Advanced Mobile Application Development

Recipe App: Room Database

Today we are going to be using a tool called Room to create a local database that will store the users favorite recipes. We'll just be setting up the database today and verify that it works, and then we'll add the necessary UI changes next week.

Room

Room is an abstraction layer that sits on top of SQLite. It makes it easier to create SQL databases and manage tables. There are three main components that are needed in order to use Room:

- **Entities**: These represent SQL tables and they are created using data classes with special annotations that help room configure the different properties of the SQL table. Each property in the class represents a column in the table.
- **Data Access Objects**: This is how you perform SQL operations on your tables such as select, insert, update, and delete. These will be interfaces.
- **Database:** Holds the database and serves as the access point for the underlying connection to the app's persisted relational data. This is a class that extends RoomDatabase.

SQL

For our purposes, you won't need to be a SQL master since our queries and database design are pretty basic, but you should be somewhat familiar with general SQL principles and relational data and how that differs. Right now, we're using lots of nested objects and lists to describe our data which means that our data structures aren't immediately compatible with a SQL database right out of the box. We'll need to account for this in our design.

Changes to Project

I made a couple changes to the project since Tuesday so I'm going to run through those real quick just so we're all on the same page. Nothing too crazy — just cleaned things up a bit and fixed some minor bugs.

Open up Android Studio and **ensure** that you have checked out the 04-room-start branch.

Take a look at the following changes:

- Added Room dependencies — see build.gradle for the app module
    - Just got the dependencies from the Room guide on the Android website
- Fixed bottom nav disappearing bug — show MainActivity.kt

- Added new packages in ui/search for the details and results views
- Changed instructions to use a RecyclerView on detail view just like we did for the ingredients RecyclerView
    - Reused Adapter and ViewHolder from the Ingredients RecyclerView

Run and show current state of app as reminder.

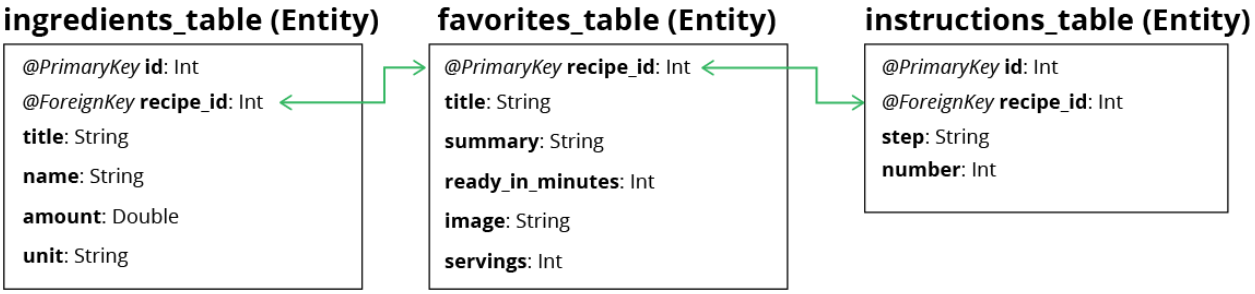
Integrate Room Database

Database Design

Navigate to Recipe.kt and take a look at our model data structure. You can see that it is relatively complex and many of the objects include other objects and lists and so on. If you know anything about SQL, you'll know that this sort of data is not directly compatible with a relational database like SQL. We need to design our database in a way that makes it easy to map between our app's model and our Room entities for persistence. I've spent some time on this and determined a pretty good solution that uses concepts like Room relations and Foreign Keys.

It's important to note that the design step is a crucial precursor to the implementation step when working with SQL. It is extremely difficult and annoying to make major modifications to your database scheme after it's put in place and migrating data from the old schema to a new one can be time consuming and frustrating (not that I've ever done a migration, but so I've heard).

Here's the schema that I've chosen for our data:



Entities

Let's start by creating our entities based on our schema.

First, **create** a *database* package inside our *data* package. Next, **create packages** inside the new *database* package for each of the entities: *favorite*, *ingredient*, and *instruction.*

Favorite

We'll start with the Entity for our *favorites_table*. **Ctrl-click** on the *favorite* package and **select** New->Kotlin File/Class to create a class **called** *Favorite.*
We'll **create** a data class for the *Favorite* entity and use Room annotations to define some of the properties specific to Room entities:

```
@Entity(tableName = "favorites_table")
data class Favorite (
    @PrimaryKey val recipe_id: Int,
    val title: String,
    val summary: String,
    val image: String,
    val ready_in_minutes: Int,
    val servings: Int
)
```

You can see that I use the tableName property of the @Entity annotation to define a table name that is different from the class name. I did this to conform to SQL naming conventions. I've also included an annotation for the Primary Key.

Ingredient

Next, we'll do the Ingredient Entity. This is a little bit more tricky than the Favorite entity because we're using a foreign key restriction and auto-generating the primary key, but still not too bad.

Create new file for class called Ingredient in the *ingredient* package
Add the following code:

```
@Entity (foreignKeys = [ForeignKey(
    entity = Favorite::class,
    parentColumns = ["recipe_id"],
    childColumns = ["recipe_id"]
)], tableName = "ingredients_table"
)
data class Ingredient(
    @PrimaryKey(autoGenerate = true) val id: Int = 0,
    val recipe_id: Int,
    val name: String,
    val amount: Double,
    val unit: String
)
```

Instruction

Now we'll create the final Instruction Entity. It will look very similar to the Ingredient Entity.

Create a new class called Instruction in the *instruction* package
Add the following code:

```
@Entity (foreignKeys = [ForeignKey(
    entity = Favorite::class,
    parentColumns = ["recipe_id"],
    childColumns = ["recipe_id"]
)],
    tableName = "instructions_table"
)
data class Instruction(
    @PrimaryKey(autoGenerate = true) val id: Int = 0,
    val recipe_id: Int,
    val number: Int,
    val step: String
)
```

Relation

Next, I'm going to create a relation object that will combine all of my entities into a single object. This isn't absolutely necessary, but it could be very useful for certain use cases so I wanted to show how it's done.

Create a new package in the *database* package called *relation*
Create a new class in the package called *FavoriteWithDetails* — this will be a plain Kotlin object
Add the following code to construct the relational object:

```
data class FavoriteWithDetails (
    @Embedded @PrimaryKey val favorite: Favorite,

    @Relation(
        parentColumn = "recipe_id",
        entityColumn = "recipe_id",
        entity = Instruction::class
    ) val instructions: List<Instruction>,

    @Relation(
        parentColumn = "recipe_id",
```

```
        entityColumn = "recipe_id",
        entity = Ingredient::class
    ) val ingredient: List<Ingredient>
)
```

Data Access Objects (DAOs)

Now let's create the Data Access Objects, or DAOs. We need a DAO for each of our entities as well as one for our relation. These will be interfaces with methods that specify SQL queries for performing operations like select, insert, delete, and update on our data. Our queries are going to be very basic so there's no need to worry if you haven't ever written a SQL query. Room also does a lot of work to simplify things on our end so it all works seamlessly.

One important thing to note about DAO methods is that they need to be executed from a background thread — we'll need to remember this when we start to use them later on in the demo.

FavoriteDAO

First, **create a new interface** in our *favorite* package called *FavoriteDAO*
For now, we'll just **add methods declarations to insert and select data** from our Favorite entity

```
@Dao
interface FavoriteDAO {
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    fun insertFavorite(favorite: Favorite)

    @Query("SELECT * FROM favorites_table")
    fun getAllFavorites(): LiveData<List<Favorite>>
}
```

We can return a LiveData object from getAllFavorites to listen to changes in the entity. This can cause performance issues and should be used wisely.

We don't need to provide implementations for the methods as Room will handle all that for us!

Ingredient DAO

Create a new interface in the *ingredient* package called *IngredientDAO*
Add methods to insert and select data from our *ingredients_table*

```
@Dao
interface IngredientDAO {
```

```
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    fun insertIngredient(ingredient: Ingredient)

    @Query("SELECT * FROM ingredients_table WHERE recipe_id = :id")
    fun getIngredientsForRecipe(id: Int): List<Ingredient>
}
```

Instruction DAO

Create a new interface in the *instruction* package called *InstructionDAO*
Add methods to insert and select data from our instruction_table

```
@Dao
interface InstructionDAO {
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    fun insertInstruction(instruction: Instruction)

    @Query("SELECT * FROM instructions_table WHERE recipe_id = :id")
    fun getInstructionsForRecipe(id: Int): List<Instruction>
}
```

FavoritesWithDetailsDAO

Create a new interface in the *relation* package called *FavoriteWithDetailsDAO*
Add a method to select all the favorites with their details (instructions and ingredients)

```
@Dao
interface FavoriteWithDetailsDAO {
    @Transaction
    @Query("SELECT * FROM favorites_table")
    fun getAllFavoritesWithDetails(): LiveData<List<FavoriteWithDetails>>
}
```

Database

Next we need to create our database class to serve as a reference point to the persisted, relational data. We'll use a Singleton design to make sure there is only one instance of this class at any given time. This code mostly came from the official Android Room guide.

Create a class called *AppDatabase* in the *database* package
Add the following code

```
@Database(entities = arrayOf(Ingredient::class, Instruction::class,
Favorite::class), version = 3, exportSchema = false)
abstract class AppDatabase : RoomDatabase() {
```

```
    abstract fun favoriteWithDetailsDAO(): FavoriteWithDetailsDAO
    abstract fun favoriteDAO(): FavoriteDAO
    abstract fun instructionDAO(): InstructionDAO
    abstract fun ingredientDAO(): IngredientDAO

    companion object {
        var INSTANCE: AppDatabase? = null

        fun getDatabase(context: Context): AppDatabase {
            val tempInstance = INSTANCE
            //return the instance if it already exists
            if(tempInstance != null) return tempInstance
            //otherwise need to create the instance
            //lock the monitor of the companion object so it can't be
access by another thread while we create the database (makes this part of
the code thread safe)
            synchronized(this) {
                val instance =
Room.databaseBuilder(context.applicationContext, AppDatabase::class.java,
"recipe_database").fallbackToDestructiveMigration().build()
                INSTANCE = instance
                return instance
            }
        }
    }
}
```

Repository Class

In order to work with our data, we need to create a repository class that will handle all of the reads and writes to the database. We'll create a new repository class that is seperate from our RecipeRepository since this is a seperate portion of our app and it doesn't depend on any of the recipe searching logic.

Create a new class called *FavoriteRepository* in the *data* package

For now, we will use this repository to get access to all the DAOs, get access to the `LiveData<List<FavoriteWithDetails>>` object so that we can listen for changes to the database, and insert some dummy data. Eventually, there will be more methods and such once we start actually writing real data, but this will be a good proof that our database is functioning properly.

Add the following code:

```
class FavoriteRepository(val app:Application) {
    private val db = AppDatabase.getDatabase(app)

    //dao references
    private val favoriteWithDetails = db.favoriteWithDetailsDAO()
    private val favoriteDAO = db.favoriteDAO()
    private val ingredientDAO = db.ingredientDAO()
    private val instructionDao = db.instructionDAO()

    //get reference to the live data object
    val allFavorites: LiveData<List<FavoriteWithDetails>> =
favoriteWithDetails.getAllFavoritesWithDetails()

    init {
        //put some dummy data into our data base
        CoroutineScope(Dispatchers.IO).launch {
            favoriteDAO.insertFavorite(Favorite(1, "pasta", "my dinner",
"bad.png", 13, 2))
            instructionDao.insertInstruction(Instruction(recipe_id = 1,
number = 1, step = "cry"))
            instructionDao.insertInstruction(Instruction(recipe_id = 2,
number = 1, step = "boil water"))
            ingredientDAO.insertIngredient(Ingredient(recipe_id = 1, name =
"pasta", amount = 1.0, unit = "whole"))
        }

    }
}
```

ViewModel

Next, we'll use *FavoritesViewModel* to instantiate the repository class and get a reference to the *allFavorites* LiveData object

Add the following code:

```
class FavoritesViewModel(app: Application) : AndroidViewModel(app) {
    private val favRepo = FavoriteRepository(app)
    val favoriteList = favRepo.allFavorites

    ...
}
```

Fragment

Finally, **add an Observer** to the LiveData object in the *FavoritesFragment*. We'll do this in the *onCreateView()* method after we get the reference to the *FavoritesViewModel* instance

```
favoritesViewModel.favoriteList.observe(viewLifecycleOwner, Observer {
    Log.i(LOG_TAG, it.toString())
})
```

Run the app and you should see the dummy data being printed to the log when you select the favorites tab from the bottom nav.

We have yet to actually implement the database in our code and allow users to save a recipe to their favorites, but we have the app architecture setup for this and the database is created. Next week we'll add in the ability to favorite/save recipes.