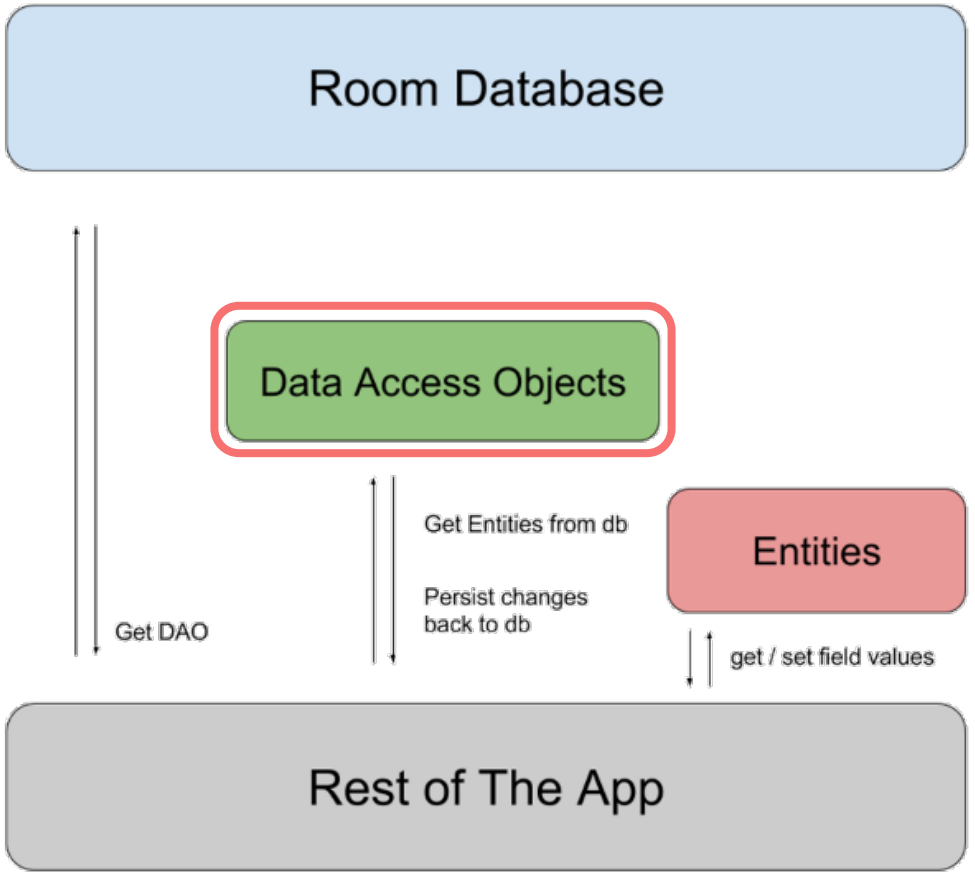
data.database.Entities.kt

```
@Entity
data class Todo(
    @PrimaryKey(autoGenerate = true) val id: Int = 0,
    @ColumnInfo val content: String,
    @ColumnInfo val isComplete: Boolean = false
)
```

- Impostando un valore di default per l'**`id`** non sarà necessario passarlo al costruttore dell'entità
- Il valore **`0`** non sarà in realtà mai utilizzato come **`id`** perché viene sovrascritto da **`autoGenerate = true`**
  - Se l'**`id`** è impostato manualmente a un valore diverso da **`0`**, allora Room utilizza quello invece di auto-generarlo
- Anche **`isComplete`** ha un valore di default, che è **`false`**

1.2. Creazione database – DAOs

- A questo punto, dobbiamo creare un **Data Access Object** (**DAO**)
  - Solitamente, si definisce un DAO per ogni tabella/entità
- Nel DAO, vanno definite le operazioni consentite sulla tabella **Todo**. Nel nostro caso abbiamo bisogno delle seguenti:
  - Reperire la lista di todo
  - Creare un nuovo todo
  - Modificare un todo esistente, marcandolo come completo/incompleto
  - Eliminare un todo esistente



1.2. Creazione database – DAOs

data.database.DAOs.kt

```
interface TodosDAO {
    @Query("SELECT * FROM todo")
    fun getAll(): Flow<List<Todo>>

    @Upsert
    suspend fun upsert(todo: Todo)

    @Delete
    suspend fun delete(todo: Todo)
}
```

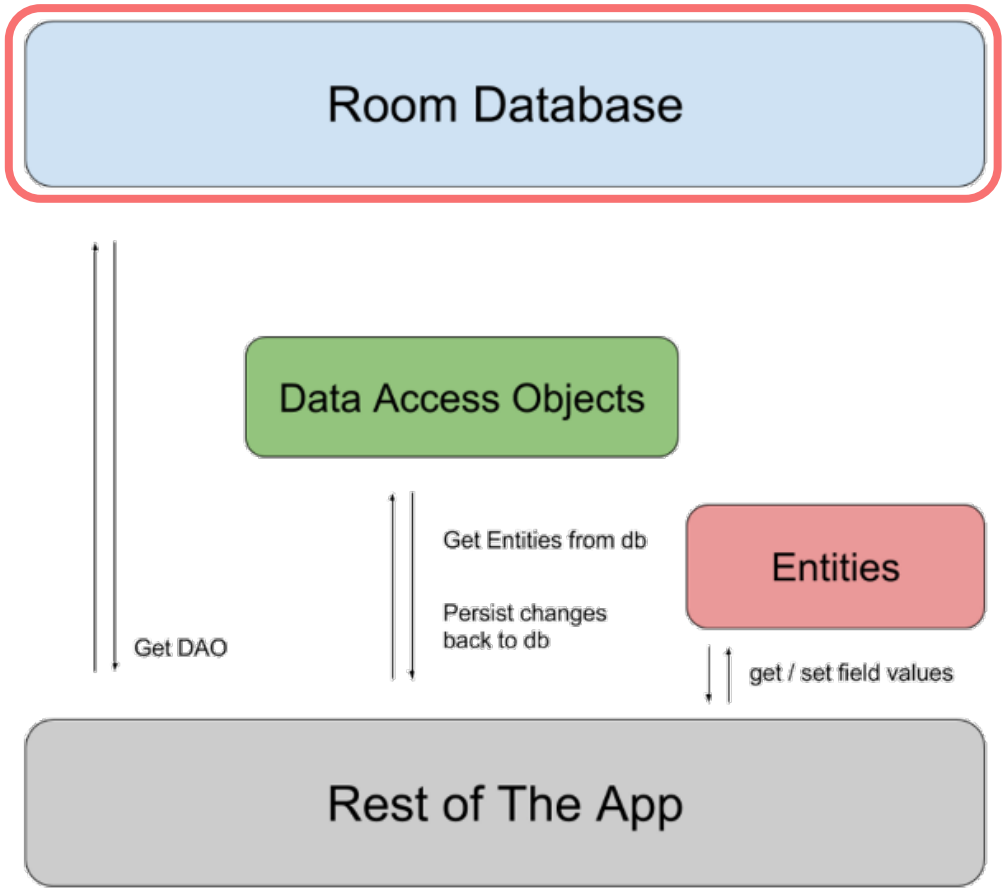
- Se un'operazione di lettura come **`getAll`** restituisce un **`Flow`**, l'implementazione generata da Room emette automaticamente un nuovo valore in seguito a ogni modifica dei dati
- Le operazioni in scrittura vanno marcate come **`suspend`**, in modo da renderle utilizzabili solo all'interno di una coroutine, evitando di bloccare il main thread
- L'annotazione **`@Upsert`**, aggiunta a Room nella versione 2.5.0, combina le operazioni di insert e update in un unico metodo: se il record passato come parametro non esiste, allora effettua una create; in caso contrario, esegue una update

1.2. Creazione database

- Entities e DAOs vanno raggruppati in un'unica classe astratta che eredita da RoomDatabase

data.database.TodoListDatabase.kt

```
@Database(entities = [Todo::class], version = 1)
abstract class TodoListDatabase : RoomDatabase() {
    abstract fun todosDAO(): TodosDAO
}
```



1.2. Creazione database

- Abbiamo bisogno di un'istanza singleton del nostro database, tramite la quale reperire i DAO e, di conseguenza, le funzioni per leggere e scrivere dati
  - Possiamo implementare il pattern singleton all'interno del companion object di **`TodoListDatabase`**
  - Ma è necessario?

data.database.TodoListDatabase.kt

```kotlin
@Database(entities = [Todo::class], version = 1)
abstract class TodoListDatabase : RoomDatabase() {
    abstract fun todosDAO(): TodosDAO

    companion object {
        @Volatile
        private var instance: TodoListDatabase? = null

        fun getDatabase(ctx: Context) =
            instance ?: synchronized(this) {
                instance = Room.databaseBuilder(
                    ctx,
                    TodoListDatabase::class.java,
                    "todo-list"
                ).build()
                instance
            }
    }
}
```

1.2. Creazione database

- Abbiamo bisogno di un'istanza singleton del nostro database, tramite la quale reperire i DAO e, di conseguenza, le funzioni per leggere e scrivere dati
  - Possiamo implementare il pattern singleton all'interno del companion object di **`TodoListDatabase`**
  - Ma è necessario?
  - Nel nostro caso no, perché abbiamo già l'helper **`single`** di Koin!

data.database.TodoListDatabase.kt

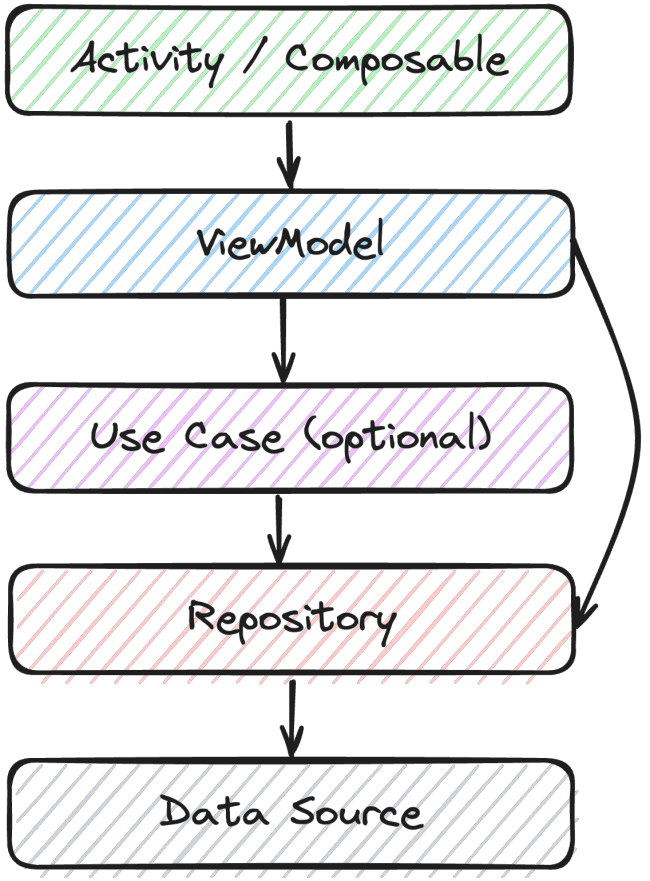
```kotlin
@Database(entities = [Todo::class], version = 1)
abstract class TodoListDatabase : RoomDatabase() {
    abstract fun todosDAO(): TodosDAO
}
```

AppModule.kt

```kotlin
val appModule = module {
     single {
          Room.databaseBuilder(
               get(),
               TodoListDatabase::class.java,
               "todo-list"
          ).build()
     }

     // ...
}
```

1.3. Creazione repository

- Aggiungiamo una **`TodosRepository`** che faccia da intermediario tra il database e il resto dell'app
- La repository ha bisogno di un riferimento al database – o meglio, al DAO – per le operazioni sui todo



`TodosRepository`

`TodoListDatabase`

1.3. Creazione repository

data.repositories.TodosRepository.kt

```
class TodosRepository(private val todosDAO: TodosDAO) {
    val todos: Flow<List<Todo>> = todosDAO.getAll()

    suspend fun upsert(todo: Todo) = todosDAO.upsert(todo)

    suspend fun delete(todo: Todo) = todosDAO.delete(todo)
}
```

AppModule.kt

```
val appModule = module {
    // ...

    single { TodosRepository(get<TodoListDatabase>().todosDAO()) }

    // ...
}
```

1.3. Creazione repository

data.repositories.TodosRepository.kt

```
class TodosRepository(private val todosDAO: TodosDAO) {
    val todos: Flow<List<Todo>> = todosDAO.getAll()

    suspend fun upsert(todo: Todo) = todosDAO.upsert(todo)

    suspend fun delete(todo: Todo) = todosDAO.delete(todo)
}
```

Accettiamo il DAO come parametro nella repository

AppModule.kt

```
val appModule = module {
    // ...

    single { TodosRepository(get<TodoListDatabase>().todosDAO()) }

    // ...
}
```

Configuriamo la dependency injection per la repository

1.4. Aggiunta repository al ViewModel

```
data class TodosState(val todos: List<Todo>)

interface TodosActions {
    fun addTodo(todo: Todo): Job
    fun removeTodo(todo: Todo): Job
    fun toggleComplete(todo: Todo): Job
}

class TodosViewModel(private val repository: TodosRepository) : ViewModel() {
    val state = repository.todos.map { TodosState(it) }.stateIn(
        scope = viewModelScope,
        started = SharingStarted.WhileSubscribed(),
        initialValue = TodosState(emptyList())
    )

    val actions = object : TodosActions {
        override fun addTodo(todo: Todo) = viewModelScope.launch {
            repository.upsert(todo)
        }

        override fun removeTodo(todo: Todo) = viewModelScope.launch {
            repository.delete(todo)
        }

        override fun toggleComplete(todo: Todo) = viewModelScope.launch {
            repository.upsert(todo.copy(isComplete = !todo.isComplete))
        }
    }
}
```

1.4. Aggiunta repository al ViewModel

```kotlin
data class TodosState(val todos: List<Todo>)

interface TodosActions {
    fun addTodo(todo: Todo): Job
    fun removeTodo(todo: Todo): Job
    fun toggleComplete(todo: Todo): Job
}

class TodosViewModel(private val repository: TodosRepository) : ViewModel() {
    val state = repository.todos.map { TodosState(it) }.stateIn(
        scope = viewModelScope,
        started = SharingStarted.WhileSubscribed(),
        initialValue = TodosState(emptyList())
    )

    val actions = object : TodosActions {
        override fun addTodo(todo: Todo) = viewModelScope.launch {
            repository.upsert(todo)
        }

        override fun removeTodo(todo: Todo) = viewModelScope.launch {
            repository.delete(todo)
        }

        override fun toggleComplete(todo: Todo) = viewModelScope.launch {
            repository.upsert(todo.copy(isComplete = !todo.isComplete))
        }
    }
}
```

La repository viene fornita come parametro al costruttore e va passata in AppModule.kt

```kotlin
val appModule = module {
    // ...

    viewModel { TodosViewModel(get()) }
}
```

1.4. Aggiunta repository al ViewModel

```
data class TodosState(val todos: List<Todo>)

interface TodosActions {
    fun addTodo(todo: Todo): Job
    fun removeTodo(todo: Todo): Job
    fun toggleComplete(todo: Todo): Job
}

class TodosViewModel(private val repository: TodosRepository) : ViewModel() {
    val state = repository.todos.map { TodosState(it) }.stateIn(
        scope = viewModelScope,
        started = SharingStarted.WhileSubscribed(),
        initialValue = TodosState(emptyList())
    )

    val actions = object : TodosActions {
        override fun addTodo(todo: Todo) = viewModelScope.launch {
            repository.upsert(todo)
        }

        override fun removeTodo(todo: Todo) = viewModelScope.launch {
            repository.delete(todo)
        }

        override fun toggleComplete(todo: Todo) = viewModelScope.launch {
            repository.upsert(todo.copy(isComplete = !todo.isComplete))
        }
    }
}
```

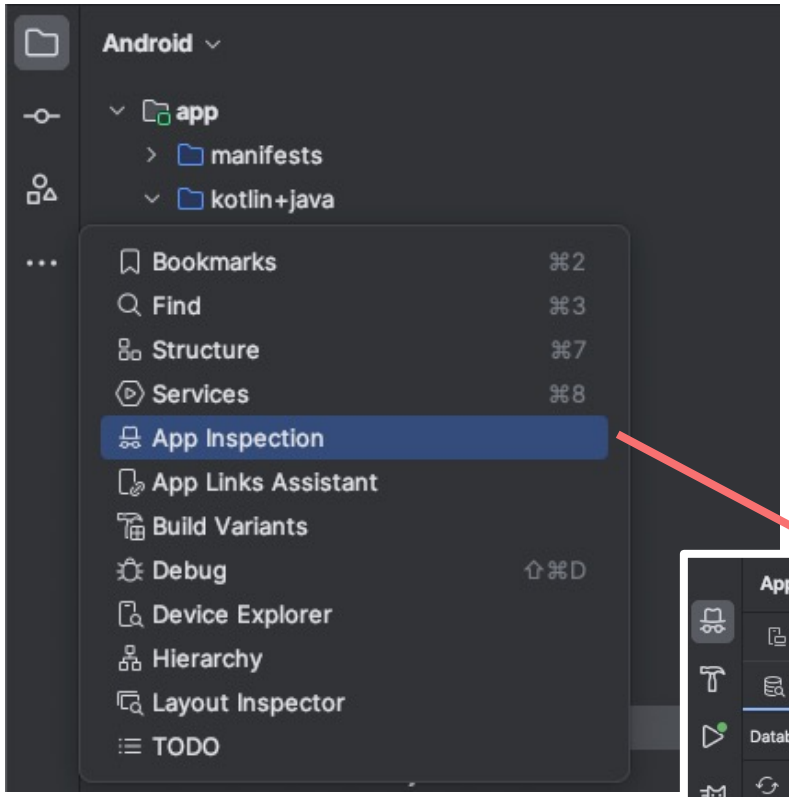
Il Flow della repository viene collezionato in uno StateFlow, come visto nel lab precedente con il DataStore

1.4. Aggiunta repository al ViewModel

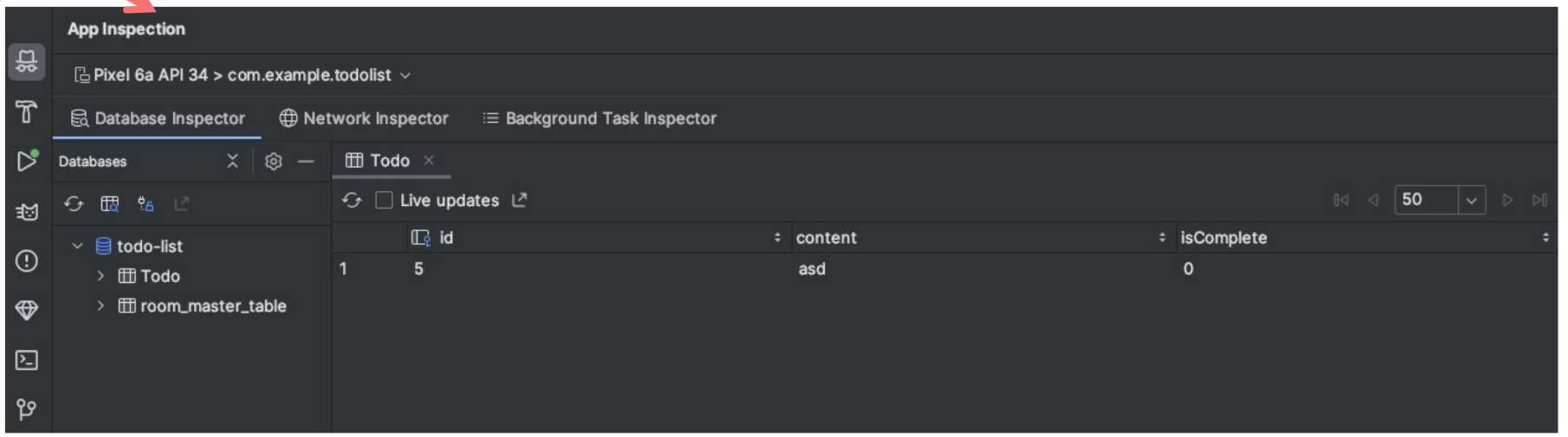
```
data class TodosState(val todos: List<Todo>)

interface TodosActions {
    fun addTodo(todo: Todo): Job
    fun removeTodo(todo: Todo): Job
    fun toggleComplete(todo: Todo): Job
}

class TodosViewModel(private val repository: TodosRepository) : ViewModel() {
    val state = repository.todos.map { TodosState(it) }.stateIn(
        scope = viewModelScope,
        started = SharingStarted.WhileSubscribed(),
        initialValue = TodosState(emptyList())
    )

    val actions = object : TodosActions {
        override fun addTodo(todo: Todo) = viewModelScope.launch {
            repository.upsert(todo)
        }

        override fun removeTodo(todo: Todo) = viewModelScope.launch {
            repository.delete(todo)
        }

        override fun toggleComplete(todo: Todo) = viewModelScope.launch {
            repository.upsert(todo.copy(isComplete = !todo.isComplete))
        }
    }
}
```

Le azioni restituiscono un Job e vengono eseguite in una coroutine nello scope del ViewModel

Database inspector



- Android Studio fornisce un tool integrato per effettuare operazioni di lettura e scrittura di qualsiasi database SQLite utilizzato da un'app.
- Molto utile in fase di debugging.



2. TravelDiary – Salvataggio dati con Room

- Utilizzare Room per salvare la lista di luoghi dell'app TravelDiary su un database locale
- Questo permetterà l'aggiunta delle seguenti funzionalità:
  - Visualizzazione della lista dei luoghi nella home
  - Visualizzazione di un luogo nella schermata di dettaglio
  - Aggiunta di un nuovo luogo
  - Opzionalmente: cancellazione di un luogo (in alternativa è possibile utilizzare il database inspector per cancellare i luoghi tramite Android Studio)

2. TravelDiary – Salvataggio dati con Room

- **Hint**: per il setup di Room basta fare riferimento all'esercizio precedente: la configurazione è praticamente identica
- Servirà una sola entity **`Trip`** con le seguenti proprietà:
  - **`id`**: numero intero
  - **`name`**: stringa
  - **`date`**: stringa (per semplicità)
  - **`description`**: stringa
- Servirà un **`TripsDAO`** per le operazioni di query, creazione e cancellazione

2. TravelDiary – Salvataggio dati con Room

- **Hint**: serviranno una **`TripsRepository`** e un **`TripsViewModel`**
- **`TripsViewModel`** in particolare è necessario in varie schermate:
  - **`HomeScreen`** ne ha bisogno per la lista di luoghi
  - **`TravelDetailsScreen`** per i dettagli su un singolo luogo
  - **`AddTravelScreen`** per aggiungere un nuovo luogo
- Per questo motivo, il punto migliore in cui crearlo è subito all'interno del **`NavGraph`**

```
@Composable
fun TravelDiaryNavGraph(navController: NavHostController) {
    val tripsVm = koinViewModel<TripsViewModel>()
    val tripsState by tripsVm.state.collectAsStateWithLifecycle()

    // ...
}
```

Tocca a voi!

Riferimenti

- Room
  https://developer.android.com/training/data-storage/room
  https://developer.android.com/training/data-storage/room/defining-data
  https://developer.android.com/training/data-storage/room/accessing-data
  https://developer.android.com/training/data-storage/room/async-queries