Lab 4

My Cocktails

Databases and Network Access in Android

UNIVERSITAT JAUME I

Mobile Device Applications
Degree in Video Game Design and Development

Goals of this Lab

In this lab you will develop a simple application to get and display basic information about drinks and cocktails from around the world provided by TheCocktailDB site. This site offers a free JSON REST API which we will use to get information about the composition and preparation of cocktails. The access to this API is free: they have a test API key which allows for a somewhat limited access to cocktails data. During the process you will apply what you have learned about application structure and you will practise using databases and accessing the network.

1. Introduction

In this lab you will develop an app that will query some REST services provided by TheCocktailDB site to get some data about the composition and preparation of popular cocktails around the world. The user of the app will be able to search for cocktails by categories or ingredients: for instance, "cocktails belonging to the Shake category" or "cocktails where Irish whiskey is one of its ingredients". The search will be performed either on Internet (accessing to the REST services provided by *TheCocktailDB*) or in a phone's local database where the user can store her favourite cocktails. The purpose of the local database is not only to reduce data usage (and, hence, the consumed bandwidth) but also let the user rate her favourite cocktails while storing their info about composition and the instructions for their preparation. That is why in this app you will practise both databases and network access in Android.

In a separate activity (screen), the app will show the results of the search performed. The layout of this activity will remain the same no matter whether an Internet or a local database query was made. Finally, at user request, the app will shown detailed information about the cocktail selected in another activity. This third activity will let the user add its data after rating it in an pop-up dialog. To get the complete picture of the working of the app, please **watch the video** available in *Aula Virtual*.

In the next section we will explain the working of the app, and then we will give you some guidelines to carry out the implementation. In the final section, we will instruct you on how the app will be graded. As always, you are advised to read this document completely before starting the actual implementation to have a better understanding of the whole project. Besides, it is advisable to watch the videos in *Aula Virtual* to get a full understanding of the working of the app and to learn how you could use Postman to inspect the JSON responses from the server's REST API.

2. Working of the App

The app will start with a screen in which the user selects the search criterion —by category name or ingredient— and the target of the search —the local database or the Internet server— (see figures 1 and 2). When she clicks any of the "Search" buttons, a screen appears in which a `RecyclerView` shows the cocktails which fulfil the given search criterion, one data item per cocktail (see Figure 3). When one item is pressed, the user is redirected to a screen showing detailed information about the cocktail (see Figure 4), mainly its ingredients and preparation. In this screen, the user is able to rate the cocktail through a dialog (see Figure 5). While the "Cancel" button of this dialog dismisses it, the "OK" button will store the score given to the cocktail through a `RatingBar`. Then, the user will be able to press a button to add the cocktail and all of its data to the local database. When this button is pressed an `AlertDialog` will inform the user of the actions performed in the local database (see Figure 6): pressing its "OK" button will finish the activity, hence returning to the screen showing the search results (Figure 3).

The search can be performed either by ingredient or category name but not by both. Also, the search can be carried out either in the local database or in the Internet server. The category names are stored in a `Spinner` while the ingredient names feed an `AutoCompleteTextView`. The type of search is selected through the corresponding `RadioButton`. Both radio buttons are grouped into a `RadioGroup`.

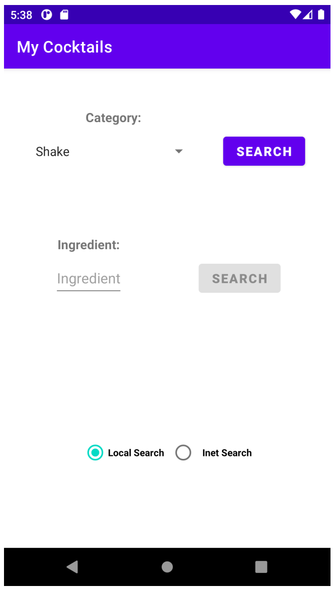


Figure 1: The screen to input the criterion search

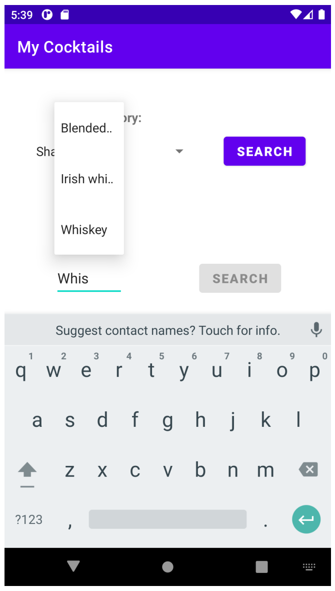
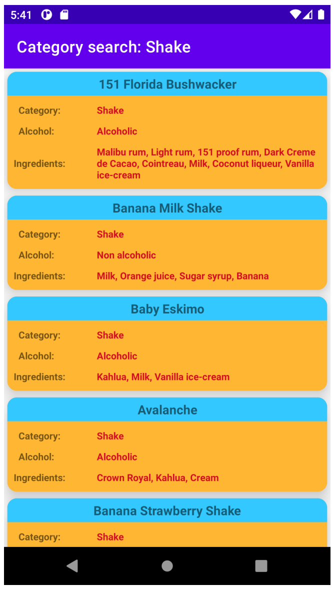


Figure 2: Setting the ingredient (partial input)



151 Florida Bushwacker
Category: Shake
Alcohol: Alcoholic
Ingredients: Malibu rum, Light rum, 151 proof rum, Dark Creme de Cacao, Cointreau, Milk, Coconut liqueur, Vanilla ice-cream

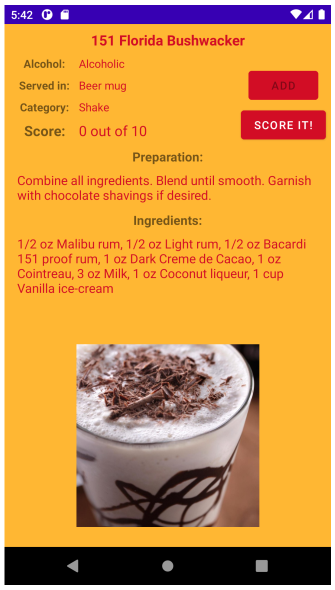
Banana Milk Shake
Category: Shake
Alcohol: Non alcoholic
Ingredients: Milk, Orange juice, Sugar syrup, Banana

Baby Eskimo
Category: Shake
Alcohol: Alcoholic
Ingredients: Kahlua, Milk, Vanilla ice-cream

Avalanche
Category: Shake
Alcohol: Alcoholic
Ingredients: Crown Royal, Kahlua, Cream

Banana Strawberry Shake
Category: Shake

Figure 3: The screen showing the search results



151 Florida Bushwacker
Alcohol: Alcoholic
Served in: Beer mug
Category: Shake
Score: 0 out of 10
Preparation:
Combine all ingredients. Blend until smooth. Garnish with chocolate shavings if desired.
Ingredients:
1/2 oz Malibu rum, 1/2 oz Light rum, 1/2 oz Bacardi 151 proof rum, 1 oz Dark Creme de Cacao, 1 oz Cointreau, 3 oz Milk, 1 oz Coconut liqueur, 1 cup Vanilla ice-cream

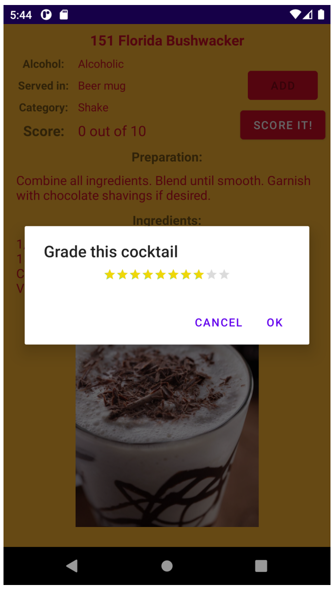
Figure 4: The screen showing a cocktail's data

According to the `View` selected —`Spinner` or `AutoCompleteTextView`— the corresponding `Button` at the right is enabled and the other disabled. When pressing the enabled button the search is performed.

The information about the existing categories and the ingredients (just 100 because of the limitations of the test API key employed) can be recovered from the API. The idea is that the first time the app is used, both lists are downloaded from the net and stored in the database. In subsequent runs, both lists are recovered from the database. Bear in mind that while the list of categories will not grow, this is not the case of the list of ingredients since the user can add cocktails to the local database at her will and hence ingredients not downloaded in the first run could be added to this list.

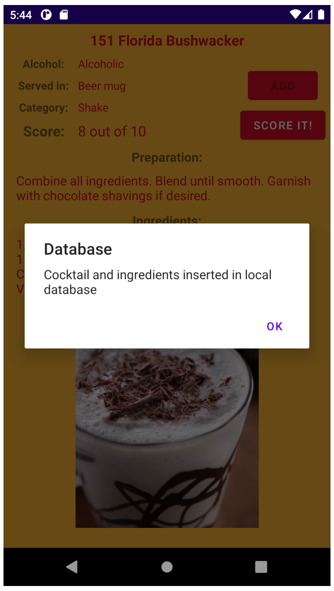
3. Implementation Guidelines

Before the implementation proper, we will comment on a couple of important aspects for preparing the dependencies of the project.



151 Florida Bushwacker
Alcohol: Alcoholic
Served in: Beer mug
Category: Shake
Score: 0 out of 10
Preparation:
Combine all ingredients. Blend until smooth. Garnish with chocolate shavings if desired.
Ingredients:
Grade this cocktail

Figure 5: The dialog to rate a cocktail



151 Florida Bushwacker
Alcohol: Alcoholic
Served in: Beer mug
Category: Shake
Score: 8 out of 10
Preparation:
Combine all ingredients. Blend until smooth. Garnish with chocolate shavings if desired.
Database
Cocktail and ingredients inserted in local database

Figure 6: The dialog which appears after adding a cocktail to the local database

3.1. Preparation of the project

After you have created the project, open the app `build.gradle`. This is the file that begins with

```
plugins {
    id 'com.android.application'
    id 'org.jetbrains.kotlin.android'
}
```

Add to these plugins a new line so that it reads:

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

Add to it, at the end of the `dependencies` section the following:

```
    def room_version = "2.4.2"
    implementation "androidx.room:room-runtime:$room_version"
    kapt "androidx.room:room-compiler:$room_version"
    implementation 'com.android.volley:volley:1.2.1'
    implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-android:1.3.9'
```

The last step of preparation is to add the `INTERNET` permission to the manifest. Add the line

```
    <uses-permission android:name="android.permission.INTERNET"/>
```

just before the `application` tag in `AndroidManifest.xml`.

As always, you are free to choose any order in implementing the app, but the nature of the activities involved suggests an order in which you first develop the `Activity` for setting the search criterion and the type of search, then the `Activity` for showing the search results, and, finally, the `Activity` for showing the data of the selected cocktail with their corresponding dialogs. Note that the first `Activity` will have to communicate only the parameters relative to the desired search: the name of the category/ingredient and whether the search will be performed in the local database or not. Also, the second `Activity` will have to pass the information about the cocktail, its preparation and its ingredients to the third `Activity` This lends itself to a simple organisation in which there are three packages, one for each activity. Each of the activities will have the corresponding presenter. For this app, you can have a single model for all of the activities. In that case, you will have an additional package for the model. However, if you feel more comfortable using a different model for each of the activities, do so. Simply tell us in your memory how you have finally organised your code.

Also, as you will use the database to store cocktail's data, a sensible order is to implement a first version without database and later add the ability to store the data related to cocktails and its elaboration. Therefore, we will start by describing how to access the server.

3.2. Network Access

We will use the Volley library to access the network. As you know, it deals with most of the hassle of accessing to the Internet. The basic idea is to keep a single instance of a `RequestQueue` and use it to ask for the pages needed. The information provided by the API comes in JSON format so you will use a `JsonObjectRequest` as explained here. Then the result can be parsed using the `org.json` library.

3.2.1. The Queries

The documentation for the API can be found here. We will use the test API key "1" to perform the queries. There are some limitations with this key: only some API requests are allowed and results are limited to 100 items. However, it does not require to sign up (no personal data are involved) and it suffices for our educational purposes.

We are interested in three types of queries. The first type is the one used to list search filters, such as the categories or ingredients: let's call them *list filter queries*. The second type searches *TheCocktailDB* database according to a concrete value of some of the previous filters, such as the name of a category or ingredient, and returns the corresponding cocktails: let's call them *filter queries*. The third type returns the data stored in the *TheCocktailDB* database for a given item, for instance a cocktail or an ingredient: let's call them *lookup queries*. All of the queries begin with the same common prefix, which includes the value of the test API key:

`https://www.thecocktaildb.com/api/json/v1/1/`

Also, bear in mind that all of these queries return a JSON object which contains a *single* JSON array labelled with the key `drinks`. The items resulting from the query can be found inside this array as JSON objects (one object per item).

List Filter Queries These type of queries have a simple structure:

- To get all available categories use
  `https://www.thecocktaildb.com/api/json/v1/1/list.php?c=list`
- To get the ingredients (limited to a maximum of 100 items) use
  `https://www.thecocktaildb.com/api/json/v1/1/list.php?i=list`

You can see in the Appendix A.1 the results achieved by both queries.

Filter Queries The filter queries are also simple and have a similar structure.

- To get the cocktails which belong to the category named *Soft Drink*, the query is
  `http://www.thecocktaildb.com/api/json/v1/1/filter.php?c=Soft Drink`
- To get the cocktails which include the ingredient named *Irish whiskey* in their preparation, the query is
  `http://www.thecocktaildb.com/api/json/v1/1/filter.php?i=Irish whiskey`

You can see in the Appendix A.2 the results achieved by both queries.

Lookup Queries We will use this kind of queries just to return all of the data stored for a given cocktail. Therefore, we will use a query like the one shown below, which returns the data stored for the cocktail whose identifier is *16405*.

`http://www.thecocktaildb.com/api/json/v1/1/lookup.php?i=16405`

Also, you can see in the Appendix A.3 the results returned by this query.

3.2.2. A Class for Performing the Queries

A way of easing the implementation of the model is to implement a class to take care of all the aspects of querying the server. This class, say `Network`, would have two public functions for the *list filter queries*: one to get the list of categories and another to retrieve the list of ingredients. Also, there would be two public functions for the *filter queries*: both let us search for the cocktails but one will retrieve the ones which belong to a category while the other will retrieve the ones which have the given ingredient.

Finally, you will have to implement a function to retrieve the cocktail's data (*lookup queries*). You are free to make this function public or private. In the latter case, the function will be called by the public functions for the *filter queries* so that they can return a list of cocktails with all the required data. Also, bear in mind that this function can be split into two: one to retrieve a given cocktail's data and another to traverse the list returned by the functions that implement the *filter queries*. Feel free to implement the *lookup queries* as you please but do not forget to briefly describe the option chosen in the memory to be submitted (see Section 5).

It is important to realise that `Network` does not access to the local database. It is the responsibility of the model to use the server only when required. For instance, the initial loading of the lists of categories and ingredients should be done from the network server while in the next uses of the app should be done from the database, as explained in Section 3.4.2.

All the functions will return their results using a listener. Since we are using Volley, the best is to pass an instance of `Response.Listener` for normal results and a `Response.ErrorListener` for announcing possible errors.

`Network` has a single property to store the `RequestQueue` that will be created in the constructor. You may need some private functions to help the implementation of the public functions. A suggestion is to use two functions for each query:

- A function to actually send the request, this function will build the `JsonObjectRequest` and `add` it to the queue.
- A function to parse the JSON response.

`Network` will be a singleton to ensure that there is only one instance of `RequestQueue`. Read the Appendix B to see a simple technique that can be used both for `Network` and for the class of the database.

When parsing the results of the *list filter queries*, it is useful to return lists storing instances of the same classes used as tables in the database, that is, to return a list of categories or ingredients, use a `List<Category>` or a `List<Ingredient>` where `Category` and `Ingredient` are the entities corresponding to a category or an ingredient respectively. This will simplify the management of the database. These classes are explained in Section 3.3.1.

For the results of the combination of the *filter queries* with the *lookup queries*, i.e., lists of cocktails with their ingredients and their respective measures in the mix, it could be convenient to define a class that groups together instances of other classes which are entities in the database (see Section 3.3.1). For example, you could define a class which had two attributes, the instance of a `Cocktail` and a `List<CocktailIngredient>`, being `Cocktail` an entity in the database and `CocktailIngredient` the implementation of the many-to-many relationship between `Cocktail` and `Ingredient`.

Finally, bear in mind that not always the data stored in the *TheCocktailDB* are consistent, see for instance the result of the *lookup query* in Appendix A.3. You can see that there is no value for the measure of the fourth ingredient (`null`). Observe also, that there is a maximum of fifteen ingredients to be considered: obviously the vast majority of cocktails stored in the *TheCocktailDB* has less ingredients.

Desired Data As you can see in the Appendix A.3, there are several data available for cocktails. It is not necessary that your app process all of them. It suffices that you store the identifier of the cocktail, the name of the category to which belongs, its name, whether it has alcohol or not, the type of glass where it should be served, the instructions to prepare it (only in English) and the list of ingredients that it has along with the corresponding measures to make the mix.

Be aware that some of the labels in the JSON structure has `null` values. Also, all of the data are stored as strings. This shouldn't be a problem since when reading this null values using the `getString` function the returned value is the string `"null"`. Had you store another string for these `null` values, let's say `"None"`, you could use a code similar to:

```kotlin
val ingredient: String = if (element.isNull("$INGREDIENT_NAME_LABEL${i}"))
                             "None"
                         else element.getString("$INGREDIENT_NAME_LABEL${i}")
```

Also, remember that you can protect your code against possible missing JSON labels by using the `optString` function.

3.3. The Classes for the Database

The library that you will use for the implementation of the database is Room. The way of working with a database using this library consists in defining a class for each of the tables in the database. These classes are used in turn to define a DAO (Data Access Object), which is an interface, and the Database itself, which is an abstract class. Then the static method `databaseBuilder` of the `Room` class is used to build the actual database.

3.3.1. The Entities

In *My Cocktails*, the entities of the database will be `Category`, `Ingredient`, `Cocktail`, and `CocktailIngredient`. As `Category` and `Ingredient` are very simple data classes, they can be used to represent the filter entities used in *list filter queries* throughout the whole application. So even if you don't start by writing the database, it is useful that you create a package called `database` within the `model` package and create in it each of these classes. Follow the instructions given here and take into account these aspects:

- Although entities such as `Category` and `Ingredient` have only one field of type `String`, its name, which acts also as their primary key, they should be declared as data classes to be used in the database. It is convenient that their `toString` functions simply return the name.
- The class `Cocktail` uses the `Int` value of the identifier in the JSON code as its primary key. This class also has a foreign key, `category`, that links the cocktail with the category to which belongs and has an index on this foreign key.
- Finally, the class `CocktailIngredient` has two foreign keys since it implements the many-to-many relationship between `Cocktail` and `Ingredient`. These foreign keys are `cocktailId`, the identifier of a cocktail, and `ingredient`, the name of an ingredient. Again, indexes should be created for both foreign keys.

Please, bear in mind that it is very convenient to make classes `Cocktail` and `CocktailIngredient` implement `Parcelable`, so that instances of them can be passed from one `Activity` to other.

3.3.2. The DAO

As you can see here creating the DAO is relatively easy. For the insert functions, you only need to annotate them with `@Insert` and use the proper `OnConflictStrategy`. In *My Cocktails*, the needed functions are one for the list of categories, other for the list of ingredients, another for inserting a cocktail, and a last one for the list of `CocktailIngredient`.

Regarding the queries, you will just need three simple queries and two not so simple. For the simple queries, create a function for recovering the list of all the categories (either as a list or as an array), other for recovering the list of all the ingredients (again as a list or as an array), and another for recovering the (list or array of) cocktails which belong to a given category. In each of those queries, use `ORDER BY name` so that the results are properly sorted.

The two more elaborated queries involve the entities `Cocktail` and `CocktailIngredient`. They have to allow for the recovering of a list or an array of cocktails given an ingredient and for the recovering of a list or an array of ingredients and their respective measures given a cocktail, respectively. Consider the use of the SQL `LEFT JOIN` keyword to select the proper database records and fields in both queries. Also, use `ORDER BY cocktail.name` in the first query so that the list or array of cocktails returned is properly sorted.

3.3.3. The Database

Finally write the class for the database. A simple way to do this is to use an abstract class with the annotation `@Database` and use that to get the dao in the "real class" using the schema of the slides seen in theory.

Make the class a singleton as explained in Appendix B.

3.4. The Model

The model will offer two functions for *list filter queries*:

- One for retrieving the list of categories.
- One for retrieving the list of ingredients

Also, the model will have to use a function for retrieving a list of cocktails along with their ingredients and respective measures. In fact, it is advisable that this function is split into two: a function for retrieving these data from the local database and another for retrieving them from the network server.

Finally, the model will have a function for inserting a cocktail with its corresponding list of ingredients and their measures in the local database.

As you can imagine the first two functions are used in the first `Activity` for accessing the database (and the network if it were the first use of the app). The function (or functions if you have followed our advice) which retrieves the list of cocktails —from the database or the network– is used in the second `Activity`. Finally, the function for inserting a cocktail's data in the local database is used in the third `Activity`. Therefore, you might want to split the model in three. However, we feel that it is easier to keep a single model shared by all the activities.

For the implementation of the model, it is important to remember that the accesses to the database and the network will be done asynchronously. The next section explains the consequences of this.

3.4.1. Asynchronous Accesses

The accesses to the database and to the network have to be made asynchronously, which implies passing listeners as parameters to process the results. To unify the way it is done, we will use the `Response.Listener` interface both for the net and the database. As you can see in the example in this page, this interface has a single function, `onResponse`, that takes just one parameter with the type determined by the type parameter of the interface (e.g. if you use a `Response.Listener <String>` the parameter `onResponse` will be a function that receives a `String`).

Also, the errors will be processed using a `Response.ErrorListener`. In this case, the function processing the error will receive a `VolleyError` that can be transformed into a `String` using the `toString` function.

Therefore, each of the functions of the model will receive together with the parameters that specify the query, a listener to deal with the result and an error listener to deal with the errors. Keep in mind that you can omit the error listener in those functions of the model that access only to the database (not the network). For instance, this would be the case of the function that get the list of cocktails and their ingredients from the database if you have followed our previous advice regarding the implementation of the functions required in the model. Likewise, you could also omit the listener in functions that do not return any data. This is the case, for instance, of the function which inserts a cocktail's data in the database.

In the case of the *list filter queries*, the function will first check whether the answer is already in the database. In that case, it will return it. If not, the function will recover the answer from the network, store it in the database, and return it using the listener.

The accesses to the network through the `RequestQueue` are automatically managed so that you don't need to care about the threads. However, in the case of the database, you will need to use Kotlin coroutines. Fortunately, the functions that implement *list filter queries* will follow a very simple schema. We will exemplify the idea for the case of the function `getCategories` that returns all the categories.

Let's assume that the function `getCategories` of the property `network` has two parameters, one for the normal listener and one for the error listener. Also, let's assume that the property `database` has the database object and the property `dao`. The idea is to use `launch` to start a new coroutine to do the job. Within the coroutine, the function `withContext` is used to block and wait for a result. So the function `getCategories` is:

```kotlin
fun getCategories(listener: Response.Listener<List<Category>>,
                  errorListener: Response.ErrorListener) =
    // Launch a coroutine
    GlobalScope.launch(Dispatchers.Main) {
        // Blocking call to read the categories from the DAO
        val categories = withContext(Dispatchers.IO) {
            database.dao.getCategories()
        }
        if (categories.isEmpty()) {
            // Recover categories from the net
            network.getCategories(Response.Listener {
                    // Launch a coroutine to store the
                    // categories in the database
                    GlobalScope.launch {
                      database.dao.insertCategories(it)
                    }
                    // Pass the categories to the listener
                    listener.onResponse(it)
                }, Response.ErrorListener {
                    errorListener.onErrorResponse(it)
                })
        }
        else
            // Pass the categories to the listener
            listener.onResponse(categories)
    }
```

Note that the first parameter of `network.getCategories` is effectively building a new `Response.Listener`. And within that listener a new coroutine is launched to store the list in the database.

3.4.2. Initialisation of the Database

When the app starts for the first time the database is empty. However, when the main activity starts it has to ask its presenter for both the list (or array) of categories to fill in the `Spinner` and the list (or array) of ingredients to populate the AutoCompleteTextView. The presenter, in turn, will request these data to the model. To this end, it should use the function that queries the database for both: categories and ingredients. Obviously, and since the database is empty, an empty list will be returned. If this happened, then the presenter would have to use the functions of the model that access the network to collect these data (*list filter queries*). These data, if no network error were thrown, would also be inserted in the database. Proceeding this way, the list of categories and the (initial) list of ingredients will be stored in the local database for future uses of the app.

3.5. Some Considerations for the Activity for Setting the Search Criterion

As explained in the introduction, this is the Activity that allows the user to determine both the search criterion and the type of search (local or Internet). Now that we have commented on particulars of the database, let us cover the view and the presenter, which are both quite simple.

3.5.1. The View

The layout for this Activity has no especial difficulties. The category to search for is selected from a Spinner while the ingredient can be selected using an AutoCompleteTextView.

As you imagine, the data to fill them will come out of the model, so it is convenient that the view offers functions named like showCategories and showIngredients to populate the drop-down list and the completion from a list of Category or Ingredient, respectively. Each of these methods will build and set the corresponding ArrayAdapter. For instance, if the showIngredients function receives a list called ingredients and the AutoCompleteTextView is in ingredientTextView, it can prepare the adapter like this:

```
val adapter = ArrayAdapter(this,android.R.layout.simple_dropdown_item_1line, ingredients)
ingredientTextView.setAdapter(adapter)
```

After that, it can add a text changed listener to call the presenter when a correct ingredient is entered:

```
addTextChangedListener(object: TextWatcher {
    override fun beforeTextChanged(p0: CharSequence?, p1: Int, p2: Int, p3: Int) {}

    override fun onTextChanged(p0: CharSequence?, p1: Int, p2: Int, p3: Int) {}

    override fun afterTextChanged(p0: Editable?) {
        val ingredient = p0.toString()
        ingredients.binarySearch { it.name.compareTo(ingredient) }.let {
            if (it >= 0)
                presenter.setChosenIngredient(ingredients[it])
        }
    }
})
```

The use of binarySearch ensures that checking if the entry is an ingredient is efficient (but it requires that the list of Ingredient is alphabetically sorted).

In the case of the Spinner, the preparation of the adapter is similar, just check the documentation. The Spinner requires that an OnItemSelectedListener is created and then you should override the function onItemSelected to call the presenter and pass the category selected to it (spinnerCategories holds the Spinner):

```
object : AdapterView.OnItemSelectedListener {
  override fun onItemSelected(p0: AdapterView<*>?, p1: View?, p2: Int, p3: Long) {
      val category : Category = spinnerCategories.getItemAtPosition(p2) as Category
      presenter.setChosenCategory(category)
  }
  override fun onNothingSelected(p0: AdapterView<*>?) {}
}.also { spinnerCategories.onItemSelectedListener = it }
```

You will need a function to show possible errors. It has a parameter with the message and shows it using either a Toast or a Snackbar. Also, it could be convenient to implement a function to clear the AutoCompleteTextView.

There will be a ProgressBar that will be active during the (short) periods that the presenter is waiting for the answer from the database (or the network).

The management of the radio buttons to choose the type of search can be done through a property, as you did in the *Tennis Scores* app.

A final function will be used to start the activity for the search results. This function will receive the category or the ingredient selected and a Boolean for the type of search chosen and use them to build an intent and start the other Activity.

But as always, don't write all the functions at once, follow the flow of the program and write them as needed. We also suggest that you prepare a different layout to work in landscape mode.

3.5.2. The Presenter

The presenter will have a constructor that will get the categories and the ingredients from the model and pass them to the view. Note that the part of sending both lists to the view has to be done within a Listener.

Add functions to react to the events in the view. For instance, a function `setChosenIngredient` that will be called when a correct ingredient has been written in the corresponding input field. This function will store the ingredient to be used in a possible search and will enable or disable conveniently some elements of the view.

Apart from the functions reacting to the inputs, the presenter will have a function that is called when any of the buttons is pressed to perform the search. It can be, for instance, `doSearch`, that will call the function in the view to start the activity for the search results.

3.6. Some Considerations for the Activity for Showing the Search Results (cocktails)

This `Activity` is quite simple. It essentially consists in a `RecyclerView` showing the cocktails found as a result of the search performed that responds to clicks by starting a third activity. In this final `Activity` the details about the composition and preparation of the cocktail are shown. However, even in this case it is useful to follow the MVP pattern. Now, let us comment the view and the presenter classes.

3.6.1. The View

The view has only two elements, a `RecyclerView` that shows the search results and a `ProgressBar` that is displayed while the app is waiting for the server to return the cocktails that match the search criterion. As you have practised it in other labs, we will not insist on how the `RecyclerView` is populated. Use a layout for the items that shows the cocktail's data as you wish.

To be able to start the third `Activity` when an item is selected, you need to use `setOnClickListener` as explained in class. You will need to pass a `Cocktail` and a list of `CocktailIngredient` to the third activity. It is advisable to use a new class that groups together both elements. Besides, this new class should implement `Parcelable`. This would be something very easy to achieve if `Cocktail` and `CocktailIngredient` were made `Parcelable`.

Like in the view for entering the search criterion, there will be a function that will be used to show possible errors while accessing the API.

3.6.2. The Presenter

The presenter for this `Activity` is extremely simple. Its constructor simply asks the model for the list of cocktails that match the search criterion input (from the database or from the network). Then, and through a listener, instructs the view to hide the progress bar and display the data. Note that the access to the *TheCocktailDB* API can fail. In that case, the presenter will simply ask the view to show the corresponding error. A better error treatment could be done, but this is not necessary for our simple app. However, it would be wrong that the app seemed to hang.

The other input point is simply that when the presenter is notified that a cocktail has been selected, the view is told to start the third (and final) `Activity`.

3.7. Some Considerations for the Activity for Displaying the Details of a Cocktail

This `Activity` requires that you put work into the design of its layout so that all of the elements of the view can be properly shown (using `TextView`) with somewhat different screen resolutions. Besides, it is mandatory to create a landscape version of the layout so that the elements of the view can also be properly shown when the phone is rotated.

3.7.1. The View

The view should offer to the presenter several functions so that its different `TextView` can be fill in with the proper strings. Apart from that, it has to implement two functions to create the two dialogs that are attached to this activity:

- An `AlertDialog` that has to be shown when the user has added the cocktail to the local database. Remember that a click on the OK button should finish the `Activity`.
- A dialog, which subclasses `DialogFragment`, that is employed by the user to rate the cocktail using a RatingBar. You will have to add a listener interface and store the listener in the `onAttach` function so that a function of the presenter can be called and the new rating of the cocktail can be passed to it.

To implement both dialogs, refer to the theory slides *Short Interactions with the User* and the corresponding explanations given in class.

3.7.2. The Presenter

The presenter for this `Activity` is very simple. It has a function for calling the different functions that offers the view for filling in the `TextView` with the corresponding data.

Also, it has to provide three other functions: one for calling the function of the view which creates the dialog to rate the cocktail, one for inserting the cocktail's data in the database, and one for updating the score of the cocktail when it has been rated by the user.

3.8. The Activity Lifecycle

The asynchronous nature of the queries to the database complicates the treatment of `onSaveInstanceState` since the recovery has to rebuild the view on steps. For this application it is not really necessary since its expected use is for short queries. Therefore, you are allowed not to save the state of the application in case of changes of display orientation and similar.

4. Optional: Displaying Images of Cocktails

As an optional part which is worth an extra point (see next section) you could modify your app so that the image of the cocktail selected is shown in the third `Activity`.

You will have to declare a new attribute in the `Cocktail` class to store the URL where its image can be downloaded. When parsing the corresponding JSON code returned by the *TheCocktailDB* server use the label `strDrinkThumb` (see Appendix A.3).

Besides, you will have to modify the layout of the third `Activity` to add an ImageView. Obviously, this will also affect to the view and the presenter.

Finally, you will have to add functions in the model and in the `Network` class so that the image can be downloaded as a `Bitmap` and properly passed to the presenter. Use the ImageRequest provided by the `Volley` library.

5. Submission and Grading

It is **required** to submit the code of the project to get a grade. Those projects whose code is not delivered **will not be graded** even if the PDF memory is submitted. The code of the project **must be built** in *Android Studio* without errors and the app **must** work without significant crashes. If the code can't be built or the app crashes due to notorious mistakes your project will be graded 0.

You are expected to store your project in a **git repository**. Therefore, the submission of its code will consist in making it available to us. Use a public service like github or bitbucket to this end. Your repository **must be private** but you will have to grant read access to `jcamen@uji.es` (bitbucket) or `jcamen@lsi.uji.es` (github) so that we can grade your work. **Any project whose code is in a public repository will be graded** 0.

This is **the only means to submit the code** of your project. Any other means, like a shared folder in *Google Drive* or a ZIP archive submitted to the task in *Aula Virtual*, will not be considered and thus your project will be graded 0. Please, do not bother us with e-mails or messages at this respect: your time and ours is very valuable.

We will provide a task in *Aula Virtual* to submit a small memory in PDF, **written in English**, giving a brief overview of the work done, the design decisions and the difficulties found (4 pages at most). It is also required to submit this memory, which will be considered the submission of the project. Anyway, keep in mind what we have said above about the code of the *Android Studio* project that you have developed. You should take into account the following points:

- The submission period ends at 23.59.59 of the day before the beginning of the next project.
- You can submit your work either individually or by pairs. In the case of pairs, only one member of the pair must submit the project.

Please, take into account the following when writing the memory:

- The design decisions include those aspects that are left open above, like whether to use one or more models, the use of complementary classes to store and pass data, the SQL queries performed to the database or the way the queries to the server's REST services are organised. Also, they must include a listing of the most relevant classes that you have used.

- In case the project has been done by a pair, the memory must include the names of both members of the pair.
- It must also include the id and address of the commit corresponding to the version of the project submitted for grading.

The submission will be graded with up to **one point** for the memory, up to **three points** for the good working and quality of the activity for searching by category or ingredients (local database or Internet), up to **three points** for the good working and quality of the activity for showing the search results, and up to **three points** for the good working and quality of the activity which shows the information about the cocktail selected: this includes the proper working of the dialogs and their connections to the activity. Please, bear in mind that a layout design which (almost) properly shows in phones with different screen resolutions will be considered.

The use of images for the cocktails, downloaded from the *TheCocktailDB* server, is **optional**. If your app includes the changes required to make it work properly you will be rewarded with **one *extra* point**.

Take special care to avoid that your application crashes anytime. It is better to have less functionality correctly implemented than to have an application that crashes.

A. Results of Some Example Queries

The result of all the queries, as mentioned in Section 3.2.1, is a JSON object which contains a *single* JSON array, labelled through the key `drinks`, which, in turn, contains all of the items as a result of the query performed: each item is returned as a *single* JSON object.

A.1. List Filter Queries

The result of the query `https://www.thecocktaildb.com/api/json/v1/1/list.php?c=list` which returns the list of category names is:

```
{
    "drinks": [
        {
            "strCategory": "Ordinary Drink"
        },
        {
            "strCategory": "Cocktail"
        },
        {
            "strCategory": "Shake"
        },
       ...
        {
            "strCategory": "Soft Drink"
        }
    ]
}
```

Observe that the key `strCategory` identifies the field where the category name is stored. You will use this name to perform subsequent *filter queries* (by category name, obviously).

The query `http://www.thecocktaildb.com/api/json/v1/1/list.php?i=list` returns the list of ingredients, limited to a maximum of 100 items. Its result is shown below:

```
{
    "drinks": [
        {
            "strIngredient1": "Light rum"
        },
        {
            "strIngredient1": "Applejack"
        },
        {
            "strIngredient1": "Gin"
        },
       ...
        {
            "strIngredient1": "Creme de Cassis"
        }
    ]
}
```

Again, observe that the key `strIngredient1` identifies the field where the ingredient name is stored. You will use this name to perform subsequent *filter queries* by the name of an ingredient.

A.2. Filter Queries

These queries are used to return the list of cocktails that match the name of either the category or the ingredient employed. The structure of both queries is almost identical and the results are just the same. For instance, the result of the query `http://www.thecocktaildb.com/api/json/v1/1/filter.php?c=Soft Drink` is:

```
{
    "drinks": [
        {
            "strDrink": "Bailey's Dream Shake",
            "strDrinkThumb": "https://www.thecocktaildb.com...qxrvqw1472718959.jpg",
            "idDrink": "14510"
        },
        {
            "strDrink": "Belgian Blue",
            "strDrinkThumb": "https://www.thecocktaildb.com...jylbrq1582580066.jpg",
            "idDrink": "14071"
        },
        {
            "strDrink": "Bleeding Surgeon",
            "strDrinkThumb": "https://www.thecocktaildb.com...usuvvr1472719118.jpg",
            "idDrink": "16295"
        },
       ...
        {
            "strDrink": "Zoksel",
            "strDrinkThumb": "https://www.thecocktaildb.com...ft8ed01485620930.jpg",
            "idDrink": "15691"
        }
    ]
}
```

And the result of a query by ingredient name, `http://www.thecocktaildb.com/api/json/v1/1/filter.php?i=Irish whiskey`, has an identical structure:

```
{
    "drinks": [
        {
            "strDrink": "Hot Creamy Bush",
            "strDrinkThumb": "https://www.thecocktaildb.com...spvrtp1472668037.jpg",
            "idDrink": "14782"
        },
        {
            "strDrink": "Irish Coffee",
            "strDrinkThumb": "https://www.thecocktaildb.com...sywsqw1439906999.jpg",
            "idDrink": "13971"
        },
       ...
        {
            "strDrink": "Tipperary",
            "strDrinkThumb": "https://www.thecocktaildb.com...b522ek1521761610.jpg",
            "idDrink": "17828"
        }
    ]
}
```

This type of queries returns only the name of the cocktail (`strDrink`), the URL to download an image of the cocktail (`strDrinkThumb`), and its identifier (`idDrink`). You will have to use these identifiers in subsequent *lookup queries* to retrieve the stored data for each cocktail.

A.3. Lookup Queries

You will use this type of queries to retrieve the data stored in the *TheCocktailDB* server for a given cocktail employing its identifier. For instance, this query: `http://www.thecocktaildb.com/api/json/v1/1/lookup.php?i=16405` returns the following JSON code:

```
{
    "drinks": [
        {
            "idDrink": "16405",
            "strDrink": "A Piece of Ass",
            "strDrinkAlternate": null,
            "strTags": null,
            "strVideo": null,
            "strCategory": "Other/Unknown",
            "strIBA": null,
            "strAlcoholic": "Alcoholic",
            "strGlass": "Highball glass",
            "strInstructions": "Put ice in glass...Fill with Sour Mix.",
            "strInstructionsES": null,
            "strInstructionsDE": "Gib Eis in ein Glas...Mit Sour Mix auffüllen.",
            "strInstructionsFR": null,
            "strInstructionsIT": "Metti il ghiaccio in un...Riempi con Sour Mix.",
            "strInstructionsZH-HANS": null,
            "strInstructionsZH-HANT": null,
            "strDrinkThumb": "https://www.thecocktaildb.com...tqxyxx1472719737.jpg",
            "strIngredient1": "Amaretto",
            "strIngredient2": "Southern Comfort",
            "strIngredient3": "Ice",
            "strIngredient4": "Sour mix",
            "strIngredient5": null,
            "strIngredient6": null,
            "strIngredient7": null,
            "strIngredient8": null,
            "strIngredient9": null,
            "strIngredient10": null,
            "strIngredient11": null,
            "strIngredient12": null,
            "strIngredient13": null,
            "strIngredient14": null,
            "strIngredient15": null,
            "strMeasure1": "1 shot ",
            "strMeasure2": "1 shot ",
            "strMeasure3": "cubes",
            "strMeasure4": null,
            "strMeasure5": null,
            "strMeasure6": null,
            "strMeasure7": null,
            "strMeasure8": null,
            "strMeasure9": null,
            "strMeasure10": null,
            "strMeasure11": null,
            "strMeasure12": null,
            "strMeasure13": null,
            "strMeasure14": null,
            "strMeasure15": null,
            "strImageSource": null,
            "strImageAttribution": null,
            "strCreativeCommonsConfirmed": "No",
            "dateModified": "2016-09-01 09:48:57"
        }
    ]
}
```

Observe that the array `drinks` will always have a single JSON object containing all of the cocktail's data.

B. Creating Singletons

The recommended way to implement singletons in Kotlin is to use `object`. However, there is a problem when the constructor of the class needs parameters. In this appendix you can see a simple technique to create singletons that have a parameter in the constructor. As an example, let's suppose that you have the class `Network` that has a constructor with the parameter `context` of type `Context`:

```
class Network (context: Context) {
  ...
}
```

Transforming it to a singleton is as simple as declaring the constructor private and adding a companion object of the generic class `SingletonHolder`. That means that the new version of `Network` is:

```
class Network private constructor (context: Context) {

  companion object: SingletonHolder<Network, Context>(::Network)
  ...
}
```

Now, the only way to get an instance of `Network` is to use the function `getInstance` like this:

```
val network = Network.getInstance(context)
```

The class `SingletonHolder` is this:

```
open class SingletonHolder<out T, in A>(private val constructor: (A) -> T) {

    @Volatile
    private var instance: T? = null

    fun getInstance(arg: A): T =
        instance ?: synchronized(this) {
            instance ?: constructor(arg).also { instance = it}
        }
}
```

As you can see, it stores a single instance of an object of type T and takes care of any synchronisation problems.