Project report: Tracking My Pantry

Paolo Marzolo

June 2021

1 Introduction

As part of the 2021 edition of the Mobile Apps Laboratory at the University of Bologna, we were tasked to develop an app that allowed the user to track groceries, using the barcode as the identifier.

The association barcode $\leftrightarrow$ product details was meant to be shared among the community, as mentioned in the project requirement document:

> The mobile application is designed for those who want to keep track of the groceries they buy through their barcode and build up a collaborative database of barcodes that can be used by the community.

This document seeks to identify the reasons behind our implementative choices, and outline the creative process as a whole in order to give a streamlined but complete view of what was created.

2 Overview

2.1 Project Requirements

Before diving into the reasoning behind the choices that were made, we must make the requirements clear. This is only a short summary of what was included in the original document and in the relevant lectures. There were two main necessary features:

- **Update the barcode knowledge base.** Glossing over implementative details, the user was supposed to first request known objects corresponding to a given barcode, then, if the product is present, the details will be saved to the user's personal library and the server will be notified of the user's choice; otherwise, a new item will be created, and the remote database must be updated.
- **Login and register to access the remote database.** In order to gain access to the remote database, the user must first register and login. All following requests will then be authorized by a specially included header, which will included the session token.

- **Keep track of the Groceries bought.** The application must also include an offline database that holds the groceries selected by the user. It must be possible to delete this once they've been used.

2.2 Additional Features

Having made clear what the mandatory features were, what follows are additional features that were picked based on a specific rationale, made clear in the following section.

2.2.1 Amounts

The first feature I chose to add are amounts. The reason is probably self-explanatory: in order to track groceries it must be possible to track how many instances of one product are in your pantry: this is done via a simple integer, editable with two buttons on all items. Once an item gets to amount "zero", it is *not* deleted. This is meant as a utility feature: adding an item is more cumbersome than increasing its amount, and items with amount "zero" are used in the Dashboard, explained later.

2.2.2 Offline First

The app works correctly offline as well as online. While the phone is offline, the user can't search for new products based on a barcode, but can manually add items, browse the items they are currently holding, and change their amounts.

2.2.3 Product Types

In order to organize groceries, a type was included. The possible types are hardcoded and cannot be changed by the users, but they may be revised in the future. The types are organized **functionally**, in order to give a simple and immediate idea of what they represent, and to have a marked impact on the dashboard (explained in the following section). The possible types are Carbs, Veggies, Proteins, Cheese, Sweets, Fruit, Alcohol, Drinks.

2.2.4 Dashboard

The app includes a very simple dashboard, with two main sections: Missing Items and Product Percentages.

- **Missing Items** is a list of all the items with amount "zero" in the grocery list. This is meant to be used as a grocery list, and to that end the button marked "Copy to clipboard" copies to the user's clipboard a simplified view (in text) of the missing items. The intended usage would be to paste it into a family group chat, or send it to a housemate; because of its simple nature, it could be no doubt adapted to work with different grocery list or messaging apps.

- **Product Percentage** is a list of the item types with what percentage of the product they represent compared to the whole pantry. This is meant to be used as a "wake-up call", in the most un-intrusive and neutral way possible. There are two important design choices that went into this list: only types represented by any item in the pantry (even with amount zero) are included, to avoid unwanted categories to show up (a simple example is alcohol); types that represent "0%" of the total items are still included, as it was still considered relevant information. As an example, consider the following: the user A does not drink alcohol, and its groceries are generally health-oriented. A does not have any alcohol in his pantry, but just finished all vegetables. A does not need to know alcohol represents 0% of his groceries, but feedback about his vegetables is important.

2.2.5 Login Management

As a small addition to the login system, the authentication token is saved in private storage along with when it was created, so that, once the application is started, if and only if the token is expired then the user is prompted to login again. The login section is accessible through a login button in the top right. Logging out is possible by holding the Login button, while only tapping it will either open the login activity or give feedback about the state of the login.

2.2.6 Barcode Scanning

I also implemented a very simple barcode scanning feature, using an external library.

2.3 Choice Rationale

As it may have emerged trough the feature choices, all additional features are oriented toward a similar rationale: the objective is to provide a simple but complete user experience, while avoiding excessive developer burden. Options have simple and easily adaptable defaults, and activities are reused when deemed necessary.

At the same time, my objective during development was to learn as much as possible: this drove the main implementative choice, which is developing in Kotlin instead of Java. This allowed me to explore Kotlin coroutines, Kotlin extensions, Kotlin's handling of data classes, serialization and Scope functions, as well as learn a new language. This objective is reflected in how different situations were handled in the source code: as one can see, I used "traditional" View finding *findViewById()* in some cases, and preferred view binding or data binding in others. Based on this, I believe it is important to specify how "code consistency" was **not** one of the objectives, so similar problems have been tackled differently.

3 Implementations and Technical Challenges

In this next section, every activity will be shown in screenshots. After the image, some relevant portions of the code will be discussed. Since a lot of the code is similar throughout activities, it will be explained in detail the first time and glossed over in following references. This means that explanations will get thinner and quicker as we go along.

Activities are analyzed in creation order, but obviously some were reused or rivisited after the creation. For each activity, conceptual and technical choices will be highlighted in separate sections.

3.1 Main Activity

There are a few concepts worth exploring that will be relevant later on: the action bar, the token expiration, and some basic navigation.

3.1.1 Action Bar

The top widget is a toolbar. There are a couple sections relevant to it: after adding it to the layout file:

```
    <androidx.appcompat.widget.Toolbar
        android:id="@+id/toolbar"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:minHeight="?attr/actionBarSize"/>
```

The toolbar must then be set as the ActionBar in the onCreate method, and a corresponding method must be overridden to inflate icons and add them to the toolbar

```
override fun onCreate(savedInstanceState: Bundle?) {
    ...
    val toolbar: Toolbar = findViewById(R.id.toolbar)
        // Sets the Toolbar to act as the ActionBar for this Activity window.
        setSupportActionBar(toolbar)
    ...
}
override fun onCreateOptionsMenu(menu: Menu?): Boolean {
    ...
}
```

In order to facilitate user interaction, I used the profile icon to access the login activity: on a single tap, if the user isn't logged in, the LoginActivity is started, while if the user is already logged in a Toast is shown. To log out, I used the longClick event.

3.1.2 Authentication Token

In order to store the authentication token, I used SharedPreferences. As mentioned in the documentation, SharedPreferences allows to "Store private, primitive data in key-value pairs.". Using it is very simple: access the preferences object, then get the value you need:

```
val sharedPref = getSharedPreferences(
            getString(R.string.auth_data_file),
            Context.MODE_PRIVATE
        )
val codedInfo = sharedPref.getString(..., null)
```

Thanks to this, logging in and logging out corresponds to saving end deleting an entry from SharedPreferences.

3.1.3 Navigation

In MainActivity, I also used some simple navigation to navigate between the list fragment and the dashboard fragment. Such simple navigation is very easy to set up:

```
override fun onCreate(savedInstanceState: Bundle?) {
    ...
    // Passing each menu ID as a set of Ids because each
    // menu should be considered as top level destinations.
    val appBarConfiguration = AppBarConfiguration(
        setOf(
            R.id.navigation_home, R.id.navigation_dashboard
        )
    )
    setupActionBarWithNavController(navController, appBarConfiguration)
    navView.setupWithNavController(navController)
    ,,,
}
```

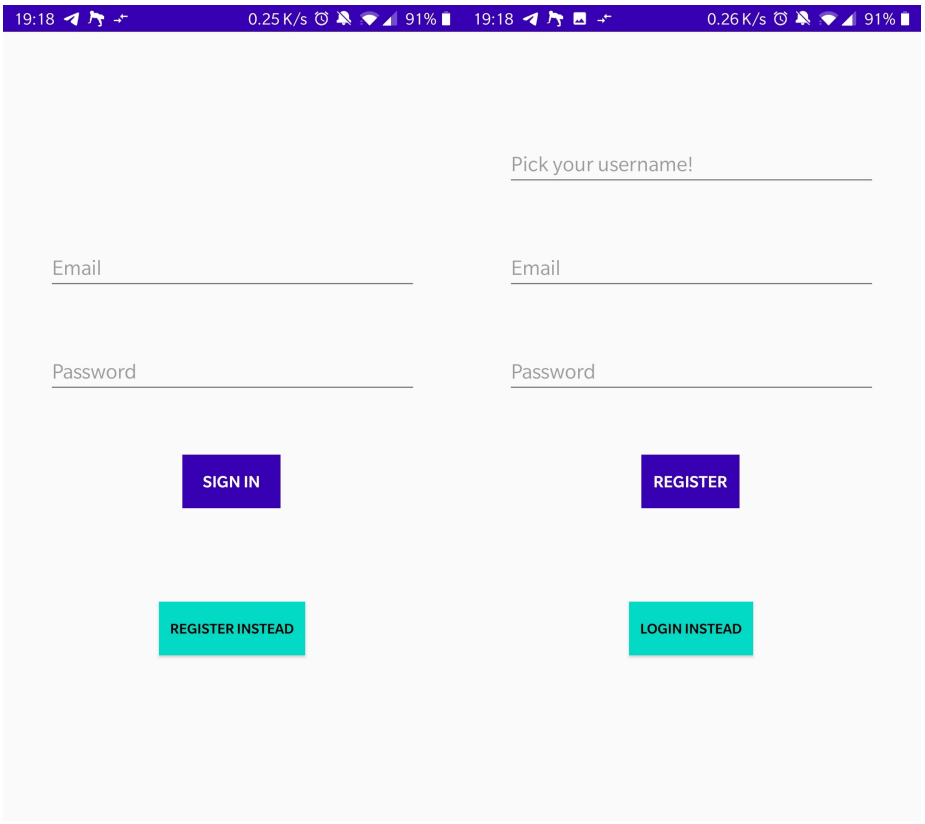
3.2 Login and Registration

Login and registration are handled by the same activity. A keen observer may notice that the back button is missing from this activity: the reason is to nudge users towards registering and logging in; that is not to say they are forced to, as the physical "back" button or the software "back" button will still take the user back to the main view.

This activity uses data-binding, partially two-way. I settled on this after trying out the new two-way data-binding APIs and finding them somewhat obscure for the reader. Concretely, this means that the xml file has a <data> section in the preface, and then variables mentioned can be used throughout the xml. Safety should be guaranteed at compile time.

```
<data>
    <variable
        name="LoginVM"
        type="com.example.pantry.ui.login.LoginViewModel" />
</data>
...
<EditText
    android:id="@+id/username"
    android:afterTextChanged="@{LoginVM::setUsernameText}"
    android:text="@{LoginVM.usernameText}"
.../>
```

To set the new values of the EditText, the attribute afterTextChanged was used. This allowed me to decouple a private, mutable state from the public, immutable



one, and mediate access through a setter. An alternative way would be to include an equal sign in the xml, between '@' and '{' to propagate changes back to the state, but I decided against it.

```
private var _usernameText = MutableLiveData<String?>()
var usernameText: LiveData<String?> = _usernameText
...
fun setUsernameText(s: Editable) {
    _usernameText.value = s.toString()
    loginDataChanged()
}
```

Within the ViewModel, the form state is kept up-to-date through a LiveData variable and the loginDataChanged function:

```
fun loginDataChanged() {
    if (!isUserNameValid(usernameText.value.toString())) {
        _loginForm.value = LoginFormState(usernameError = ...)
    } else if (!isPasswordValid(passwordText.value.toString())) {
        _loginForm.value = LoginFormState(passwordError = ...)
    } else if (!isEmailValid(emailText.value.toString())) {
        _loginForm.value = LoginFormState(emailError = ...)
    } else {
        _loginForm.value = LoginFormState(isDataValid = true)
    }
}
```

Although the viewModel is independent from the repository and the data source, login and register functions (suspend functions, until the repository) are run on the MainScope(). If the user registers, successfully, he's prompted to login (I decided this flow was clearer than logging the user in directly); if the user logs in successfully, he's moved back to the main view, and credentials are saved using a simple serializable class.

```
@Serializable
data class LoginInfo(
    val email: String,
    val token: String,
    val dateLogged: String,
)
```

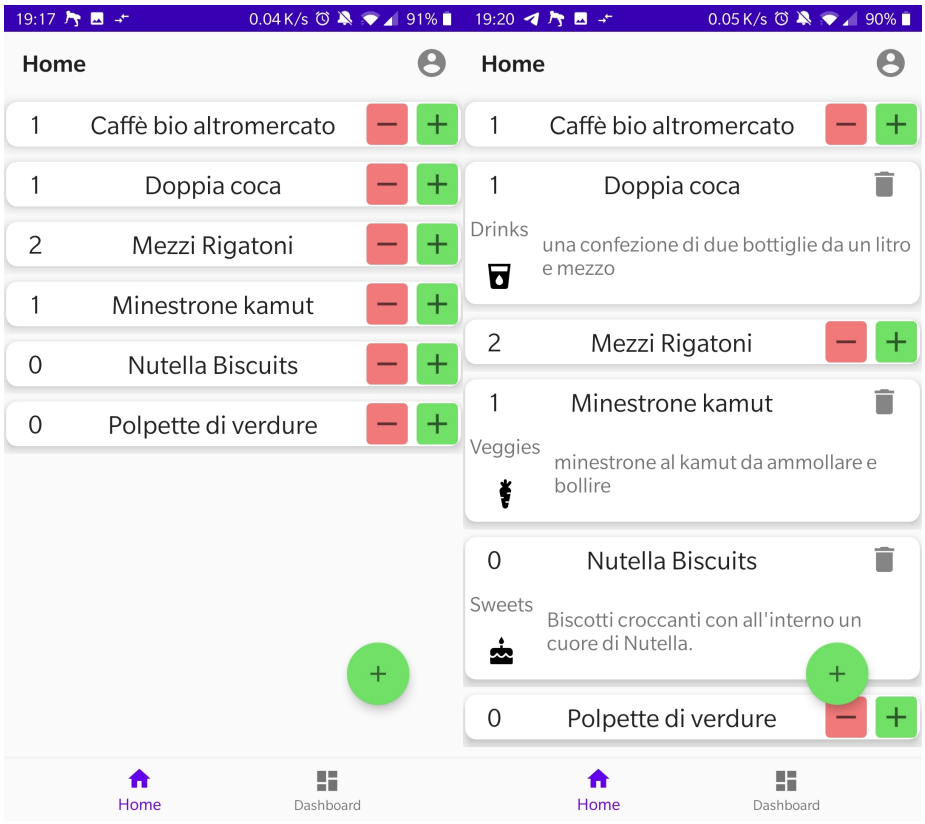
3.2.1 Network Requests

Since this is the first request that was implemented, this is where I will give a brief description of how network requests are handled. HTTP requests are handled by Volley. With a small singleton, the queue is kept unique:

```
class NetworkOp constructor(context: Context) {
    companion object {
        @Volatile
        private var INSTANCE: NetworkOp? = null
        fun getInstance(context: Context) =
            INSTANCE ?: synchronized(this) {
                INSTANCE ?: NetworkOp(context).also {
                    INSTANCE = it
                }
            }
    }
    val requestQueue: RequestQueue by lazy {
        // applicationContext is key, it keeps you from leaking the
        // Activity or BroadcastReceiver if someone passes one in.
        Volley.newRequestQueue(context.applicationContext)
    }
    fun <T> addToRequestQueue(req: Request<T>) {
        requestQueue.add(req)
    }
}
```

Finally, generating a Request is very simple. In this code snippet, the request generation and adding it to queue is implemented as a suspendCoroutine, to then run it asynchronously:

```
suspend fun login(email: String, password: String) =
        suspendCoroutine<Result<LoggedInUser>> { cont ->
            val jsonObjectRequest = JsonObjectRequest(
                Request.Method.POST, loginURL, jsonObj(null, email, password),
                { response ->
                    Log.d("NETWORKING", response.toString())
                    val res = Json.decodeFromString<LoginResponse>(
                        response.toString())
                    cont.resume(
                        Result.Success(
                            LoggedInUser(
                                token = res.accessToken,
                                displayName = email
```




```
14                          ),
15                      )
16                  )
17              },
18              { error ->
19                  Log.d("NETWORKING", error.toString())
20                  cont.resume(Result.Error(IOException(
21                                          "Error logging in")))
22              }
23          )
24
25          NetworkOp.getInstance(appContext).
26              addToRequestQueue(jsonObjectRequest)
27      }
```

3.3 ItemListFragment

This is one of the most important pieces of the app: it displays a RecyclerView with logic-driven items, and is the home of the application.

3.3.1 ItemViewModel

This ViewModel is shared between this activity and two fragments, and it contains the list of items saved in the database. The ViewModel itself is very simple:

```
class ItemViewModel(private val repository: ItemRepository) : ViewModel() {
    val allItem: LiveData<List<Item>> = repository.allItems.asLiveData()

    /**
     * Launching a new coroutine to insert the data in a non-blocking way
     */
    fun insert(item: Item) = viewModelScope.launch {
        repository.insert(item)
    }
    fun update(item: Item) = viewModelScope.launch {
        repository.update(item)
    }
    fun delete(item:Item) = viewModelScope.launch{
        repository.delete(item)
    }

}

// needed for accessing in fragments
// (if doesn't exist, create. how? with this factory)
class ItemViewModelFactory(private val repository: ItemRepository) :
    ViewModelProvider.Factory {
    override fun <T : ViewModel> create(modelClass: Class<T>): T {
        if (modelClass.isAssignableFrom(ItemViewModel::class.java)) {
            @Suppress("UNCHECKED_CAST")
            return ItemViewModel(repository) as T
        }
        throw IllegalArgumentException("Unknown ViewModel class")
    }
}
```

As one can see from the code, I strongly preferred lifecycle-aware components. This was instrumental for both learning relevant and updated information and reducing development time. The LiveData will allow us to observe changes in the list and react accordingly, while the "cast" is necessary because the repository wraps the list in a Flow, a Kotlin data type. From the documentation:

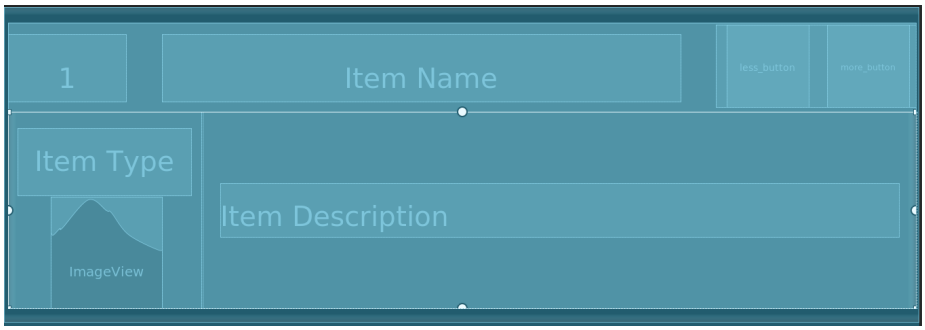
"A cold asynchronous data stream that sequentially emits values and completes normally or with an exception."

The second relevant information about this viewModel is its usage of Kotlin coroutines to manage data in a non-blocking way. The relevant kotlin documentation can be found here.

3.3.2 Item Definition

I won't analyze the entire code about the database, but it should suffice to know that it was created using Room, and the basic entity looks like this:

```
@Entity(tableName = "item_table")
@Serializable
data class Item(
    // the actual numerical barcode this corresponds to
    @ColumnInfo(name = "code") val code: String,
    // name of the product
    val name: String = "",
    @PrimaryKey(autoGenerate = true) val id: Int = 0,
```




```
9     // description of the product
10    val description: String = "",
11    // amount we have in pantry
12    val amount: Int = 1,
13    // type of the product
14    val type: String = "")
```

The most interesting parts are:

- **The key**. I decided to use a personal id, so that multiple objects with the same barcode are allowed. This was entirely based on ignorance, so I covered all my bases. The id is autogenerated.
- **Additional fields.** I used only two additional fields: the amount and the type. Icons for items are generated based on type.
- **Serializable.** Because all classes in kotlin are serializable, and I only used primitive types, I did not need to write a serializer. Throughout the code, serialized Items are often used to only pass Strings between activities (through intents), and serialization is used before HTTP calls.

Together with the Dashboard fragment, it shares a ViewModel (ItemViewModel) with its host activity. This allows all three fragments to share data. The ItemListFragment contains a RecyclerView: throughout the app, there are 4 RecyclerView's, but this one has by far the most complex adapter.

3.3.3 Item

The item layout file is used to contain data amount a single item and allow users simple actions on them. Internally, it uses a CardView for elevation, and is divided into two sections, one collapsable, the other one fixed. Here it is shown expanded. To expand and collapse it, both itemName and hiddenView View's are given onClickListener 's.

3.3.4 ItemListAdapter

The adapters have small differences between them, but because they all show similar items and inherit from ListAdapter this is the only one we will explain.

In order to propagate changes to the database, I decided to use functions. Detailed information about RecyclerView can be found here, so in this document I will highlight the three most important sections to my development:

1. **Binding function.** Because this function is responsible for (possibly) inflating the View and binding information to it, it grew linearly with the amount of data and listeners contained therein; nonetheless, at its core it is very simple: after selecting the relevant Views, it injects information from the Item into them.

2. **Switching visibility.** This function is responsible for opening and closing the collapsable item, as well as setting the correct transitions (the default ones were far too slow) and switching the visibility of icons: when the item is opened, its amount can't be changed, and the item can instead be deleted.

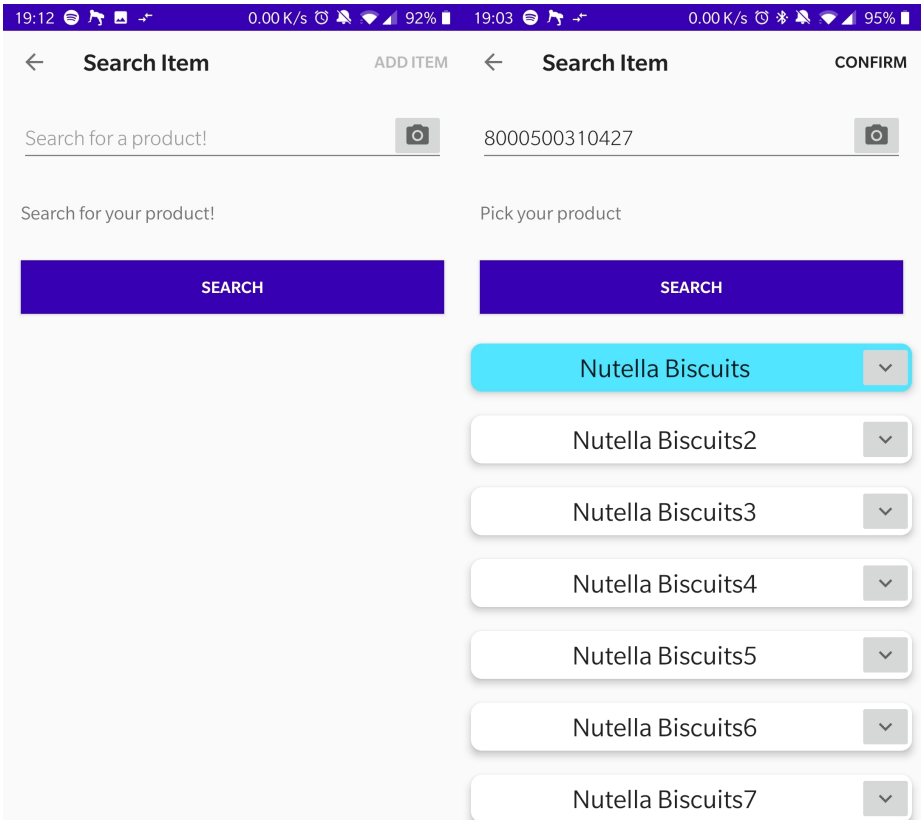
3. **Comparator.** Lastly, I believe it is important to mention the custom comparator, as it allows me to only re-render items that actually changed when a new list is submitted to the adapter.

```
class ItemsComparator : DiffUtil.ItemCallback<Item>() {
    override fun areItemsTheSame(oldItem: Item, newItem: Item):
        Boolean {
        return oldItem.id == newItem.id &&
            oldItem.amount == newItem.amount &&
            oldItem.type == newItem.type &&
            oldItem.description == newItem.description
        }
    ...
}
```

   This is one of the few choices I am not sure about: when amounts are changed, the comparator's areItemsTheSame returns false, so the view is removed and re-generated. This results in an unpleasant viewing experience, as the new View fades into existence. A possible fix would be to remove the amount check in the comparator: I left it as is to maintain as strong a bond as possible with the database, but in a real app I would consider decoupling the two for the sake of a more pleasant experience or overriding the default fade-in of the new view.

3.4 Adding items

From here on, most instrumental pieces have already been explained. Fetching items is only allowed if a user is logged in, but adding items is **possible** even offline. To avoid users skipping code (and failing to increase the size and reach of the shared codebase), creating an item is only possible after inputting a code, whether items with that code were searched or not. If, instead, the user searches for items corresponding to a specific code, then he can select an item from the list or create a new one: only after this latter action will a POST be sent, and the corresponding barcode added to the remote database.



The list is a streamlined version of the earlier RecyclerView, so I won't go into details about how it was implemented. Because the onClick is used to select an item, I used a separate collapse button to expand the description.

3.5 Creating new items

The NewItemActivity is used to both create new items and edit existing ones. If a user wants to edit an item, he or she must tap and hold on the item name. To create a new item, it is necessary to input a code first, and then fill in other necessary details. The reasoning behind this is mentioned above.

This activity does not have any special logic, but is handled carefully: because the user may reach it by editing an existing item, the corresponding item must be fetched from the Intent and default values of all editable areas must be updated accordingly:

```
override fun onCreate(savedInstanceState: Bundle?) {
    ...
    val prevValue = intent.getStringExtra(EDITED_ITEM_KEY)
    if (prevValue != null){
        val item = Json.decodeFromString<Item>(prevValue)
        ... // update all views and viewModel
```

```
    }
}
```

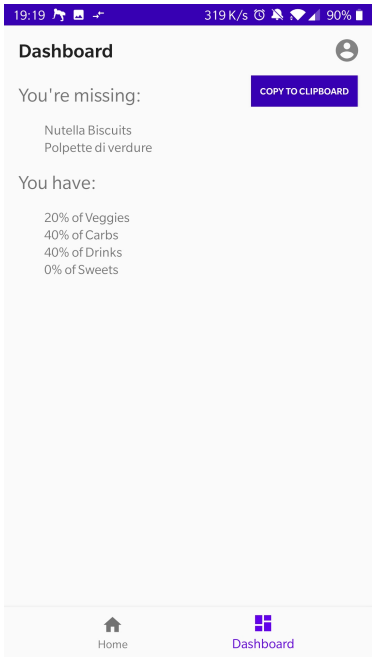
Although this activity implements formState handling and errors, requirements are very lax: names can contain letters and numbers, descriptions don't have a limit on length, and amounts have to be positive numbers.

3.6 Dashboard

As a special section, I included a simple dashboard. The functions are as described above, and their implementation follows from what we have mentioned so far: there are two recyclerView's, with *very* streamlined versions of items, that summarize what items are missing and what percentage of every type is present in your groceries.

Lastly, I included a button to copy the missing items to the clipboard, which is probably the most useful feature in the entire app. Here's how that was implemented:

```
val clip = root.findViewById<Button>(R.id.export_button)
clip.setOnClickListener {
    val clipboard = this.context?.let { it1 ->
        getSystemService(
            it1,
            ClipboardManager::class.java
        )
    } as ClipboardManager
    val clip: ClipData = ClipData.newPlainText(label, getMissingString())
    clipboard.setPrimaryClip(clip)
}
```



4 Shortcomings and Feedback

Although I believe the final result is pretty good, I believe it's important to recognize its shortcomings and identify the sources. First of all, the handling and state of the style of the app: although it is usable, the theme is inconsistent and not pretty by any stretch of the word. Custom buttons do not have ripple effects, and don't change color while pressed, and using a light blue to show what item is selected in AddItemActivity makes for an unclear and uncomfortable experience. Likewise, the expansion button is not styled well. Animations left a lot to be desired as well. In addition, the swipe gesture should have been used in lieu of buttons: either to move from the ListItemFragment to the DashboardFragment or to delete an item from the list. Lastly, adding multiple items in sequence is a cumbersome experience; this is due to the small size of the remote database: in the future, it would make sense to never leave the camera view, and only display possible choices at the bottom, in order to quickly choose the correct item and scan all products in one go.

The sources of these issues is the time that was allotted to development: I strongly believe the project should have been presented at the start of the course, so that students could have used classes to work on the project and utilize hands-on learning to its full potential. Having said that, difficulties in implementing such a course are easily understandable; unfortunately, as a third-year course, it is impossible to delay the completion of the project if the student expects to graduate in the summer session.

5 Conclusion

This summarizes the main challenges I faced, from both a conceptual and a technical point of view, during the development of "Groceries!". It also explains implementative choices with reference to the code underlying most of the app. More information can be found in the commit history of the project, located here.