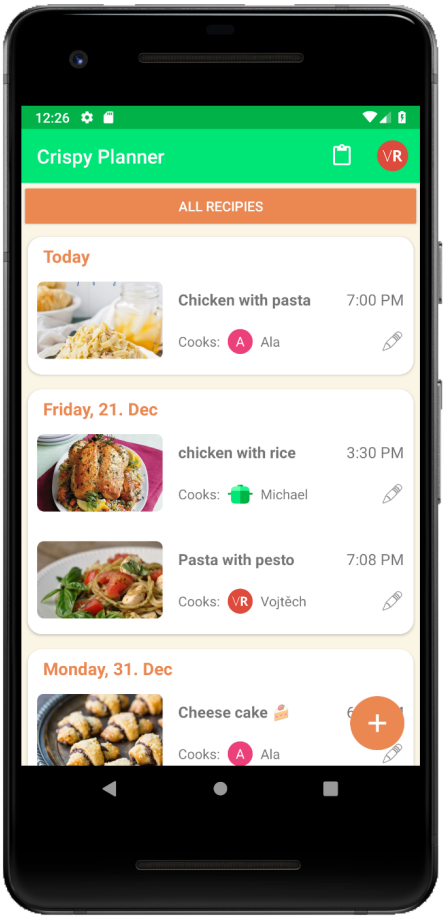
Crispy – Cookbook & Meal Planner

Alicja Kowalewska au609611
Vojtěch Rychnovský au603522
Pawel Belczak au609757

09.12.2018



Contents

1 Introduction

1.1 App vision

The mission of our app is to make storing recipes, planning meals and doing groceries, for the whole family, as easy as possible. An intuitive UI will allow users to quickly pick up a day, choose a recipe, and decide who is going to prepare it with just a few clicks. This app will make deciding about which meal should the family eat the next day – as well as who should take the actual responsibility of preparing it – smooth and efficient. Moreover it will enable users to have all of their families tasty recipes always in their pocket. Another useful feature will be automatically created shopping list that contains all the necessary ingredients for the meals planned for the next few days.

The goal of this project is to make an Android app fulfilling all features described by user stories in Requirements Specification section.

1.2 Personal vision

When we were thinking about an idea for this project, we wanted to create something we could actually use afterwards. We hope that this application will be useful also for other people. We would like to reach production level with this app and luckily deploy it on Google Store by the end of the semester.

By working on this project we want to get better at creating Android applications and using modern technologies such as Architecture Components and cloud solutions like Firebase with its storage and authentication features.

1.3 Context

There are many smartphone applications for storing, browsing and discovering new recipes – both on Android and iOS devices. However, they are not focused on solving the problem of sharing the recipes with other household/family members and deciding who has to prepare e.g. a dinner on the next day.

Our application could fill up this niche providing easy way for your family to store and share the recipes with each other and maintain transparent plan whose responsibility is to prepare next dinner.

2 Requirements Specification

2.1 Functionalities

We represent the desired functionalities with user stories:

1. As a **User** when I first open the app I can:
   (a) register as a new household/family member
   (b) log in

2. As a logged in User when I open the app I would like to manage my **meals/recipes**. I would like to:
   (a) create new recipes including their names, descriptions and required ingredients with quantities
   (b) see all my family/household recipes
   (c) delete some of my recipes
3. As a logged in User when I open the app I would like to manage my **meal plan**. I would like to:
   (a) see all planned meals for the upcoming days with information what the meal is and who is assigned to make this meal
   (b) add a meal to a plan, assign a family/household member to it and choose a date when this plan will happen
   (c) edit a planned meal
   (d) delete a planned meal
4. As a logged in User when I open the app I would like to be able to manage the **grocery list** for next few meals that I've already planned. I would like to:
   (a) see all necessary ingredients with their quantities
   (b) check the products that I already have at home or that I've already bought

2.2 Other requirements

In this project we also need:

- multiple languages support (English and Polish),
- custom app theme,
- custom app icon,
- strings externalization.

3 Design

We tried to use Architecture Components and adhere to the most modern design principles from Google developers. Our application consists of one main activity that hosts many fragments, which we replace, and navigate to, using Android Navigation Component. For backend we chose Firestore Cloud, a NoSQL document database, with which we interact using Firestore SDK. We use Repository

Pattern as our source-of-truth in communication with Firestore. For populating fragments (our Views) we use ViewModels and LiveData. For authentication we opted for Firebase Authentication service. To provide smooth signing user interface we use activity from open-source library FirebaseUI with which we communicate by Intents. Our application uses Intent Service and Alarm Manager for scheduling and displaying notifications. We also use FoodAPI to auto-complete ingredients and get images for created recipes. We are interacting with FoodAPI using Retrofit Library and for downloading the images we use Glide. Throughout the app we use asynchronous processing and callbacks, e.g. in making calls to Firestore or FoodAPI.

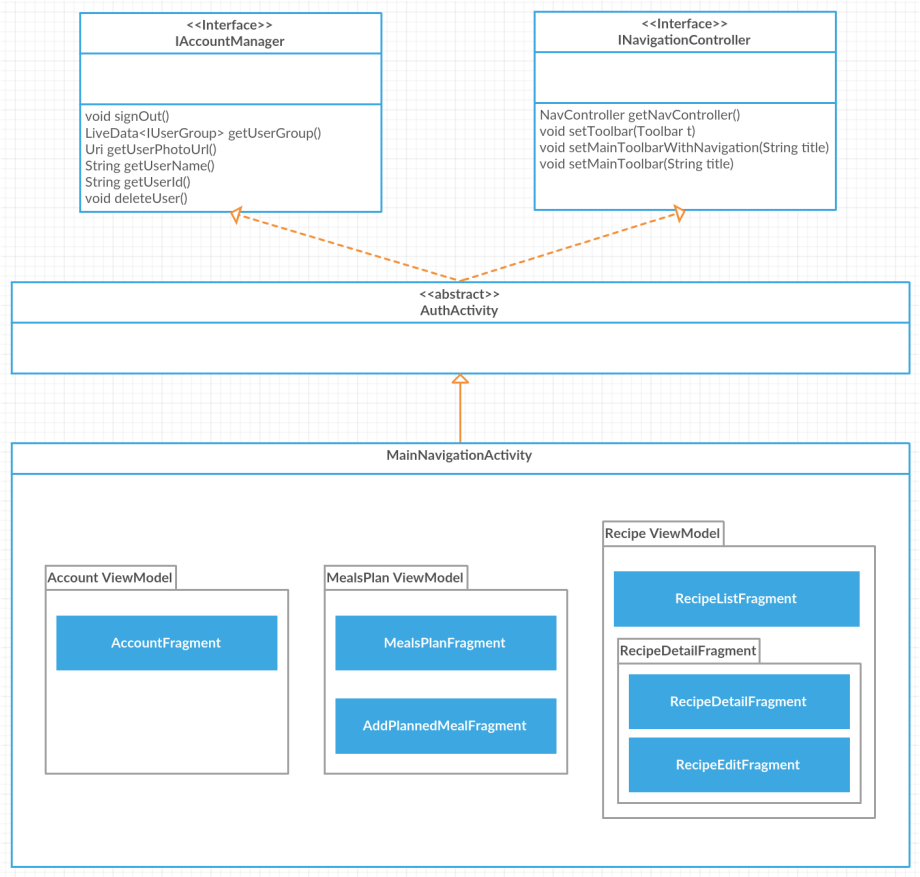


Figure 1: Diagram showing the main Activity encapsulating all fragments (screens) and their corresponding ViewModels.

3.1 Firestore Cloud & Repository

First we started implementing standard relational database using Room library. But only then, when we implemented it, we discovered that the Firestore uses NoSQL document-based database that is not easily compatible with relational model. After changes to our data model we started appreciating benefits and ease-of-use of Firestore Cloud.

Data in the Firestore is divided into collections of documents, each consisting of map of key-value pairs that range from primitive numbers and strings to arrays of other maps of values. Each document can also have its own sub-collections. Our database consists of three collections: groups, recipes and meals.

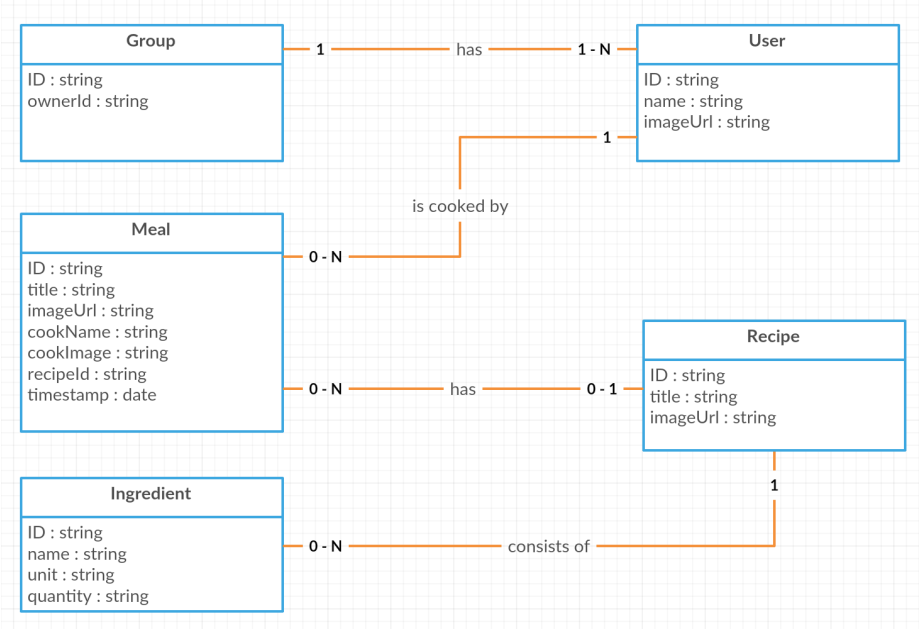


Figure 2: Diagram of entities showing relations between them.

In the groups collection we store documents mapped to UserGroup POJOs. Most importantly they consist of UIDs of users belonging to these groups and are used for authentication. The recipes collection stores data about recipes, such as a title, description and image URL. Each recipe document also have its own *ingredients* sub-collection.

All the Fragments and Activities get data from Firestore Cloud through ViewModels which are connected to the repository. Repository uses our custom FSDocument and FSCollectionLiveData classes which extend LiveData. They use Firestore real-time updates which allow them to *observe* collections, documents and queries in the database and being notified when any of the observed documents/queries changes.

3.2 Authentication

For Authentication we use Firebase Authentication because of its easy integration with Firestore Cloud as well as with popular account providers. As our application runs in the Google-friendly ecosystem we provided ability to sign in with GMail/Google account as well as simple email/password combination. For the authentication signing in/out flow we use an Activity provided by FirebaseUI library. To communicate with this activity and to ensure safety of our application, our MainNavigationActivity inherits from abstract AuthActivity which takes care of checking user status (authorized/unauthorized) and providing data about current user, and her group, for other Fragments and ViewModels. It also starts signing in/out flow by calling StartActivityForResult with specially constructed Intent for FirebaseUI Activity. Then in OnActivitResults() function we check for the status of authentication flow and take different actions depending on the current conditions.

Firebase Authentication also works seamlessly with Firestore Cloud. When the user is authenticated in the app all the requests to the database automatically include authentication headers, which then are checked on the server to allow access to particular resources only to signed in users, e.g. to mitigate risks of malicious users reverse engineering the app and trying to use our API keys to access the database.

3.3 ViewModels and LiveData

For connection Repository layer with individual fragments and Activities, we use ViewModels. There is very heavy usage of LiveData, every values comes from database in this form providing observerable object for fragments. These fragments subsequently observe these LiveData objects and can react on their changes. By this mechanism, there are always almost real-time data on each of our screens. In reality this is clearly visible when one device edit any entity and the other device can show this change in matter of seconds.

Because of these ViewModels are scoped for Activity and there is only one Activity in our app, we can use ViewModels as a way how to communicate between fragments instead of broadcasting and other "traditional" ways. To show concrete examples in our app, when there is a list of meals and user click on one meal, this selection will be stored in ViewModel as a selected meal. After navigation to MealEdit screen using Navigation Components, this MealEdit fragment can get selected meal from VM and show relevant data. We wanted to make this mechanism even better, so we have implemented a combination of Transformations in a form of switchMap functions, for example from Entity ID to Entity itself, also some ViewModel has a Mode, by which we can change Mode of Edit screens to Add screen and back. This Mode is sometimes provided in form of LiveData, so the Edit fragment can observe this value and in combination with SelectedItem it can change it behaviour. When we look back to this approach, it seems a little bit over engineered, but in the end it provides us a possibility to have more fragments next to each other, for

example in tablet layouts during landscape mode.

There are 3 main ViewModels:

- AuthViewModel - providing functions for user registration, login etc.
- MealsPlanViewModel - providing access to Meals from database FSRepository, also encapsulation logic and data of MealsPlan and Add/Edit Meal
- RecipeViewModel - providing access to Recipes from database FSRepository, also encapsulation logic and data of RecipeListFragment, RecipeDetailFragment and Add/Edit Recipe.

3.4 UI Design

Our app supports two different screen sizes. We created layouts both for smartphones and tablets, both of them works in landscape and portrait mode. Each screen is a separate *fragment* or a *composition of fragments*.

We implemented following screens:

- **LOGIN/REGISTER**: This layout is provided by Firebase, we only set there our custom icon. It allows typical login with email address or login with a Google account.
- **PLANNER**: This is the main screen of the application. It is a fragment that includes list (RecycleView) of upcoming meals. They are ordered by date and each day defines a separate card. If the user clicks on a meal it navigates to the detailed recipe screen for that meal (if the recipe exists in the user cookbook). There's also an edit icon, which redirects the user to the edit meal screen. We use there a Floating Action Button that allows to create a new meal. Above the meals we put a button that navigates the user to the list of all recipies.
- **EDIT/ADD MEAL**: This is one fragment that we use both for editing existing planned meal and creating a new one. The only difference is that in *edit mode* the fields are already filled and there's a delete button. In this screen we make use of Android Date and Time Pickers to select the meal time. The field for choosing the recipe is an *AutoCompleteTextView*, where we hint the recipe's name from the user's cookbook. We decided to use it instead of the *Spinner* because the user might have a long list of recipes, and it would be annoying to scroll it. Finally, in this fragment we also have a spinner with all members of the group that allows to select who should prepare the meal.
- **RECIPES**: It is the user's cookbok – the list of all the recipes. This view works differently on smaller and bigger screens. On smartphones it simply shows the list (*RecycleView*) of recipes and after clicking on any item the user is navigated to Recipe Detail fragment. On tablets we show *Recipe Detail fragment* next to the list fragment (side by side).

- **RECIPE DETAILS**: In this fragment we display the recipe name, the table of ingredients and the description. On smaller screens we use *Collapsing Toolbar* with the recipe picture. For readability of the title we added a scrim on the image. There's also an *Floating Action Button* which navigates the user to the edit screen.

- **EDIT/ADD RECIPE**: This fragment works both in *edit* and *add* mode, just like the fragment for planned meals. Each section: title, ingredients and description is built on another card. The field for choosing ingredient name is an *AutoCompleteTextView*, which suggests food names based on the reply from FoodAPI.

- **ACCOUNT**: On this screen we display details of the logged in user. We show the user's group name and all the members of that group. There are also buttons to logout or delete the account.

While designing all layouts we tried to follow *Material Design* best practices. On most of the screens we use card design provided by *Android CardView*. We also use a *Toolbar* that allows for an easy navigation through the entire application. Because the app is implemented with only one main activity it let us keep the same toolbar and just change the fragments, which gives a better user experience.

3.5 Navigation Architecture Components

As mentioned above, we use one main Activity called *MainNavigationActivity* containing Navigation fragment. In this fragment all transitions between screens happens.

Because of using this approach, we don't move between screens using Intents, but using Navigate() and NavigateUp() functions.

All of app fragments (destinations) can be imagined in a tree graph, destinations are represented by nodes in this graph, transition between them are shown as edges.

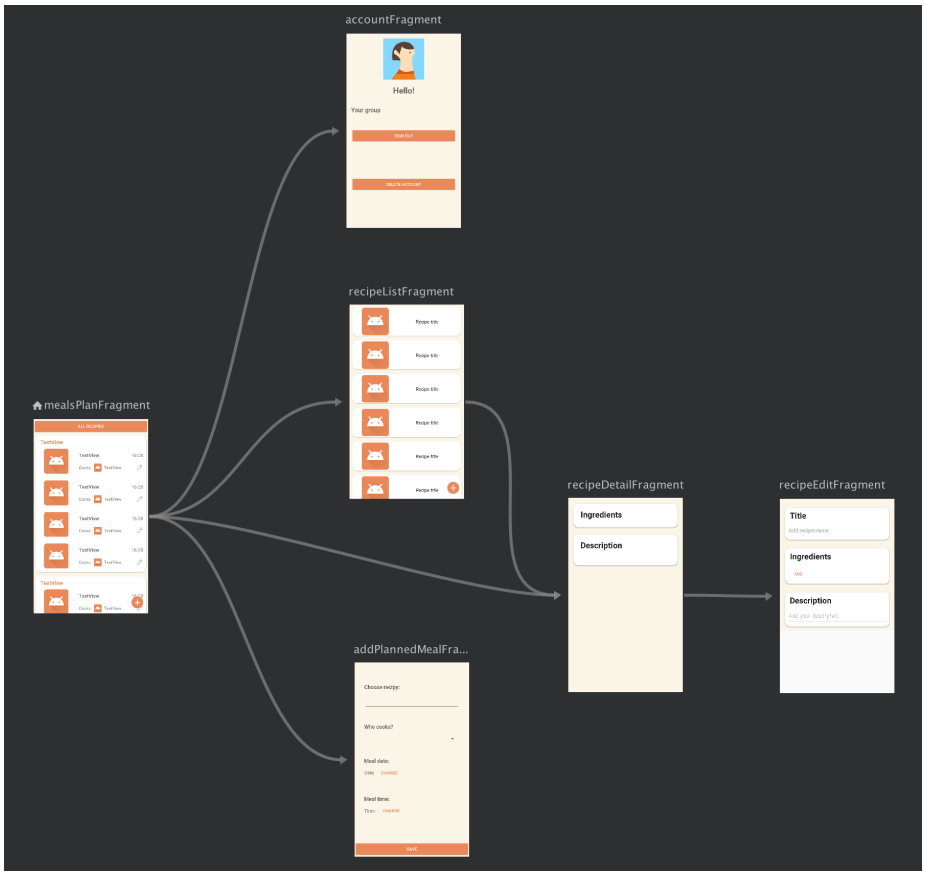


Figure 3: Tree graph of destinations

When we navigate through the app, we build a stack of visited destinations, therefore navigation up (back) is very straightforward, we just pop up the top element in the stack using LiFo strategy. There are two ways how to go up in the app, user can press the back button or up arrow in left top corner. For activity it is important to override behavior of the back button to use Navigation components Navigate Up function, we do this in the MainNavigationActivity.

Navigation Components can be also used for deep links to our app. We don't use this feature right now, but there are certainly some use cases where we can use this in the future, for example going directly to detail of a meal or a recipe.

3.6 FoodAPI & other libraries

One of many features of our application is autocompletion of ingredients. When user is typing the autocompletion helps the user to just tap a ingredient name without writing all of the characters. For that functionality we use FoodAPI from Spoonacular (powered by RapidAPI). The API is RESTful. We use library

called Retrofit to abstract the API ednpoints as Interface. For loading images we use Glide library. Not to block the main UI thread we make Retrofit API calls asynchronously using callbacks to retrieve downlaoded data.

We also autoselect images for recipes added by the user. We search foodAPI for recipes matching the ingredients of new recipe. Than we download the suggested image. It not always work, but majority of time results are good.

3.7 Notifications Intent Service

We use AlarmManager to send a PendingIntent every day at 9am to NotificationIntentService. When this service wakes up, it fetches all meals of ongoing day from Firestore. If there are any meals, it shows a notification to user about his / her today's meals. This notification is clicable, after click user goes directly into our app.

This service is for testing and demonstrating purposes changed to wake up every minute, so we can see and show more notifications.

4 Project work

4.1 Responsibilities

We divided our work for all team members. The main focus of each of us was:

- Alicja - UI design, layouts, fragments, partly ViewModels, custom theme and icon, translations
- Pawel - Firestore repository, Firebase authentication, Web APIs, LiveData, Models Interfaces
- Vojtěch - Navigation Components, ViewModels and LiveData, Notification Service, partly layouts

We worked together as a group and we helped each other a lot so it's not easy to clearly state who implemented which parts.

4.2 Difficulties

While developing this application we decided to use the newest tools available, so we had to learn about *ViewModels*, *LiveData*, *Firebase* and *NavigationComponents*. It took us some time and also caused a lot of code re-factorization. For instance we created a whole Room database just to delete it, when we realized it doesn't work well with *Firebase*. Also some parts of the code are written more complicated than they should be, but this is caused by using new APIs and frameworks. Because of those difficulties we didn't have enough time to test the app properly. We have done few tests and we think that it should generally work, but there might be some small bugs. The main functions of the app works so we consider this project us successful.

The remaining issues are sometimes hard to spot. For example signing in generally works, but deleting or user and signing out may generate a run-time error because of the Smart Lock functionality on Android and because FirebaseUI doesn't adhere to the new one Activity-many Fragments design.

4.3 Future work

For developing this application we had a strictly limited time and we didn't manage to add all the functionalities that we would like to implement. We created a short list of features that could be added:

- **Groups**: Right now all users are in one default group. Adding new groups should be enabled. It'll require implementing users invitations/requests to join a new group.
- **Grocery list**: In the toolbar we created an icon that should take the user to the shopping list screen. To display the list of ingredients required for upcoming meals we'd need to get from the repository all meals with their ingredients and sum the quantity number.
- **Notifications**: More notifications could be added, for example when the user is assigned to a new meal. This requires Google Cloud Messaging.
- **Edit account**: The user should be able to change the nick name and profile picture.
- **Recipe picture**: A dialog should be displayed in which the user can choose whether to use automatically suggested picture or user's own photo.
- **JobScheduler**: JobScheduler can be used instead of AlarmManager to wake up notification service based on internet connection and other conditions, as mentioned above this can be changed with Cloud Messaging.
- **Transitions**: The transitions and animations between fragments could be added.
- **Refactor the code**: When programming an app under time pressure, there are some parts, that should be more consistent, we would like to work on it to make our code more sustainable.

References

[1] `https://material.io/develop/android/components/material-card-view/`

[2] `https://material.io/design/components/lists.html#`

[3] `https://android.jlelse.eu/parallax-scrolling-header-tabs-android-tutorial-2cc6e40aa257`

[4] https://developer.android.com/guide/components/fragments

[5] https://developer.android.com/topic/libraries/architecture/livedata

[6] App Icon made byhttps://www.flaticon.com/authors/freepik from www.flaticon.com

[7] Firestore SDK documentation and Android Developers guides https://firebase.google.com/docs/firestore/

[8] FirebaseUI library https://github.com/firebase/FirebaseUI-Android/blob/master/auth/README.md

More references are listed directly in the code.