Documentación de la Capa de Datos y Repositorios

Responsable: Rodrigo Vargas Orellana – Ingeniero de Datos / Gestor de Flujo

1. Objetivo de la Capa de Datos

El objetivo de esta capa es proveer un modelo de datos consistente para la aplicación de gestión financiera, implementado sobre una base de datos local usando Room (SQLite) en Android. Además, se definen repositorios que actúan como interfaz única entre la lógica de negocio (ViewModels, casos de uso) y la persistencia, garantizando desacoplamiento, mantenibilidad y facilidad de pruebas.

Ningún componente de la interfaz de usuario (Activities, Fragments, Composables) debe acceder directamente a Room ni a los DAOs. Toda interacción con los datos se realiza a través de los repositorios.

---

2. Estructura de Paquetes

La organización del código sigue principios de arquitectura limpia y MVVM:

- `com.example.trial.data.local.entities`
  Contiene las clases `Entity` que representan las tablas de la base de datos.
- `com.example.trial.data.local.dao`
  Define las interfaces DAO que encapsulan las operaciones de lectura/escritura sobre la base de datos.
- `com.example.trial.data.local.AppDatabase`
  Clase central de Room que registra las entidades, expone los DAOs y aplica el patrón Singleton.
- `com.example.trial.data.repository`
  Implementa los repositorios de dominio que exponen métodos de alto nivel para la app.

---

3. Modelo de Datos (Entities)

El modelo se ha construido en base al diagrama entidad–relación entregado en las instrucciones del proyecto. Se definen seis tablas principales:

3.1 Categorias

Representa las categorías de gasto e ingreso.

- `idCategoria` (PK, autogenerado)
- `nombre`
- `descripcion`

Implementación:

kotlin
```kotlin
@Entity(tableName = "categorias")
data class CategoriaEntity(
    @PrimaryKey(autoGenerate = true)
    val idCategoria: Int = 0,
    val nombre: String,
    val descripcion: String
)
```

3.2 TipoCuentas

Catálogo de tipos de cuenta (efectivo, ahorro, cuenta corriente, etc.).

- `idTipo` (PK, autogenerado)
- `nombre`
- `descripcion`

kotlin
```kotlin
@Entity(tableName = "tipo_cuentas")
data class TipoCuentaEntity(
    @PrimaryKey(autoGenerate = true)
    val idTipo: Int = 0,
    val nombre: String,
    val descripcion: String
)
```

3.3 Cuentas

Define las cuentas financieras del usuario.

- `idCuenta` (PK, autogenerado)
- `idTipo` (FK hacia `TipoCuentas`)
- `nombre`
- `descripcion`
- `balance` (saldo actual de la cuenta)

kotlin

```kotlin
@Entity(tableName = "cuentas")
data class CuentaEntity(
    @PrimaryKey(autoGenerate = true)
    val idCuenta: Int = 0,
    val idTipo: Int,
    val nombre: String,
    val descripcion: String,
    val balance: Double
)
```

3.4 Estados

Catálogo de estados de las metas de ahorro.

- `idEstado` (PK, autogenerado)
- `nombre`
- `descripcion`

kotlin

```kotlin
@Entity(tableName = "estados")
data class EstadoEntity(
    @PrimaryKey(autoGenerate = true)
    val idEstado: Int = 0,
    val nombre: String,
    val descripcion: String
)
```

3.5 MetasAhorro

Gestión de objetivos de ahorro, asociados a categorías y estados.

- `idMeta` (PK, autogenerado)
- `idCategoria` (FK hacia `Categorias`)
- `idEstado` (FK hacia `Estados`)
- `nombre`
- `descripcion`
- `monto` (objetivo total)
- `montoActual` (progreso acumulado)
- `fechaInicio` (timestamp)
- `fechaObjetivo` (timestamp)

kotlin

```kotlin
@Entity(tableName = "metas_ahorro")
data class MetaAhorroEntity(
    @PrimaryKey(autoGenerate = true)
    val idMeta: Int = 0,
    val idCategoria: Int,
    val idEstado: Int,
    val nombre: String,
    val descripcion: String,
    val monto: Double,
    val montoActual: Double,
    val fechaInicio: Long,
    val fechaObjetivo: Long
)
```

3.6 Transacciones

Registro de todos los movimientos financieros.

- `idTransaccion` (PK, autogenerado)
- `idCategoria` (FK hacia `Categorias`)
- `idCuenta` (FK hacia `Cuentas`)
- `monto`
- `fecha` (timestamp)

- `descripcion`

kotlin
```kotlin
@Entity(tableName = "transacciones")
data class TransaccionEntity(
    @PrimaryKey(autoGenerate = true)
    val idTransaccion: Int = 0,
    val idCategoria: Int,
    val idCuenta: Int,
    val monto: Double,
    val fecha: Long,
    val descripcion: String
)
```

---

4. Capa DAO (Data Access Object)

Los DAOs definen las operaciones de acceso a la base de datos. Todos los métodos de lectura trabajan con `Flow` para integrarse de forma reactiva con los ViewModels.

4.1 CategoriaDao

kotlin
```kotlin
@Dao
interface CategoriaDao {

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertCategoria(categoria: CategoriaEntity)

    @Query("SELECT * FROM categorias ORDER BY nombre ASC")
    fun getAllCategorias(): Flow<List<CategoriaEntity>>
}
```

4.2 CuentaDao

kotlin
```kotlin
@Dao
interface CuentaDao {
```

```
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertCuenta(cuenta: CuentaEntity)

    @Update
    suspend fun updateCuenta(cuenta: CuentaEntity)

    @Query("SELECT * FROM cuentas ORDER BY nombre ASC")
    fun getAllCuentas(): Flow<List<CuentaEntity>>

    @Query("UPDATE cuentas SET balance = :nuevoBalance WHERE idCuenta
= :idCuenta")
    suspend fun actualizarBalance(idCuenta: Int, nuevoBalance: Double)
}
```

4.3 TransaccionDao

kotlin

```
@Dao
interface TransaccionDao {

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertTransaccion(transaccion: TransaccionEntity)

    @Query("SELECT * FROM transacciones ORDER BY fecha DESC")
    fun getAllTransacciones(): Flow<List<TransaccionEntity>>

    @Query("SELECT SUM(monto) FROM transacciones WHERE idCuenta =
:idCuenta")
    fun getTotalPorCuenta(idCuenta: Int): Flow<Double?>
}
```

DAOs adicionales para `TipoCuentas`, `Estados` y `MetasAhorro` pueden crearse siguiendo el mismo patrón cuando se requieran consultas específicas.

---

5. AppDatabase (Room + Singleton)

La clase `AppDatabase` centraliza la configuración de Room, aplica el patrón Singleton y expone los DAOs.

kotlin

```kotlin
@Database(
    entities = [
        CategoriaEntity::class,
        TipoCuentaEntity::class,
        CuentaEntity::class,
        EstadoEntity::class,
        MetaAhorroEntity::class,
        TransaccionEntity::class
    ],
    version = 1,
    exportSchema = false
)
abstract class AppDatabase : RoomDatabase() {

    abstract fun categoriaDao(): CategoriaDao
    abstract fun cuentaDao(): CuentaDao
    abstract fun transaccionDao(): TransaccionDao

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

        fun getDatabase(context: Context): AppDatabase {
            return INSTANCE ?: synchronized(this) {
                val instance = Room.databaseBuilder(
                    context.applicationContext,
                    AppDatabase::class.java,
                    "finanzas_db"
                )
                    .fallbackToDestructiveMigration()
                    .build()
                INSTANCE = instance
                instance
```

```
            }
        }
    }
}
```

- Se utiliza `fallbackToDestructiveMigration()` para simplificar el desarrollo: ante cambios de esquema, la base se recrea.
- En producción se recomienda definir migraciones explícitas.

---

6. Repositorios (Capa de Abstracción de Datos)

Los repositorios encapsulan la lógica de acceso y exponen una API limpia para ViewModels y casos de uso. Se integran fácilmente con Hilt mediante inyección por constructor.

6.1 TransaccionRepository

kotlin

```kotlin
class TransaccionRepository @Inject constructor(
    private val transaccionDao: TransaccionDao
) {

    fun getAllTransacciones(): Flow<List<TransaccionEntity>> =
        transaccionDao.getAllTransacciones()

    fun getTotalPorCuenta(idCuenta: Int): Flow<Double?> =
        transaccionDao.getTotalPorCuenta(idCuenta)

    suspend fun addTransaccion(transaccion: TransaccionEntity) {
        transaccionDao.insertTransaccion(transaccion)
    }
}
```

6.2 CuentaRepository

kotlin

```kotlin
class CuentaRepository @Inject constructor(
```

```
    private val cuentaDao: CuentaDao
) {

    fun getAllCuentas(): Flow<List<CuentaEntity>> =
        cuentaDao.getAllCuentas()

    suspend fun addCuenta(cuenta: CuentaEntity) {
        cuentaDao.insertCuenta(cuenta)
    }

    suspend fun actualizarBalance(idCuenta: Int, nuevoBalance: Double)
{
        cuentaDao.actualizarBalance(idCuenta, nuevoBalance)
    }
}
```

6.3 CategoriaRepository

kotlin

```
class CategoriaRepository @Inject constructor(
    private val categoriaDao: CategoriaDao
) {

    fun getAllCategorias(): Flow<List<CategoriaEntity>> =
        categoriaDao.getAllCategorias()

    suspend fun addCategoria(categoria: CategoriaEntity) {
        categoriaDao.insertCategoria(categoria)
    }
}
```

---

7. Uso desde ViewModels (ejemplo)

Ejemplo de integración con MVVM y Hilt para el caso de transacciones:

kotlin

```
@HiltViewModel
class TransaccionViewModel @Inject constructor(
    private val transaccionRepository: TransaccionRepository
) : ViewModel() {

    val transacciones: StateFlow<List<TransaccionEntity>> =
        transaccionRepository
            .getAllTransacciones()
            .stateIn(viewModelScope, SharingStarted.Lazily,
emptyList())

    fun registrarTransaccion(t: TransaccionEntity) {
        viewModelScope.launch {
            transaccionRepository.addTransaccion(t)
        }
    }
}
```

Las pantallas deben observar `transacciones` y llamar a `registrarTransaccion` para crear nuevos movimientos, sin conocer detalles de Room ni de los DAOs.

---

8. Reglas para extender la capa de datos

1. **Nuevas consultas**
    - Agregar primero el método en el DAO.
    - Luego exponerlo en el Repository.
    - Finalmente llamarlo desde el ViewModel.
2. **Nuevos campos o tablas**
    - Modificar/crear las Entities necesarias.
    - Actualizar `AppDatabase` y (en producción) la versión y migraciones.
3. **Buenas prácticas**
    - No acceder a DAOs desde UI.
    - No crear instancias de `Room.databaseBuilder` fuera de `AppDatabase`.
    - Reutilizar los Repositorios como única fuente de verdad de datos para la app.