TrackingMyPantry

STUDENT: Luca Borghi

EMAIL: luca.borghi5@studio.unibo.it

MATRICULATION: 0000880151

Android application for the course “laboratorio di applicazioni mobili” @ University of Bologna.

Index

Functionalities

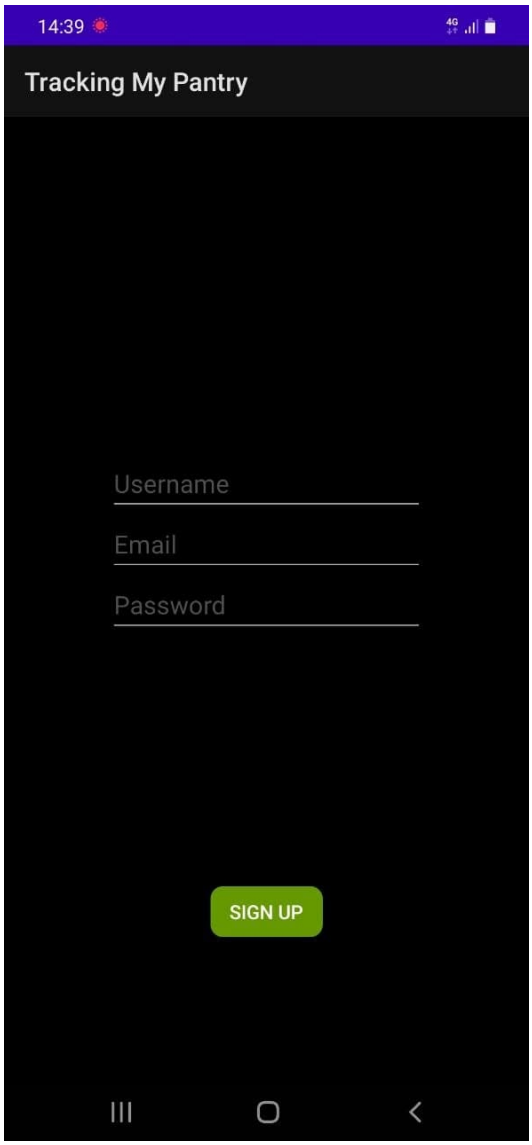
Sign-up and sign-in

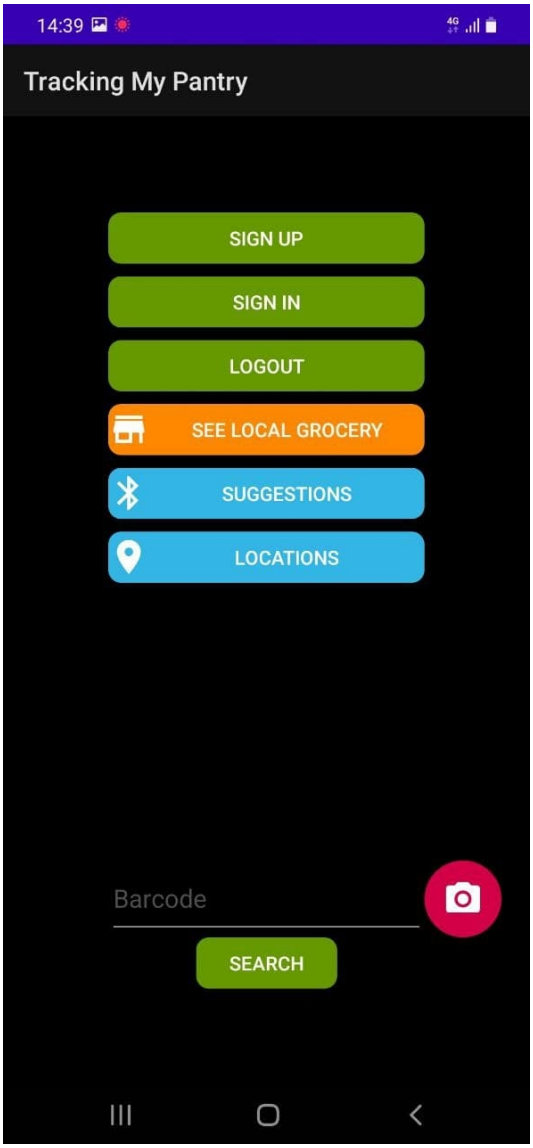
The user can sign up to the remote service, by clicking Sign up button in the “home” and inserting his personal data: username, email, and password (See the picture in the next page). Once the user created an account, he can log in through a sign-in form, by specifying his email and password. The sign-in form is the same as the sign-up one, except for it lacks of username field. When the user is logged, new buttons will appear:

- logout button, that is self-explanatory;

- a barcode input field and a button with symbol of a camera, which are useful to catch a barcode and carry out a search;

- search button just to execute a search.





Products search

The user can carry out a search either by typing the barcode manually…

Barcode scanning

…or by clicking the camera button and scanning one or more barcodes. As mentioned before, this is possible only when the user is logged. Once the user ended the scanning, he can stop, select a barcode and execute a search or trying once again, by clicking retry button. Home button is useful to come back to home page. Permissions for camera will be asked to the user if he is not already owning them.

Buying a product

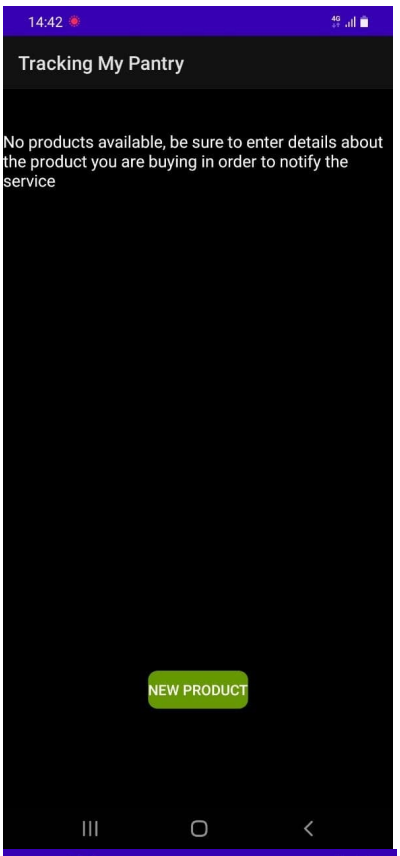
A search can result in zero or more products, anyway the user can decide to buy an already existent one or he can “create” a new product by clicking the “New product” button.

Expiration date

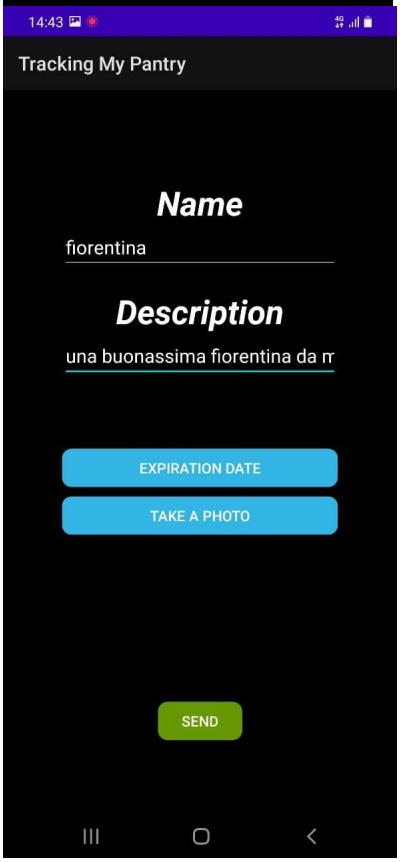
When the user is buying a product or he’s creating a new one, he can choose (it’s not mandatory) to specify an expiration date for that product. The expiration date is something that is tracked locally and it is not concerned with the remote service.

Taking a picture for a product

When the user is creating a new product, he can insert a name for the product, a description, an expiration date as well as taking a picture that is representative for the product. Permissions for camera will be asked to the user if he is not already owning them.


14:42
Tracking My Pantry
No products available, be sure to enter details about the product you are buying in order to notify the service
NEW PRODUCT


22:06
Tracking My Pantry
Choose the product you wish to buy or notify the service about a new one
BOTTIGLIA D'ACQUA
acqua fresca
SET AN EXPIRATION DATE AND CHOOSE
CHOOSE
ACQUA FRIZZANTE
NEW PRODUCT


14:43
Tracking My Pantry
Name
fiorentina
Description
una buonassima fiorentina da m
EXPIRATION DATE
TAKE A PHOTO
SEND

Local pantry

The user can see and manage the products he bought previously, by clicking “Local grocery” button in the homepage and then clicking “See your products” button. Note that expired products will appear with red color.

Collections

Moreover, he can gather a certain number of products in custom collections; the user can create a new collection by clicking “Create collection” button: it will be asked him to insert a name for the collection.

Managing products

The operations that can be carried out on products are:

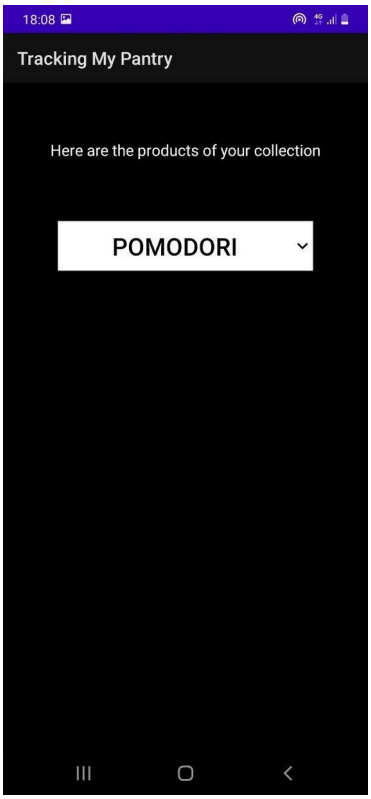
- See the details of the product (e.g. name, description, etc.);

- increase or decrease the quantity of a product;

- adding/removing a product to/from a collection;

- set a new expiration date for a product.

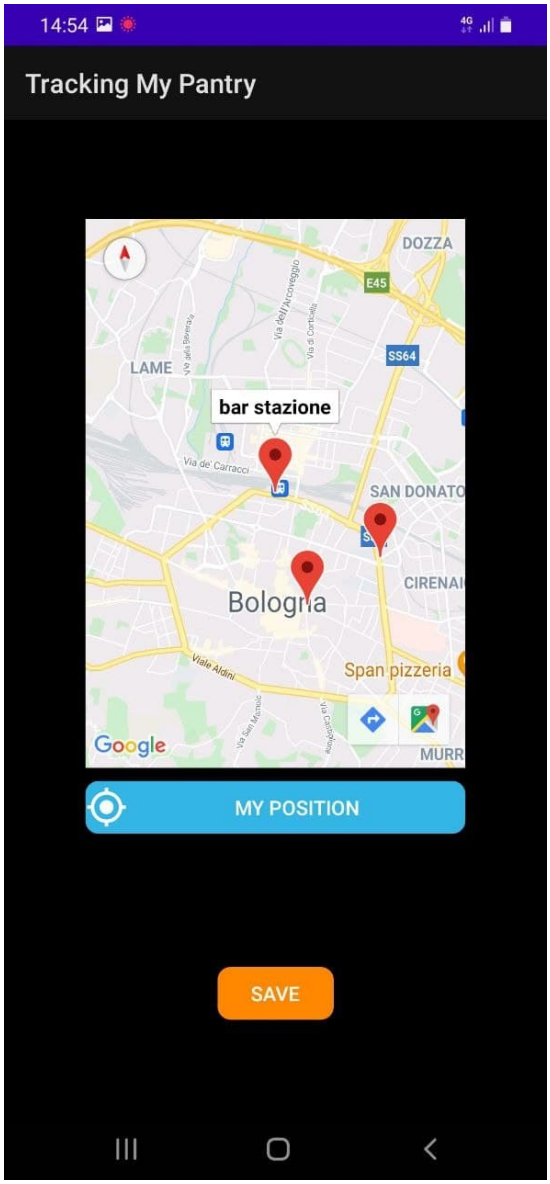
Obviously, if a collection has products, by clicking on the button of that collection, all products belonging to the collection will appear.



Collections

Saving places

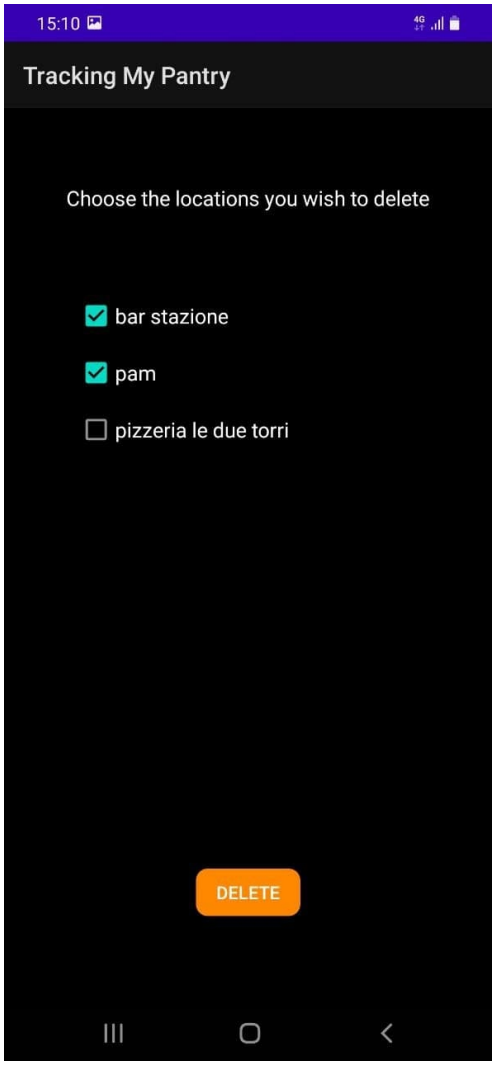
Another interesting functionality can be seen by clicking “Locations” button in the homepage and then to “Add new locations” entry. What will be shown to the user is a google map and a pair of button. When the map is loaded, the user can navigate it and clicking it. A click on the map makes the user performs the creation of a marker; it will be asked to insert the name of the point of interest. By clicking “My position” button, it will be asked permissions to fetch current position of the device (and to enable location, if it is not enabled), in order to create a marker in the current position. The save button is useful to store new added markers, indeed, every time the user use this functionality will see markers created and saved in the past on the map. If the user try to come back without saving changes, it will be asked if he’s sure to do it.



Locations on the map

Deleting places

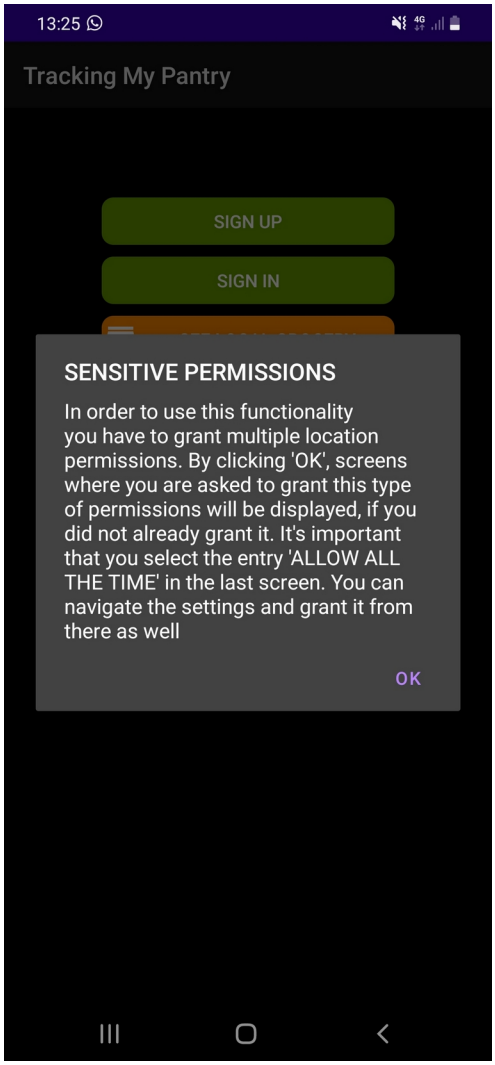
The user can delete stored places as well (maybe a market gets closed forever or the user is not interested anymore). This is done by clicking “Locations” button in the homepage and selecting the “Remove your locations” entry. Then, now it’s up to the user to choose which locations to delete.



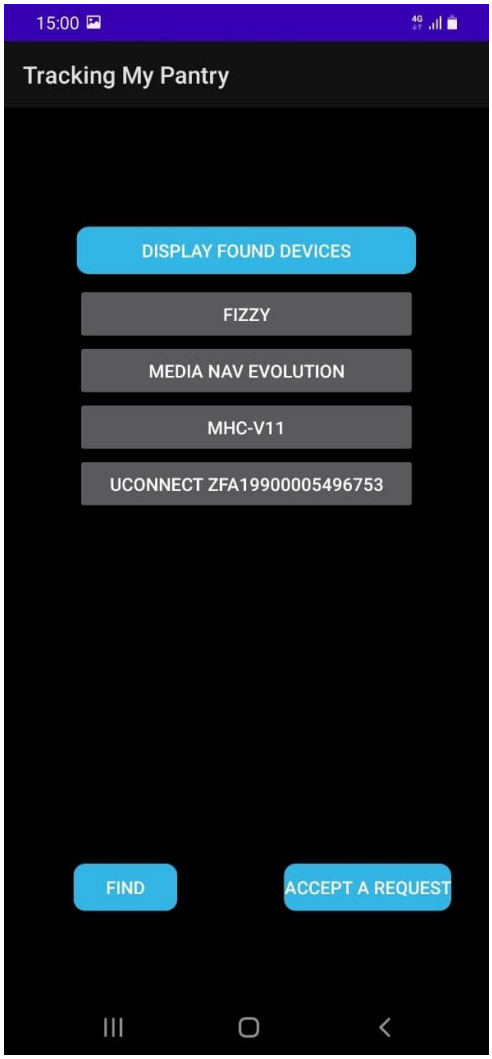
Bluetooth connection with other users

In the homepage, there is another button: “Suggestions”. Well, the user can exchange suggestions with other users about items and places. Before exchanging data, users have to connect their devices via a bluetooth channel; in order to this, a set of permissions has to be granted and the way of granting it depends from android version.

This is shown only on Android XXX, where XXX >= 10:



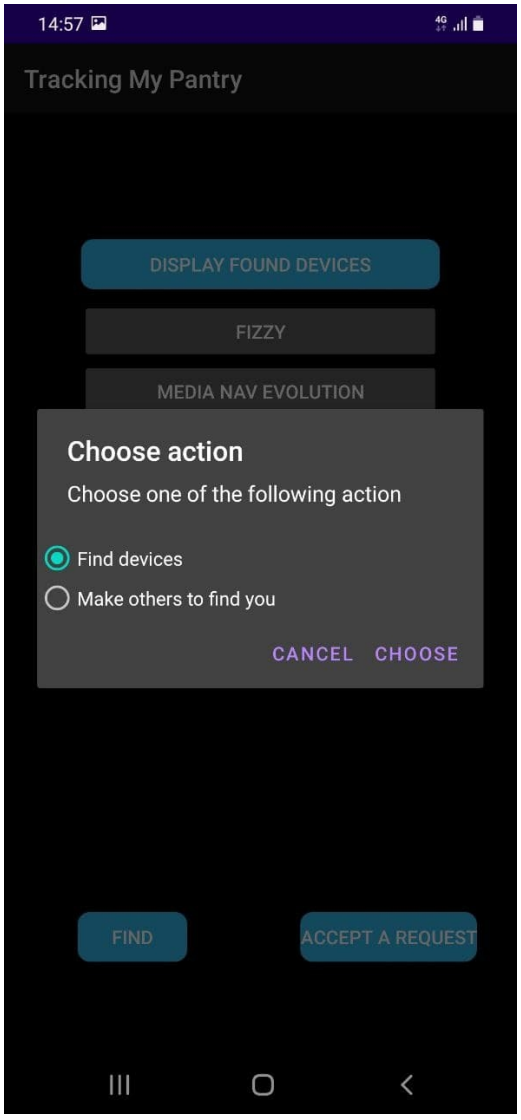
So, by clicking the “Suggestions” button, a request for enabling bluetooth will be performed, if bluetooth is not already enabled, then the required permissions will be asked. At this point the user will see the paired devices and by clicking one of them, he will try to initiate a connection.



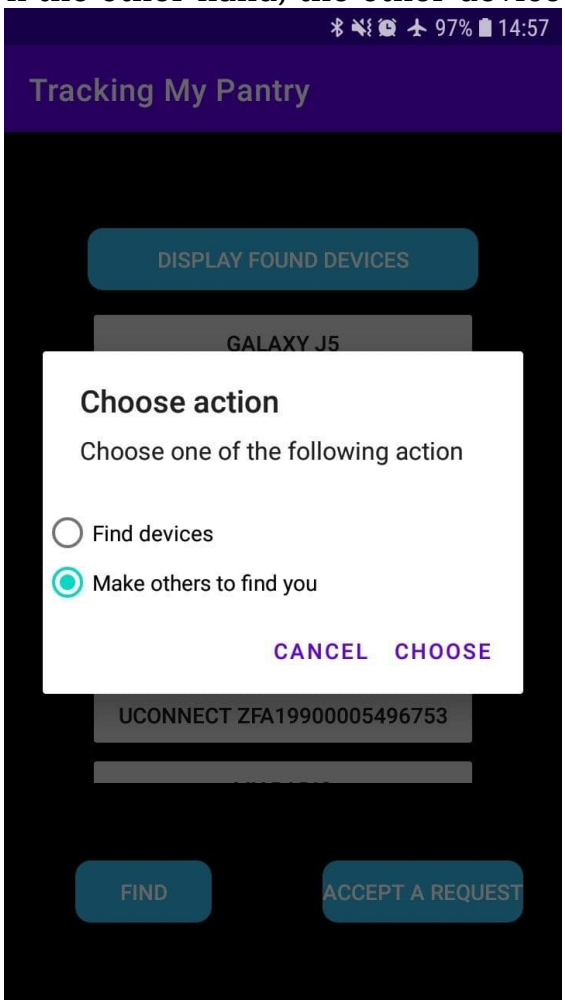
Paired devices

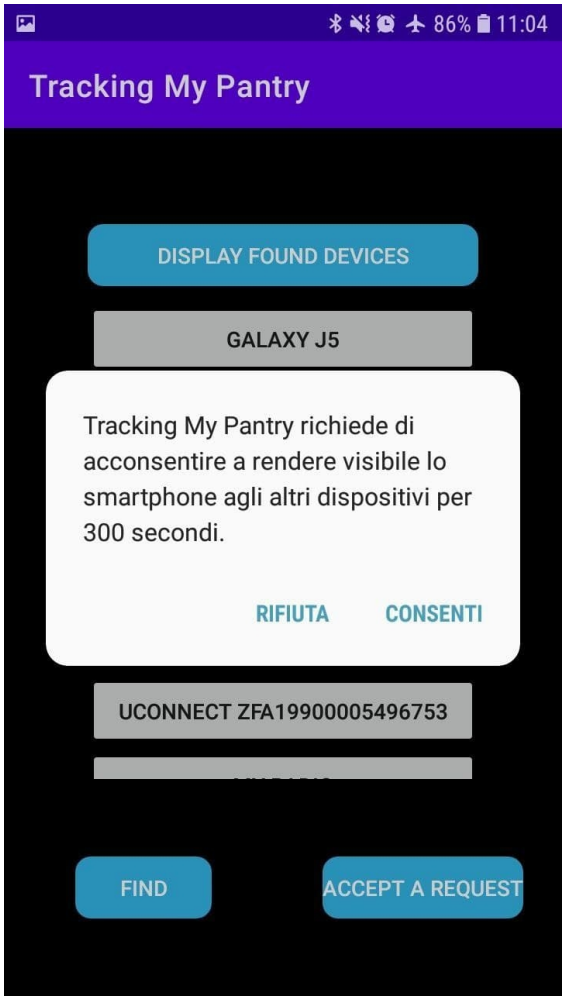
Otherwise, he can click the “Find” button that makes the user choose between two entries: “Find devices”; “Make others to find you”. The former starts a discovery to find available devices, while the latter makes your device available.

One device starts a discovery:



On the other hand, the other device makes itself visible:

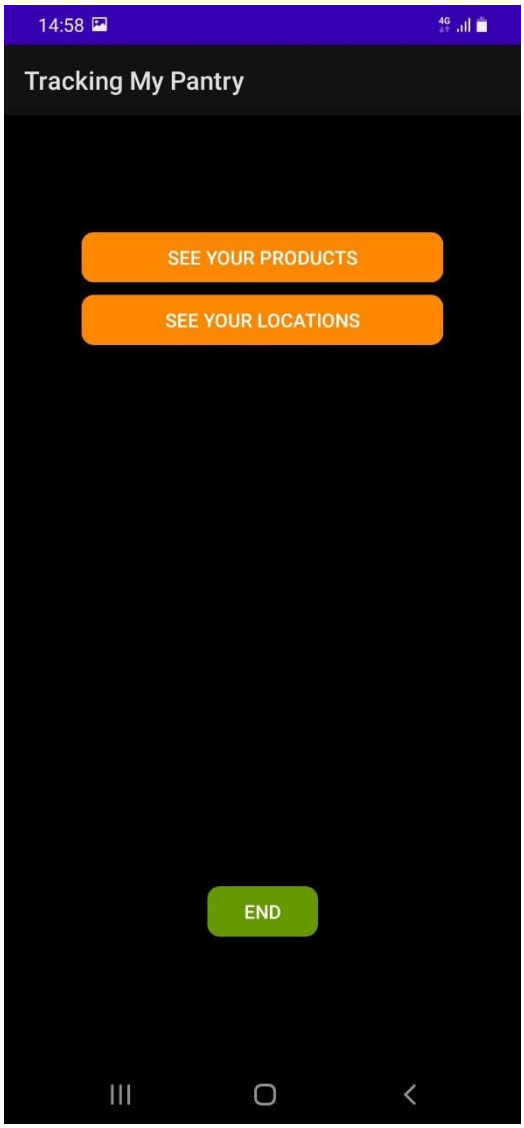




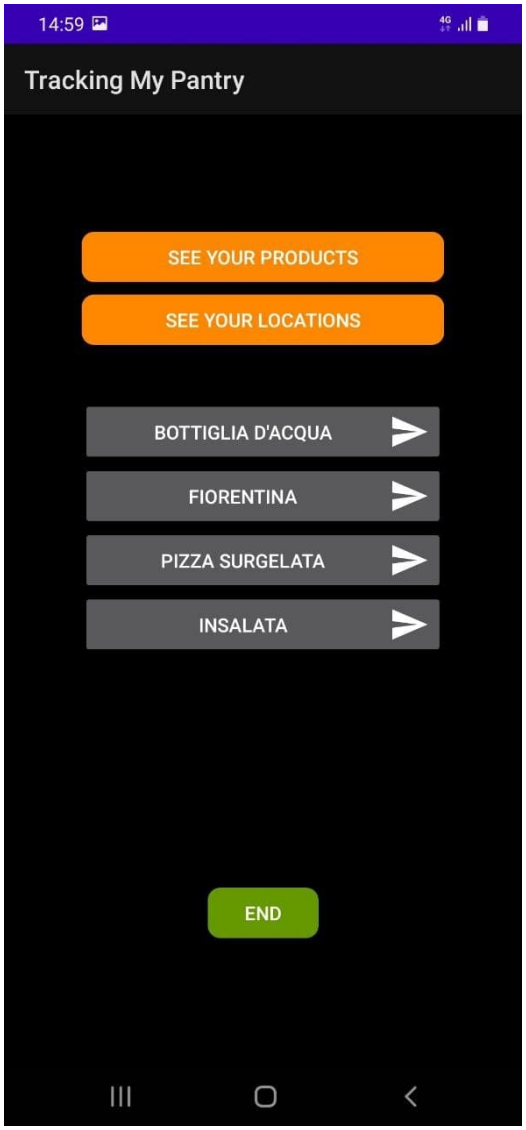
Now, if the user that started the discovery finds the other one, he will see a toast and can click the “Display found/paired devices” to see what discovered. As before, by clicking the device, he will try to initiate a connection. Another button on the screen is the “Accept a request” button that is necessary to accept connection requests. If the user makes his device discoverable, he is already ready to accept requests.

Suggestions exchange

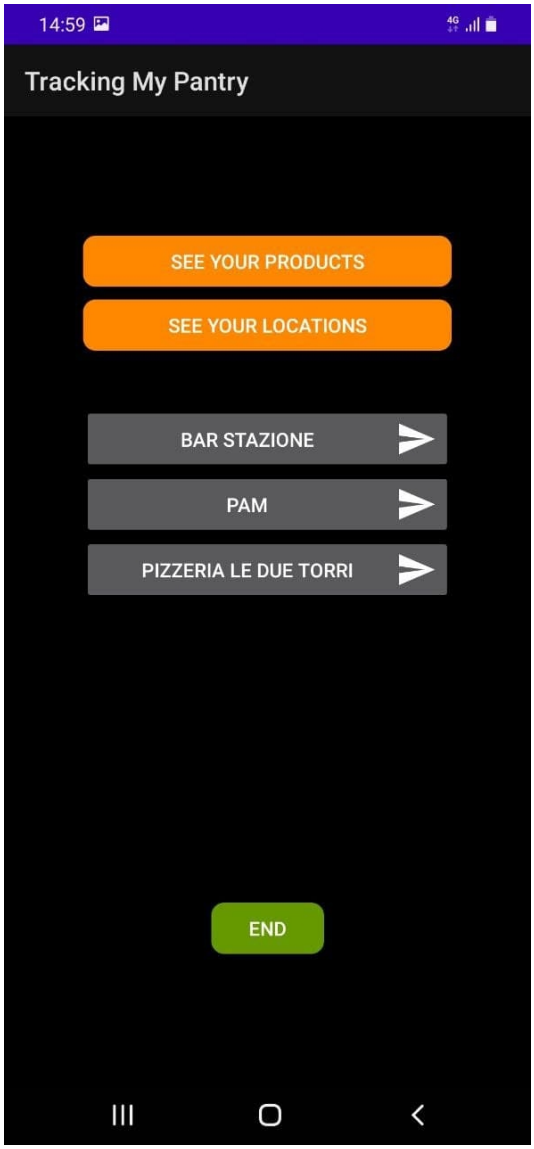
The device that initiate a connection can only send suggestions…



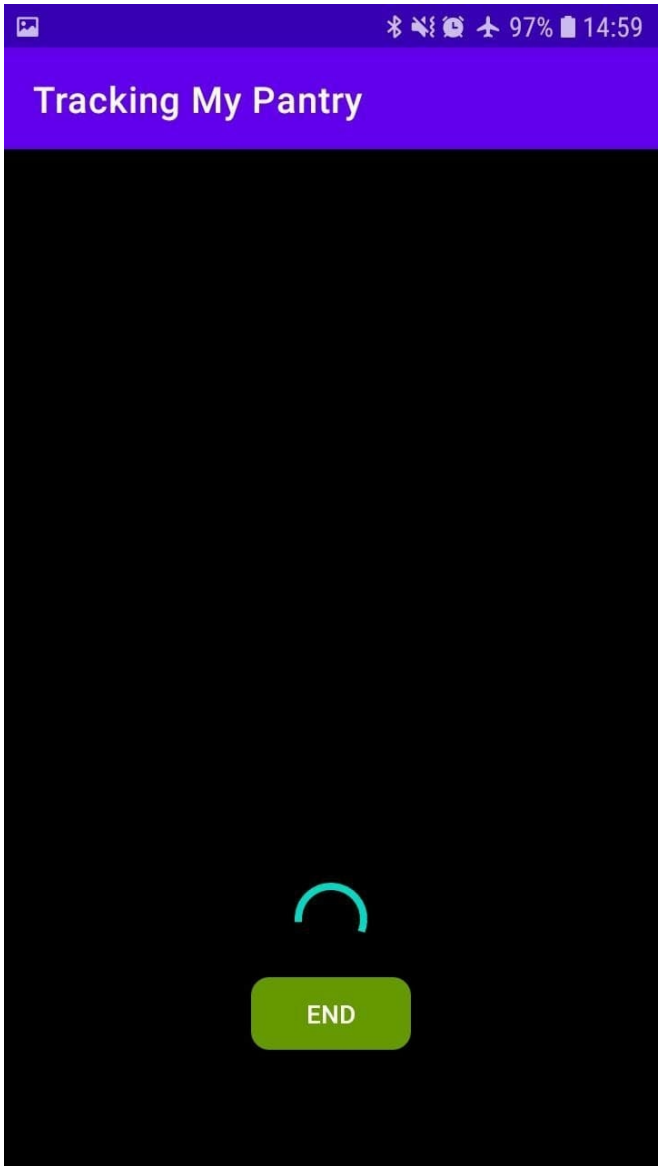
(By clicking “See your products” button)



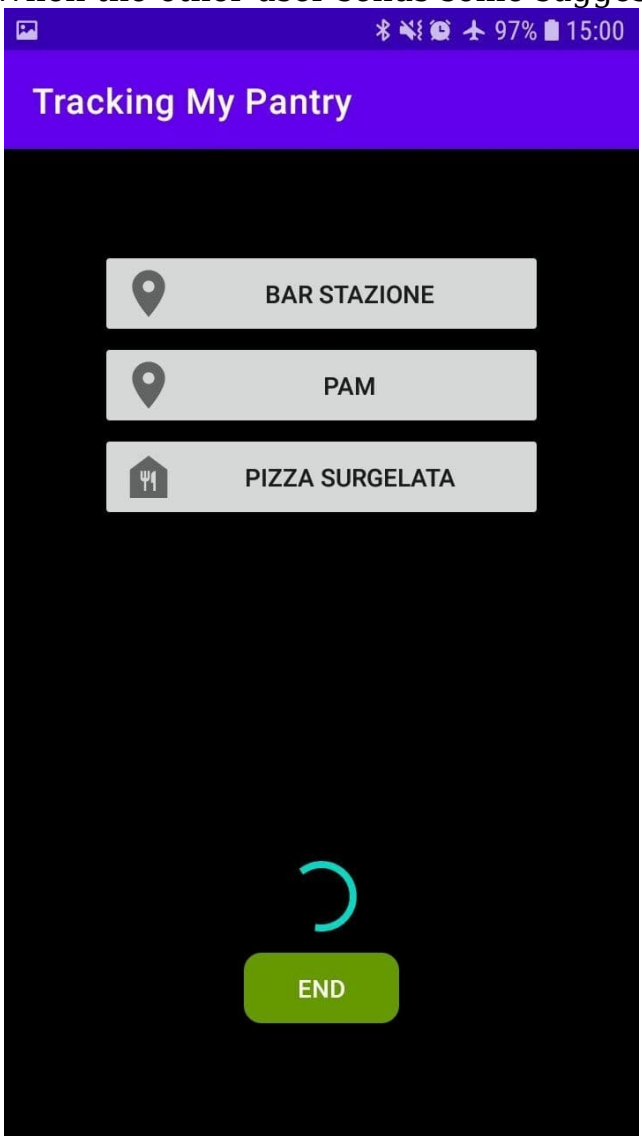
(By clicking “See your locations” button)



…and who accepted a connection request can only receive



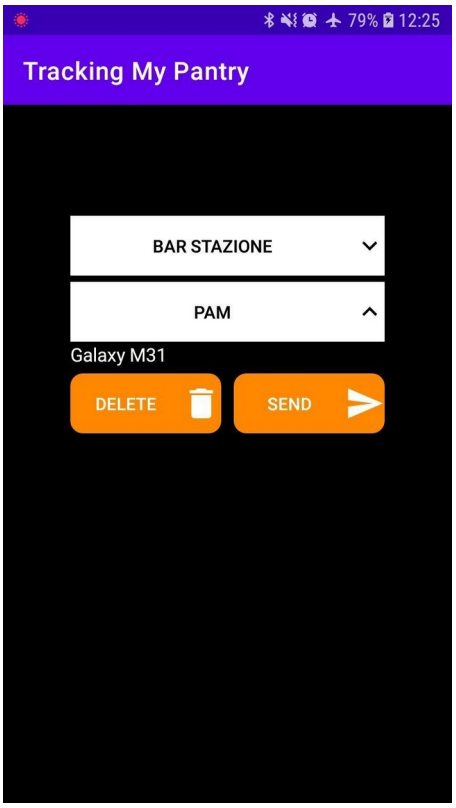
(When the other user sends some suggestions)



By clicking “End” button on both sides, the connection will end.

Managing received suggestions

User can manage the received suggestions and this is done by clicking “See local grocery” button in the homepage and then “See received suggestions” button. Now, the user can select one of the two proposed entries (products or places) and see the suggestions about them.



Note that a suggestion about a place has two buttons, one for deleting (that is self-explainatory) and one for sending the suggestion to the set of stored local places. On the other hand, a suggestion about a product can only be deleted, because a product can be bought only notifying the remote service. A suggestion of that type is useful to the user, because he can search the barcode specified in the suggestion details.

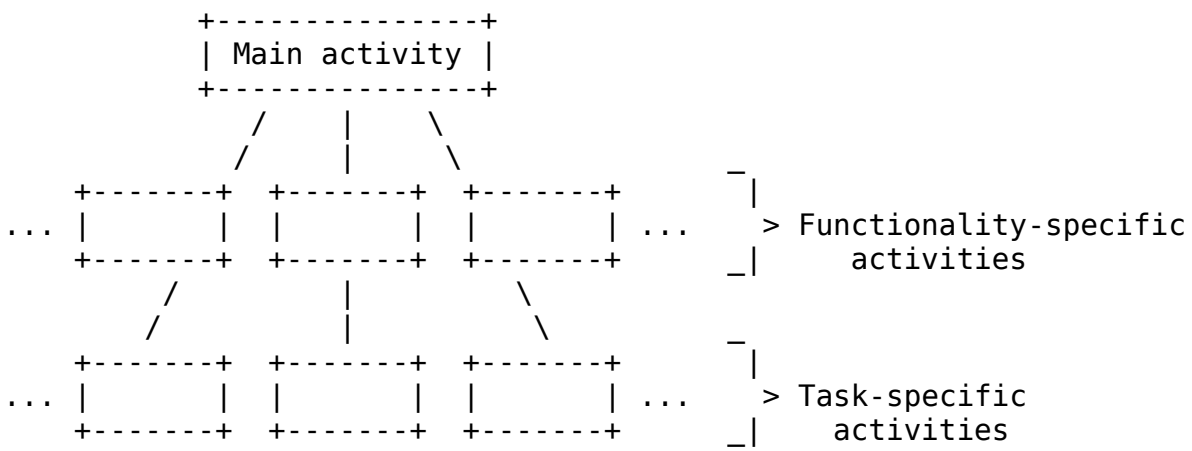
Overall architecture

The project is developed with Kotlin programming language. The `targetSdkVersion` is 31 and the minCompileSdk is 24 (encryption library needs minCompileSdk is 24). In this chapter, I will explain the general concepts and organization of the project, without analyzing the code in the deep.

Activities hierarchy

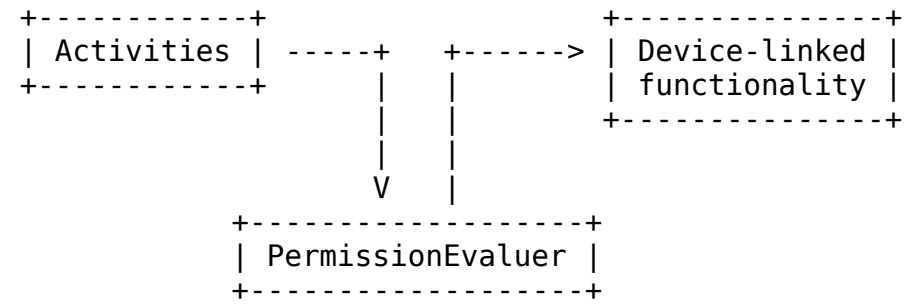
As far as the activities are concerned, I tried to follow nearly always this rule: one activity corresponds to one screen. For instance, the previously quoted “homepage” corresponds to the main activity; from here, the other activities are distributed depending on the functionality they have to implement. Moreover, each of these activities may need to execute specific tasks related to the functionality they implement, or simply, they need to

execute something that continues what they started, but this has to be done in a different screen. Thus, in a certain sense, there is a hierarchical structure of the project:



Permissions

To carry out tasks, activities need of external resources like devices (not in the sense of an entire mobile device, an example is the camera or geolocation), network resources, database, etc.. As far as devices are concerned, activities have to gain permissions to use them, so I wrote a simple utility class (PermissionEvaluer) that only checks if an activity has a specific permission and eventually it can ask for it:

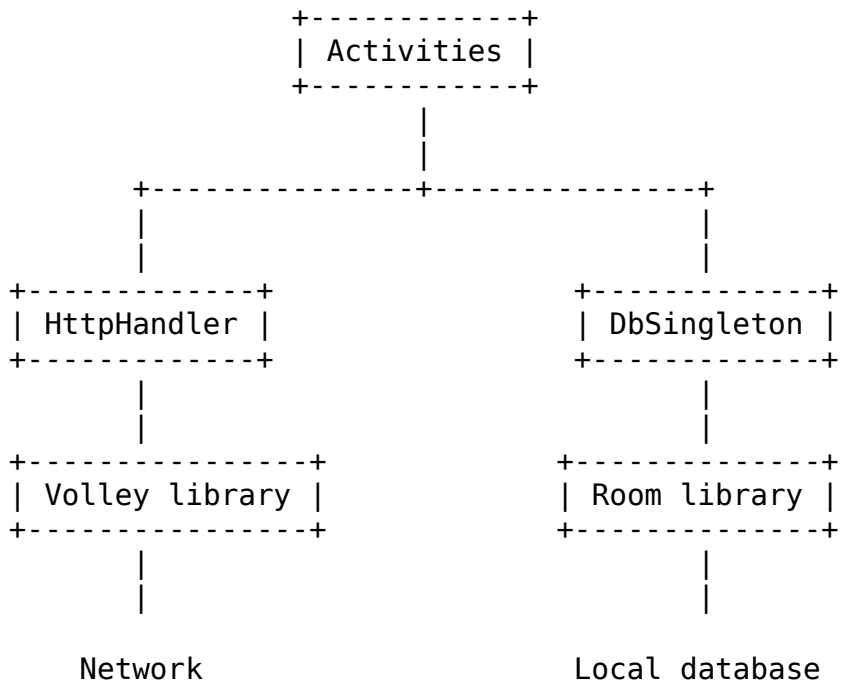


External resources

Moving on resources like network or the database, I decided to make a single endpoint of access and this achieved by particular design pattern that I will explain later. The single endpoints are:

- HttpHandler, for network (in particular, http operations);

- DbSingleton, for the database.



Just as the scheme shows, HttpHandler and DbSingleton does not access directly to the resources, but they uses some libraries (Volley for Http operations and Room to query the local database) to make easier the operations they have to implement.

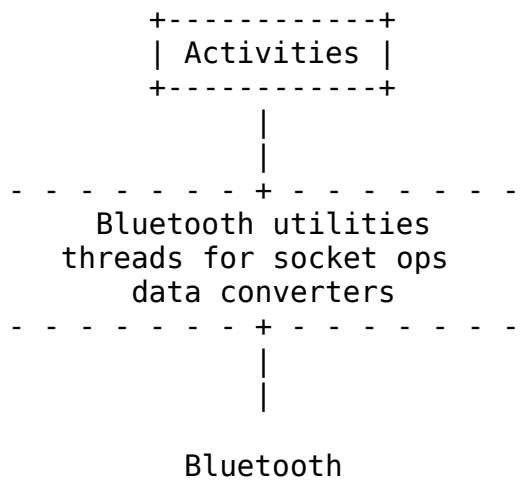
Bluetooth primitives

For bluetooth usage, android framework offers only very raw primitives that operates with sockets, so `connect`, `accept`, `read` (that is `receive`) and `write` (that is `send`). Moreover, it offers API for starting the discovery of devices and for making a device discoverable as well. The data exchange is also really at a low level. In order to face this sitaution, I made a subtle layer to make a little easier the usage of bluetooth API; this layer includes:

- BlueUtils, a module that wraps some useful operations for bluetooth usage (like allocating a broadcast receiver for the case during a discovery, a new bluetooth device is found);

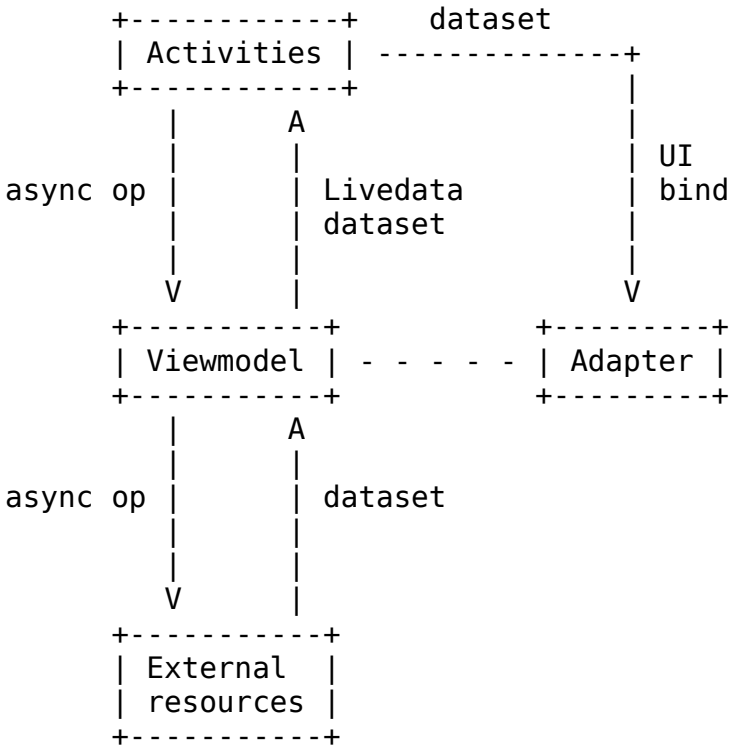
- threads for sockets primitives;

- methods for converting structured data to exchange into byte arrays and viceversa; these methods resides in the Utils module.



UI organization

For each activity, there is an associated layout file (in `res/layout/` package). In order to organize the UI of an activity, I decided to follow a very simple and trivial rule: an activity has to have all and only UI elements that it needs to, not less, not more. Thus, when I was planing the structure of an activity, first I listed all the functionalities it has to implement, then I chose the better UI elements that fulfill my needs and finally I united the parts. Often, I faced a situation where I have to show and manage a dataset that comes from asynchronous executions, usually represented by a list of elements. In order to solve this common problem, I used the Model-View-Viewmodel design pattern (that I'll try to explain nextly) with the `Adapter` class, which is useful to handle the UI part. Almost always, one viewmodel corresponds to one adapter and (almost always) one use case corresponds to one viewmodel. Let's make an example: I had to show to the user the set of products which is returned by an asynchronous operation that makes a http request to the remote service (use case), thus, I made a viewmodel class named `ReceivedItemsViewModel` that handles a livedata that wraps the list of products returned by the network operation (viewmodel) and finally, I created an adapter class called (with a lot of imagination) `ReceivedItemsAdapter` which binds my dataset to the UI (adapter). The general scheme is the following:



This ends the presentation of the overall architecture, in the next sections I will explain the implementation choices and details.

General activities organization

In this section, I show the general organization of an activity in the project. Sometimes there are important differences among activities, however, in general, they all follow the implementation design I’m going to explain.

Layout

Usually, an activity extends `AppCompatActivity` and overrides `onCreate` method and one of the first stuff that executes in this method is the “layout binding”, namely it defines the logic of the UI.

```
class SomethingActivity(): AppCompatActivity() {
    ...

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate()

        ...
        this.setContentView(R.layout.activity_something)

        val view1 = this.findViewById(R.id.someView1)
        val view2 = this.findViewById(R.id.someView2)
        ...
        val viewN = this.findViewById(R.id.someViewN)
        ...
    }

    ...
}
```

On need, activities override other methods of the form `on<event>`, for instance, `LocationsActivity` overrides `onBackPressed`, because it needs of customizing the behaviour of the app when the user click the back command.

Extras

Another common feature of the project activities is the definition of the keys of the extras. That's because, in this way, an activity tells which "parameters" it needs when created (or when other specific events occur). Let's make an example: when the user made a product search through a barcode, the `MainActivity` "starts" the `BuyActivity` that defines the logic of a product search. However, `BuyActivity` needs a barcode to implement its functionality, indeed, it defines a constant that represents the key of the extra that it expects to find: `BARCODE_EXTRA`.

```
BuyActivity(): AppCompatActivity() {
    companion object {
        ...

        const val BARCODE_EXTRA = "barcode"
    }

    ...
}
```

Activity launchers

As I mentioned before, activities “start” other activities. Sometimes, they expect a result back. Thus, they define a launcher that is nothing more something that can be launched with an intent. In particular, the intent must be the one that “starts” the activity of interest. This is all offered by android framework by the `registerActivityForResult` method, so in the activities, it is common to have launchers with custom handlers which manage the result from activities.

Custom result codes

Android defines only two result codes (`RESULT_OK`, `RESULT_CANCELED`), but sometimes, activities need to handle situations with finer “granularity”, so Android `Activity` class offers a constant useful to define new constants (`RESULT_FIRST_USER`) that will so granted not to clash with system symbols. `ResultCode` is the class that defines these constants:

```
class ResultCode {
    companion object {
        //        RESULT_CANCELED
        //        RESULT_OK
        const val NETWORK_ERR = Activity.RESULT_FIRST_USER + 1
        const val EXISTENT_USER = Activity.RESULT_FIRST_USER + 2
        const val EXPIRED_TOKEN = Activity.RESULT_FIRST_USER + 3
        const val INVALID_SESSION_TOKEN = Activity.RESULT_FIRST_USER+4
        const val DENIED_PERMISSIONS = Activity.RESULT_FIRST_USER + 5
        const val DEVICE_ERR = Activity.RESULT_FIRST_USER + 6
    }
}
```

Thereby, handlers in the launchers allocation will be of the form:

```
{ result ->
    when (result.resultCode) {
        RESULT_OK -> {
            // Some code
        }

        RESULT_CANCELED -> {
            // Other code
        }

        ResultCode.<CUSTOM_CODE_1> -> {
            // Other code again
        }

        ...

        ResultCode.<CUSTOM_CODE_n> -> {
            // Some other code
        }
    }
}
```

Http operations

Volley library and Singleton design pattern

To perform http requests, I decided to use Volley library, which allows to execute network (http) operations granting some important facilities and without worrying about low-level details, like threads organization or the scheduling of http requests. As the Android documentation suggests, I use this library along with the Singleton design pattern. Before explaining it, it's right to remember that Volley uses a queue of requests, which are then delivered to sub-threads that process them. Singleton design pattern is foundamental to have a single instance of that queue in the whole app, in order to have a single access point to the http operations, here's the code of `ReqQueueSingleton`:

```
class ReqQueueSingleton(context: Context) {
    companion object {
        @Volatile
        private var queueInstance: ReqQueueSingleton? = null

        fun getInstance(context: Context): ReqQueueSingleton {
            return queueInstance ?: synchronized(this) {
                queueInstance ?: ReqQueueSingleton(context).also {
                    queueInstance = it
                }
```

```
            }
        }
    }

    private val requestQueue =
         Volley.newRequestQueue(context.applicationContext)

    fun <T> addRequest(req: Request<T>) {
        requestQueue.add(req)
    }
}
```

It’s interesting to see that in spite of there is a single queue in the whole app (thereby, multiple threads may have access to it), it is thread safe and this is granted by `synchronized` method, which holds a lock on the object passed as parameter.

HttpHandler

There is also a higher level of abstraction represented by `HttpHandler` class. Its API offers methods to communicate with the remote web service. Such methods need always a context, a success callback and an error callback; moreover, they need specific data to perform the specific request they have to implement. To sum up, `HttpHandler` exposes an interface (not in the sense of Java or Kotlin interfaces) which allows to make the requests to the remote service and nothing more. It’s up to `HttpHandler` the construction of http request, the programmer has not to worry about it. A `HttpHandler` method has the form:

```
fun methodName(
    context: Context,
    ...<data>...,
    successCallback: (<res>) -> Unit,
    errorCallback: (<error>) -> Unit
) {
    ...
    val request = <request_construction>

ReqQueueSingleton.getInstance(context.applicationContext).addRequest(r
equest)
}
```

Credentials handling

This is a sensitive topic and despite it was not requested to implement security features in the project specifications, I decided to add a layer that manages the tokens and the user credentials anyway. The way I handle this type of data is the (encrypted) shared preferences.

Encrypted shared preferences

Shared preferences are just a mechanism that hides the storing of data of the form (key, value) in files. In the case of the project, shared preferences were used along with encryption (`EncryptedSharedPreferences` class). This avoids accidental external reads of sensitive data. `Encrypted` class creates an instance of encrypted `SharedPreferences`. The encryption, as Android documentation shows, works in the following way:

- Keys are encrypted with deterministic algorithms;

- Values are encrypted using AES-256 GCM and are non-deterministic.

TokenHandler

`TokenHandler` class is an utility for access token and session token management. It just exposes methods to save, read and remove access or session token.

CredentialsHandler and retryOnFailure method

`CredentialsHandler` class is an utility to handle user's email and password. When the user logs in, he "receives" an access token back from the remote service (it is saved through `TokenHandler`) and this access token is consumed whenever the user does an operation on the remote service that changes the state of the server, so before the successive operation, he has to re-log in. But this can annoying user experience, if he has to authenticate himself every time. To simulate a "keep-login" semantic, the first time the user logs in successfully, his credentials are saved through `CredentialsHandler`. Moreover, access token is automatically removed when changing-state http request is performed and executed successfully.
`HttpHandler.retryOnFailure` can perform one of the three operations which need of access token, but it first checks if an access token is saved: if it does not exist, it does not try to perform the http request anyway, it would be a waste of energy, so it first executes the operation to log in, fetching credentials from `CredentialsHandler` and finally, if this latter execution is successful, it tries to perform the operation that it had to perform previously. This is the piece of code that does the check:

```
if (!<operation_that_need_access_token>(context, barcode, success,
error)) {
    val email = CredentialsHandler.getEmail(context)
    val password = CredentialsHandler.getPassword(context)

    if (email == null || password == null) {
        Log.e("Unexpected", "Credentials email and password should
have already been set")
        ...
    } else {
```

```
        serviceAuthenticate(
            context,
            email,
            password,
            { _ ->
                if (!<operation_that_need_access_token>(context,
barcode, success, error)) {
                    // Should not happen
                }
            },
            { _, _ ->
                Log.e(
                    "Unexpected",
                    "Authentication should not have problems"
                )
                ...
            }
        )
    }
}
```

Note two important details: the first one is that in the first line, in the `if` guard, it calls one of the methods of `HttpHandler` that need the access token. Their return type is boolean, which tells if the computation has ended successfully (it returns `true`) or, otherwise, it is not executed at all (`false`, this is the case where the access token does not exist anymore). The second important thing to note is that the code of the retry of the operation is passed as the success callback of authentication request (`serviceAuthenticate` call).

Removing credentials

I added one more security feature to “close the circle”. `ClearCredentialsService` is a service that does literally nothing, except when the app is closed by swiping: it removes tokens and credentials. Why using a service and not putting the code to remove sensitive data inside `onDestroy` method of `MainActivity`? The latter would be a wrong solution, because Android does not guarantee that `onDestroy` method of an activity will be called always. Moreover, `onDestroy` is called when the home button is clicked and in that point, I would not remove sensitive data, because maybe the user can come back to use the application. The solution I reached is implementing a service with the lowest priority, started whenever the `MainActivity` is created and overriding method `onTaskRemoved`. In this latter method, I put the code to clear credentials and tokens. Note the following important detail in AndroidManifest.xml:

```
// AndroidManifest.xml

<service
    android:name=".ClearCredentialsService"
    android:enabled="true"
    android:exported="false"
    android:stopWithTask="false" />
```

`stopWithTask` flag set to false allows the service to survive to death of the activity where it is allocated. `onTaskRemoved` will be called when the user swipes the app from recent apps' list.

Local database

Room library and package organization

To implement the local pantry, I decided to use the Room library which gives a high level of abstraction on database operations. The Room documentation shows that there are three main concepts:

- Entity, to create tables;

- Dao, to define operations on entity;

- Room database, to unify the whole.

Thereby, I simply organized the db package respecting these important concepts. Moreover, I added a component , `Converter` class, which allows to store even non-trivial data types (like `Date`, for example) in the local database.

Entities

There are five entities and each of them has an associated dao that defines the possible operations on those entities: - `Item`, which represents a product that the user can buy in the local pantry; - `Place`, which represents a point in a (Google) map; - `Collection`, which represents a set of products gathered by the fact they belong to the same collection; - `ItemSuggestion`, which represents a suggestion about a specific product; - `PlaceSuggestion`, which represents a suggestion about a specific place.

Singleton design pattern

As for http requests, I used Singleton design pattern for the database as well. But this time, there's not a queue for requests, indeed, it is the Room database to be instanciated once and locked when accessed. This leads to have a single thread-safe access point to the database: `DbSingleton`. Moreover, `DbSingleton` act as a sort of wrapper of operations defined by daos, because it exposes them, this is the structure of the code:

```
class DbSingleton(context: Context) {
    companion object {
        // Single instance of db
    }

    ...
    // private fields for db and daos storing

    ...
    // methods of daos
}
```

Kotlin coroutines

The framework needs to run queries to the local database in separate threads from the UI thread (main thread) , because a query run in the main thread may lock the UI for a long period of time annoying user experience, indeed, it is not allowed! To solve this problem, it is necessary to build up concurrent programs. Kotlin offers coroutines, which are nothing more than light-weight processes that run concurrently among them and can suspend their execution in one thread and resume it in another one. The advantage of using a coroutine instead of a thread is that threads are not as light-weight as coroutines, because they are bound to the OS resources, while coroutines are just a language-level abstraction. Moreover, Room runs suspend queries off the main thread. This led as consequence on database environment implementation, that non-returning operations on the local database (defined in daos) are suspending functions, for instance:

```
// In ItemDao

@Insert
suspend fun insertItems(vararg items: Item)
```

Flow objects

I previously mentioned non-returning operations on the db, but what about a query which “returns” some rows of a table? Well, `Flow` class comes to the aid: it wraps a value and it represents a stream of something computed asynchronously. Indeed, daos operations which returns have `Flow<T>` as return type. This is exploited by DbSingleton that turns `Flow<T>` into `LiveData<T>`, which is very easy to handle, especially with MVVM design pattern. Let’s make an example:

```
// In PlaceDao

@Query("select * from Place")
fun getAllPlaces(): Flow<List<Place>>

// In DbSingleton

fun getAllPlaces(): LiveData<List<Place>> {
    return placeDao.getAllPlaces().asLiveData()
}
```

UI implementation

Implementing the UI, I tried to keep it as responsive as possible, following some simple guide lines and using Model-View-Viewmodel design pattern.

Responsive UI

The first problem I faced to keep the UI responsive is the choices about activities layout: each layout xml file associated to an activity has a `ConstraintLayout` as view at the root. `ConstraintLayout` makes much easier the organization of the views inside the layout, because it is sufficient to define constraints among views to place them. Moreover, another common pattern in the project is the usage of `Guidelines` with a particularity: the position of each `Guideline` is not defined with an absolute unit, but using the percentage.

```
<androidx.constraintlayout.widget.Guideline
    android:id="@+id/guideline4"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:orientation="horizontal"
    app:layout_constraintGuide_percent="0.85" />
```

Following this pattern is surely better than dimensions defined with absolute quantities and it assures that most of mobile devices won’t have problems displaying the UI. Moreover, views which are constrained to guidelines often have `0dp` shorthand for one or both dimensions. This is useful to stretch a UI element with a sufficient consistency with the dimensions of screen.

MVVM design pattern

MVVM leads to several benefits: besides it makes a dataset to survive to configuration changes, it makes a clear separation of concerns. The followed guideline is that every time a component was needing to show a certain dataset that comes from external resources (like database), a new

Viewmodel was necessary, indeed, the role of the Model is played by external resources. This is the general scheme of a Viewmodel:

```
class SomethingViewModel(...): ViewModel() {
    ...
    private val something = // async operation
    ...

    fun getSomething(): LiveData<T> {
        return something
    }
}
```

Really simple; note that there are no references on views and that's foundamental not to break the design pattern. Often, there is the following variant:

```
class SomethingViewModel(app: Application, ...): AndroidViewModel(app)
{
    private val context = app.applicationContext
    ...

    // Same as before
}
```

A possible complaint about this version is that the Viewmodel, in this case, is holding a context, which would seem something dangerous! However, this is a particular context, namely it is the application context. Remember that Viewmodel has to survive to configuration changes, like activities changes, but if the entire application dies, everything inside it will die, so it is not something dangerous to keep the application context. On the activities side, they behave as the View in the MVVM, indeed they observe changes coming from the Viewmodel. The mechanism to observe these changes is very simple and this is one of the greatest benefit that MVVM in Android leads: a `ViewModel` instance, as the previous code shows, returns a `LiveData<T>`, that is nothing but a class that wraps data and can be observed, with the add it is bound to a lifecycle (usually an activity lifecyle).

```
class SomethingActivity(): AppCompatActivity() {
    ...

    override fun onCreate(...) {
        ...
        val recyclerView = this.findViewById(...)

        ...
        val model: SomethingViewModel by viewModels {
            SomethingViewModelFactory(...)
        }
```

```
        model.getSomething().observe(this, { it ->
            recyclerView.adapter = SomethingAdapter(it)
        })
    }
}
```

The above code shows how `LiveData` are observed. Furthermore, it shows other interesting implementation details: the Viewmodel (that is called `model` for simplicity in the code, even if it is not Model of MVVM) allocation is delegated to `viewModels` (shortcut for `lazy` function combined with `ViewModelProvider`), which is passed a lambda that takes a `ViewModelFactory` instance object to, that is necessary for allocating the `ViewModel` object; `ViewModel` is used in combination with `RecyclerView` and `RecyclerView.Adapter`, this is explained in the next section.

Adapters and RecyclerView

As mentioned in the **Overall Architecture** chapter, almost always a `ViewModel` comes with an `Adapter`. The latter is the glue component between the dataset and the views. It's foundamental to say that neither `ViewModel` knows about the existence of `Adapter` nor `Adapter` knows about the existence of `ViewModel`. This is important for separation of concerns, it's up to the activity to link together the two parts. Furthermore, adapters behave very well with `RecyclerView`, which is a view that excellently show large dataset, recycling the views it has. This is the typical form of a class extending `RecyclerView.Adapter`:

```
class SomethingAdapter(
    private val callback_1: ...
    ...
    private val callback_n: ...
    ...
    private val elements: Array<Something>
): RecyclerView.Adapter<SomethingAdapter.ViewHolder>() {

    inner class ViewHolder(view: View): RecyclerView.ViewHolder(view)
{
        // Views initialization

        init {
            // Views listeners, using callback_1, ..., callback_n
        }
    }

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
ViewHolder {
        val view = LayoutInflater.from(parent.context).inflate(
            R.layout.something_row,
```

```
            parent,
            false
        )

        return ViewHolder(view)
    }

    override fun onBindViewHolder(holder: ViewHolder, position: Int) {
        // Bindings
    }

    override fun getItemCount(): Int {
        return elements.size
    }
}
```

Hacking RecyclerView

Sometimes, adapters come with a little bit more complicated code. Observe this variant:

```
class SomethingAdapter(...):
RecyclerView.Adapter<SomethingAdapter.ViewHolder>() {
    private var isExpanded = BooleanArray(products.size) { _ ->
false }

    inner class ViewHolder(view: View): RecyclerView.ViewHolder(view)
{
        val nameButton = ...
        val nameExpandedButton = ...

        ...
    }

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
ViewHolder {
        ...
    }

    override fun onBindViewHolder(holder: ViewHolder, position: Int) {
        // Bindings

        if (this.isExpanded[position]) {
            // Changing visibility, nameButton is not visible anymore
        } else {
            // Changing visibility, nameExpandedButton is not visible
anymore
        }

        holder.nameButton.setOnClickListener {
```

```
            this.isExpanded[position] = true
            this.notifyItemChanged(position)
        }

        holder.nameExpandedButton.setOnClickListener {
            this.isExpanded[position] = false
            this.notifyItemChanged(position)
        }
    }

    override fun getItemCount(): Int {
        ...
    }
}
```

This is the case where the activity has to show a series of buttons which, if clicked, a set of views has to appear and when cliked again, the previous set of views has to disappear. This type of logic with `RecyclerView` is a bit difficult to insert, because `RecyclerView` recycle its views, so without the code above, what would happen is that if a button is clicked in a `RecyclerView`, all the buttons “linked” with the recycled view will be affected. The way this event is bypassed consists in storing a boolean array in the `Adapter` to keep track of which buttons are clicked (“is expanded”, because a set of views has to appear, so it gives the illusion of something expanded), then, instead of having one button for each button that appears in the recyclerview, there are two buttons (`nameButton`, `nameExpandedButton`) which are never visible both in the same moment. When one is clicked, it has to become invisible and the other has to appear and viceversa and this achieved with `notifyItemChanged(position)` call, which forces the update of the item at a certain position.

Bluetooth

In the project, bluetooth technology is used to implement a serverless exchange of data (even if the implementation has a server-client architecture, indeed, “serverless” here is meant to an exchange of data among users without the web service as mediator).

Permissions for bluetooth actions

Bluetooth has some delicated points to keep in consideration. One of the most critic is the one about permissions. There are several permissions that can be given and often they depend on the functionality to implement and on the API level. This leads to a more difficult management of permissions requests. Let’s have a look to the set of useful permissions for bluetooth technology declared in `AndroidManifest.xml`:

- `BLUETOOTH` allows applications to connect to paired bluetooth devices;

- BLUETOOTH_ADMIN allows applications to discover and pair bluetooth devices;

- BLUETOOTH_SCAN is required to be able to discover and pair nearby bluetooth devices, it has a dangerous protection level;

- BLUETOOTH_ADVERTISE is required to be able to make a device discoverable to nearby devices, it has a dangerous protection level;

- BLUETOOTH_CONNECT is required to be able to connect to paired bluetooth devices, it has a dangerous protection level;

- ACCESS_COARSE_LOCATION is required to perform discovery of nearby devices, it has a dangerous protection level;

- ACCESS_BACKGROUND_LOCATION is required if the app can run on Android 10 or Android 11 to perform discovery of nearby devices, it has a dangerous protection level and it is necessary the user grants this permission with ALLOW ALL THE TIME entry, otherwise it is not possible to start a device discovery. As far as this latter permission is concerned, if the app is running Android 10 or 11, it will show an AlertDialog to inform the user about how to give this type of permission:

```
private fun showPermissionsInfoAndRequest() {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.Q &&
        !PermissionEvaluator.got(this,
android.Manifest.permission.ACCESS_COARSE_LOCATION)) {

        AlertDialog.Builder(this)
            .setTitle("SENSITIVE PERMISSIONS")
            .setMessage(
                "In order to use this functionality you have to grant
multiple location permissions. " +
                        "By clicking 'OK', screens where you are asked
to grant this type of " +
                        "permissions will be displayed, if you did not
already grant it. " +
                        "It's important that you select the entry
'ALLOW ALL THE TIME' in the last " +
                        "screen. You can navigate the settings and
grant it from there as well"
            )
            .setPositiveButton(R.string.positiveOk,
DialogInterface.OnClickListener() { _, _ ->
                PermissionEvaluator.request(
                    this,

android.Manifest.permission.ACCESS_COARSE_LOCATION,
                    BLUETOOTH_REQUEST_COARSE
                )
```

```
            })
            .show()

    } else if (Build.VERSION.SDK_INT < Build.VERSION_CODES.Q &&
        !PermissionEvaluator.got(this,
android.Manifest.permission.ACCESS_COARSE_LOCATION)) {

        PermissionEvaluator.request(
            this,
            android.Manifest.permission.ACCESS_COARSE_LOCATION,
            BLUETOOTH_REQUEST_COARSE
        )

    } else {
        this.locationCheck()
    }
}
```

Here, `ACCESS_COARSE_LOCATION` is asked before and `ACCESS_BACKGROUND_LOCATION` is only hidden in `locationCheck` call.

BlueUtils

`BlueUtils` is just a help library to make easier bluetooth operations, like getting a bluetooth adapter (that has nothing to do with `RecyclerView.Adapter`) or allocating a `BroadcastReceiver` for when a device is found, after discovery has started or launching a bluetooth enabling request. It just allows to write less code.

Threads for bluetooth primitives and UUID

To make device communicating and exchanging data with bluetooth technology, there is a very low-level facility, namely the sockets. The primitives are the usual `connect`, `accept`, `read` and `write`. But before talking about them, it's important to mention something about the used bluetooth protocol: the protocol is named RFCOMM and it is made on top of other more low-level protocols; furthermore, it offers a reliable data stream, so it could be comparable to TCP for networking (obviously, it's just a parallelism, bluetooth stack and network stack are very different). A RFCOMM channel relies on a code called UUID (Universal Unique Identifier), a standardized 128-bit-long string to uniquely identify information, that is the agreement between clients.

```
// In UUID.kt
private const val UUID_STR_REPRESENTATION = "39ea43fb-b2bb-49d2-bb48-
85aad992611f"
val APP_UUID: UUID by lazy
{ UUID.fromString(UUID_STR_REPRESENTATION) }
```

```
// In ConnectThread
device.createRfcommSocketToServiceRecord(APP_UUID)

// In AcceptThread
adapter.listenUsingRfcommWithServiceRecord(SERVICE_NAME, APP_UUID)
```

Most of socket primitives are blocking calls, so executing them in the UI thread would lock the UI. Therefore, as Android documentation suggests, I implemented threads for the execution of socket primitives. Each thread takes a `Handler` to notify an activity of a certain event, for instance, when it starts accepting or when it sent a bunch of data.

```
private val btInfoHandler = object : Handler(Looper.getMainLooper()) {
    override fun handleMessage(msg: Message) {
        when (msg.what) {
            MessageType.START_CONNECT -> {
                val connectThread = msg.obj as ConnectThread
                Utils.saveValue(BLUETOOTH_CONNECT_THREAD_KEY,
connectThread)
            }

            MessageType.START_ACCEPT -> {
                val acceptThread = msg.obj as AcceptThread
                Utils.saveValue(BLUETOOTH_ACCEPT_THREAD_KEY,
acceptThread)
            }

            MessageType.CONNECTED -> {
                val socket = msg.obj as BluetoothSocket
                Utils.saveValue(BLUETOOTH_CONNECTED_SOCKET_KEY,
socket)

                val intent = Intent(this@BluetoothManagerActivity,
ShareActivity::class.java)
                intent.putExtra(
                    ShareActivity.BLUETOOTH_SOCKET_EXTRA,
                    BLUETOOTH_CONNECTED_SOCKET_KEY
                )
                intent.putExtra(
                    ShareActivity.BLUETOOTH_THREAD_EXTRA,
                    BLUETOOTH_CONNECT_THREAD_KEY
                )

intent.putExtra(ShareActivity.BLUETOOTH_USERNAME_EXTRA,
btAdapter.name)
                this@BluetoothManagerActivity.startActivity(intent)
            }

            MessageType.ACCEPTED -> {
                val socket = msg.obj as BluetoothSocket
```

```
                Utils.saveValue(BLUETOOTH_ACCEPTED_SOCKET_KEY, socket)

                val intent = Intent(this@BluetoothManagerActivity,
AcceptSuggestionsActivity::class.java)
                intent.putExtra(
                    AcceptSuggestionsActivity.BLUETOOTH_SOCKET_EXTRA,
                    BLUETOOTH_ACCEPTED_SOCKET_KEY
                )
                intent.putExtra(
                    AcceptSuggestionsActivity.BLUETOOTH_THREAD_EXTRA,
                    BLUETOOTH_ACCEPT_THREAD_KEY
                )
                this@BluetoothManagerActivity.startActivity(intent)
            }

            MessageType.ERROR_CONNECT -> {
                Utils.toastShow(this@BluetoothManagerActivity, "Could
not connect to this device")
            }

            MessageType.ERROR_ACCEPT -> {
                Utils.toastShow(this@BluetoothManagerActivity, "Could
not connect to other devices")
            }
        }
    }
}
```

The above code is in `BluetoothManagerActivity` class and it is the example of the event handling previously mentioned.

Data conversion

In the app, users can share products and places, but `read` and `write` may only deal with arrays of bytes, thereby it is necessary to have conversion methods, indeed, these methods resides in `Utils` class. The needed methods are:

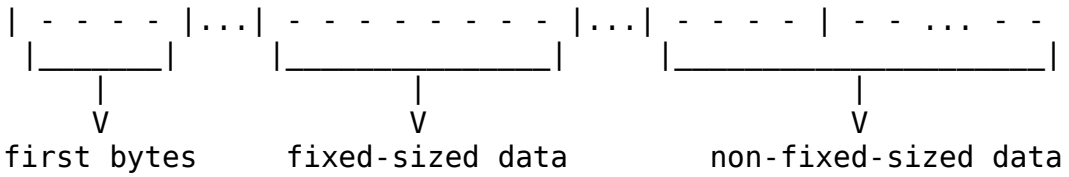
- `Item -> ByteArray`

- `Place -> ByteArray`

- `ByteArray -> ItemSuggestion`

- `ByteArray -> PlaceSuggestion`

There are a few problems: the exchange of non-fixed-sized types of data (like strings); understanding which type of data has been received (`ItemSuggestion` or `PlaceSuggestion`?); the endianess. The general scheme for the first two methods is the following:



The first bytes (in Kotlin should be always four) is an integer to recognize the type of suggestion. Then, fixed-sized data (like doubles) are converted directly to an array of bytes. For non-fixed-sized data, it is not possible to do the same as before, so data is firstly converted into an array of bytes, then the size of the array is taken (it is an integer) and converted into an array of bytes that is concatenated in front of the first array. The second two methods just perform the contrary transformation, knowing the order of data in an array of bytes.

Maps

Google Maps API for Android

For the project, I used Google Maps API (for Android). This API provides a Google Map that the user can interact with, indeed, by clicking on a point of the map, a marker is added to the map.

```
// In LocationsActivity

private fun addMarkedPosition(map: GoogleMap, title: String, location:
LatLng) {
    this.markedLocations.add(
        MarkerAndPosition(
            map.addMarker(
                MarkerOptions()
                    .position(location)
                    .title(title)
            ),
            location.latitude,
            location.longitude
        )
    )
}

...

override fun onMapReady(map: GoogleMap) {
    this.map = map
    map.setOnMapClickListener { location ->
        this.showSetNameDialog(location.latitude, location.longitude,
map)
```

```
    }

    DbSingleton.getInstance(this).getAllPlaces().observe(this,
{ places ->
        places.forEach {
            this.map.addMarker(
                MarkerOptions()
                    .position(LatLng(it.latitude, it.longitude))
                    .title(it.title)
            )
        }
    })

    val myPositionButton: AppCompatButton =
this.findViewById(R.id.myPositionButton)
    myPositionButton.setOnClickListener {
        if (!PermissionEvaluer.got(this,
android.Manifest.permission.ACCESS_FINE_LOCATION)) {
            PermissionEvaluer.request(
                this,
                android.Manifest.permission.ACCESS_FINE_LOCATION,
                LOCATION_REQUEST_FINE
            )
        } else {
            this.getCurrentLocation()
        }
    }
}
```

`LocationsActivity` overrides `onMapReady` method, which is called whenever the map is loaded. The above method `addMarkedPosition` shows how a marker is added on the map. To benefit the Google Map API, it is necessary to generate an API key, adding it to the project and then adding some dependencies.

Getting the current position

Besides clicking the map, there is another way to add markers on the map. By clicking “My Position” button, the current position of the device will be taken, but first `ACCESS_FINE_LOCATION` permission will be asked, that is similar to `ACCESS_COARSE_LOCATION` except that the latter allows to get an approximate position, while the former allows to get a really precise position of the device. Furthermore, a location enabling request will be performed, if location is not already enabled and this is done by directing the user to a screen like the following:

Other core topics

Barcode scanning

Another important functionality of the app is the barcode scanning for fetching barcodes. It is provided by ML kit and camerax library and it is implemented in `BarcodeScannerActivity`. In particular, this activity extends the abstract class `CameraActivity` which is defined like this:

```
abstract class CameraActivity() : AppCompatActivity() {
    protected val PERMISSION_REQUEST_CODE = 1

    override fun onRequestPermissionsResult(
        requestCode: Int,
        permissions: Array<out String>,
        grantResults: IntArray
    ) {
        super.onRequestPermissionsResult(requestCode, permissions,
grantResults)
        when (requestCode) {
            PERMISSION_REQUEST_CODE -> {
                if (grantResults.isNotEmpty() && grantResults[0] ==
PackageManager.PERMISSION_GRANTED) {
                    this.startCamera()
                }
            }
        }
    }

    protected abstract fun startCamera()
}
```

`startCamera` should be the method that defines the logic of the camera usage. The following code is the implementation of `startCamera` in `BarcodeScannerActivity`:

```
override fun startCamera() {
    val cameraProviderFuture = ProcessCameraProvider.getInstance(this)

    cameraProviderFuture.addListener(Runnable {
        val cameraProvider = cameraProviderFuture.get()

        val preview = Preview.Builder().build().also {
            it.setSurfaceProvider(this.viewFinder.surfaceProvider)
        }

        val cameraSelector = CameraSelector.DEFAULT_BACK_CAMERA

        try {
            cameraProvider.unbindAll()
            val imageAnalysis = ImageAnalysis.Builder()
                .build()
                .also {
                    it.setAnalyzer(this.cameraExecutor,
BarcodeAnalyzer({ scanned ->
                        if (scanned != null) {
                            this.barcodes.addAll(scanned)
                        }
                    },
```

```
                    {
                        this.setContentBarcodes()
                    }))
                }
            cameraProvider.bindToLifecycle(this, cameraSelector,
preview, imageAnalysis)
        } catch(exception: Exception) {
            this.setContentBarcodes()
        }
    }, ContextCompat.getMainExecutor(this))
}
```

`ProcessCameraProvider` is used to bind the lifecycle of cameras to the lifecycle of the activity (in this case). Then, the surface of the screen is taken to "project" what is seen by the camera (`Preview`); afterwards, the back camera is selected and an analyzer is set (`BarcodeAnalyzer`).

```
class BarcodeAnalyzer(
    private val successListener: (barcodes: List<Barcode>?) -> Unit,
    private val errListener: () -> Unit
): ImageAnalysis.Analyzer {

    val options = BarcodeScannerOptions.Builder()
        .setBarcodeFormats(Barcode.FORMAT_EAN_13,
Barcode.FORMAT_EAN_8)
        .build()
    private val scanner = BarcodeScanning.getClient()

    @SuppressLint("UnsafeOptInUsageError")
    override fun analyze(imageProxy: ImageProxy) {
        val mediaImage = imageProxy.image

        if (mediaImage != null) {
            val image = InputImage.fromMediaImage(mediaImage,
imageProxy.imageInfo.rotationDegrees)

            scanner.process(image)
                .addOnSuccessListener { barcodes ->
                    this.successListener(barcodes)
                }
                .addOnFailureListener {
                    this.errListener()
                }
                .addOnCompleteListener {
                    imageProxy.close()
                }
        }
    }
}
```

The implementation of `BarcodeAnalyzer` just consists in overriding `analyze` method, processing an image and setting handlers for success and for failure as well (maybe the scanning won’t terminate successfully).