Extracted from:

Kotlin and Android Development featuring Jetpack

Build Better, Safer Android Apps

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Kotlin and Android Development

featuring Jetpack

Build Better, Safer Android Apps

Michael Fazio

edited by Michael Swaine

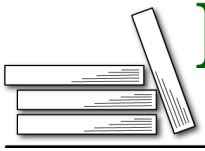
Kotlin and Android Development featuring Jetpack

Build Better, Safer Android Apps

Michael Fazio

The Pragmatic Bookshelf

Raleigh, North Carolina



For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Michael Swaine
Copy Editor: Sakhi MacMillan
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics
Founders: Andy Hunt and Dave Thomas

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-815-4
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—June 2021

Create a Repository Class

First off, a repository isn't a specific Room-related class but rather a recommended Jetpack convention. The advantage of using a repository is that the ViewModel classes have a single location for all data access, no matter the source of that data. This means that if we were to have web service calls along with our database calls, the front-end components would go to the same place for both and not even care about which was which. Having a repository separates out how you get data with how you use the data.

In our case, having a repository is admittedly overkill since we only have a single data source and, as you'll see, the functions in our repository will just be pass-throughs to PennyDropDao functions. Still, it's good practice to have the repository in place once you move on to more complicated apps, like the Android Baseball League app.

To start, create the PennyDropRepository class in the data package and send in an instance of PennyDropDao as a private value.

```
class PennyDropRepository(private val pennyDropDao: PennyDropDao) { ... }
```

The PennyDropRepository class will have four functions to start: getCurrentGameWithPlayers(), getCurrentGameStatuses(), startGame(), and updateGame(). This highlights another advantage of a repository: we can limit the functions that are shown to the rest of the app while allowing the DAO to have extra internal functionality. While in most cases we could just make those functions protected, this would restrict us from using them in other DB-related activity, such as during RoomDatabase initialization.

As I mentioned before, each of these functions will be calling the corresponding function in PennyDropDao. Also, the latter two functions will be marked with the suspend keyword, as they're modifying the database. The entire PennyDropRepository looks like this:

```
class PennyDropRepository(private val pennyDropDao: PennyDropDao) {

  fun getCurrentGameWithPlayers() =
    pennyDropDao.getCurrentGameWithPlayers()

  fun getCurrentGameStatuses() =
    pennyDropDao.getCurrentGameStatuses()

  suspend fun startGame(players: List<Player>) =
    pennyDropDao.startGame(players)

  suspend fun updateGameAndStatuses(
    game: Game,
    statuses: List<GameStatus>
```

```
  ) = pennyDropDao.updateGameAndStatuses(game, statuses)
}
```

The PennyDropRepository is now complete, but unfortunately none of the DAO functions currently exist. Let's head back over to PennyDropDao and add them in.

Add PennyDropDao Functions

We've got four functions to add, and we're going to go in order. First up, getCurrentGameWithPlayers(). This function is similar to what we saw earlier (in particular, the getPlayer() function) but it also includes the @Transaction annotation. This annotation tells Room that the function you're calling references multiple tables and the data should be retrieved in a single atomic operation. In our case, we're getting data from both the games and players tables.

```
@Transaction
@Query("SELECT *  FROM games ORDER BY startTime DESC LIMIT 1")
abstract fun getCurrentGameWithPlayers(): LiveData<GameWithPlayers>
```

While the query only mentions the games table, we're pulling in data from both tables due to the @Relation annotation and the @Junction on the GameStatus class. That tells Room to get the associated Player records for the Game without having to write out that piece of the SQL query.

The next function is getCurrentGameStatuses(). Unfortunately, there isn't an easy way with Room to grab a game, the players, *and* the statuses for each player in a single query and map the results to an object. So we need to pull in the GameStatus objects separately. This @Query will get the latest GameStatus instance by performing a subquery on the games table. We get the most recent open game, then sort the statuses by the gamePlayerNumber property (to ensure players are in the right play order). Note that this will also be an @Transaction since we're referencing multiple tables.

```
@Transaction
@Query(
  """
  SELECT * FROM game_statuses
  WHERE gameId = (
    SELECT gameId FROM games
    WHERE endTime IS NULL
    ORDER BY startTime DESC
    LIMIT 1)
  ORDER BY gamePlayerNumber
  """
)
abstract fun getCurrentGameStatuses(): LiveData<List<GameStatus>>
```

As you can see, the @Query annotation gives us a lot of flexibilty in retreiving data from the database. But sometimes an @Query still isn't enough and we need to call multiple functions in a single @Transaction. For that scenario, we can instead create an open function and implement the function ourselves rather than letting Room do that for us. The ability to have fully implemented functions in our DAO is the reason PennyDropDao is an abstract class rather than an interface (as is commonly seen in the Room documentation).

In the case of startGame(), we're going to bring in a List<Player>, close up any existing games, create a new Game, get or insert the entered Player objects from/into the database, then add new GameStatus entries for each player before returning the newly created game's ID. To do all this, we'll call other functions inside PennyDropDao to do the work for us. We already created insertGame() and insertPlayer() when first building PennyDropDao, so we just need two additional new functions.

The first function is called closeOpenGames(), which goes through the database and sets the current time as the endTime and state of Cancelled for any still-open games. We previously saw named bind parameters in Create a DAO Class, on page ?, but here they're more interesting.

We can't send in complex types by default, but since we previously created type converters for both types, this works just fine.

```
@Query("""
  UPDATE games
  SET endTime = :endDate, gameState = :gameState
  WHERE endTime IS NULL""")
abstract suspend fun closeOpenGames(
  endDate: OffsetDateTime = OffsetDateTime.now(),
  gameState: GameState = GameState.Cancelled
)
```

Note the use of default values for each property. We can include parameters we have no intention of overwriting purely to be able to include them as parameters in a query, yet still keep the flexibility to overwrite if needed for any reason.

Also, since this function is modifying the database, it needs to be a transaction. But instead of having to add the @Transaction annotation, Room automatically wraps all modifying actions as a transaction. This includes functions with an @Insert, @Update, or @Delete annotation.

The other function we still need is insertGameStatuses(), which just requires the @Insert annotation.

```
@Insert
abstract suspend fun insertGameStatuses(gameStatuses: List<GameStatus>)
```

This highlights another nice Room feature: we can send in a List<GameStatus> and all GameStatus records are entered into the database instead of manually having to insert them one by one.

Now that all the functions we're using are created, we can get back to startGame() itself. Note that even though we have the implementation for startGame() in here, it still has to be marked as open since it has the @Transaction annotation.

The function code looks like this:

```
@Transaction
open suspend fun startGame(players: List<Player>): Long {
  this.closeOpenGames()

  val gameId = this.insertGame(
    Game(
      gameState = GameState.Started,
      currentTurnText = "The game has begun!\n",
      canRoll = true
    )
  )

  val playerIds = players.map { player ->
    getPlayer(player.playerName)?.playerId ?: insertPlayer(player)
  }

  this.insertGameStatuses(
    playerIds.mapIndexed { index, playerId ->
      GameStatus(
        gameId,
        playerId,
        index,
        index == 0
      )
    }
  )

  return gameId
}
```

The one piece I want to call out here is how we're getting the playerIds value. We check the database for a player and either use that to get the playerId or, if the player's not found, we create the player and then send back its player ID. Since insertPlayer() returns a Long, we get back the database ID right away without having to do a secondary lookup. Plus, the Elvis operator allows us to keep everything in one expression instead of having to include extra conditional logic.

We have one remaining function to cover in the PennyDropDao, which is the updateGameAndStatuses() function. This function does exactly what you'd expect: it updates the DB versions of the entered Game and GameStatus objects, all wrapped in a single @Transaction. updateGame() already exists, but we need to create updateGameStatuses() quickly:

```
@Update
abstract suspend fun updateGameStatuses(gameStatuses: List<GameStatus>)
```

From there, updateGameAndStatuses() is calling those two functions, wrapped in a @Transaction:

```
@Transaction
open suspend fun updateGameAndStatuses(
  game: Game,
  statuses: List<GameStatus>
) {
  this.updateGame(game)
  this.updateGameStatuses(statuses)
}
```

The last part of the PennyDropRepository I want to cover is adding the ability to have a singleton instance of the repository for use anywhere. This is optional, but it'll be useful to avoid creating multiple instances of PennyDropRepository for different views.

The idea is the same as with PennyDropDatabase: we get the existing instance variable unless it's null, then we create a new instance and return that. We're also going to take advantage of the synchronized() block to avoid having multiple simultaneous attempts at creating the PennyDropRepository. All of this will live inside PennyDropRepository in its companion object:

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

  fun getInstance(pennyDropDao: PennyDropDao) =
    this.instance ?: synchronized(this) {
      instance ?: PennyDropRepository(pennyDropDao).also {
          instance = it
      }
    }
}
```

With that, the PennyDropRepository and PennyDropDao classes are now all set. We have all the logic we need to persist our game data in a local database. The remaining piece is to pull that data back out of the database and use it inside GameViewModel.