DreamCatcher App (Part 3 of 3)

In this third of three parts for this project, we will extend the app further by adding the following new features:

1. Add a New Dream menu item to the DLF app bar (Chapter 15)
2. Implement a swipe-to-delete feature to delete Dreams from DLF
3. Show message with Button to Add Dream when DLF is empty (Chapter 15 Challenge)
4. Share a Dream via an implicit intent (Chapter 16)
5. Take a photo via implicit intent for display in DDF (Chapter 17)
6. Display a zoomed photo in a dialog when clicked in DDF (Chapter 17 Challenge)
7. Implement a RecyclerView to hold the DreamEntry list within DDF
8. Implement a swipe-to-delete feature to delete reflections from DDF

Several of these features correspond closely to BNRG Chapters 15 through 17, as noted in the list above. However, there are several places where DreamCatcher requires significant changes from the approaches described in the textbook for CriminalIntent, with the most notable divergences detailed below by feature and chapter.

Getting Started

First, please be sure to retain a backup copy of your P2B 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 2C:

* **build.gradle** (Module :app)
  + A new dependency has been added to support the testing process; you may need to use File | Sync Project with Gradle Files as a result
* **P2CTest.kt**
  + This new instrumented test file will help ensure that your app meets all of the P2C assignment requirements
* **PictureUtils.kt**
  + This utility function is directly from BNRG Listing 17.14, provided here for convenience

Feature 1: Add a New Dream menu item to the DLF app bar (Chapter 15)

The goal of this feature is to allow the user to add a new Dream from a menu in the DLF.

Start Chapter 15, and follow Listings 15.1 and 15.2 (although our string should be "New Dream"). Please change the color specification of the ic\_add vector android:tint value from the hard-coded default of #000000 to @color/design\_default\_color\_on\_primary instead. This will ensure a high contrast with the app bar. When defining the menu, please call it "res/menu/fragment\_dream\_list.xml" and ensure that the Item id value is @+id/new\_dream.

Stop at "Creating the menu" and skip Listings 15.3 and 15.4. The approach in BNRG was replaced in late 2021, and is now deprecated and difficult to use. Instead of overriding onCreateOptionsMenu() we'll call addMenuProvider() on the activity from within onCreateView(). This function takes a MenuProvider and a LifecycleOwner:

* MenuProvider: We'll define this as an anonymous MenuProvider object (meaning object : MenuProvider). The MenuProvider itself requires the implementation of two functions:
  + onCreateMenu() creates the menu by inflating it with a provided inflater and menu.
  + onMenuItemSelected() will react to the menu clicks, and in this case there's only one item in the menu.
* LifecycleOwner: This parameter is optional, but by default it uses the Activity lifecycle. Since we're using a single-Activity app, the menu would be retained even when this fragment is replaced by the detail fragment. We want our menu to be removed when the fragment is replaced, so we explicitly specify viewLifecycleOwner here.

We'll get started with just the basics, and no implementation yet:

\_binding = ...

**requireActivity().addMenuProvider(object : MenuProvider {**

**// blank for now**

**}, viewLifecycleOwner)**

return binding.root

The "object" above will be highlighted with an error. Hover and click "Implement members" then select both abstract functions to provide placeholders with the required headers. Please remove the TODO notations.

In onMenuItemSelected() we can just return false for now:

requireActivity().addMenuProvider(object : MenuProvider {

~~// blank for now~~

**override fun onCreateMenu(menu: Menu, menuInflater: MenuInflater) {**

**}**

**override fun onMenuItemSelected(menuItem: MenuItem): Boolean {**

**return false**

**}**

}, viewLifecycleOwner)

In onCreateMenu() add something similar to Listing 15.3:

override fun onCreateMenu(menu: Menu, menuInflater: MenuInflater) {

**menuInflater.inflate(R.menu.fragment\_dream\_list, menu)**

}

Please run the app now, and follow *Figures* 15.6, 15.7, and 15.8 from the texbook.

Listing 15.5 - **database/DreamDao.kt** - Please change the name of the @Insert function to internalInsertDream(), and also add a @Transaction function to add all the entries along with the dream:

**@Insert**

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

**@Transaction**

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

**// You must implement this on your own**

**}**

Listing 15.6 - **DreamRepository.kt** - Same as BNRG, but remember to call the @Transaction function:

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

**database.dreamDao().insertDreamAndEntries(dream)**

**}**

Listing 15.7 - **DreamListViewModel.kt** - Same as BNRG, with different names:

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

**dreamRepository.addDream(dream)**

**}**

Now we can fill in the code to create the new dream and navigate to it.

Listing 15.8 - **DreamListFragment.kt** - The code is nearly identical, but it belongs in the onMenuItemSelected() function we created above

override fun onMenuItemSelected(menuItem: MenuItem): Boolean {

~~return false~~

**return when (menuItem.itemId) {**

**R.id.new\_dream -> {**

**showNewDream()**

**true**

**}**

**else -> false**

**}**

}

...

**private fun showNewDream() {**

**viewLifecycleOwner.lifecycleScope.launch {**

**val newDream = Dream()**

**vm.addDream(newDream)**

**findNavController().navigate(**

**DreamListFragmentDirections.showDreamDetail(newDream.id)**

**)**

**}**

**}**

Important: Please **skip** Listing 15.9. Please don't disable or delete the prepopulated dream-database asset. It's still useful for us throughout this project, especially for testing purposes.

Run the app and you should be able to add a new Dream from the app bar menu.

Feature 2: Implement a swipe-to-delete feature to delete Dreams from DLF

The goal of this feature is for the user to swipe any Dream to the left to delete the dream and its entries from the system.

First we must add a way to delete a dream (and its entries) from the database, which requires updating several files.

**database/DreamDao.kt**:

**@Delete**

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

**@Transaction**

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

**// You must implement this on your own**

**}**

**DreamRepository.kt**:

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

**// You must implement this on your own**

**}**

**DreamListViewModel.kt**:

**fun deleteDream(dream: Dream) {**

**// You must implement this on your own**

**// Note that this is NOT a suspend function**

**}**

**DreamListAdapter.kt** - In order for the swipe handler to know which dream to delete, the DreamHolder class must be adjusted to expose the bound dream object as a public read-only property, set by its bind() function. This is the Kotlin equivalent to exposing a public getter with a private setter in Java:

class DreamHolder(private val binding: ListItemDreamBinding) :

RecyclerView.ViewHolder(binding.root) {

**lateinit var boundDream: Dream**

**private set**

...

fun bind(...) {

**boundDream = dream**

...

}

**DreamListFragment.kt** - Consult the API for ItemTouchHelper along with the autocomplete of the IDE to create a private function that returns a new ItemTouchHelper object:

**private fun getItemTouchHelper(): ItemTouchHelper {**

**return ItemTouchHelper(object : ItemTouchHelper.SimpleCallback(0, ItemTouchHelper.LEFT) {**

**override fun onMove(...): Boolean = true**

**override fun onSwiped(...) {**

**// You must implement this on your own**

**}**

**}**

**}**

Hint: In the onSwiped() function above, the viewHolder must be cast to a DreamHolder type, using the as operator of Kotlin. To actually delete the dream and its entries, you must call the function you added to the view model. Finally, don't forget to attach the helper to the recycler view in onViewCreated().

Feature 3: Show message with Button to Add Dream when DLF is empty (Chapter 15 Challenge)

The goal of this feature is to display a brief message to the user if there are no dreams displayed in the DLF. There should also be a Button displayed to allow the user to add a new Dream (in addition to the menu item in the app bar implemented above).

**layout/fragment\_dream\_list.xml** - Currently the RecyclerView may be the top-level view of the entire layout, or it may reside within a LinearLayout. Either way, we now need to place a ConstraintLayout at the top level, and place the RecyclerView within the ConstraintLayout. This is necessary, because we need to add two new views that will overlap with the RecyclerView. ConstraintLayout allows overlaps, but LinearLayout doesn't. Because the RecyclerView height and width are both match\_parent it doesn't require any constraints. However, the new view components will require you to specify constraints. You can review Chapter 11 for details about working with constraints in the ConstraintLayout.

Please use the following naming conventions for the two new view components:

TextView:

android:id="@+id/no\_dream\_text"

android:text="@string/no\_dreams"

Button:

android:id="@+id/no\_dream\_add\_button"

android:text="@string/new\_dream"

app:icon="@drawable/ic\_add"

**DreamListFragment.kt** - Set an onClickListener for the new button to call the showNewDream() function implemented earlier.

Hints: Before constructing the adapter, but within the collect{} block for dreams, set the visibility of the new views to either View.VISIBLE or View.GONE, based on whether the collected dreams list is empty or not. Clicking the new button should behave exactly the same as the New Dream app bar menu from Feature 1.

To test this feature, swipe away all the dreams and ensure the new components appear (and that the button works as expected). To add all the dreams back, uninstall the app and then run it again. This will prepopulate the dreams from the dream-database asset file.

Feature 4: Share a Dream via an implicit intent (Chapter 16)

The goal of this feature is for the user to use an app bar menu from the detail screen to share dream details with others via SMS, email, or copy-and-paste.

Chapter 16: Read – but please *don't* implement – BNRG up through the "Using a Format String" section.

Follow the concepts of Listings 16.6 and 16.7, but shared dream will have the following format:

[Dream Title]

[Date String]

Reflections:

\* [Reflection 1]

\* [Reflection 2]

\* [Reflection 3]

This dream has been [Deferred|Fulfilled].

Example:

Ride in a hot air balloon

Last updated 2024-02-04 at 02:04:24 PM

Reflections:

\* One

\* Two

This dream has been Deferred.

Formatting Notes:

* The Date String should be the same format as in the last\_updated\_text of DDF
* Only show as many Reflection lines as there are Reflection entries
* If there are no reflections, the "Reflections:" header must be omitted
* The last line must be omitted if the dream is neither fulfilled nor deferred
* The joinToString() function is very useful for handling the reflections

**DreamDetailFragment.kt** - BNRG uses a button placed in the layout for to trigger the implicit intent. Instead, we'll add a new menu to the DDF app bar using exactly the same approach that we used to add a menu to the app bar of the DLF in Feature 1 above.

Use the Resource Manager to add a new Vector Asset from clip art called ic\_share (later setting the android:tint to @color/design\_default\_color\_on\_primary), and create a new Menu resource called menu/fragment\_dream\_detail.xml. Please use the following naming conventions in the attributes:

Item:

android:id="@+id/share\_dream\_menu"

android:icon="@drawable/ic\_share"

android:title="@string/share\_dream"

app:showAsAction="ifRoom|withText"

Make sure to fully implement code similar to the menu provider code of Feature 1 at this point.

Listing 16.8 and 16.9 - **DreamDetailFragment.kt** - Follow the BNRG concepts, but place the listener code from Listing 16.8 into a helper function private fun shareDream(dream: Dream) instead. We'll call this function from the onMenuItemSelected function, after extracting the current dream value from the StateFlow held in the view model:

R.id.share\_dream\_menu -> {

**vm.dream.value?.let { shareDream(it) }**

**true**

}

else -> false

After this, please read — but *don't implement* anything — starting from the "Asking Android for a contact" section through the end of this chapter.

Feature 5: Take a photo via implicit intent for display in DDF (Chapter 17)

The goal of this feature is to provide an app bar menu item to the DDF for the user to add a photo (or replace an existing photo) for display in the detail screen.

Note that there are a lot of steps here. However, we're going to take a few shortcuts so that the overall procedure is actually a bit more straightforward than what's in BNRG.

*Figure* 17.2 - Create new Vector Asset with a camera icon, called ic\_camera. Please note that we are only using an ImageView, not an ImageButton.

Listing 17.1 and 17.2 - **layout/fragment\_dream\_detail.xml** (both default and landscape variants) - Include an ImageView component of size 80dp x 80dp. Please include the following naming conventions and attributes:

android:id="@+id/dream\_photo"

android:background="@color/material\_dynamic\_neutral30"

android:contentDescription="@string/dream\_photo"

android:cropToPadding="true"

android:scaleType="centerInside"

Layout Notes: The suggested placement for the ImageView is in the upper-left of the Dream section, but the overall layout is completely up to you. Just make sure that everything looks relatively neat, in both default and landscape orientations. You can generally follow the textbook, but you'll likely need to move the last\_updated\_text so that it doesn't line-wrap. Again, please don't add an ImageButton as in BNRG.

**menu/fragment\_dream\_detail.xml** - As one minor simplification, we're going to use a menu item to activate the camera, rather than a button in the layout. To do this, we must add a new Item to the existing menu, with the following attributes:

android:id="@+id/take\_photo\_menu"

android:icon="@drawable/ic\_camera"

android:title="@string/take\_photo\_menu"

app:showAsAction "ifRoom|withText"

Note: The R.string.take\_photo\_menu resource must have the value "Take Photo"

Listings 17.3, 17.4, and 17.5 - **manifests/AndroidManifest.xml** - Follow BNRG, but use edu.vt.cs5254.dreamcatcher.fileprovider as the authorities, and use dream\_photos for the files-path name.

Listing 17.6 - **Dream.kt** - Instead of adding a new String property, please just add a computed property to the Dream class:

...

val isDeferred get() = ...

**val photoFileName get() = "IMG\_$id.JPG"**

}

Listings 17.7 and 17.8 - **[SKIP THESE]** - BNRG is trying to emphasize database migrations, but it's really not necessary here because we're computing the name directly from the UUID of the dream.

Listing 17.9 - **DreamDetailFragment.kt** - Follow BNRG here.

Listing 17.10 - **DreamDetailFragment.kt** - BNRG uses an onClickListener, but we're using a menu. We're also getting the file name from the dream itself, rather than using the date. Our implementation is mostly the same, but please review the differences below.

Insert the following into the "when" statement within onMenuItemSelected() of the MenuProvider:

when(...) {

R.id.share\_dream\_menu -> {

vm.dream.value?.let { shareDream(it) }

true

}

**R.id.take\_photo\_menu -> {**

**vm.dream.value?.let {**

**val photoFile = File(**

**requireContext().applicationContext.filesDir,**

**it.photoFileName**

**)**

**val photoUri = FileProvider.getUriForFile(**

**requireContext(),**

**"edu.vt.cs5254.dreamcatcher.fileprovider",**

**photoFile**

**)**

**takePhoto.launch(photoUri)**

**}**

**true**

**}**

else -> false

}

Listing 17.11 - **[SKIP THIS]** - We don't need to update the database, but we'll deal with updating the photo near the end of this feature.

Listing 16.16 - **DreamDetailFragment.kt** - Please go *back* to Listing 16.16 (that's not a typo; this step is from Chapter 16) and add the canResolveIntent() helper function to the DDF.

Listing 17.12 - **DreamDetailFragment.kt** - BNRG simply disables a view if the camera isn't available, whereas we need to hide a menu item instead. The code is mostly the same, but please review the differences. This code is added to the onCreateMenu() function of the MenuProvider:

override fun onCreateMenu(menu: Menu, menuInflater: MenuInflater) {

menuInflater.inflate(R.menu.fragment\_dream\_detail, menu)

**val captureImageIntent = takePhoto.contract.createIntent(**

**requireContext(),**

**Uri.EMPTY // NOTE: The "null" used in BNRG is obsolete now**

**)**

**menu.findItem(R.id.take\_photo\_menu).isVisible = canResolveIntent(captureImageIntent)**

}

Listing 17.13 - **AndroidManifest.xml** - We're a bit out of sync with the manifest in BNRG, so we must add the queries section that the textbook has already added, along with the new intent query for the camera:

...

</application>

**<queries>**

**<intent>**

**<action android:name="android.media.action.IMAGE\_CAPTURE" />**

**</intent>**

**</queries>**

</manifest>

Listing 17.14 - **PictureUtils.kt** - This file, which is identical to BNRG, has been provided to you via Git update.

Listing 17.15 and 17.16 - **DreamDetailFragment.kt** - This is similar to BNRG. There are a few syntax simplifications, but the main difference is how we handle the filename:

private fun updateView(dream: Dream) {

...

**updatePhoto(dream)**

}

**private fun updatePhoto(dream: Dream) {**

**with(binding.dreamPhoto) {**

**if (tag != dream.photoFileName) {**

**val photoFile =**

**File(requireContext().applicationContext.filesDir, dream.photoFileName)**

**if (photoFile.exists()) {**

**doOnLayout { measuredView ->**

**val scaledBM = getScaledBitmap(**

**photoFile.path,**

**measuredView.width,**

**measuredView.height**

**)**

**setImageBitmap(scaledBM)**

**tag = dream.photoFileName**

**}**

**} else {**

**setImageBitmap(null)**

**tag = null**

**}**

**}**

**}**

**}**

**DreamDetailFragment.kt** - We skipped Listing 17.11 earlier, because we don't need to update the Dream object or the database upon taking a photo. However, we do need to clear the tag that might have been set by taking an earlier photo for the same Dream, and we need to call our updatePhoto() function to display the just-taken photo. Here's our replacement for that listing:

if (didTakePhoto) {

**binding.dreamPhoto.tag = null**

**vm.dream.value?.let { updatePhoto(it) }**

}

Listing 17.17 - **manifests/AndroidManifest.xml** - Be sure follow BNRG to declare that our app uses the camera, if it's available, but still works even without it.

Feature 6: Display a zoomed photo in a dialog when clicked in DDF (Chapter 17 Challenge)

The goal of this feature is to allow the user to click the dream photo in DDF to see a dialog with a larger version of the image.

Create **layout/fragment\_photo\_dialog.xml** - This is a simple layout with a single LinearLayout that spans its entire dialog parent. An ImageView within the layout must use the following attributes:

android:id="@+id/photo\_detail"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:scaleType="fitXY"

android:padding="8dp"

Create **PhotoDialogFragment.kt** - This class has the same structure as the ReflectionDialogFragment class. The AlertDialog here doesn't need a title nor any buttons, so the builder is quite basic at the end of onCreateDialog():

return AlertDialog.Builder(requireContext())

.setView(binding.root)

.show()

Update **navigation/nav\_graph.kt** - Add PDF to the navigation graph, and drag another arrow from the right side of DDF to the PDF. Change the default action name to show\_photo\_detail. Also create a new argument to the PDF called dream\_photo\_filename of type String.

Update **DreamDetailFragment.kt** - Set an OnClickListener for the ImageView that navigates to the PDF, passing the photo filename to the PDF. Note that the navigation shouldn't occur if the current dream is null.

Further, we don't want the listener to display a dialog without any image, so we must enable/disable the dream\_photo component if there is/isn't a photo associated with the current dream. As a hint, this should be done within the updatePhoto() function.

Update **PhotoDialogFragment.kt** - Above onCreateDialog(), you must first fetch the photo filename from the argument passed to it, using by navArgs(). Then, within onCreateDialog(), use the code from Listing 17.15 as a guide for how to load the file from its name, scale it properly, and set the ImageView to display the scaled bitmap. The code here won't be as complex as Listing 17.15, because here no tags are involved, but everything you need is there. Note that to maximize the size and quality of the zoomed image, you must use binding.root (rather than the ImageView itself) as the receiver of the doOnLayout() function.

Note: The aspect ratio of the largest possible dialog follows the orientation of the device, so a portrait image is best viewed when the device is also in portrait mode.

Feature 7: Implement a RecyclerView to hold the DreamEntry list within DDF

The goal of this feature is to use a recycler view to display an unlimited number of scrolling dream entries per dream, rather than a fixed maximum of 5 entries.

**layout/list\_item\_dream\_entry.xml** - This is a basic layout containing just a single disabled Button with no padding or margins. The layout and the button should each take up all available width, but the height should only be the height of the button. The button should have an ID of dream\_entry\_button.

**layout/fragent\_dream\_detail.xml** (default and landscape) - Replace the five Dream Entry buttons with a single RecyclerView. This view must use all available vertical space below the "Entries" header and above the FAB. At least 8 entries must fully fit on the screen at the same time in portrait mode. Please use the following name in both layout files:

android:id="@+id/dream\_entry\_recycler"

**DreamEntryAdapter.kt** (also includes the DreamEntryHolder class) - Use the DreamListAdapter as a guide.

**DreamDetailFragment.kt** - In updateView() Remove all unnecessary code, for example to zip the buttons to the entries. In that same function, set the layout manager and construct the adapter for the new recycler view.

Hint: Review BNRG Chapter 10 and the Module 5 Part 2 slides to recall how to implement a recycler view.

Feature 8: Implement a swipe-to-delete feature to delete reflections from DDF

The goal is to allow the user to remove any Reflection entry from a Dream by swiping left on the DreamEntry recycler view. No other DreamEntry kind should respond to any swipe/move attempt.

Hints:

* Use Feature 2 above as a guide. Feature 8 is much less complicated because no new database-related code is needed. All you need is vm.updateDream { ... }.
* Update DreamEntryHolder to expose a boundEntry property. This is the same as how you exposed boundDream above in Feature 2.
* Pass in (0, 0) to the constructor of the ItemTouchHelper.SimpleCallback object, then override the getSwipeDirs() of this object. Here we must return ItemTouchHelper.LEFT if the entry is a Reflection kind, or return 0 otherwise.