DreamCatcher App (Part 2 of 3)

In this second of three parts for this project, we will extend the app by incorporating a database for persistence, providing intuitive navigation back and forth between the list view and the detail view, and allowing new reflections to be added to any dream. This part closely corresponds to BNRG Chapters 12 through 14.

Getting Started

First, please be sure to retain a backup copy of your P2A project submission.

Next, you must get some new updates from the remote Git repository that you configured when starting Project 2A. To do this, select Git | Update Project and choose the option to Merge the incoming changes.

These updates provide the following changes specific to Project 2B:

* **build.gradle.kts** (Module :app) and (Project)
  + These configuration files have been updated to include several additional dependencies and plugins necessary to support this assignment
* **Dream.kt**
  + Annotations have been added to the existing Dream class to support its use with the Room database library (see details in Chapter 12 below)
* **dream-database**
  + This new file provides a prepopulated database for use when no existing database is found (see details in Chapter 12 below)
* **P2BTest.kt**
  + This new instrumented test file will help ensure that your app meets all of the P2B assignment requirements (please see the important naming conventions in the Testing section below)

Please use the above files as provided, without making any changes to them.

As you work through the BNRG chapters, there are several fundamental differences between DreamCatcher and CriminalIntent that require significant divergence from the textbook. The divergences are detailed below, by chapter and listing.

Phase 1: Chapter 12

Please be sure to skip the Chapter 12 sections before "Creating a Database" when developing the DreamCatcher app. These steps are important to understand when developing CriminalIntent, because they're meant to demonstrate low-level coroutines. However, the configuration files for this assignment won't support these steps.

Listing 12.10 - Dream.kt - Due to the complexity of our data structure, we need to include a few more annotations to our Dream.kt file than are needed for the relatively simple Crime class. Please carefully review the file provided for this project. The @Ignore annotation asks Room to ignore the entries property of the Dream object. Room can't directly support storing collections, so we'll instead store the entries in a separate table, in keeping with normal relational database practices. As a result, we must also define two distinct @Entity annotations, one for each table, and for convenience we'll explicitly provide the table names. Finally we must explicitly specify a unique column name for the DreamEntry id property, to avoid a Room quirk regarding duplicate column names (even in distinct tables).

Listing 12.11 - **database/DreamDatabase** - As noted above, we have two entities, so we must specify both of these in the @Database annotation:

**@Database(entities = [Dream::class, DreamEntry::class], version = 1)**

Listing 12.15 - **database/DreamDao** - This file is substantially more complex than BNRG. To fetch the list of dreams with their entries, the most straightforward way is to perform a join, which Room will return as a multimap structure. To fetch a single dream, a transaction is defined to first fetch the dream and then fetch its entries.

Notice that some functions start with "internal" as the name. These are functions that shouldn't be used outside of this class, but they can't actually be declared as private functions because this is an interface. The naming convention acts as a reminder to use them only locally.

**@Query("SELECT \* FROM dream d JOIN dream\_entry e ON e.dreamId = d.id ORDER BY d.lastUpdated DESC")**

**suspend fun getDreams(): Map<Dream, List<DreamEntry>>**

**@Query("SELECT \* FROM dream WHERE id=(:id)")**

**suspend fun internalGetDream(id: UUID): Dream**

**@Query("SELECT \* FROM dream\_entry WHERE dreamId = (:dreamId)")**

**suspend fun internalGetEntriesForDream(dreamId: UUID): List<DreamEntry>**

**@Transaction**

**suspend fun getDreamAndEntries(id: UUID): Dream {**

**return internalGetDream(id).apply { entries = internalGetEntriesForDream(id) }**

**}**

Listing 12.21 - **DreamRepository** - For getDreams() we must transform the multimap into a list of Dream objects. For the getDream() we may simply call the @Transaction function:

// Transform the DAO multimap into a list of dreams with their entries:

suspend fun getDreams(): List<Dream> {

**val dreamMap = database.dreamDao().getDreams()**

**return dreamMap.keys.map { dream ->**

**dream.apply { entries = dreamMap.getValue(dream) }**

**}**

}

// Call the DAO transaction function, to get the dream and its entries:

suspend fun getDream(id: UUID): Dream {

**return database.dreamDao().getDreamAndEntries(id)**

}

In the "Importing Prepopulated Data" section of Chapter 12, the dream-detail asset file has been provided when updating the project from the Git repository. This file prepopulates the database with 20 sample dreams, with varying combinations of entries per dream (but no more than five).

Listing 12.25 - **DreamRepository** - When implementing the Flow, we must adjust the getDreams() logic, in order to manipulate the structure within the Flow. The first map call below is actually a coroutine extension, so it will require an import from kotlinx.coroutines.flow.map, whereas the second map call below is the standard map function:

// Transform the DAO multimap within the Flow

**~~suspend~~** fun getDreams(): **Flow<**List<Dream>**>** {

val **~~dreamMap~~ dreamMapFlow** = database.dreamDao().getDreams()

return **dreamMapFlow.map { dreamMap ->**

dreamMap.keys.map { dream ->

dream.apply { entries = dreamMap.getValue(dream) }

}

**}**

}

Listing 12.27 (which should be titled "Collecting your Flow" since StateFlow is introduced in Listing 12.28): Because we skipped the initial sections of this chapter, the onViewCreated() function of our DreamListFragment won't match exactly what's in BNRG. Specifically, you'll first need to implement onViewCreated() from Listing 12.6, then adjust it according to Listing 12.27.

Chapter Requirements

At the end of this chapter, the DreamCatcher app should meet the following requirements:

* DLF should display the prepopulated dream list, as fetched from the database
* DLF should immediately reflect any changes made to any dream titles via the App Inspection's Database Inspector tool

Chapter Notes

* The Chapter 12 challenge has been implemented already within the provided configuration files
  + Still, as a challenge, use the Project view to find and review the generated 1.json schema file

Phase 2: Chapter 13

Listing 13.19 - **DreamDetailFragment** - This update will break the listeners, so please just comment-out everything in onViewCreated() for now (other than what's in Listing 13.19). Adjust updateView() to accept a dream parameter of type Dream, and change all the references within the updateView() function from vm.dream to dream.

Listing 13.20 - **DreamDetailViewModel** - When adding the updateDream() function, there some additional checks to be made. If the dream itself and its list of entries hasn't changed, no further changes should be made. However, if anything has changed, we must copy the dream, with a new last updated date, along with its entries:

**\_dream.update { oldDream ->**

**val newDream = oldDream?.let { onUpdate(it) } ?: return**

**if (newDream == oldDream && newDream.entries == oldDream.entries) {**

**return**

**}**

**newDream.copy(lastUpdated = Date()).apply { entries = newDream.entries }**

**}**

Listing 13.21 - **DreamDetailFragment** - Now you can un-comment the listeners again. You'll need to adjust the logic to use the new vm.updateDream() function, which takes a lambda that transforms the oldDream into the updated Dream. Keep in mind that you **must** make a copy the oldDream. Otherwise you're just making a change to the original object, in which case vm.updateDream() won't detect any changes when comparing the old to the updated.

As a bit of exciting news, you'll no longer need to call updateView() from the end of each listener, nor from the end of onViewCreated()! This is all handled by collecting on the StateFlow. Every change (including the initial loading from the database) will automatically call updateView() with the updated Dream. You'll be able to tell it's working because the last updated timestamp will change upon any successful update.

Any changes made to the Dream object at this point will survive device rotation. However, we're still not updating the database. When navigating Back from the detail fragment to the list fragment, all the changes will be lost.

Listing 13.22 - **database/DreamDao** - When adding support for updateDream(), additional database functions must be added to handle the dream and the dream entries. Here again we use a naming convention as a reminder not to use the @Transaction function only, rather than any of the individual functions:

**@Update**

**suspend fun internalUpdateDream(dream: Dream)**

**@Insert**

**suspend fun internalInsertDreamEntry(dreamEntry: DreamEntry)**

**@Query("DELETE FROM dream\_entry WHERE dreamId = (:dreamId)")**

**suspend fun internalDeleteEntriesFromDream(dreamId: UUID)**

**@Transaction**

**suspend fun updateDreamAndEntries(dream: Dream) {**

**internalDeleteEntriesFromDream(dream.id)**

**dream.entries.forEach { internalInsertDreamEntry(it) }**

**internalUpdateDream(dream)**

**}**

Listing 13.23 and 13.25 - **DreamRepository** - The @Transaction function of the DAO must be called from the updateDream() function of the DreamRepository:

// Listing 13.23:

**suspend fun updateDream(dream: Dream) {**

**database.dreamDao().updateDreamAndEntries(dream)**

**}**

// Listing 13.25:

**~~suspend~~** fun updateDream(dream: Dream) {

**coroutineScope.launch {**

database.dreamDao().updateDreamAndEntries(dream)

**}**

}

Requirements

At the end of this chapter, the DreamCatcher app should meet the following requirements

* Clicking a dream in DLF should display that dream's details in the DDF view
* Any changes to the dream made in DDF should change the lastUpdated timestamp display
  + The change in timestamp should be seen immediately in DDF
  + However, the display is in seconds, so multiple updates in quick succession might not appear to change the display
* When the user navigates Back from DDF to DLF via button/gesture:
  + Any changes to the dream title or deferred/fulfilled status should appear in DLF, and the dream should move to the top of the list
  + If no changes have been make in DDF, then the dream should remain the same in DLF, and the dream should not move within the list

Chapter Notes

* Please don't implement the Chapter 13 challenge (No Untitled Crimes)

Phase 3: Chapter 14

Our app will diverge even more from CriminalIntent in this chapter. Instead of picking a date for a Crime, we'll be allowing the user to add a new reflection kind of DreamEntry to a Dream. However, this feature should only be allowed if the dream is not fulfilled.

Before getting started, we need to declare a layout for the dialog, which is a step that isn't necessary in CriminalIntent. The layout file should be named fragment\_reflection\_dialog.xml and it should only contain a single EditText within a LinearLayout.

The LinearLayout should only be as tall as its contents, but as wide as possible. It should also include a padding (not margin) of 16dp in all directions.

The EditText must have an ID value of reflection\_text. It should only be as tall as its contents, but as wide as possible. Please define a string resource such as "Enter Reflection Text" to be displayed as a hint.

Listing 14.1 - **ReflectionDialogFragment** - We will be developing a ReflectionDialogFragment rather than the DatePickerFragment from BNRG. This component will allow a user to add a new reflection to the current dream. The ReflectionDialogFragment will display an AlertDialog to request the title text for the new reflection:

class ReflectionDialogFragment : DialogFragment() {

**override fun onCreateDialog(savedInstanceState: Bundle?): Dialog {**

**val binding = FragmentReflectionDialogBinding.inflate(layoutInflater)**

**return AlertDialog.Builder(requireContext())**

**.setView(binding.root)**

**.setTitle(R.string.reflection\_dialog\_title) // "Add Reflection"**

**.setPositiveButton(R.string.reflection\_dialog\_positive, null) // "Add"**

**.setNegativeButton(R.string.reflection\_dialog\_negative, null) // "Cancel"**

**.show()**

**}**

}

Listing 14.2 - **nav\_graph.xml** - Specify the ID value of the new action as "add\_reflection" and specify the dialog (with its layout) accordingly. There are no arguments to the action.

Listing 14.3 and 14.4 - **DreamDetailFragment** - We will be adding a new FloatingActionButton (FAB) component to the both fragment\_dream\_detail.xml layouts. This view may be found in the Button section of the design palette.

The ID value of the FAB must be add\_reflection\_button. Placement should be as follows:

* Portrait/default: Place the FAB in the bottom-right corner of the screen
* Landscape: Place the FAB in the bottom-right corner of the DREAM section

Finally, in the updateView() function of DreamDetailFragment, we must call show() on the FAB if the dream is not fulfilled, and call hide() on the FAB if the dream is fulfilled.

Listing 14.5 and 14.6 (*Figure* 14.4 and 14.5) - please skip these steps for DreamCatcher, as nothing needs to be passed into the dialog.

Listing 14.9 - **ReflectionDialogFragment** - We need to use a different approach from BNRG. In DreamCatcher the user must click the positive button to indicate that the text has been entered and confirm that the intention is to add a reflection:

override fun onCreateDialog(savedInstanceState: Bundle?): Dialog {

val binding = FragmentReflectionDialogBinding.inflate(layoutInflater)

**val positiveListener = DialogInterface.OnClickListener { \_, \_ ->**

**val resultText = binding.reflectionText.text.toString()**

**setFragmentResult(**

**REQUEST\_KEY,**

**bundleOf(BUNDLE\_KEY to resultText)**

**)**

**}**

...

.setPositiveButton(R.string.reflection\_dialog\_positive, **~~null~~positiveListener**)

Listing 14.10 - **DreamDetailFragment** - We're only fetching a String from our dialog, rather than a Date as in BNRG, so please use bundle.getString() rather than bundle.getSerializable().

Requirements

At the end of this chapter, the DreamCatcher app should meet the following requirements:

* The user should be able to add a reflection via the FAB shown in the DreamDetailFragment
  + This is only available if the dream is not fulfilled
  + If the dream is fulfilled, the FAB should be hidden via hide()
* The new reflection must appear with the entered text in DDF immediately upon clicking Add in the dialog
* Upon returning to the DLF by navigating Back via button/gesture, the reflection count should be updated

Chapter Notes

* DDF can still only display a maximum of five total entries per dream
  + Any entries beyond five should still be stored, but won't be displayed in DDF
  + The reflection count in DLF should remain accurate, despite the DDF display limitations
* If a dream is deferred before adding a reflection, the reflection entry will appear below the deferred entry
  + If the deferred checkbox is then unchecked, the only the deferred entry should be removed
    - This means you can't always just remove the last entry; you must use a filter instead
    - Any reflections located after removed deferred entry should simply shift upward by one place
    - If the deferred checkbox is then checked again, the deferred entry should appear at the bottom of all entries

Testing

Please remove the P2A test files (DreamDetailFragmentTest and DreamListFragmentTest) from the project before you begin testing P2B. Those tests won't compile, now that we're using a database to fetch and store all the dream data.

Also please confirm that your version of Android Studio (according to Help | About) is Hedgehog. Any other versions are likely not to work properly. Please use Help | Check for Updates to upgrade if necessary.

For the test file to compile and run properly, the following resource ID values must be used. Most of these should already be in place from Project 2A or Project 2B as specified above, but please be sure to confirm all of these:

| **Resource ID** | **Type** | **Location** |
| --- | --- | --- |
| R.id.add\_reflection\_button | FloatingActionButton | fragment\_dream\_detail.xml (both!) |
| R.id.title\_text | EditText | fragment\_dream\_detail.xml (both!) |
| R.id.last\_updated\_text | TextView | fragment\_dream\_detail.xml (both!) |
| R.id.deferred\_checkbox | CheckBox | fragment\_dream\_detail.xml (both!) |
| R.id.fulfilled\_checkbox | CheckBox | fragment\_dream\_detail.xml (both!) |
| R.id.entry\_0\_button | Button | fragment\_dream\_detail.xml (both!) |
| R.id.entry\_1\_button | Button | fragment\_dream\_detail.xml (both!) |
| R.id.entry\_2\_button | Button | fragment\_dream\_detail.xml (both!) |
| R.id.entry\_3\_button | Button | fragment\_dream\_detail.xml (both!) |
| R.id.entry\_4\_button | Button | fragment\_dream\_detail.xml (both!) |
| R.id.dream\_recycler\_view | RecyclerView | fragment\_dream\_list.xml |
| R.id.list\_item\_image | ImageView | list\_item\_dream.xml |
| R.id.reflection\_text | EditText | fragment\_reflection\_dialog.xml |
| R.drawable.ic\_dream\_deferred | Drawable | DreamHolder (DreamListAdapter.kt) |
| R.drawable.ic\_dream\_fulfilled | Drawable | DreamHolder (DreamListAdapter.kt) |

Important: If your CPU doesn't meet the minimum requirements listed in the Syllabus, your tests will be unreliable because the emulator can't properly handle the background threads necessary to work with the coroutines for the database. You will find that one or more tests may sometimes pass and sometimes fail, even if without any implementation changes. You may find this happens sometimes even with a modern development machine, if you attempt testing while other resource-intensive applications are running during the testing process. In these cases, running the tests on a physical Android device rather than the emulator will generally help.