
BUSINESS ACADEMY
AARHUS



Report on
Mobile Application - "ShoppingListApp"
Exam Project

Web Development

Author

Valentin Yordanov

Contents

| | | |
|----------|---|-----------|
| 1 | Introduction | 2 |
| 2 | Main Functionalities | 2 |
| 2.1 | ”RecyclerView” for listing all products | 2 |
| 2.2 | Insert new product | 3 |
| 2.3 | Delete all products | 4 |
| 2.4 | Delete special product | 6 |
| 2.5 | Icon for the application | 7 |
| 2.6 | Share the whole list of products | 8 |
| 2.7 | Implemented Settings Activity | 9 |
| 2.8 | Sorting capabilities | 10 |
| 2.9 | Extra feature | 11 |
| 3 | Interesting code in ”ShoppingListApp” | 12 |
| 4 | Data Model | 13 |
| 5 | Testing | 14 |
| 6 | Monetization | 15 |
| 7 | Conclusion | 15 |
| | References | 16 |

1 Introduction

The content of this report is about an Android application that represents a solution for an exam project. The "ShoppingListApp" is an application for storing data for different products that are stored in a list of products. The main idea behind this project is to give the user an application where he can keep track of the product list that he needs to buy from a specific shop. The app gives the user possibilities like adding a product, deleting a single product, deleting the whole list, and updating an already existing product. In other words, this application gives one solution to the people where they can manage their product list for their next shopping. Each new product that they can add or manage, contains a name, quantity, and name of the shop that the user will buy the specific product. The application is built with all knowledge that was obtained during the second semester in the subject Mobile Development with the guidelines from the teacher.

2 Main Functionalities

2.1 "RecyclerView" for listing all products

In the application "ShoppingListApp" one of the main requirements is that all products have to be listed using "RecyclerView". Using this kind of listing products gives us a very easy[1] and efficient way of displaying data. Based on a layout file that is set for the visualization of a single product, each product is displayed with its specific data. Every product in the application is defined with a view holder object. In the method "onCreateViewHolder()" which is overridden in the "ProductAdapter" class, the layout for displaying every product is set. An example of this layout file can be seen in the figure[1] below.



Figure 1: Displaying single product.

The information that the layout is displaying is the name of the product, the quantity, the shop, and two buttons for deleting and updating the product.

```
override fun onBindViewHolder(holder: ProductAdapter.ViewHolder, position: Int) {  
    holder.itemName.text = products[position].name  
    holder.itemDetail.text = "Quantity: " + products[position].quantity + "\nShop: " + products[position].shop  
}
```

Figure 2: Code where the data is set for each product based on its position.

Figure[2] present the code where the "holder" object from the inner class "ViewHolder" is used to set the two text views inside the layout "itemName" and "itemDetail" with the data from the list of products with the name "products", based on the position of each item. Here as it can be seen the "itemDetail" contains information on two lines. The first displays the quantity and on the next line displays the shop again based on the product position in the list.

2.2 Insert new product

The main idea is of course the end-user to be able to store different products in this application so he doesn't need to take care of a paper where he writes all products for the next shopping. Instead, he needs to insert the necessary data in the application, and in doing so he will know in each moment where is his list of products. In the current version of the application, the user has to insert information for all three fields name, quantity, and shop. If he misses a field, the application will print a Toast message where it will say "You need to fill all 3 fields!". So far the application doesn't check if the product name already exists since the user can insert another shop name where he wants to buy it from.

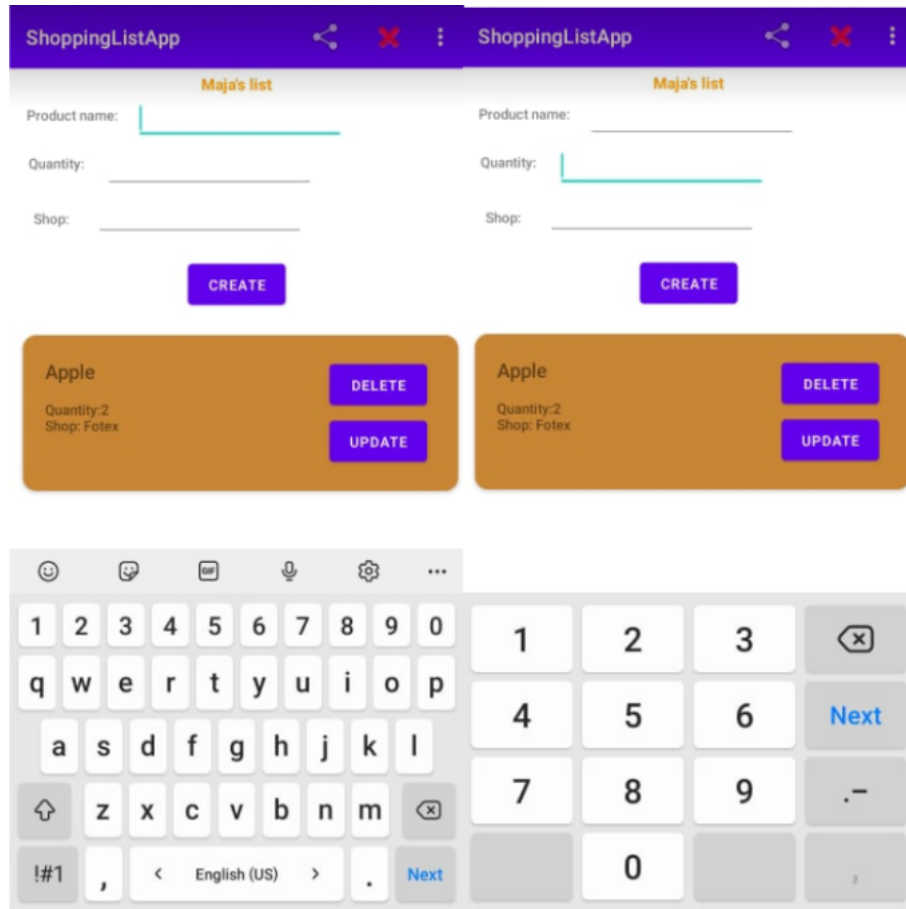


Figure 3: Picture of a possible insertion.

In figure[3], the possible insertion of a new product can be seen. Based on the field that the user will insert a new product, of course, the keyboard is different. For the product name and for the shop the user needs to insert text data but for the quantity field he needs to insert a number.

```

fun addProduct(product: Product) {
    db.collection( collectionPath: "product")
        .add(product)
        .addOnSuccessListener { documentReference ->
            Log.d( tag: "Error", msg: "DocumentSnapshot written with ID: " + documentReference.id)
        }
        .addOnFailureListener { e -> Log.w( tag: "Error", msg: "Error adding document", e) }

    readDataFromFireBase()
}

```

Figure 4: Code for adding new product to the list.

Of course, the main action is happening when the "Create" button is pressed. Since the button is declared inside the "activity_main" layout file, all operations are created inside an "addProduct()" method in the Main Activity. By pressing the button, a "setOnClickListener" is called with the method for adding product. The first thing that the method is doing is checking if all fields are completed. If they are not a Toast message is shown. If they are filled, an object of type Product is created with the data that is inserted. After that, the method calls the "createProduct()" method with the product as a parameter from the "ProductAdapter" class. Inside this method, a repository method is called with the same product object. Finally, in the method "addProduct()" inside the Repository class, an instance of the database is used to add the product which is coming as a parameter and is added inside the collection "product" into the database. The code for this method can be seen in the figure[4] above. To return the newest version of the collection from the database a method "readFromFireBase()" is called which returns all data from the database and loads it inside the application. Of course, this is not the best solution to return the data into the application since if there is a large number of documents to return, it is possible to load them slowly. The better solution for this is basically to add a new element into the local list with all products and load it again.

2.3 Delete all products

One of the functional requirements for the "ShoppingListApp" is to be possible to delete all products from the list. For this functionality, an icon was implemented as an option from the menu inside the app. Inside the "xml" layout file "menu_main.xml" the icon is set to "app:showAsAction="ifRoom"" to appear in the action bar if there is a place since some of the other functionalities like the share button is set to appear always in this panel. So far the icon is shown since all other functionalities are presented when the user presses the menu icon.



Figure 5: Action bar in the application.

As it can be seen in figure[5] the action bar contains the name of the application, the share icon, the icon for deleting all products in the list, and of course the three dots where all other options in the menu are stored.

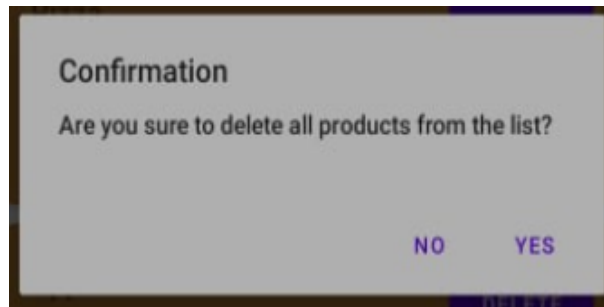


Figure 6: Dialog to prevent or make sure for deleting all products.

Of course, deleting all products can be a very bad idea if the icon is pressed by mistake. That's why when a user presses the icon for deleting the products, a dialog shows up with a message "Are you sure to delete all products from the list?" and the application just gives one more chance to the user if he really wants to delete all products or he just pressed the button by mistake. This prevents deleting huge data by mistake based on the user's interaction. The dialog can be seen in figure[6].

```
fun deleteAllProducts() {
    deleteAllFirebase()
    products.clear()
    productListener.value = products
}

private fun deleteAllFirebase() {
    val batch = db.batch()
    for (product in products) {
        val ref = db.collection(collectionPath: "product").document(product.id)
        batch.delete(ref)
    }
    batch.commit().addOnCompleteListener {}
}
```

Figure 7: Code for deleting all products.

Behind the scene, there are two methods that are doing all the work. The code in figure[7], represents their implementation. The idea of the first method is to delete all products from the local list "products" where all data from the database is saved and present in the application. When the list is cleared then the listener is informed that there is a new value for it. The first thing that this method is doing is to call the second method "deleteAllFirebase()" which deletes all data from the database by traversing each product from the "products" list and based on its id is deleted from the collection "product" in the database. Here the second method is executed before the first method or in other words, firstly the data is deleted from the database then from the local list in the application.

2.4 Delete special product

To give full control over the user's list of products, one of the actions in the application is necessary if the user inserts wrong data or the user to be able to delete a single product from the list. This action is implemented as a button "Delete" into every single product into the "RecyclerView". The button is implemented into the "card_layout.xml" layout file whereas it was mentioned every single product is printed with its data as a single item in the list.



Figure 8: Button to delete single product.

Figure[8] shows how the button looks like in the single product view. Here there is no dialog for making sure if the user wants to delete his product since this deletion is only for one product and won't cost a lot of work for the user if he wants it again.

```
fun deleteProduct(index: Int) {  
    deleteProductFromFirebase(index)  
    products.removeAt(index)  
    productListener.value = products  
}  
  
fun deleteProductFromFirebase(index: Int) {  
    val product = products[index]  
    db.collection(collectionPath: "product").document(product.id).delete().addOnSuccessListener { it: Void!  
        Log.d(tag: "Snapshot", msg: "DocumentSnapshot with id: ${product.id} successfully deleted!")  
    }  
    .addOnFailureListener { e -> Log.w(tag: "Error", msg: "Error deleting document", e) }  
}
```

Figure 9: Code for deleting single product.

Here the solution for deleting product is implemented in the same way as deleting all products. Two methods are implemented where the first one "deleteProduct()" deletes the product from the local list by its index or position in the list. The second method "deleteProductFromFirebase()" is again called from the first one where based on the position in the list takes the whole product object and based on its id deletes it from the database. The button click listener is implemented into the "ProductAdpater" because there is the only place where the position of the desired product is known. The click listener then called the repository method "deleteProduct()" whereas a parameter is sent the position of the index of the product inside the list "products".

2.5 Icon for the application

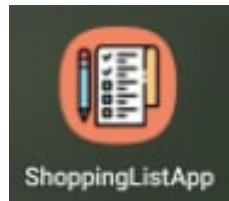


Figure 10: Icon for the application.

To be more discoverable, the application has its launcher icon. The icon was chosen from a web resource[2] where a large gallery of icons was traversed. The new launcher icon was implemented using the option to insert a new Image Asset into the application module. After inserting the new image file inside the window that can be seen in the figure[11] the resize option was used so to set the right size of the image and also the background layer was set to the special orange color. After that, the image was set as an icon into the "AndroidManifest.xml" file within the application tag with the "android:icon="@mipmap/shoppinglist"" line of code.

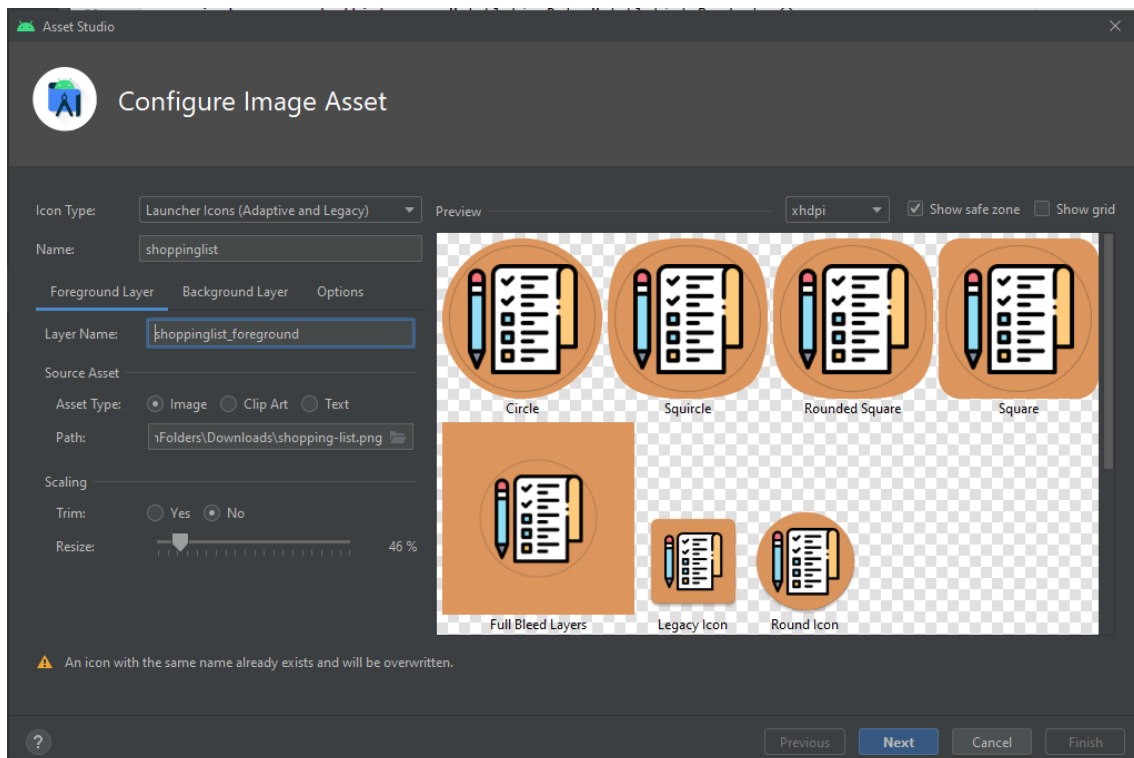


Figure 11: Configure Image Asset window.

2.6 Share the whole list of products

As another important functionality to be implemented was a possible sharing button to be used for sending as a text the whole shopping list. Using that option the user can send his list to another person through other applications that can handle the "text/plain" type.

```
R.id.item_action_share -> {  
    var allProducts = ""  
  
    for (product in Repository.products) {  
        allProducts=allProducts + "${product.name} | ${product.quantity} | ${product.shop} \n"  
    }  
    val intent=Intent()  
    intent.action=Intent.ACTION_SEND  
    intent.putExtra(Intent.EXTRA_TEXT,allProducts)  
    intent.type="text/plain"  
    startActivity(Intent.createChooser(intent,"Share"))  
    return true  
}
```

Figure 12: Code using intents for sharing the list of product.

As it can be seen in figure[12], the implementation of this functionality is set as another option in the menu in the action bar. By pressing the sharing icon, a for loop is used to traverse each product from the repository list "products". For each product object, a new line of text is inserted into a string variable where the name, the quantity, and the shop name are separated by the "|" character. After saving all products and their details in this string, an Intent is used for communication with other apps. The action is "ACTION_SEND" and the string with all products details is set into the method "putExtra()". Then the type is set and all android systems that can handle this type will be listed to the user.

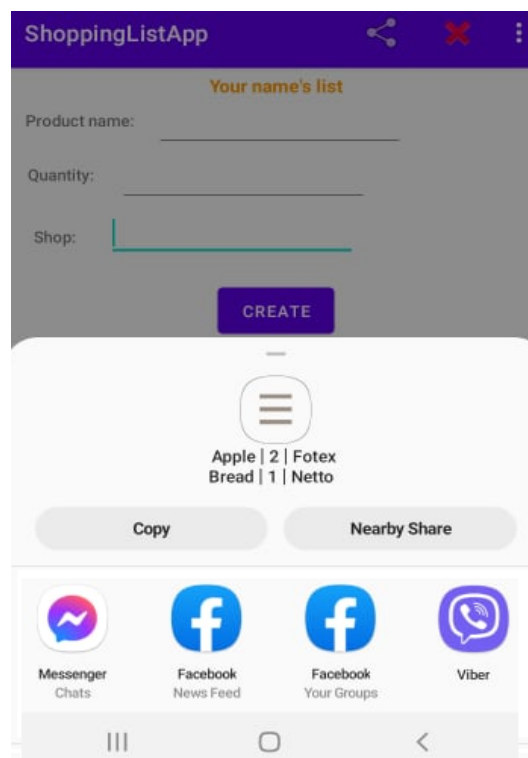


Figure 13: Using Share icon - display.

2.7 Implemented Settings Activity

To make the application more user-friendly and to make it more personal to the person that is using the application, a settings activity was implemented. The user can open the settings window from the menu in the action bar which is listed as the "Settings" option item. In the settings, the user can set his name, and when he sets it and come back to the main activity window he will see a Toast message "Welcome, Name, to your shopping list?". Also, to improve the user-friendly design, over the fields for insertion a new product or on the top of the activity, a title will be set based on the name like "Name's list". Figure[14] represents the main activity after setting the name in the settings activity.

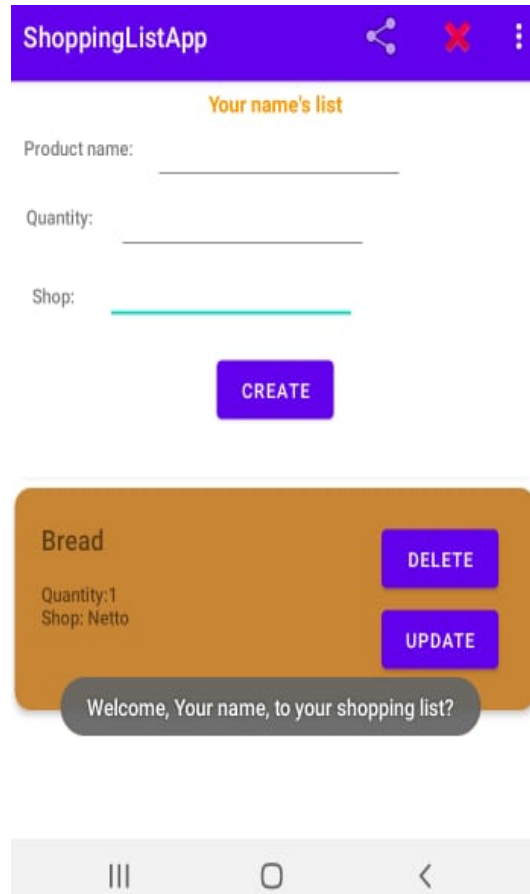


Figure 14: Effect based on the used settings activity.

For making this working, three files are used "MyPreferenceFragment", "PreferenceHandler" and "SettingsActivity". The "MyPreferenceFragment" extends the standard preference fragment and inside this class, a "prefs.xml" file is inflated or loaded as a window in the application. In this window, there is a field where the user can insert his own name. The "SettingsActivity" class is used as the main content in the application when it is started. In this class, there is no declaration of the specific layout file because a "MyPreferenceFragment" is called and it does all the work for showing the right layout file.

```

val name = PreferenceHandler.getName(context: this)
val message = "Welcome, $name, to your shopping list?"
val toast = Toast.makeText(context: this, message, Toast.LENGTH_LONG)
toast.show()
updateUIPref(name)

```

Figure 15: Code that affects the UI after using the settings.

Figure[15] shows, the code that is running after a new name is set into the settings activity. Firstly the data that is inserted for the name is saved into a variable based on a "getName()" method which is called from the "PreferenceHandler" class since it is stored in it. After that, a message of type string is created with the name and it is printed as a Toast message. A function "updateUIPref()" is called which updates the text view with the user name on the top of the list.

2.8 Sorting capabilities

Another requirement that is implemented in the "ShoppingListApp" app is the options for sorting the list of products. So far the user can sort his products based on the names of the products or their quantity. The user can choose the desired options from the menu in the action bar. Figure[16] shows how the products in the list are displayed if the user presses the option from the menu "Sort by quantity".

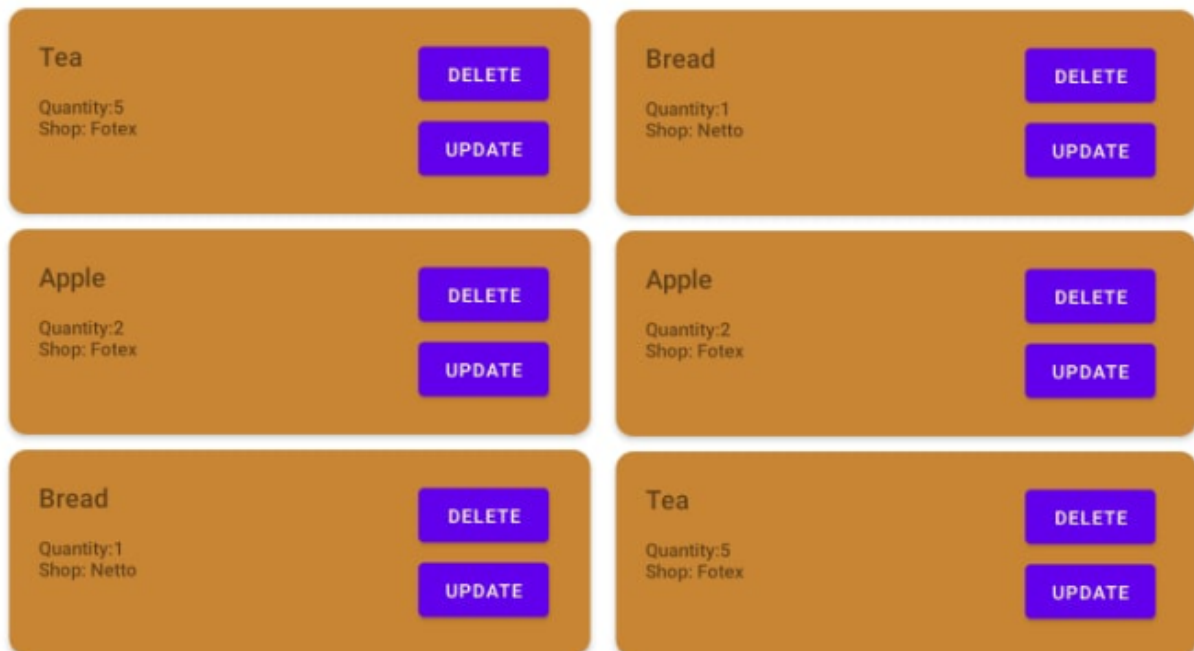


Figure 16: Sorting by quantity.

```

R.id.item_sortByName -> {
    Repository.products.sortBy { it.name }
    adapter.notifyDataSetChanged()
}

R.id.item_sortByQunatity -> {
    Repository.products.sortBy { it.quantity }
    adapter.notifyDataSetChanged()
}

```

Figure 17: Code for sorting the list.

As it can be seen in figure[17], the code implementation for sorting the list by specific criteria is very simple. By pressing one of the two sorting options from the menu, the list in the "Repository" class is sorted by using the "sortBy" function with a parameter which in this case is to sort the products in the list by their name or quantity. After that, the adapter is notified that the data set is changed to display all products with their specific position.

2.9 Extra feature

The extra feature that is implemented is functionality for updating a single product. An application where the user will store his data is an every time good idea to have all CRUD operations since the user can make a mistake when inserting a new product. The functionality is implemented with the "Update" button which is set for every single product in the list like the button "Delete". When the user clicks on the "Update" button a new dialog window is displayed. Inside this dialog, the user can see the current data for the particular product. He can rewrite the data that is wrong or he wants to update and press the "Update" button in that dialog. If he decides to leave the data as it is, he just needs to press the "Cancel" button as it can be seen in figure[18].

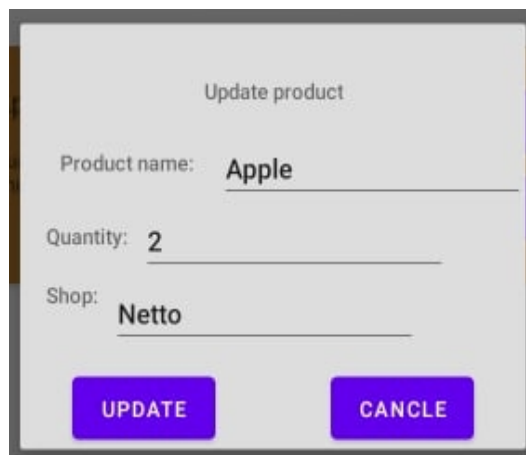


Figure 18: Dialog for updating product.

As it was mentioned, the only place where the position of the products is known is in the "ProductAdapter" class. That's why the click listener for the "Update" button is implemented in the adapter like the "Delete" button. The specification here is that to be able to show the dialog, a callback function is sent in the adapter class as a parameter. This callback function with the name "callbackFunc()" is defined in the main activity class.

3 Interesting code in "ShoppingListApp"

The most interesting code that was for me was based on the extra feature that was implemented in the application. Of course, all new features and techniques that correspond to programming with Kotlin and especially how to implement a specific functionality in Android for me are very interesting but something that I spend a lot of time working on was the callback functions in Kotlin. In the beginning, I faced a problem where calling a function from another activity class was not possible with my current knowledge at that time. Then I decided to set a click listener in the main activity but the button id was not visible since it was defined in another layout file. This problem was faced when I tried to implement the update dialog for a specific product and it was because a dialog fragment class had no information about the product that was about to be updated. After a discussion with the teacher and a hint to use callback functions or to send a function as a parameter into the specific class, the problem was solved.

```
fun callBackfunc(product: Product)
{
    val dialog = UpdateProductDialog(product, ::callUpdateFunc, ::hideKeyboard)
    dialog.show(supportFragmentManager, tag: "UpdateProductDialog")
}

fun callUpdateFunc(product: Product, newName: String, newQuantity: Int, newShop: String)
{
    adapter.updateProduct(product, newName, newQuantity, newShop);
}
```

Figure 19: Code for two callback functions used in the updating product.

Figure[19] contains the code for the two callback functions that are used to update a single product. The first function "callBackfunc()" is sent as a parameter into the adapter class since after pressing the update button for a specific product, the adapter class knows the position of the product and it has to open the dialog with the data for the product. This callback function opens the dialog fragment and also sends as a parameter the product so the fields can show the current information, the "callUpdateFunc()" and the function "hideKeyboard()" which is used to hide the keyboard when the user updates the fields. After the dialog is opened and all updating over the data of the product is done, the user needs to press the "Update" button from the dialog. By pressing the button, the "callUpdateFunc()" function is called with the product as a parameter and also all data from the fields. Inside the adapter class, a "updateProduct()" function is called which passes all parameters to a function with the same name from the Repository class. Inside that method, a database instance is used to find the document with the same id as the product one and to update the document with the product name, quantity, and the shop name that was sent as parameters from the dialog to the Repository class.

4 Data Model

The place where the data is stored from the app is in the Firestore Database. The decision to use Firebase was because of the all useful features provides for the developer. Another feature that was implemented in the application is Crashlytics where every time the application doesn't work or crash at some point, it gives a lot of information for where the problem in the code occurs and even the number of the line where the code crashes. Another very cool feature that was in advantage is that the Firebase supports offline on-device caching and automatic re-synchronization.

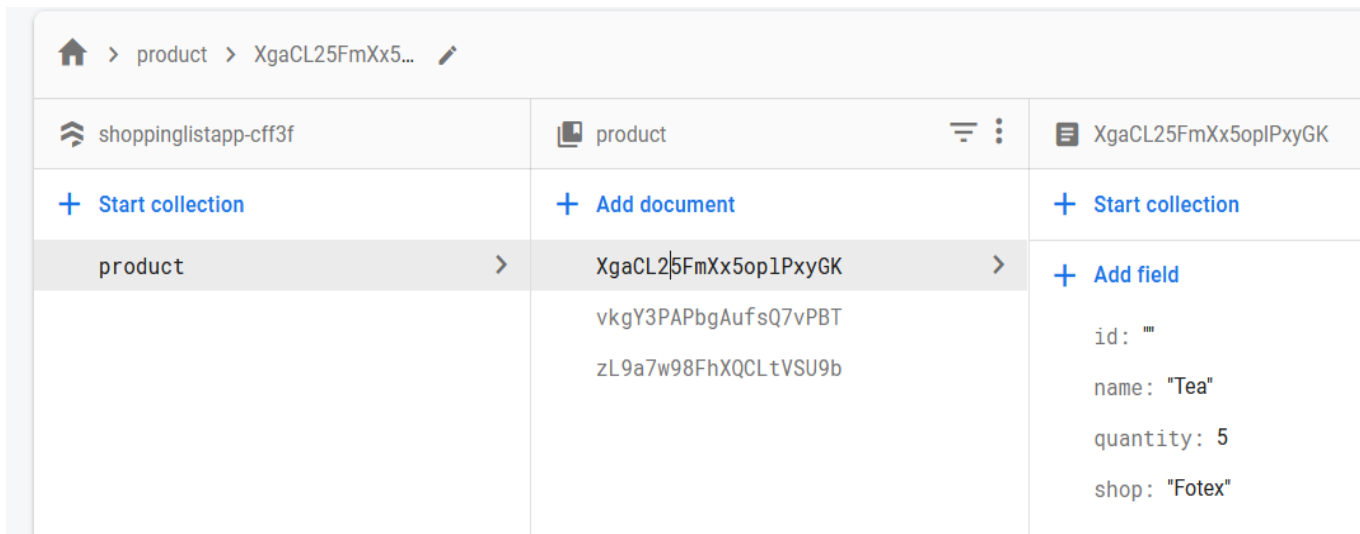


Figure 20: Cloud Firestore for "ShoppingListApp".

In figure[20] a screenshot from the Firebase is shown. As it is visible, the application so far is using one collection which is named "product". This collection contains documents with a specific ID. Each document contains data for a single product or data for its name, quantity, and shop.

5 Testing

When it comes to testing, the application "ShoppingListApp" was tested on two physical devices. The first device is the Samsung Galaxy S7 edge which has Android version 8 or in other words, the application was tested on API Level 26. The second device is a Samsung Galaxy Note 20+ which has Android version 11 or API Level 30. The application was working completely fine within both of the devices. All functionalities and design layouts are running as expected and displayed as expected. As it can be seen in figure[21], the application is running on both of the devices. The only difference is that the device Samsung Galaxy S7 is running on Dark mode.

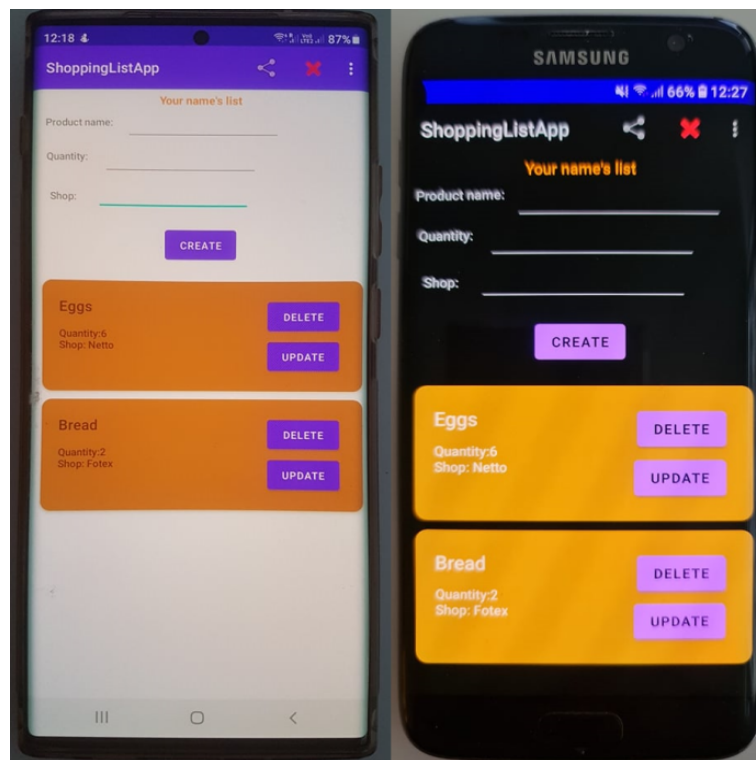


Figure 21: Testing the application using Test Lab.

A feature from Firebase was used in order to check the application functionality. A teacher's advice was to use Test Lab where the system provides automatically testing over the application by simulating real usage. The test was passed successfully as it can be seen in figure[22].

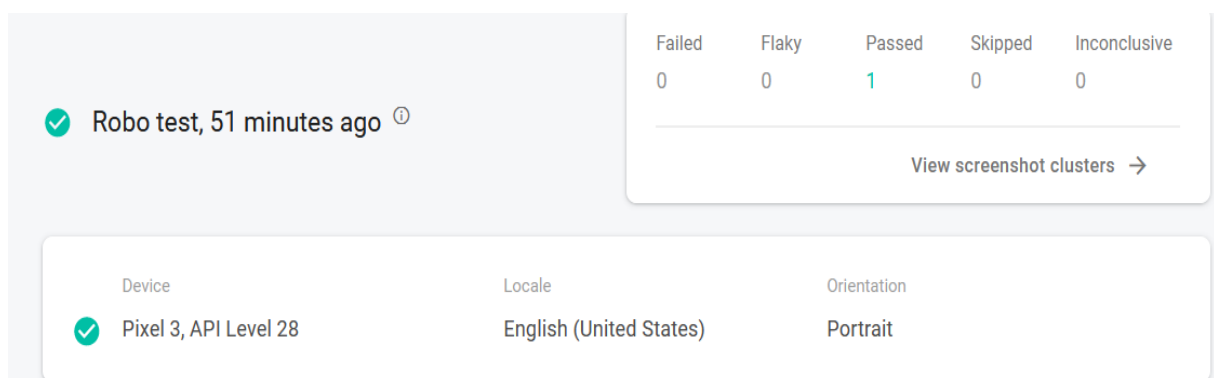


Figure 22: Testing the application using Test Lab.

6 Monetization

To be able to make some kind of profit from an application is very important what kind of the application is and what functionalities has. Based on the "ShoppingListApp" functionalities and main idea of it, as a person who has no previous experience, I would say that the main way to make a profit from this application will be a possible business model like In-App-Billing. The idea behind that is because of the main goal of the app. The main goal is for the user to be able to store data for his shopping list. For now, he can store as much as he needs products in this application. A possible way for me to monetize this application is by implementing the In-App-Billing function where the user will be able to pay for more space to store data. In other words, the user will be able to store ten products in the application for making his list without paying anything. But if the user needs more space he can pay in the app to be able to store more products. A possible solution is when the user pays for example 2DKK to store five more products or in total fifteen products in his list. Possible users that can pay in the app could be big companies or people who manage a food store and they would need more place to store all the products that they need to buy in order to load their store with new products.

7 Conclusion

As a student who is really interesting in the area of Mobile Applications, the whole subject of Mobile Development - Android, the support from the teacher, and the learning process during the second semester in the Web Development program was key factor in my future career. The expectations at the beginning for this final project were to implement as many as possible functionalities since this application is the first application with all CRUD functionalities and external database or in other words the most complicated one in my mobile development experience. With the guidelines and the support from the teacher and the many efforts that were made, at the very end the exam project "ShoppingListApp" mobile application is a fully functional application with a cloud database. If there was more time, a possible authentication functionality would be implemented where the user will be able to make a profile. Of course, this course for me was one of the most interesting ones and the three most important things that I learned for Android development were the Intent System, the Database communication, and the communication between the different fragments in the Android application. Thanks to the teacher and the whole learning process, now I feel more competent and able to develop Mobile Applications.

References

- [1] *RecyclerView*.
<https://developer.android.com/guide/topics/ui/layout/recyclerview>.
- [2] *Icon Resource*.
https://www.flaticon.com/premium-icon/shopping-list_1892603?term=shopping%20list&page=1&position=11&page=1&position=11&related_id=1892603&origin=search.