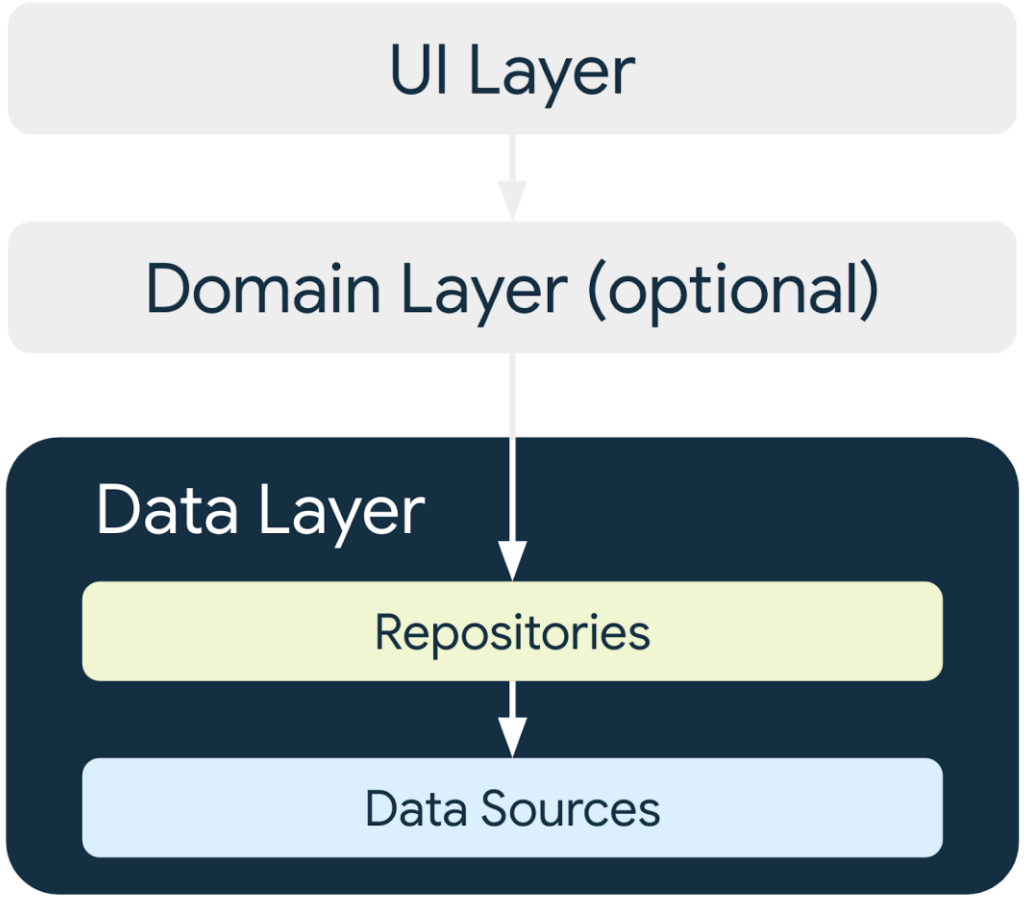
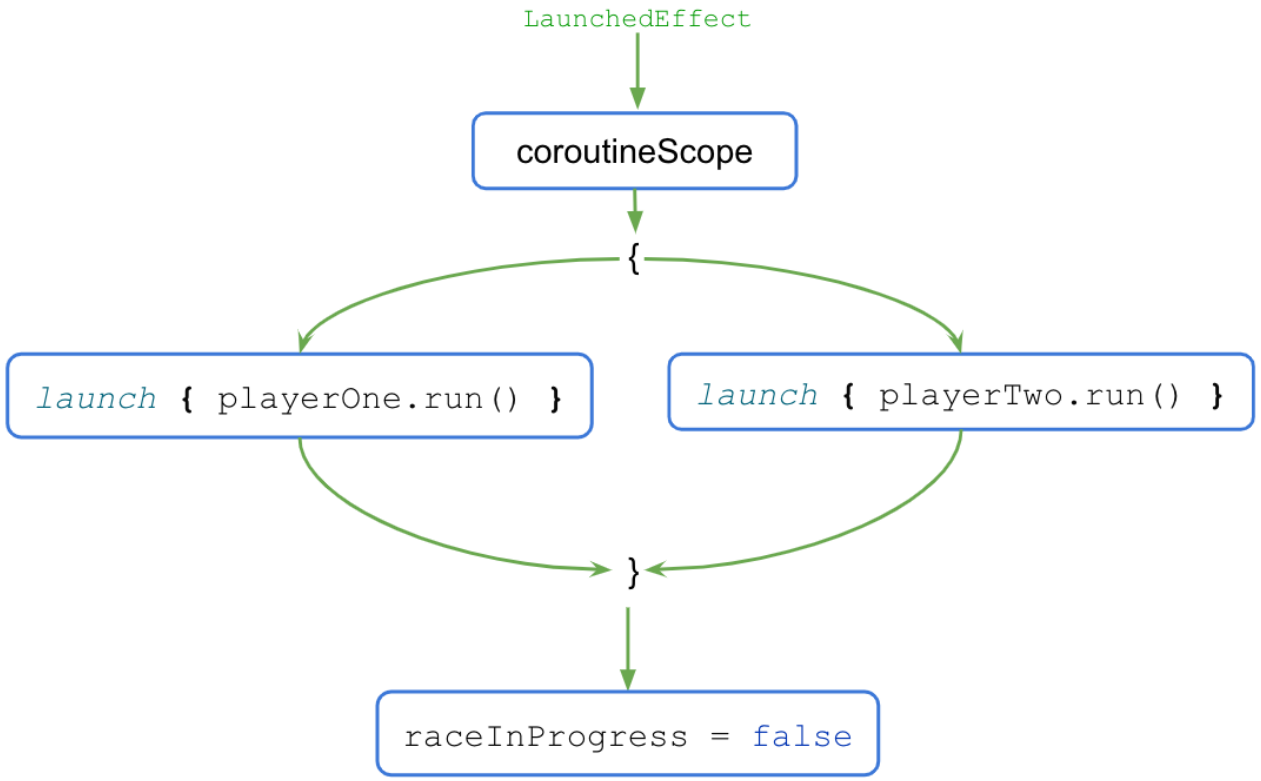
ROOM – Persistent Data Store

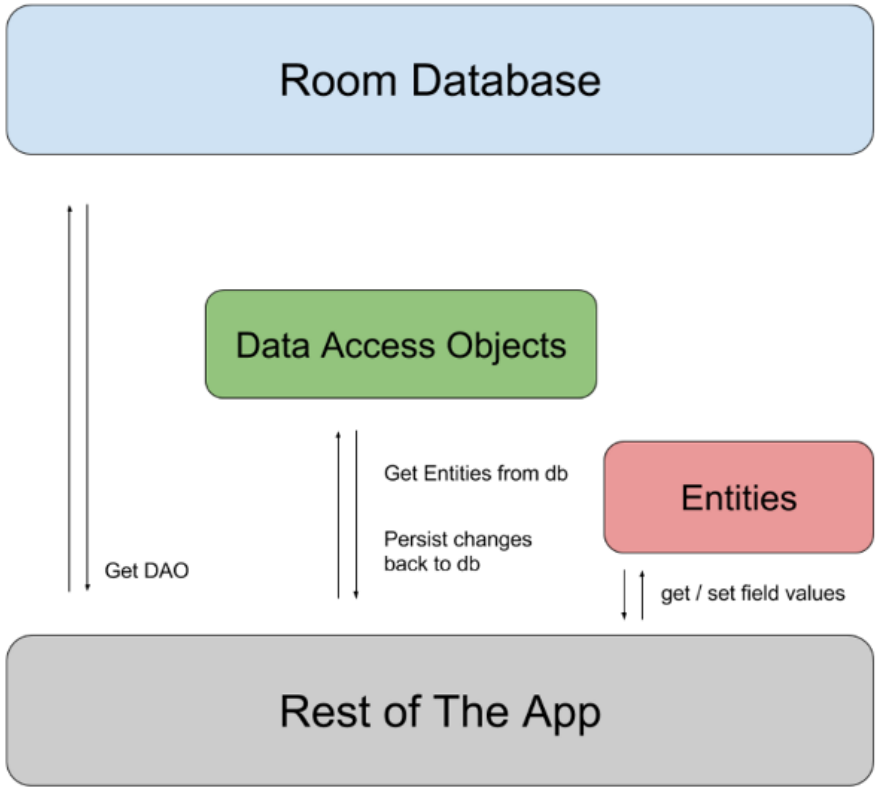
Agung Sediyono, Ph.D



Data Sources: ROOM


LaunchedEffect
coroutineScope
{
launch { playerOne.run() }
launch { playerTwo.run() }
}
raceInProgress = false

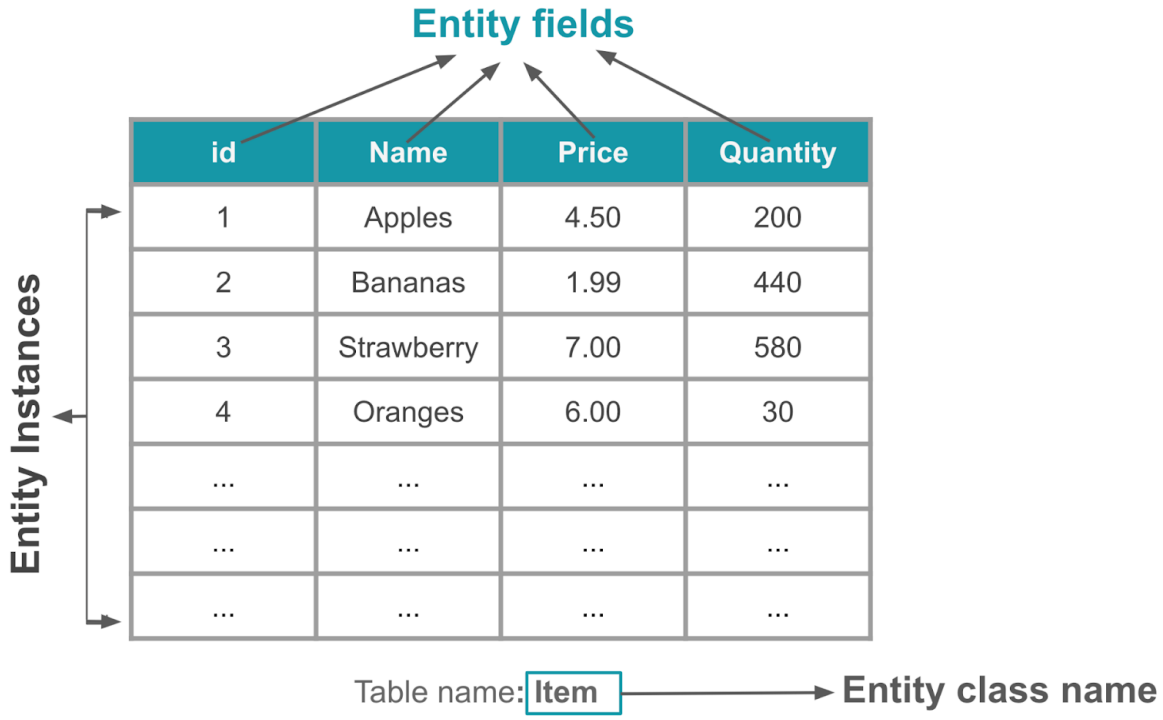
Co Routine



Component of Room

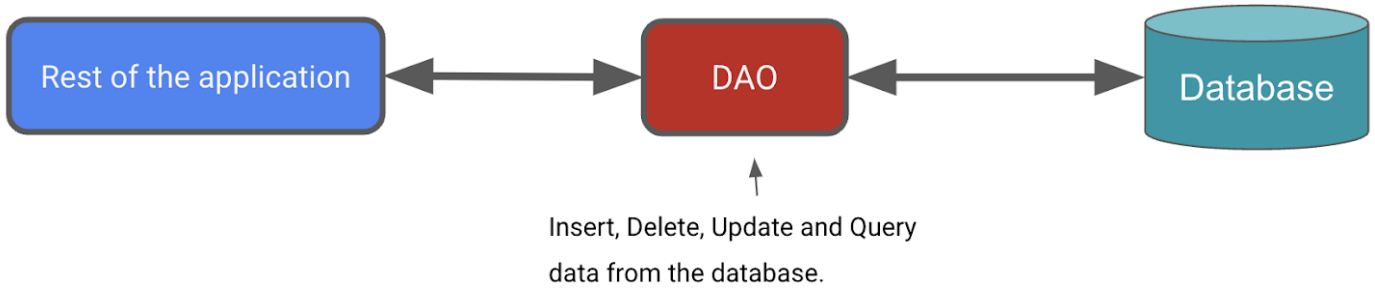
```
//Room
implementation("androidx.room:room-
runtime:${rootProject.extra["room_version"]}")
ksp("androidx.room:room-compiler:${rootProject.extra["room_version"]}")
implementation("androidx.room:room-ktx:${rootProject.extra["room_version"]}")
```

Dependensi: build.gradle.kts




```
@Entity (tablenName="items")
data class Item(
   @PrimaryKey
    val id: Int,
    val name: String,
    val price: Double,
    val quantity: Int
)
```

Entity



DAO: Data Access Object

```
 @Dao
interface ItemDao {
    @Insert(onConflict = OnConflictStrategy.IGNORE)
    suspend fun insert(item: Item)

    @Update
    suspend fun update(item: Item)

    @Delete
    suspend fun delete(item: Item)

    @Query("SELECT * from items WHERE id = :id")
    fun getItem(id: Int): Flow<Item>

    @Query("SELECT * from items ORDER BY name ASC")
    fun getAllItems(): Flow<List<Item>>
}
```

DAO Declaration

```
@Database(entities = [Item::class], version = 1, exportSchema = false)
abstract class InventoryDatabase : RoomDatabase() {

    abstract fun itemDao(): ItemDao

    companion object {
        @Volatile
        private var Instance: InventoryDatabase? = null

        fun getDatabase(context: Context): InventoryDatabase {
            // if the Instance is not null, return it, otherwise create a new database instance.
            return Instance ?: synchronized(this) {
                Room.databaseBuilder(context, InventoryDatabase::class.java, "item_database")
                    .build()
                    .also { Instance = it }
            }
        }
    }
}
```

Create Database Class

```
import kotlinx.coroutines.flow.Flow

/**
* Repository that provides insert, update, delete, and retrieve of [Item] from a given data source.
*/
interface ItemsRepository {

    fun getAllItemsStream(): Flow<List<Item>>

    fun getItemStream(id: Int): Flow<Item?>

    suspend fun insertItem(item: Item)

    suspend fun deleteItem(item: Item)

    suspend fun updateItem(item: Item)
}
```

Repository Interface

```
import kotlinx.coroutines.flow.Flow

class OfflineItemsRepository(private val itemDao: ItemDao): ItemsRepository{

override fun getAllItemsStream(): Flow<List<Item>> = itemDao.getAllItems()

 override  fun getItemStream(id: Int): Flow<Item?> = itemDao.getItem(id)

 override suspend fun insertItem(item: Item) = itemDao.insert(item)

 override suspend fun deleteItem(item: Item) = itemDao.delete(item)

 override suspend fun updateItem(item: Item) = itemDao.update(item)
}
```

Repository Class

```
/**
 * App container for Dependency injection.
 */
interface AppContainer {
    val itemsRepository: ItemsRepository
}

/**
 * [AppContainer] implementation that provides instance of [OfflineItemsRepository]
 */
class AppDataContainer(private val context: Context) : AppContainer {
    /**
     * Implementation for [ItemsRepository]
     */

override val itemsRepository: ItemsRepository by lazy {
    OfflineItemsRepository(InventoryDatabase.getDatabase(context).itemDao())
}
}
```

App Container

```
class ItemEntryViewModel : ViewModel() {

    /**
     * Holds current item ui state
     */
    var itemUiState by mutableStateOf(ItemUiState())
        private set

    /**
     * Updates the [itemUiState] with the value provided in the argument. This method also triggers
     * a validation for input values.
     */
    fun updateUiState(itemDetails: ItemDetails) {
        itemUiState =
            ItemUiState(itemDetails = itemDetails, isEntryValid = validateInput(itemDetails))
    }

    private fun validateInput(uiState: ItemDetails = itemUiState.itemDetails): Boolean {
        return with(uiState) {
            name.isNotBlank() && price.isNotBlank() && quantity.isNotBlank()
        }
    }
}
```

View Model: State holder

```
/**
 * Represents Ui State for an Item.
 */
data class ItemUiState(
    val itemDetails: ItemDetails = ItemDetails(),
    val isEntryValid: Boolean = false
)

data class ItemDetails(
    val id: Int = 0,
    val name: String = "",
    val price: String = "",
    val quantity: String = "",
)
```

View Model: Items

```
/**
 * Extension function to convert [ItemDetails] to [Item]. If the value of [ItemDetails.price] is
 * not a valid [Double], then the price will be set to 0.0. Similarly if the value of
 * [ItemDetails.quantity] is not a valid [Int], then the quantity will be set to 0
 */
fun ItemDetails.toItem(): Item = Item(
    id = id,
    name = name,
    price = price.toDoubleOrNull() ?: 0.0,
    quantity = quantity.toIntOrNull() ?: 0
)

fun Item.formatedPrice(): String {
    return NumberFormat.getCurrencyInstance().format(price)
}
```

View Model: Ekstensi

```
class MyViewModel(
    private val myRepository: MyRepository,
    private val savedStateHandle: SavedStateHandle
) : ViewModel() {
    // ViewModel logic

    // Define ViewModel factory in a companion object
    companion object {
        val Factory: ViewModelProvider.Factory = viewModelFactory {
            initializer {
                val savedStateHandle = createSavedStateHandle()
                val myRepository = (this[APPLICATION_KEY] as MyApplication).myRepository
                MyViewModel(
                    myRepository = myRepository,
                    savedStateHandle = savedStateHandle
                )
            }
        }
    }
}
```

Dependensi