Chapter 6: Adding Offline Capabilities with Jetpack Room

In this chapter, we're starting our journey of exploring ways to architecture our apps by first making sure that our application can be used without an internet connection.

In the *Introducing Jetpack Room* section, we will briefly note the various caching mechanisms that are on Android. Then, we will introduce the Jetpack Room library and its core elements.

Next, in the *Enabling offline usage by implementing Room* section, we will implement Room in our Restaurants app and allow users to use the application without an internet connection. In the *Applying partial updates to the Room database* section, we will learn how to partially update data inside Room so that we can save selections such as whether the restaurants were favorited by the user.

Finally, in the *Making local data the single source of truth for app content* section, we will understand why having a single source of truth for app data is beneficial, and then we will set the Room database as the single source of content for our app.

To summarize, in this chapter, we're going to cover the following main topics:

- Introducing Jetpack Room
- Enabling offline usage by implementing Room
- Applying partial updates to the Room database
- Making local data the single source of truth for app content

Before jumping in, let's set up the technical requirements for this chapter.

Technical requirements

Usually, building Compose-based Android projects with Jetpack Room will require your day-to-day tools. However, to follow along with the examples smoothly, make sure you have the following:

- The Arctic Fox 2020.3.1 version of Android Studio. You can also use a newer Android Studio version or even Canary builds but note that the IDE interface and other generated code files might differ from the ones used throughout this book.
- The Kotlin 1.6.10, or newer, plugin installed in Android Studio
- The Restaurants app code from the previous chapter.
- Minimal knowledge of SQL databases and queries

The starting point for this chapter is represented by the Restaurants application that was developed in the previous chapter. If you haven't followed the implementation described in the previous chapter, access the starter code for this chapter by navigating to the **`Chapter_05`** directory of the repository. Then, import the Android project entitled **`chapter_5_restaurants_app`**.

To access the solution code for this chapter, navigate to the **`Chapter_06`** directory:

https://github.com/PacktPublishing/Kickstart-Modern-Android-Development-with-Jetpack-and-Kotlin/tree/main/Chapter_06/chapter_6_restaurants_app.

Introducing Jetpack Room

Modern applications should be available for use in any conditions, including when the user is missing an internet connection. This allows apps to provide a seamless user experience and usability even when the user's device cannot access the network.

In this section, we will discuss the following:

- Exploring the caching mechanism on Android
- Introducing Jetpack Room as a solution for local caching

So, let's begin!

Exploring the caching mechanism on Android

To cache specific content or application data, reliable Android apps make use of the various offline caching mechanisms that are suitable for different use cases:

- Shared preferences are used to store lightweight data (such as user-related selections) as key-value pairs. This option shouldn't be used to store objects that are part of the app's content.
- Device storage (either internal or external) is used for storing heavy-weight data (such as files, pictures, and more).
- SQLite database is used for storing app content in a structured manner inside a private database. **SQLite** is an open source SQL database that stores data in private text files.

In this chapter, we will focus on learning how to cache structured content (which is, usually, held by Kotlin `data class` objects) within a SQLite database. In this way, we allow the user to browse the app's data while remaining offline.

NOTE

Android comes with a built-in SQLite database implementation that allows us to save structured data.

In our app, we can consider the array of restaurants to be a perfect candidate for app content that can be saved inside a SQLite database. Since the

data is structured, with SQLite, we get the advantage of being able to perform different actions such as searching for restaurants within the database, updating particular restaurants, and more.

By caching app content in this way, we can allow users to browse the app's restaurants while offline. However, for this to work, the users need to have previously opened the app using an active internet connection, thereby allowing the app to cache the contents for future offline use.

Now, to save the restaurants to the SQLite private database, we need to make use of the SQLite APIs. These APIs are powerful. However, by using them, you face quite a few disadvantages:

- The APIs are of a low level and are relatively difficult to use.
- The SQLite APIs provide no compile-time verification of SQL queries, which can lead to unwanted runtime errors.
- There is a lot of boilerplate code involved in creating a database, performing SQL queries, and more.

To mitigate these issues, Google provides the Jetpack Room library. This library is nothing more than a wrapper library that simplifies the way we access and interact with the SQLite database.

Introducing Jetpack Room as a solution for local caching

Room is a persistence library that is defined as an abstraction layer over SQLite and provides simplified database access while taking advantage of the power of the SQLite APIs.

As opposed to using the raw SQLite APIs, Room abstracts most of the complexity associated with working with SQLite. The library removes most of the unpleasant boilerplate code that is required to set up and interact with SQLite databases on Android while also providing the compile-time checking of SQL queries.

To make use of the Room library and cache contents using its API, you need to define three primary components:

- **Entities** that define tables within the private database. In our Restaurants app, we will consider the `Restaurant` data class as an entity. This means that we will have a table populated with `Restaurant` objects. In other words, the rows of the table are represented by instances of our restaurants.
- A database class that will contain and expose the actual database.
- **Data Access Objects (DAOs)** that represent an interface. This allows us to get, insert, delete, or update the actual content within the database.

The database class provides us with a reference to the DAO interface associated with the SQLite database:

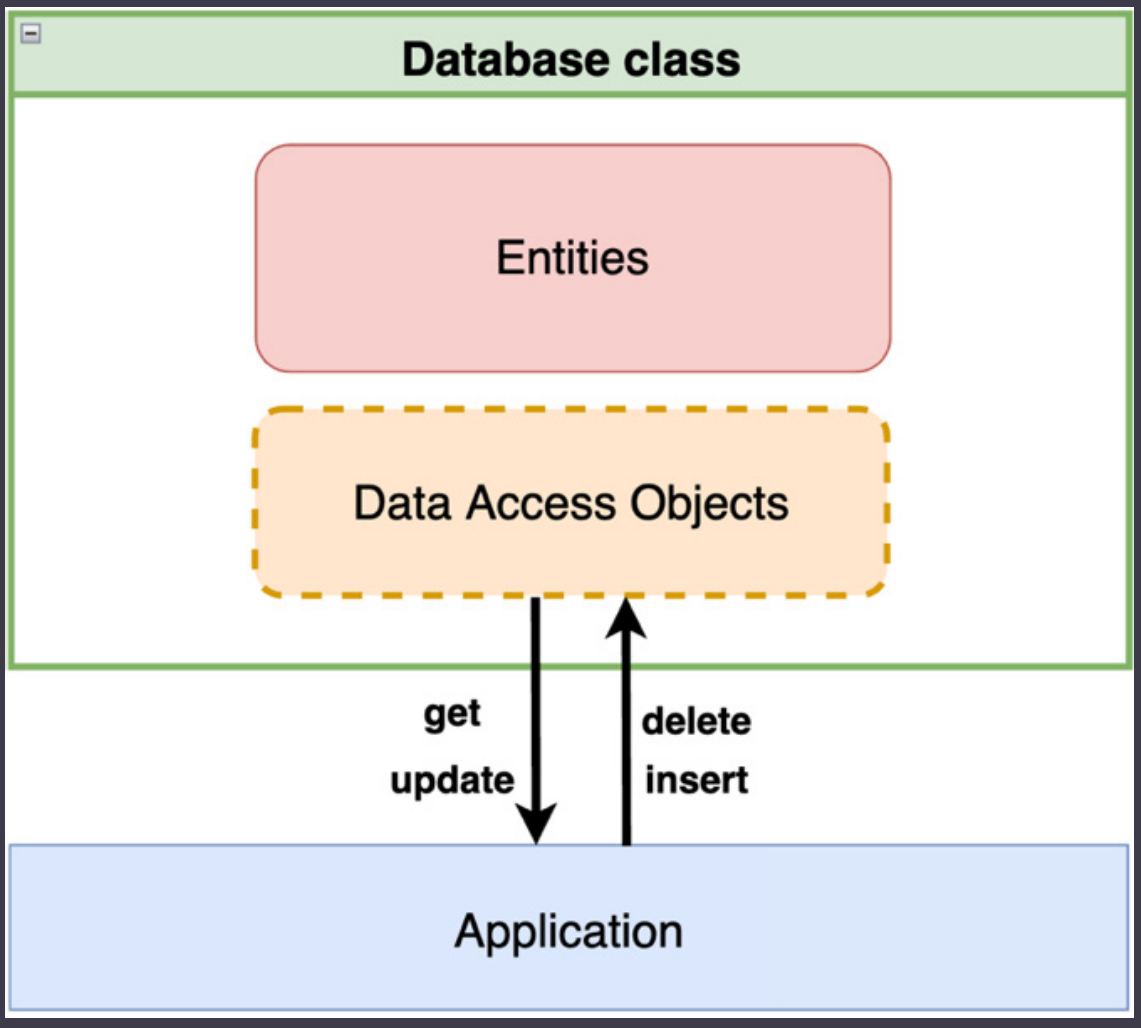

Figure 6.1 – The interaction between the application and the Room database

As previously illustrated, we can use the DAO to retrieve or update the data from the database in the form of entity objects – in our case, the entity is the restaurant, so we will be applying such operations to restaurant objects.

Now that we have a basic understanding of how Room works and how we can interact with it, it's time to see it in action for ourselves and implement Room in our Restaurants app.

Enabling offline usage by implementing Room

We want to locally cache all the restaurants that we receive from our Firebase database. Since this content is structured, we want to use Room to help us with this task.

Essentially, we are trying to save the restaurants when the user is browsing our Restaurants app while online. Then, we will reuse them when the user browses the app while being offline:

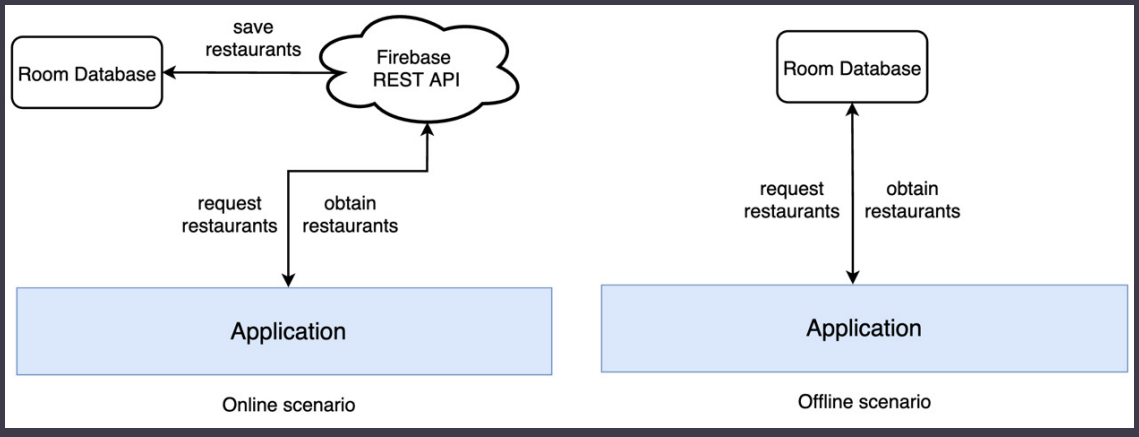

Figure 6.2 – Data retrieval for the Restaurants app with two sources of truth

When online, we retrieve the restaurants from our web API. Before displaying them to the user, first, we will cache them to our Room database. If offline, we will retrieve the restaurants from the Room database and then display them to the user.

Essentially, we are creating two sources of truth for our app:

- The remote API for when the user is online
- The local Room database for when the user is offline

In the next section, we will discuss why this approach is not ideal. However, until then, we are content with the fact that we will be able to use the app while remaining offline.

Let's start implementing Room, and then let's cache those restaurants! Perform the following steps:

1. Inside the **`build.gradle`** file in the app module, add the dependencies for Room inside the **`dependencies`** block:

```
implementation "androidx.room:room-runtime:2.4.2"
kapt "androidx.room:room-compiler:2.4.2"
implementation "androidx.room:room-ktx:2.4.2"
```

2. While you are still inside the **`build.gradle`** file, add the **`kotlin-kapt`** plugin for Room inside the **`plugins`** block:

```
plugins {
    id 'com.android.application'
    id 'kotlin-android'
    id 'kotlin-kapt'
}
```

The **`kapt`** plugin stands for **Kotlin Annotation Processing Tool**. This allows Room to generate annotated code at compile time while hiding most of the associated complexity from us.

After updating the **`build.gradle`** files, make sure to sync your project with its Gradle files. You can do that by pressing on the **File** menu option and then by selecting **Sync Project with Gradle Files**.

3. Since we want to store restaurant objects inside our local database, let's instruct Room that the **`Restaurant`** data class is an entity that must be saved. Head inside the **`Restaurant.kt`** file, and add the **`@Entity`** annotation on top of the class declaration:

```
@Entity(tableName = "restaurants")
data class Restaurant(…)
```

Inside the **`@Entity`** annotation, we have passed the name of the table via the **`tableName`** parameter. We will use this name when making queries.

4. Now that Room will create a table with **`Restaurant`** objects as rows, it's time to define the columns (or fields) of the entity. While we are still inside the **`Restaurant.kt`** class, let's add the **`@ColumnInfo`** annotation on top of each field that we're interested in, and that should represent a column:

```
@Entity(tableName = "restaurants")
data class Restaurant(
    @ColumnInfo(name = "r_id")
    @SerializedName("r_id")
    val id: Int,
    @ColumnInfo(name = "r_title")
    @SerializedName("r_title")
    val title: String,
    @ColumnInfo(name = "r_description")
    @SerializedName("r_description")
    val description: String,
    var isFavorite: Boolean = false
)
```

For each field we're interested in saving, we've added the **`@ColumnInfo`** annotation and passed a **`String`** value to the **`name`** parameter. These names

will correspond to the name of the table's columns. Right now, we are not interested in saving the **`isFavorite`** field; we will do that a bit later.

5. The entity that represents a table should have a primary key column that ensures uniqueness within the database. For this, we can use the **`id`** field that was configured from our Firebase database to be unique. While still inside the **`Restaurant.kt`** class, let's add the **`@PrimaryKey`** annotation to the **`id`** field:

```
@Entity(tableName = "restaurants")
data class Restaurant(
    @PrimaryKey()
    @ColumnInfo(name = "r_id")
    @SerializedName("r_id")
    val id: Int,
    …)
```

Now we have defined the entity for our database and configured the table's columns.

It's time to create a DAO that will serve as the entry point to our database, allowing us to perform various actions on it.

6. Create a DAO by clicking on the application package, selecting **New**, and then selecting **Kotlin Class/File**. Enter **`RestaurantsDao`** as the name, and select **Interface** as the type. Inside the new file, add the following code:

```
import androidx.room.*
@Dao
interface RestaurantsDao { }
```

Since Room will take care of implementing any actions that we need to interact with the database, the DAO is an interface, just like Retrofit also had an interface for the HTTP methods. To instruct Room that this is a DAO entity, we've added the **`@Dao`** annotation on top of the interface declaration.

7. Inside the **RestaurantsDao** interface, add two **suspend** functions that will help us to both save the restaurants and retrieve them from the database:

```
@Dao
interface RestaurantsDao {
    @Query("SELECT * FROM restaurants")
    suspend fun getAll(): List<Restaurant>
    @Insert(onConflict =
OnConflictStrategy.REPLACE)
    suspend fun addAll(restaurants:
List<Restaurant>)
}
```

Now, let's analyze the two methods that we've added:

- **getAll()** is a query statement that returns the restaurants that were previously cached inside the database. Since we need to perform a SQL query when calling this method, we've marked it with the **@Query** annotation and specified that we want all the restaurants (by adding *) from the **restaurants** table defined in the **Restaurant** entity data class.
- **addAll()** is an **insert** statement that caches the received restaurants inside the database. To mark this as a SQL **insert** statement, we've added the **@Insert** annotation. However, if the restaurants being inserted are already present in the database, we should replace the old ones with the new ones to refresh our cache. We instructed Room to do so by passing the **OnConflictStrategy.REPLACE** value into the **@Insert** annotation.

Both methods are marked as **suspend** functions because any interaction with the Room database can take time and is an async job; therefore, it shouldn't block the UI.

Now, we have defined an entity class and a DAO class, we must define the last component that Room needs in order to function, the database class.

8. Create a Room database class by clicking on the application package. Select **New**, and then select **Kotlin Class/File**. Enter **`RestaurantsDb`** as the name, and select **File** as the type. Inside the new file, add the following code:

```
@Database(
    entities = [Restaurant::class],
    version = 1,
    exportSchema = false)
abstract class RestaurantsDb : RoomDatabase() { }
```

Now, let's analyze the code that we've just added:

- **`RestaurantsDb`** is an abstract class that inherits from **`RoomDatabase()`**. This will allow Room to create the actual implementation of the database behind the scenes and hide all the heavy implementation details from us.
- For the **`RestaurantsDb`** class, we've added the **`@Database`** annotation so that Room knows that this class represents a database and provides an implementation for it. Inside this annotation, we've passed the following:
- The **`Restaurant`** class to the **`entities`** parameter. This parameter tells Room which entities are associated with this database so that it can create corresponding tables. The parameter expects an array, so you can add as many entity classes as you wish, as long as they are annotated with **`@Entity`**.
- **`1`** as the **`version`** number of the database. We should increment this version number whenever the schema of the database changes. The **schema** is the collection of database objects, such as the tables that correspond to entities. If we change the **`Restaurant`** class, since it's an entity, we might change the schema of the database, and Room needs to know that for migration purposes.
- **`false`** to the **`exportSchema`** parameter. Room can export the schema of our database externally; however, for simplicity, we chose not to do so.

9. Inside the **RestaurantsDb** class, add an abstract **RestaurantsDao** variable:

```
@Database(…)
abstract class RestaurantsDb : RoomDatabase() {
    abstract val dao: RestaurantsDao
}
```

We know that the database class should expose a DAO object so that we can interact with the database. By leaving it abstract, we allow Room to provide its implementation behind the scenes.

10. Even though we declared a variable to hold our DAO object, we still need to find a way to build the database and obtain a reference to the **RestaurantsDao** instance that Room will create for us. Inside the **RestaurantsDb** class, add **companion object** and then add the **buildDatabase** method:

```
@Database(…)
abstract class RestaurantsDb : RoomDatabase() {
    abstract val dao: RestaurantsDao
    companion object {
        private fun buildDatabase(context:
Context):
            RestaurantsDb =
            Room.databaseBuilder(
                context.applicationContext,
                RestaurantsDb::class.java,
                "restaurants_database")
                .fallbackToDestructiveMigration()
                .build()
    }
}
```

Essentially, this method returns a **RestaurantsDb** instance. To construct a Room database, we need to call the **Room.databaseBuilder** constructor, which expects the following parameters:

- A **`Context`** object that we provided from the **`context`** input argument of our **`buildDatabase`** method.
- The class of the database you're trying to build, that is, the **`RestaurantsDb`** class.
- A name for the database – we named it **`"restaurants_database"`**.

The builder returns a **`RoomDatabase.Builder`** object on which we called **`.fallbackToDestructiveMigration()`**. This means that, in the case of a schema change (such as performing changes in the entity class and modifying the table columns), the tables would be dropped (or deleted) instead of trying to migrate the contents from the previous schema (which would have been a bit more complex).

Finally, we called **`build()`** on the builder object so that our **`buildDatabase()`** method returns a **`RestaurantsDb`** instance.

It's time to finally get a reference to our DAO so that we can start using the database.

11. While still inside the **`companion object`** of the **`RestaurantsDb`** class, add the following code:

```
companion object {
    @Volatile
    private var INSTANCE: RestaurantsDao? = null
    fun getDaoInstance(context: Context):
RestaurantsDao
    {
        synchronized(this) {
            var instance = INSTANCE
            if (instance == null) {
                instance =
buildDatabase(context).dao
                INSTANCE = instance
            }
            return instance
```

```
            }
        }
        private fun buildDatabase(…) = …
    }
```

Now, let's break down what we've done:

- We added an **`INSTANCE`** variable of type **`RestaurantsDao`**. Since this variable is inside the companion object, **`INSTANCE`** is static. Additionally, we marked it with **`@Volatile`**. This means that writes to this field are immediately made visible to other threads. Don't worry too much about these multithreading concepts – we will get rid of this boilerplate code soon enough.
- We created a **`getDaoInstance()`** method where we added a block of code that calls the **`buildDatabase()`** method and gets the DAO object by calling the **`.dao`** accessor.

Since we want only one memory reference to our database (and not create other database instances in other parts of the app), we made sure that our **`INSTANCE`** variable conforms to the singleton pattern. Essentially, the **singleton pattern** allows us to hold a static reference to an object so that it lives as long as our application does.

By following this approach, anytime we need to access the Room database from different parts of the app, we can call the **`getDaoInstance()`** method, which returns an instance of **`RestaurantsDao`**. Additionally, we can be sure that it's always the same memory reference and that no concurrency issues will occur since we have wrapped the instance creation code inside a **`synchronized`** block.

12. You might have noticed that to get a reference to our DAO and cache our restaurants in the database, the **`RestaurantsDb.getDaoInstance()`** method expects a **`Context`** object. This is needed to create the instance of the database. However, we want to get our DAO in the

`RestaurantsViewModel` class, and we have no context there, so what should we do?

Let's expose the application context from the application class! Create the application class by clicking on the application package, selecting **New**, and then selecting **Kotlin Class/File**. Enter **`RestaurantsApplication`** as the name, and select **File** as the type. Inside the new file, add the following code:

```
class RestaurantsApplication: Application() {
    init { app = this }
    companion object {
        private lateinit var app:
RestaurantsApplication
        fun getAppContext(): Context =
            app.applicationContext
    }
}
```

This class now inherits from **`android.app.Application`** and exposes its context through the static **`getAppContext()`** method. The only issue is that even though we have an application class, we still haven't configured the project to recognize it.

13. In the **`AndroidManifest.xml`** file, inside the **`<application>`** element, add the **`android:name`** identifier that sets our **`RestaurantsApplication`** class as the application class:

```
<application
    android:allowBackup="true"
    android:name=".RestaurantsApplication"
    android:icon="@mipmap/ic_launcher"
    …
    <activity> … </activity>
</application>
```

It's time to finally start working on caching those restaurants in our database.

14. Inside the **`RestaurantsViewModel`** class, add a **`restaurantsDao`** variable. Then, instantiate it via the static **`RestaurantsDb.getDaoInstance`** method:

```
class RestaurantsViewModel(…) : ViewModel() {
    private var restInterface:
RestaurantsApiService
    private var restaurantsDao = RestaurantsDb
        .getDaoInstance(
            RestaurantsApplication.getAppContext()
        )
     ....
}
```

Make sure that you pass the application context through the newly created **`getAppContext()`** method inside the application class.

15. Now we're ready to save the restaurants locally! While you are still in the **`RestaurantsViewModel`** class, inside the **`getRemoteRestaurants()`** method, add these new lines of code:

```
private suspend fun getRemoteRestaurants():
    List<Restaurant> {
    return withContext(Dispatchers.IO) {
        val restaurants =
restInterface.getRestaurants()
        restaurantsDao.addAll(restaurants)
        return@withContext restaurants
    }
}
```

Essentially, what we are doing is the following:

I. Getting the restaurants from the remote API (here, it's the Retrofit **`restInterface`** variable).

II. Caching those restaurants inside the local database through Room by calling **`restaurantsDao.addAll()`**.

III. Finally, returning the restaurants to the UI.

16. Run the app while you have a working internet connection.

In terms of the UI, nothing should change – you should still see the restaurants. That said, behind the scenes, the restaurants should now have been cached.

17. Run the app again but without internet.

The chances are that you won't see anything. The restaurants are not there.

This happens because, while we are offline, we never try to get the previously cached restaurants from the Room database. Moreover, when offline, the **`restInterface.getRestaurants()`** suspending function throws an error because the HTTP call that fetches the restaurants has failed – this exception should arrive inside **`CoroutineExceptionHandler`**. The exception is thrown by Retrofit because the associated network request has failed.

18. Let's leverage the fact that, while we're offline, the **`restInterface.getRestaurants()`** function call throws an exception. This is so that we can wrap the whole block of code inside **`getRemoteRestaurants()`** inside a **`try-catch`** block:

```
  private suspend fun getRemoteRestaurants():
          List<Restaurant> {
      return withContext(Dispatchers.IO) {
          try {
              val restaurants = restInterface
                  .getRestaurants()
```

```
                restaurantsDao.addAll(restaurants)
                return@withContext restaurants
            } catch (e: Exception) {
                when (e) {
                    is UnknownHostException,
                    is ConnectException,
                    is HttpException -> {
                        return@withContext
                            restaurantsDao.getAll()
                    }
                    else -> throw e
                }
            }
        }
    }
```

Essentially, what happens now is that if the user is offline, we catch the exception thrown by Retrofit. Alternatively, we return the cached restaurants from the Room database by calling **`restaurantsDao.getAll()`**.

As an extra, we also check whether the exception we've caught has been thrown because of the user's poor or inexistent internet connectivity. If the **`Exception`** object is of type **`UnknownHostException`**, **`ConnectException`**, or **`HttpException`**, we're loading the restaurants from Room through our DAO; otherwise, we propagate the exception so that it's caught later by **`CoroutineExceptionHandler`**.

19. Before running the app, let's refactor our **`getRemoteRestaurants()`** method a bit. Now, the name of the method implies that it retrieves restaurants from a remote source. However, in reality, it also retrieves restaurants from Room if the user is offline. Room is a local data source, so the name of this method is no longer appropriate.

Rename the **`getRemoteRestaurants()`** method to **`getAllRestaurants()`**:

```
private suspend fun getAllRestaurants():
    List<Restaurant> {  }
```

Additionally, remember to rename its usage in the **`getRestaurants()`** method where the coroutine is launched:

```
private fun getRestaurants() {
    viewModelScope.launch(errorHandler) {
        val restaurants = getAllRestaurants()
        state.value = restaurants.restoreSelections()
    }
}
```

20. Run the app again without an internet connection.

Because the restaurants were previously cached and now the user is off-line, we are fetching them from Room. You should see the restaurants even without the internet. Success!

Even though we've come a long way and have managed to make the Restaurants app usable even without internet, there is still something that we've missed. To reproduce it, perform the following steps:

1. Try running the application (either online or offline), and then mark a couple of restaurants as favorites.
2. Disconnect your device from the internet and make sure you are now offline.
3. Restart the application while remaining offline.

You will get to see the restaurants, but your previous selections have been lost. More precisely, even though we have marked some restaurants as favorites, all restaurants now appear as not favorites. It's time to fix this!

Applying partial updates to the Room database

Right now, our application is saving the restaurants that we receive from the remote web API directly inside the Room database.

This is not a bad approach; however, whenever we are marking a restaurant as a favorite, we aren't updating the corresponding restaurant inside Room. If we take a look inside the **`RestaurantsViewModel`** class and we check its **`toggleFavorite()`** method, we can see that we're only updating the **`isFavorite`** flag of a restaurant inside the **`state`** variable:

```
fun toggleFavorite(id: Int) {
    val restaurants = state.value.toMutableList()
    val itemIndex = restaurants.indexOfFirst { it.id
== id }
    val item = restaurants[itemIndex]
    restaurants[itemIndex] = item.copy(isFavorite =
        !item.isFavorite)
    storeSelection(restaurants[itemIndex])
    state.value = restaurants
}
```

We aren't updating the corresponding restaurant's **`isFavorite`** field value inside Room. So, whenever we use the application offline, the restaurants will no longer appear as favorites, even though when we were online, we might have marked some as favorites.

To fix this, whenever we mark a restaurant as a favorite or not a favorite, we need to apply a partial update on a particular **`Restaurant`** object inside our Room database. The partial update should not replace the entire **`Restaurant`** object, but it should only update its **`isFavorite`** field value.

Let's get started! Perform the following steps:

1. Create a partial entity class by clicking on the application package, selecting **New**, and then selecting **Kotlin Class/File**. Enter **`PartialRestaurant`** as the name, and select **File** as the type. Inside the new file, add the following code:

```
@Entity
class PartialRestaurant(
    @ColumnInfo(name = "r_id")
    val id: Int,
    @ColumnInfo(name = "is_favorite")
    val isFavorite: Boolean)
```

In this **`@Entity`** annotated class, we've only added two fields:

- An **`id`** field with a **`@ColumnInfo()`** annotation that has the same value (**`"r_id"`**) passed to the **`name`** parameter as the **`Restaurant`** object's **`id`** field. This allows Room to match the **`Restaurant`** object's **`id`** field with the one from **`PartialRestaurant`**.
- An **`isFavorite`** field with a **`@ColumnInfo()`** annotation that has the name set to **`"is_favorited"`**. So far, Room can't match this field with the one from **`Restaurant`**, because inside **`Restaurant`**, we haven't annotated the **`isFavorite`** field with **`@ColumnInfo`** – we'll do that next.

2. Now that our partial entity, called **`PartialRestaurant`**, has a column corresponding to the **`isFavorite`** field, it's time to also add a **`@ColumnInfo()`** annotation with the same value (**`"is_favorite"`**) for the **`isFavorite`** field of the **`Restaurant`** entity:

```
@Entity(tableName = "restaurants")
data class Restaurant(
    …
    val description: String,
    @ColumnInfo(name = "is_favorite")
    val isFavorite: Boolean = false
)
```

As a good practice, we've also made the **`isFavorite`** field **`val`** instead of **`var`** to prevent its value from being changed once the object has been created. Because **`Restaurant`** is an object passed to a Compose **`State`** object, we want to promote immutability across its fields to ensure recomposition events happen.

NOTE

By having a data class field as **`var`***, we can easily change its value at run-time and risk having Compose miss a well-needed recomposition. Immutability ensures that whenever an object field's value changes, a new object is created (just as we do with the* **`.copy()`** *function), and Compose is notified so that it can trigger recomposition.*

3. Since the **`isFavorite`** field is now **`val`**, the **`restoreSelections()`** extension function inside **`RestaurantViewModel`** has broken. Update its code as follows:

```
private fun List<Restaurant>.restoreSelections(): …
{
    stateHandle.[…]let { selectedIds ->
        val restaurantsMap = this.associateBy {
it.id }
            .toMutableMap()
        selectedIds.forEach { id ->
            val restaurant =
                restaurantsMap[id] ?:
return@forEach
            restaurantsMap[id] =
                restaurant.copy(isFavorite = true)
        }
        return restaurantsMap.values.toList()
    }
    return this
}
```

Essentially, what we have done is make sure our **`restaurantsMap`** of type **`Map<Int, Restaurant>`** is mutable so that we can replace elements inside it. With this approach, we are now replacing the restaurant at entry **`id`** by passing a new object reference with the **`copy`** function. We are not going to go into much detail since this portion of the code will soon be removed.

4. Now that we have a partial entity defined, we need to add another function inside our DAO that will update a **`Restaurant`** entity through a **`PartialRestaurant`** entity. Inside **`RestaurantsDao`**, add the **`update()`** function:

```
@Dao
interface RestaurantsDao {
    …
    @Insert(onConflict =
OnConflictStrategy.REPLACE)
    suspend fun addAll(restaurants:
List<Restaurant>)
    @Update(entity = Restaurant::class)
    suspend fun update(partialRestaurant:
        PartialRestaurant)
}
```

Let's understand, step by step, how the new **`update()`** function works:

I. It's a **`suspend`** function because, as we know by now, any interaction with the local database is a suspending job that should not run on the main thread.

II. It receives a **`PartialRestaurant`** entity as an argument and returns nothing. The partial entity's field values correspond to the restaurant that we're trying to update.

III. It's annotated with the **`@Update`** annotation to which we passed the **`Restaurant`** entity. The update process has two steps, as follows:

i. First, **`PartialRestaurant`** exposes the **`id`** field, whose value matches the **`id`** field's value of the corresponding **`Restaurant`** object.

ii. Once the match is complete, the **`isFavorite`** field's value is set to the **`isFavorite`** field of the matched **`Restaurant`** object.

These matches are possible because the **id** and **isFavorite** fields of both entities have the same **@ColumnInfo** name values.

5. Now that our DAO knows how to partially update our **Restaurant** entity, it's time to perform the update.

First, inside **RestaurantsViewModel**, add a new suspending function, called **toggleFavoriteRestaurant()**:

```
private suspend fun toggleFavoriteRestaurant(id: Int,
oldValue: Boolean) =
    withContext(Dispatchers.IO) {
        restaurantsDao.update(
            PartialRestaurant(
                id = id,
                isFavorite = !oldValue
            )
        )
    }
```

Let's understand, step by step, what this new method does:

I. It receives the **id** field of the restaurant that we're trying to update, along with the **oldValue** field which represents the value of the **isFavorite** field just before the user has toggled the heart icon of the restaurant.

II. To partially update a restaurant, it needs to interact with the Room DAO object. This means that the **toggleFavoriteRestaurant** method must be a **suspend** function. As a good practice, we wrapped it inside a **withContext** block that specifies its work must be done inside the **IO** dispatcher. While Room ensures that we wrap our suspending work with a special dispatcher, we explicitly specified the **Dispatchers.IO** dispatcher to better highlight that such heavy work should be done in an appropriate dispatcher.

III. It builds a **`PartialRestaurant`** object, which it then passes to the DAO's **`update()`** method that was created earlier. The **`PartialRestaurant`** object gets the **`id`** field of the restaurant we're updating, along with the negated value of the **`isFavorite`** flag. If the user previously didn't have the restaurant marked as favorite, upon clicking the heart icon, we should negate the old (**`false`**) value and obtain **`true`**, or vice versa.

Now that we have the method in place to update a restaurant, it's time to call it.

6. While you are still in **`RestaurantsViewModel`**, make the **`toggleFavorite`** method launch a coroutine at the end of its body. Then, inside it, call the new **`toggleFavoriteRestaurant()`** suspending function:

```
fun toggleFavorite(id: Int) {
    …
    restaurants[itemIndex] = item.copy(isFavorite
=
       !item.isFavorite)
    storeSelection(restaurants[itemIndex])
    state.value = restaurants
    viewModelScope.launch {
        toggleFavoriteRestaurant(id,
item.isFavorite)
    }
}
```

To the **`toggleFavoriteRestaurant()`** function, we've passed the following:

- The **`id`** parameter, which represents the ID of the restaurant the user is trying to mark as favorite or not favorite
- The old value of the favorite status of the restaurant, as defined by the **`isFavorite`** flag of the **`item`** field

Now, whenever the user presses on the heart icon, we not only update the UI but also cache this selection inside the local database through a partial

update.

7. Build and run the application because it's time to test what we've just implemented! Unfortunately, the app crashes. Can you think of one reason why this happens? If we look at the stack trace of the error, we will see the following message:

```
java.lang.IllegalStateException: Room cannot verify
the data integrity. Looks like you've changed
schema but forgot to update the version number.
```

This error message makes total sense because we've changed the schema of the database, and now Room doesn't know whether to migrate the old entries or delete them. But how did we change the schema?

Well, we changed the schema when we defined a new column for the Restaurants table by adding the **`@ColumnInfo()`** annotation to the **`isFavorite`** field.

8. To mitigate this issue, we must increase the **`version`** number of the database. Inside the **`RestaurantsDb`** class, increase the **`version`** number from **`1`** to **`2`**:

```
@Database(
    entities = [Restaurant::class],
    version = 2,
    exportSchema = false)
abstract class RestaurantsDb : RoomDatabase() { ..
}
```

Now, Room knows that we've changed the schema of the database. In turn, because we haven't provided a migration strategy, and instead, we've called the **`fallbackToDestructiveMigration()`** method in the **`Room.databaseBuilder`** constructor when we initially instantiated the database, Room will drop the old contents and tables and provide us with a fresh start.

9. Try running the application online, and then mark a couple of restaurants as favorites.
10. Disconnect your device from the internet and make sure you are now offline.
11. Restart the application while remaining offline.

Great news! The selections were now kept, and we can see which restaurants were previously marked as favorites!

12. To continue testing, while you are offline, you can try marking other restaurants as favorites.

Then, still in offline mode, restart the app and you will notice that these new selections have also been saved.

13. Connect your device to the internet and run the application – while you are online.

Oops! The restaurants that we have previously marked as favorites no longer appear as such, even though we previously cached these selections inside the Room database.

Essentially, every time we open the application while being connected to the internet, we lose all the previous selections, and no restaurant is marked as favorite anymore.

There are two issues in our code that are causing this! Can you think of why this is happening?

In the next section, we will identify and address them. Additionally, we will make sure that Room is the single source of truth for the content of our application.

Making local data the single source of truth for app content

Whenever we launch the app with the internet, all the restaurants appear as not favorites, even though we previously marked them as favorites and cached the selections in the Room database.

To identify the issue, let's navigate back inside **`RestaurantsViewModel`** and inspect the **`getAllRestaurants()`** method:

```
private suspend fun getAllRestaurants():
List<Restaurant> {
    return withContext(Dispatchers.IO) {
        try {
            val restaurants =
restInterface.getRestaurants()
            restaurantsDao.addAll(restaurants)
            return@withContext restaurants
        } catch (e: Exception) {
            when (e) {
                is UnknownHostException, […] -> {
                    return@withContext
restaurantsDao.getAll()
                }
                else -> throw e
            }
        }
    }
}
```

Now, when we launch the app while online, we do three things:

- We load the restaurants from the server by calling **`restInterface.getRestaurants()`**. For these restaurants, we don't receive the **`isFavorite`** flag, so we automatically have it set to **`false`**. This happens because our **`Restaurant`** class defaults the value of **`isFavorite`** to false if no value is passed from the Gson deserialization:

```
@Entity(tableName = "restaurants")
data class Restaurant(
```

```
    …
    @ColumnInfo(name = "is_favorite")
    val isFavorite: Boolean = false)
```

- Then, we save those restaurants to Room by calling **`restaurantsDao.addAll(restaurants)`**. However, because we've used the **`REPLACE`** strategy inside our DAO's **`addAll()`** function, and because we received the same restaurants from the server, we override the **`isFavorite`** flags of the corresponding restaurants inside the database to **`false`**. So, even though our restaurants in Room might have had the **`isFavorite`** flag set to **`true`**, because we receive restaurants with the same **`id`** fields from the server, we end up overriding all those values to **`false`**.
- Next, we pass the **`restaurants`** list that we've received from the server to the UI. As we already know, these restaurants have the **`isFavorite`** field's value of **`false`**. So, anytime we start the app while connected to the internet, we will always see no restaurants marked as favorites.

If we think about it, there are two main issues here:

- Our application has two sources of truth:
  - When online, it displays the restaurants from the remote server.
  - When offline, it displays the restaurants from the local database.
- Whenever we cache restaurants that already exist inside the local database, we override their **`isFavorite`** flag to **`false`**.

If we can fix these two issues by having our UI receive content from a single source of data, we will also be able to remove the need for **`SavedStateHandle`** and all the special handling related to process recreation – we will see why in a moment.

Essentially, in this section, we will be doing the following:

- Refactoring the Restaurants app to have a single source of truth for data

- Removing the logic of persisting state inside `SavedStateHandle` in the case of process recreation

So, let's begin with the first issue at hand!

Refactoring the Restaurants app to have a single source of truth for data

The approach of having multiple sources of data can lead to many inconsistencies and subtle bugs – just like how our app is now inconsistent in terms of what data it displays when the user is either online or offline.

NOTE

The concept of designing systems to rely on only one data source used for storing and updating content is related to a practice that is called **Single Source of Truth** *(***SSOT***). Having multiple sources of truth for data that the UI consumes can lead to inconsistencies between what's expected to be shown to the UI and what is actually shown. The SSOT concept helps us to structure the data access so that only one data source is trusted to provide the app with data.*

Let's make sure that our application only has one source of truth, but which one should we choose?

On the one hand, we cannot control the data that is being sent from our Firebase database, and we also can't update the restaurants stored inside it when the user marks one as a favorite.

On the other hand, we can do that with Room! In fact, we are already doing that – every time a user marks a restaurant as a favorite or not a favorite, we're applying a partial update to that restaurant inside the local database.

So, let's make the local Room database our only source of data:

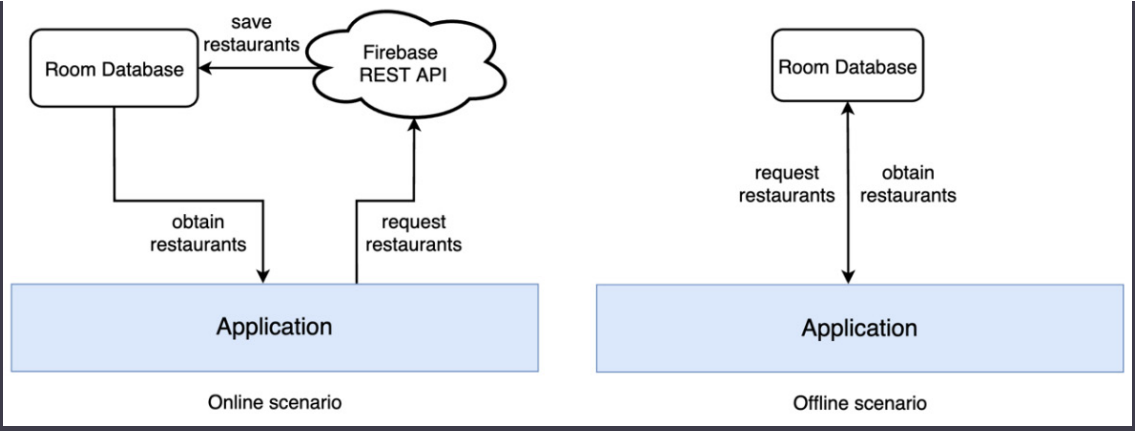

Figure 6.3 – Data retrieval for the Restaurants app with the local database as an SSOT

When the user is online, we should get the restaurants from the server, cache them into Room, and then obtain the restaurants again from Room to finally send them to the UI.

Similarly, if the user is offline, we simply obtain the restaurants from Room and display them.

NOTE

Alternatively, instead of always asking your Room database for the most up-to-date content, you could update the DAO interface to provide you with a reactive data stream that we can observe. This way, upon every data update, you would automatically be notified with the most up-to-date content in a reactive manner, without having to manually ask for it. To achieve that, you must use special data holders provided by libraries such as Jetpack LiveData, Kotlin Flow, or RxJava. We will explore Kotlin Flow in Chapter 11, Creating Infinite Lists with Jetpack Paging and Kotlin Flow.

The similarity between our two scenarios is that now, regardless of the internet connectivity of the user, our UI always displays the restaurants from inside our Room database. In other words, the local database is our SSOT!

Let's start implementing! Perform the following steps:

1. Inside **`RestaurantsViewModel`**, refactor the **`getAllRestaurants()`** function to always return the restaurants from the Room database:

```
private suspend fun getAllRestaurants():
     List<Restaurant> {
    return withContext(Dispatchers.IO) {
        try { … } catch (e: Exception) { […] }
        return@withContext restaurantsDao.getAll()
    }
}
```

Here, our app tries to display the restaurants from the local database in any condition.

2. Now, it's time to refactor the **`try – catch`** block inside the **`getAllRestaurants()`** method! Essentially, what we want to do is to get the restaurants from the server and then cache them locally.

Replace the contents within the **`try { }`** block with a new **`refreshCache()`** method:

```
return withContext(Dispatchers.IO) {
    try {
        refreshCache()
    } catch (e: Exception) { […] }
    return@withContext restaurantsDao.getAll()
}
```

3. Additionally, we want to define the **`refreshCache()`** function to get the restaurants from the remote server and then cache them inside the local database, thereby refreshing their contents:

```
private suspend fun refreshCache() {
    val remoteRestaurants = restInterface
        .getRestaurants()
    restaurantsDao.addAll(remoteRestaurants)
}
```

4. We know that if the refresh of the cache fails, we will still show the local restaurants from Room. But what if the local database is empty?

Continue refactoring the **`getAllRestaurants()`** method by updating its **`catch`** block. You can do this by removing the **`return@withContext restaurantsDao.getAll()`** call (which is now redundant) from the **`is UnknownHostException, is ConnectException, is HttpException`** branch and by replacing it with the following code:

```
try { … } catch (e: Exception) {
    when (e) {
        is UnknownHostException, is ConnectException,
        is HttpException -> {
            if (restaurantsDao.getAll().isEmpty())
                throw Exception(
                    "Something went wrong. " +
                            "We have no data.")
        }
        else -> throw e
    }
}
```

Essentially, if a network exception has been thrown, we can check whether we have any local restaurants saved in the Room database:

- If the list is empty, we return from the parent method early by throwing a custom exception to inform the user that we have no data to display.
- However, if the local database has elements, we do nothing and let the **`getAllRestaurants()`** method return the cached restaurants to the UI.

Now, inside the **`toggleFavorite()`** function of **`ViewModel`**, whenever we toggle a restaurant as a favorite or not, we can observe that we're updating the Room database with a partial update. However, we're not fetching the restaurants again from Room and so the UI is never informed of this change:

```
fun toggleFavorite(id: Int) {
    …
    restaurants[itemIndex] = item.copy(isFavorite =
        !item.isFavorite)
    storeSelection(restaurants[itemIndex])
    state.value = restaurants
    viewModelScope.launch {
        toggleFavoriteRestaurant(id, item.isFavorite)
    }
}
```

Instead, we're updating the **`state`** variable's value – so the UI receives the updated restaurants in-memory. This means that we are not conforming to the SSOT practice in which we opt to always feed the UI with restaurants from the local database. Let's fix this.

5. Make the **`toggleFavoriteRestaurant()`** function return the restaurants from our local database. You can do this by calling the **`restaurantsDao.getAll()`** function from inside the **`withContext()`** block:

```
private suspend fun toggleFavoriteRestaurant(
    id: Int,
    oldValue: Boolean
) = withContext(Dispatchers.IO) {
        restaurantsDao.update(
           PartialRestaurant(id = id, isFavorite =
               !oldValue))
        restaurantsDao.getAll()
    }
```

6. Inside the **`toggleFavorite()`** method, store the updated restaurants returned by the **`toggleFavoriteRestaurant()`** method inside an **`updatedRestaurants`** variable, and then move the **`state.value = restaurants`** line from outside the coroutine to inside it while, this time, making it receive the value stored by the **`updatedRestaurants`** variable:

```
fun toggleFavorite(id: Int) {
    val restaurants = state.value.toMutableList()
    […]
```

```
        storeSelection(restaurants[itemIndex])
        viewModelScope.launch(errorHandler) {
            val updatedRestaurants =
                toggleFavoriteRestaurant(id,
    item.isFavorite)
            state.value = updatedRestaurants
        }
    }
```

Here, we have not updated the **`state`** object value with the **`restaurants`** value from the previous state value. Instead, we passed the restaurants from the local database, which were obtained from the **`toggleFavoriteRestaurant()`** function.

Now that we have made our local database the single source of truth for data, we might assume that our issues have been solved. However, remember that we are still overriding the **`isFavorite`** field values of the local restaurants whenever we cache restaurants with the same IDs from the server.

That's why the final problem lies in the **`refreshCache()`** method:

```
  private suspend fun refreshCache() {
      val remoteRestaurants = restInterface
          .getRestaurants()
      restaurantsDao.addAll(remoteRestaurants)
  }
```

We must find a way to preserve the **`isFavorite`** field of the restaurants whenever we call **`restaurantsDao.addAll(remoteRestaurants)`**.

We can fix this issue by complicating the logic that is happening inside the **`refreshCache()`** function.

7. Inside the **`refreshCache()`** function, add the following code:

```
  private suspend fun refreshCache() {
```

```
        val remoteRestaurants = restInterface
            .getRestaurants()
        val favoriteRestaurants = restaurantsDao
            .getAllFavorited()
        restaurantsDao.addAll(remoteRestaurants)
        restaurantsDao.updateAll(
            favoriteRestaurants.map {
                PartialRestaurant(it.id, true)
            })
    }
```

Now, let's break down what we've just done:

i. First, just as before, we got the restaurants from the server (which will all have the **`isFavorite`** fields set to **`false`** as their default values) by calling **`restInterface.getRestaurants()`**.

ii. Then, from Room, we obtained all the restaurants that were favorited by calling **`restaurantsDao.getAllFavorited()`** – we haven't added this function yet so don't worry if your code doesn't compile yet.

iii. Next, just as before, we saved the remote restaurants in Room by calling **`restaurantsDao.addAll(remoteRestaurants)`**. With this, we override the **`isFavorite`** field (to **`false`**) of the existing restaurants that have the same ID as **`remoteRestaurants`**.

iv. Finally, we partially updated all the restaurants within Room by calling **`restaurantsDao.updateAll()`**. To this method (which we have yet to implement), we are passing a list of **`PartialRestaurant`** objects.

These objects resulted from mapping the previously cached **`favoriteRestaurants`** objects of type **`Restaurant`** to objects of type **`PartialRestaurant`**, which have their **`isFavorite`** fields set to **`true`**. With this approach, we have now restored the **`isFavorite`** field's value for those favorited restaurants that were initially cached.

8. Inside **`RestaurantsDao`**, we must implement the two methods used earlier:

```
@Dao
interface RestaurantsDao {
    […]
    @Update(entity = Restaurant::class)
    suspend fun updateAll(partialRestaurants:
        List<PartialRestaurant>)
    @Query("SELECT * FROM restaurants WHERE
        is_favorite = 1")
    suspend fun getAllFavorited(): List<Restaurant>
}
```

We have added the following:

- The **`updateAll()`** method: This is a partial update that works in the same way as the **`update()`** method. Here, the only difference is that we update the **`isFavorite`** field for a list of restaurants instead of only one.
- The **`getAllFavorited()`** method: This is a query just like the **`getAll()`** method but more specific, as it obtains all the restaurants that have their **`isFavorite`** field values equal to **`1`** (which stands for **`true`**).

We are finally done! It's time to test out the app!

9. Try running the application offline and then mark a couple of restaurants as favorites.
10. Connect your device to the internet and run the application – while you are online.

You should now be able to see the previous selections – all the restaurants that were originally marked as favorites are now persisted across any scenario.

However, we have one more thing to address!

Removing the logic of persisting state in the case of process recreation

Now our application has a single source of truth, that is, the local database:

- Whenever we receive restaurants from the server, we cache them to Room and then refresh the UI with the restaurants from Room.
- Whenever we mark a restaurant as a favorite or not, we cache the selection to Room, and similarly, we then refresh the UI with restaurants from Room.

This means that if a system-initiated process death occurs, we should be able to restore the UI state easily because, now, the restaurants in Room also have the **`isFavorite`** field cached.

In other words, our app no longer needs to rely on **`SavedStateHandle`** to restore the restaurants that have been favorited or not; the local source of data for our application will now handle this automatically.

Let's remove our special handling for a system-initiated process death:

1. Inside **`RestaurantsViewModel`**, remove the **`stateHandle: SavedStateHandle`** parameter:
   ```
   class RestaurantsViewModel() : ViewModel() { … }
   ```
2. Inside **`RestaurantsViewModel`**, remove the **`storeSelection()`** and the **`restoreSelections()`** methods.
3. Remove the **`companion object`** of the **`RestaurantsViewModel`** class.
4. While you are still inside **`ViewModel`**, remove all the logic related to the **`stateHandle`** variable from within the **`toggleFavorite()`** method. The method should now look like this:
   ```
   fun toggleFavorite(id: Int) {
       viewModelScope.launch(errorHandler) {
           val updatedRestaurants =
   ```

```
            toggleFavoriteRestaurant(id,
item.isFavorite)
            state.value = updatedRestaurants
        }
    }
```

The issue is that we no longer have the **`item`** variable, so we don't know what to pass to the **`toggleFavoriteRestaurant()`** function's **`oldValue`** parameter instead of **`item.isFavorite`**. We need to fix this.

5. Add a new parameter to the **`toggleFavorite()`** method, called **`oldValue`**:

```
fun toggleFavorite(id: Int, oldValue: Boolean) {
    viewModelScope.launch(errorHandler) {
        val updatedRestaurants =
            toggleFavoriteRestaurant(id, oldValue)
        state.value = updatedRestaurants
    }
}
```

This **`Boolean`** argument should tell us whether the restaurant was previously marked as favorite or not.

6. Following this, refactor the **`getRestaurants()`** method to no longer use the **`restoreSelections()`** method. The method should now look like this:

```
private fun getRestaurants() {
    viewModelScope.launch(errorHandler) {
        state.value = getAllRestaurants()
    }
}
```

7. Next, navigate to the **`RestaurantsScreen`** file. Then, inside the **`RestaurantItem`** composable, add another **`oldValue`** parameter to the **`onFavoriteClick`** callback function:

```
@Composable
fun RestaurantItem([…],
```

```
        onFavoriteClick: (id: Int, oldValue:
Boolean)
            -> Unit,
        onItemClick: (id: Int) -> Unit) {
    ...
    Card(…) {
        Row(…) {
            [...]
            RestaurantDetails(...)
            RestaurantIcon(icon,
Modifier.weight(0.15f))
            {
                onFavoriteClick(item.id,
item.isFavorite)
            }
        }
    }
}
```

Also, make sure that you pass the **`item.isFavorite`** value to the newly added parameter when the **`onFavoriteClick`** function is called.

8. Inside the **`RestaurantsScreen()`** composable, make sure you register and then pass the newly received **`oldValue`** function parameter to the **`toggleFavorite`** method of **`ViewModel`**:

```
@Composable
fun RestaurantsScreen(onItemClick: (id: Int) ->
Unit) {
    val viewModel: RestaurantsViewModel =
viewModel()
    LazyColumn(…) {
        items(viewModel.state.value) { restaurant -
>
            RestaurantItem(
                restaurant,
```

```
                onFavoriteClick = { id, oldValue ->
                    viewModel
                        .toggleFavorite(id,
oldValue)
                },
                onItemClick = { id ->
onItemClick(id) })
            }
        }
}
```

We're done! Now it's time to simulate the system-initiated process death scenario.

9. Build the project and run the application.
10. Mark some restaurants as favorites.
11. Place the app in the background by pressing the home button on the device/emulator.
12. Select the **Logcat** window and then press the red rectangular button on the left-hand side to terminate the application:

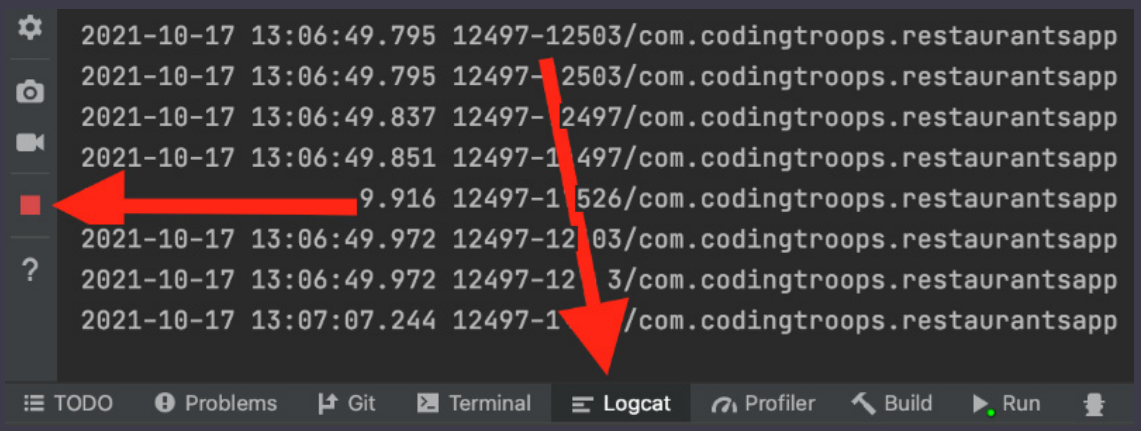

Figure 6.4 – Simulating a system-initiated process death

13. Relaunch the application from the application drawer.

Because the app relies on the content saved in the local database, it should now correctly display the UI state with the previously favorited restaurants from before the system-initiated process death.

ASSIGNMENT

In this chapter, we made sure to cache the restaurants in Room so that the first screen of the application could be accessed without the internet. As a homework assignment, you can try to refactor the details screen of the application (where the details of a specific restaurant are displayed) to obtain its own data from Room if the user enters the app without the internet.

Summary

In this chapter, we gained an understanding of how Room is an essential Jetpack library because it allows us to offer offline capabilities to our applications.

First, we explored the core elements of Room to see how a private database is set up. Second, we implemented Room inside our Restaurants application and explored how to save and retrieve cached content from the local database.

Afterward, we discovered what partial updates are and how to implement them to preserve a user's selections within the app.

Toward the end of the chapter, we understood why having a single source of truth for the application's content is beneficial and how that helps us in edge cases such as a system-initiated process death.

In the next chapter, we're going to dive deeper into various ways of defining the architecture of our applications by exploring architectural presentation patterns.