QDOCS

Software Design Document

DESIGN AND IMPLEMENTATION OF MOBILE APPLICATION

Academic Year 2018/2019

Made by:
Lamparelli Andrea, Chittò Pietro

Contents

Chapter 1

Introduction

1.1 Purpose

The purpose of the Software Design Document (SDD) is to provide a description of the design of a system fully enough to allow for software development to proceed with an understanding of what is to be built and how it is expected to be built. The Software Design Document provides information necessary to provide description of the details for the software and system to be built.

It is a technical document that exposes techniques, decisions, implementations, components and the architecture that we decided to adopt in the *QDocs* application system development. In this introduction chapter we provide a description of the application and for what it was thought to be used, some definition and the structure of the whole document.

The technical aspects of the *QDocs* implementation are addressed starting from the section Architectural Design

Figure 1.1: *QDocs* launcher icon

1.2 Scope

The *QDocs* app (figure 1.1) is a file-storage mobile application which help users to organize, visualize and, above all, find in a faster way their personal stored files. In fact, there are many files that users use rarely and after some time they no longer remember where they were placed. *QDocs* allows user to basically associate a stored file to a specific and unique QR-code that can be printed and placed in a site of the "real world" that is related to that file. When the QR-code is scanned, the related file is instantly shown on the smartphone. In order to show how this application can be used in real word we provide some usage examples:

- A student takes notes on his notebook and he wants to extend his notes with some digital articles or some digital book pages. He can upload the articles on his *QDocs* storage and print the generated QR-code on the personal notebook.

When he must study and want to read the articles, he can easily scan the QR-code with the application scanner and immediately the file appears on the smartphone.

- Since nowadays more and more electronic tools are sold with digital instruction booklet, *QDocs* can become very useful for keeping associate the tool with its instruction booklet. A simple example can be the following, suppose to buy a camera and suppose that after long time you don't remember what the functionality of a specific button is, very often you don't remember where the booklet was stored. So to overcome this problem you can store the instruction booklet on the *QDocs* storage and print its QR-code on the camera and then whenever you have to access this document you can easily scan its QR-code (that you know is placed on the camera) and instantly read the document.

Obviously, the application can be used as any other cloud-storage application like Dropbox, Google Drive and so on since it allows users to manage their own files (e.g. uploading and/or deleting files), you can also create directories for a clearer storage procedure. The representation of the images is directly managed by the application itself, without using external application, the same for the audio. You can also save your online files on the internal storage of the device such that whenever you have to open that file you have not to download it each time. In order to use the application, you must register to it and create your own account, all the accounts are independent, hence they are separated such that someone cannot access other user's files.

1.3 Definition, Acronyms and Abbreviations

- ***QDocs* :** Mobile Application
- **Cloud:** The term is generally used to describe data centers available to many users over the Internet.
- **Client/User:**
- **Firebase:** Mobile development platform sponsored by Google [2]
- **Internal Storage:** Local storage of the current device that is using the application
- **Java:** Object-Oriented Programming Language
- **NoSQL:** Non-relational database which provides a mechanism for storage and retrieval of data that is modeled in means other than the tabular relations used in relational databases.
- **JSON:** JavaScript Object Notation, is an open-standard file format that uses human-readable text to transmit data objects consisting of attribute-value pairs and array data types.
- **SDD:** Software Design Document
- **API:** Application Programming Interface, it is a set of clearly defined methods of communication among various components.

- **OS:** Operating System
- **DB:** Database, it is an organized collection of data.
- **UML:** Unified Modeling Language [16]
- **FMVC:** Framework-Model-View-Controller paradigm, which is an MVC's variation
- **GUI/UI:** Graphic User Interface / User Interface
- **XML:** eXtensible Markup Language, is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable.

1.4 Overview

The Software Design Document is composed by 8 chapters with various sections and subsections. Each chapter emphasize a specific aspect of the *QDocs* system, all chapters are the following:

1. **Introduction:** In this part we basically introduce the application itself specifying, which is its goal, and providing some simple, real-world, usage examples.
2. **Use Cases:** In this section we provide user's interaction with the system that shows the relationship between the user and the different use cases in which the user is involved, identifying different actors that are involved and all use cases (operations) that each actor can perform.
3. **Architectural Design:** The goal of this chapter is to provide and discuss the technical aspects adopted for the application development, it analyses the architecture of the whole system and then it focuses the attention on the client-side (mobile app) implementation.
4. **User Interface Design:** In this part we will provide a less technical analysis about the mobile application, we will focus the attention on the graphical interfaces of it discussing the life-cycle of the application from the point of view of the external interfaces.
5. **Requirements Traceability:** This chapter provides the list of requirements that this application was thought to satisfy providing also which are the specific components that are in charge to satisfy them.
6. **Implementation, Integration and Test Plan:** In this part we focus the attention on the testing part of the development, analysing how the tests were thought and performed.
7. **Tools Used:** This is the last section of the document where we provide the tools used for the application system and SDD development.
8. **Bibliography** In this section there are all external links that we have used during this project, considering both the implementation and this documentation.

Chapter 2

Use Cases

This section provides user's interaction with the system that shows the relationship between the user itself and the different use cases in which the user is involved.

2.1 Actors

In the *QDocs* system there are basically four main actors:

- **Unregistered User:** this generally represents the user who interact with this system for the first time. There are no accounts that are associated to this user, for this reason the only thing that he can do is to register to *QDocs* system. (figure 2.1).
- **Registered and Unsigned-In User:** this represent the user who has been already registered into this system but that has not been signed-in yet. Another similar situation is when a user tries to access the application from a different device where he has not been signed-in yet. (figure 2.2).
- **Signed-In User:** this represent the user who has already been signed-in in the system and who can interact with system's functionality for it was intended to be used. (figure 2.3).
- **External Application:** this actor represents a generic external application that can be used for accomplish some operation (e.g Facebook, Google etc.).

2.2 Use Case Diagrams

In this subsection you can find all use cases diagrams, which are split into three main diagrams, where each of one focuses on a specific kind of user-actor.

As you can notice in figure 2.1, since the system requires an authentication, the user who has not already been registered yet cannot directly interact with the system exposed functionality, but he can only register into the system or leave the application.

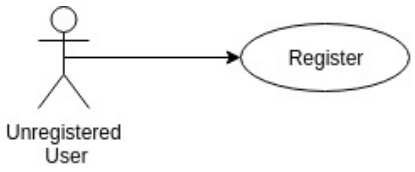

Figure 2.1: Use case for a user who has not been registered yet

In case in which the user has been already registered but not signed-in, as before since the system requires an authentication, he can only login into the system or leave the application.

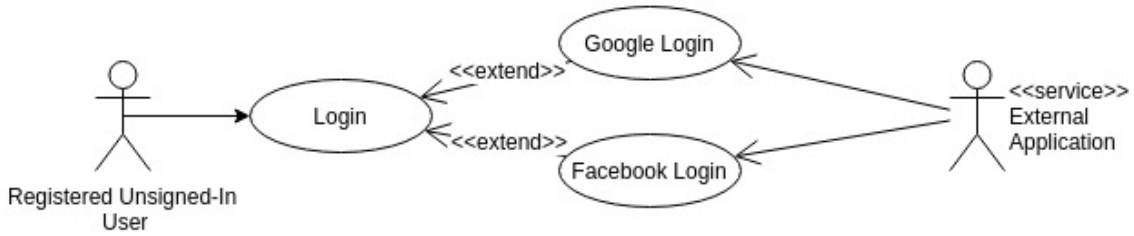

Figure 2.2: Use case for a registered user who has not been signed-in yet

Here, instead, the user can do anything that the system offer, obviously following its directive. In this scenario the user has the highest level, obviously he can downgrade himself executing the logout operation or he can do all other exposed operation, such as upload, download, file scan and so on. The use cases are quite self-explanatory and the more important will be deeper analysed in Dynamic Model section.

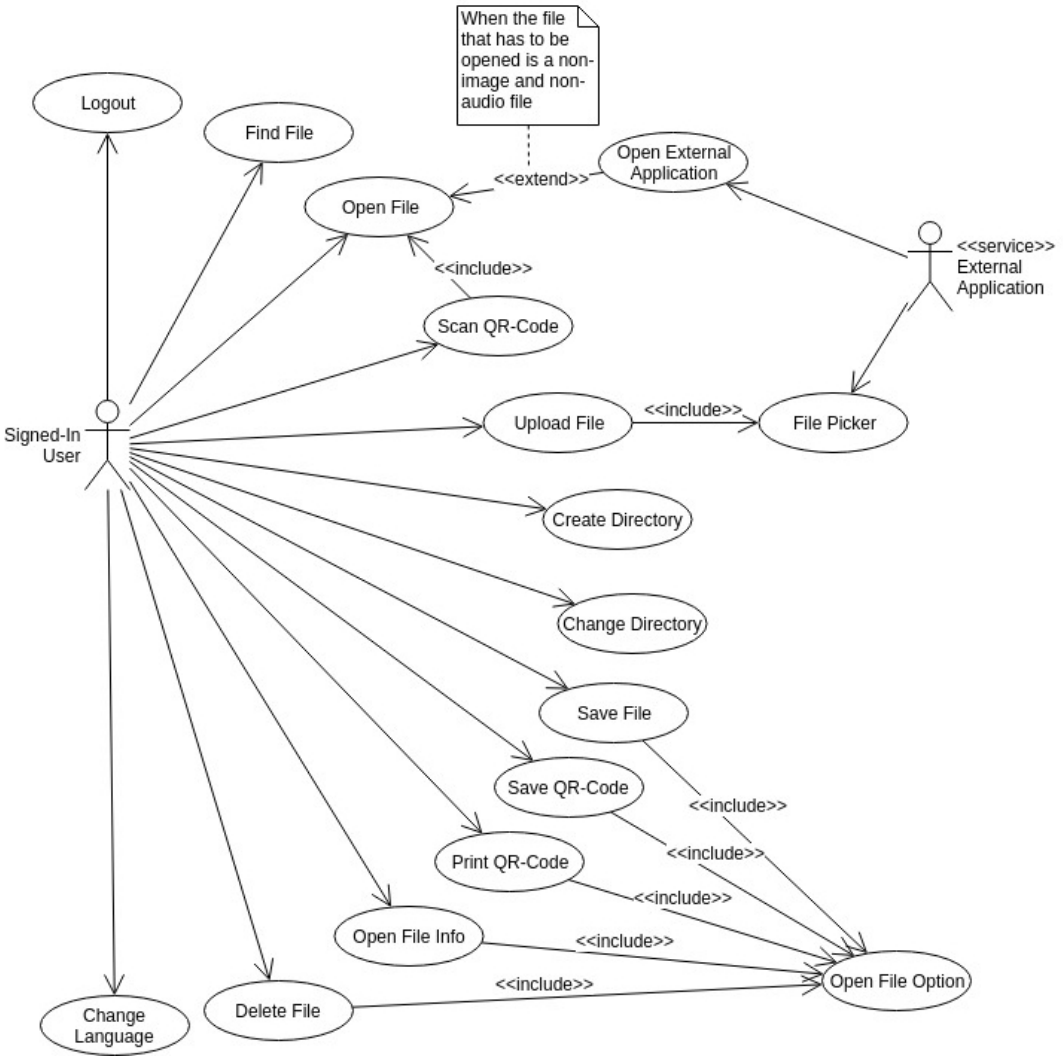

Figure 2.3: Use case for a signed-in user

Chapter 3

Architectural Design

3.1 Overview

The goal of this chapter is to introduce and describe all the technical aspects adopted in the *QDocs* Application System Development, we will provide information about the overall architecture structure, considering both sides, server and client, with more attention on the client-side (i.e. Mobile App).

3.2 Selected Architectural Styles and Patterns

The *QDocs* application was developed using the 3-tier client-server architecture (figure 3.1).

"3-tier architecture is a client-server architecture in which the functional process logic, data access, computer data storage and user interface are developed and maintained as independent modules on separate platforms."

- **Presentation Tier:** This tier occupies the top level and displays information related to services that the system provides. This tier only communicates with the application tier through classic internet request/response mechanism. The user interacts with the application system through this tier, hence the Mobile application *QDocs* is in this tier.
- **Application Tier:** Also called the middle tier, logic tier, business logic or logic tier. This tier is pulled from the presentation tier. It controls application functionality by performing detailed processing, such as calculations, logical decisions, data and model manipulation. This is basically the server application that provides APIs used by the presentation tier software. It communicates with both presentation tier and data tier.
- **Data Tier:** Houses storage servers where information is stored and retrieved. Data in this tier is kept independent of application servers or business logic. For this tier we used the services offered by Firebase: Real-time Database and Cloud Storage.

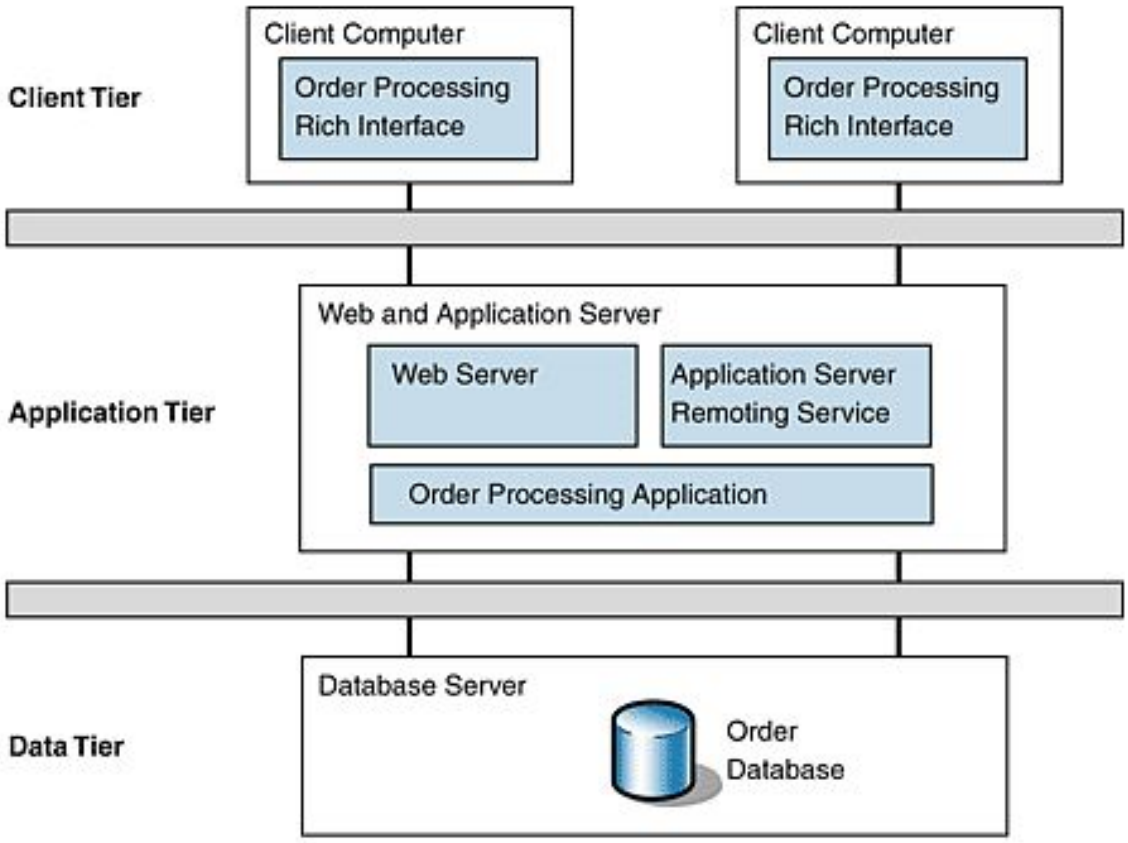

Figure 3.1: General 3-Tier Architecture

For our specific case the Presentation Tier corresponds to the Mobile Application, the Application Tier corresponds to the Firebase Server and finally the Data Tier is represented by a Real-time Database and a Cloud Storage, both offered by the Firebase platform.
Now we will inspect all tiers providing their technical details and how the communication among them are handled and performed.

Server: This represents the main back-end logic of the whole system, it stores information in a centralized way such that users can access their own data from different access-point (i.e. different devices), it allows multiple users to interact with the application system simultaneously and independently each other. From the technical point of view the server was built using the Firebase Mobile Development Platform (fig. 3.2) which is basically a cloud platform that offers several services. The main logic behind the server is fully implemented and managed by the Firebase platform, such as handling multiple users and multiple accesses to the data, simply providing different APIs each of one related to a specific service. For a deeper analysis on the services that we have used for the server refer to the section System Logic Architecture.

Figure 3.2: Firebase platform icon

Storage: This tier is entirely provided by the Firebase platform and it is completely masked from the client-side (i.e. Mobile App). The only way in which the user can access data stored in the storage is through the previous exposed server APIs (i.e. Realtime Database and Cloud Storage APIs). The communication between the server and the storage tier are completely handled by the Firebase platform itself. In this tier we have two different organized collection of data:

1. **Realtime Database:** This is a cloud-hosted NoSQL database, which is used to store and sync data information about files stored in the Cloud Storage. The data is stored as JSON and synchronized in realtime to every connected client.

2. **Cloud Storage:** This is a powerful, simple, and cost-effective object storage service, This is adopted since *QDocs* system needs to store and serve user-generated content, such as photos or videos.

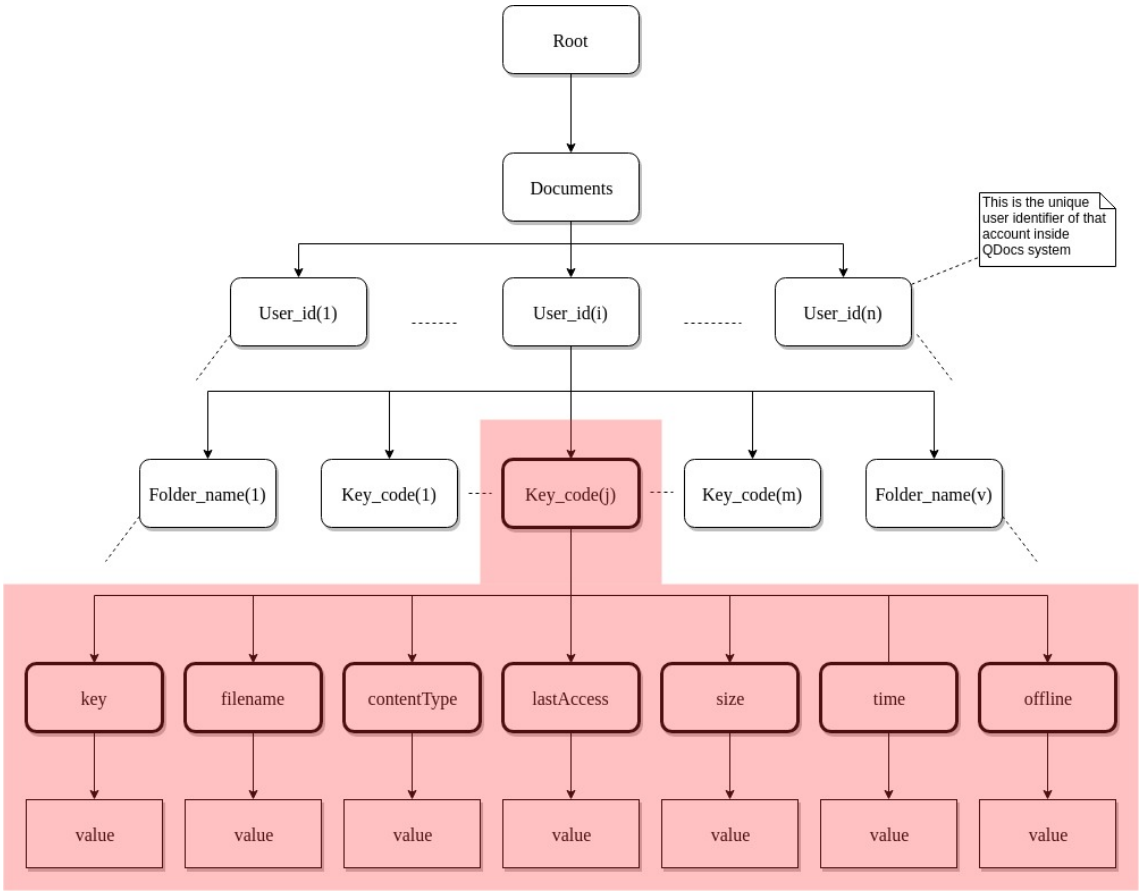

Figure 3.3: Database structure represented as tree figure

The database structure is quite simple, due to the fact that it is a NoSQL database, the values are always stored as ¡key-value¿ pairs for this reason we can easily represent its structure through a tree-base model (fig. 3.3). As you can notice rounded boxes represent nodes that can have children, while rectangular boxes represent leaf-values, this means that they cannot have other children. Since the atomic element that we store inside the database is a file, let's start analysing how it is structures: as the red box highlights, a file is represented by a child (e.g. Key_code) named as its unique associated code, which has a fixed number of children

that represent file's metadata (i.e key, filename etc.), all these children (i.e. features) have an associated value, that is the feature's value (e.g. name of the file for the *filename* feature). See table 3.1 for more details about features.

Feature	**Value Type**	**Description**
key	String	This is the code associated to its related file, it has to match with the parent name, so the key of the file object (e.g. Key_code).
filename	String	Name of the uploaded file (e.g. IMG_74839.jpg).
contentType	String	It represent the file's type, also called mime-type (e.g. image/jpeg).
lastAccess	Long	As the name says, it is the time in which the user has made an interaction with this file (e.g. open).
size	Int as String	This is the size dimension of the file, in byte.
time	Date as String	This is the time in which the file was uploaded.
offline	Boolean	Boolean value that tells whether that file is stored offline or not.

Table 3.1: *QDocs* android settings

There is another kind of object, the folder object, that basically acts as the *user_id* object since it can have as children, both files and other folders. Finally notice that each user can access files that are stored in a sub-tree where the root is its own user identifier, this simple procedure avoid that a user accesses files owned by other users.

Client: The presentation tier is implemented with the Mobile Application itself, it communicates with the server through the internet, using the exposed APIs that we have already discussed in the server section. This tier is the only way in which the user can interact with this system. More technical details about the mobile application design are described and analysed in section Mobile Application Design.

3.3 System Logic Architecture

This section will provide an overview of the overall logic architecture, considering the whole system. As already said the application system logic is based on a simple client-server architecture (i.e. 2-layers architecture).

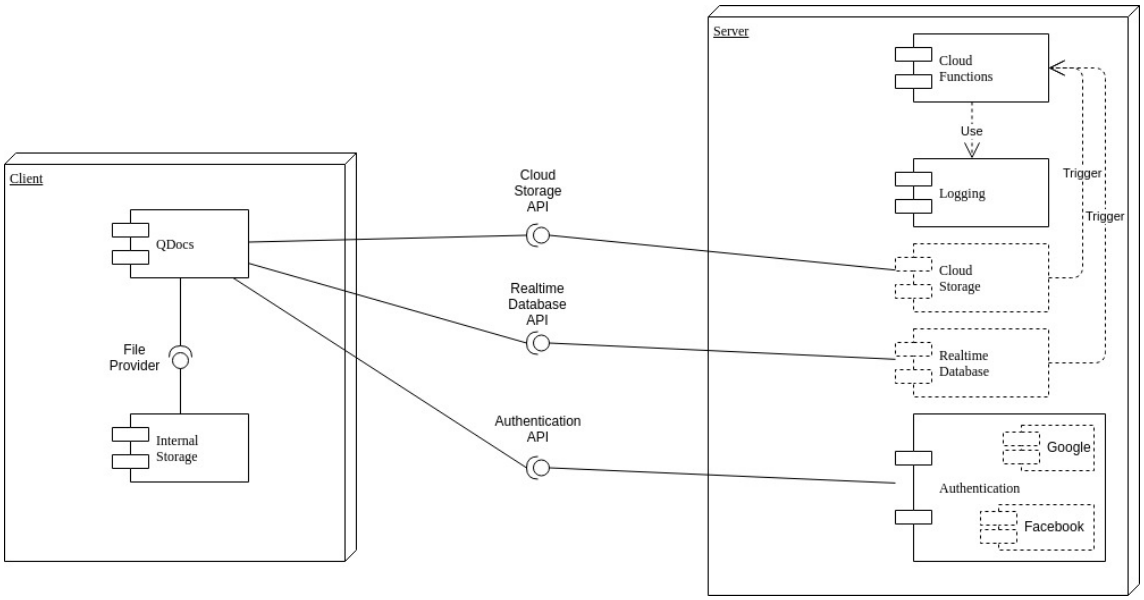

Figure 3.4: Component Structure diagram that shows the Client-Server logic architecture

As you can see in figure 3.4 the architecture is composed by two main building blocks: Client and Server.

3.3.1 Client

This block is composed by the mobile application and the device's hardware where the application is running. *QDocs* communicates with underneath hardware level through APIs offered by the Android OS, in this particular case accesses the internal storage of the device through the File Provider and it triggers external application when required (e.g. File Picker). The app also communicates with the application server over internet through the exposed APIs (i.e. Cloud Storage, Realtime Database and Authentication). More details about the implementation of the Mobile App can be found in section Mobile Application Design.

3.3.2 Server

The server was built using the Firebase platform that, like Google Cloud Platform, is full of cloud services for complex software systems. Differently from a classic App Engine which acts as PaaS (Platform as a Service) where you must create some APIs that will be called by the client, here instead, you cannot create your own APIs since clients talk directly to the services in Firebase, using special SDKs for web and mobile environments, which provide their own APIs that you have to use but that you cannot modify, hence, Firebase server acts as BaaS (Backend as a Service). In addition to those services we have added additional backend logic (through the Cloud Functions service), notice that this logic is triggered by other firebase services

and cannot be called by the clients. The Firebase server architecture is defined in figure 3.5.

Services

In according to the requirements of the *QDocs* Mobile App the server exposes the following services:

1. **Authentication:** In order to use the application system an authentication is required, for this reason we have adopted an authentication mechanism in order to allow users to register in our system and to login whenever they want. The server allows different registration mechanism, using external services (i.e. Facebook and Google) or using a classic email-password registration mechanism which is handled inside the firebase platform. The server automatically keeps tracks of all registered users.

2. **Cloud Storage:** Due to the fact that our application system is a file-storage application we adopted this API that is used by the mobile application to interact with the Cloud Storage through the server (e.g. uploading files, downloading files).

3. **Realtime Database:** This API allows user to interact with the Firebase Database, used to query information about online-stored files in a faster way. This API allows, mobile application, to keep listen database changes and instantaneously update the GUI in according to the listened data.

Cloud Functions

As stated in firebase documentation [2]: "Cloud Functions for Firebase let you automatically run backend code in response to events triggered by Firebase features and HTTPS requests. Your code is stored in Google's cloud and runs in a managed environment." Its implementation is based on a trigger-model where each trigger is executed in response to some Firebase events, here we have implemented two simple triggers, both have the goal to keep the Database and the Storage coherent each other. Let's analyse what both trigger do:

- **insertFileTrigger:** this is executed when a new file is uploaded in the Cloud Storage, it has to unpack data stored in the metadata of the file and add a new child in the database containing all file's information.

- **deleteFileTrigger:** executed upon file deletion on the Cloud Storage, it has to delete the corresponding child in the database.

Both these triggers use a logging service, provided by Firebase, used in development phase for debugging.

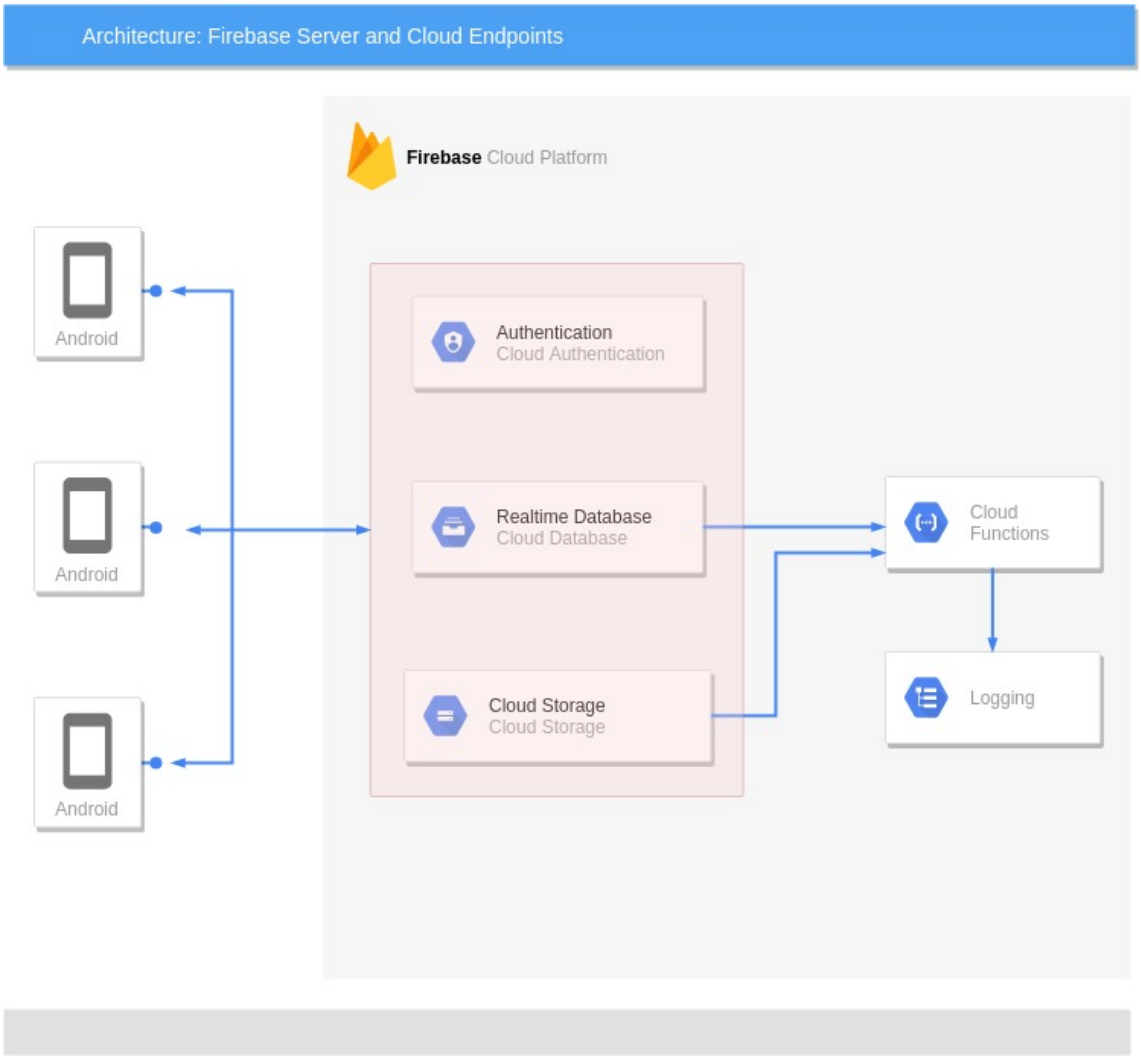

Figure 3.5: Firebase platform server

3.4 Mobile Application Design

The Mobile Application is a native Android mobile application written in Java programming language, its structure follows the standard provided by Android, as you can see in figure 3.6 the implementation is organized in three main directories: *app*, *functions* and *gradle*.

The **app** directory contains all things strictly related to the mobile application implementation:

- **Source Code:** The source code defines the logic behind the mobile application, it is totally written in Java programming language.
- **Resources:** All resources files needed by the application, such as layout, images, colors, font, menus etc. All these file are formatted in xml files, as standardized by Android.
- **Manifest:** Defined by the AndroidManifest.xml file which describes essential information about your app to the Android build tools, the Android OS, and Google Play. These information specify the Activity classes, the File Provider,

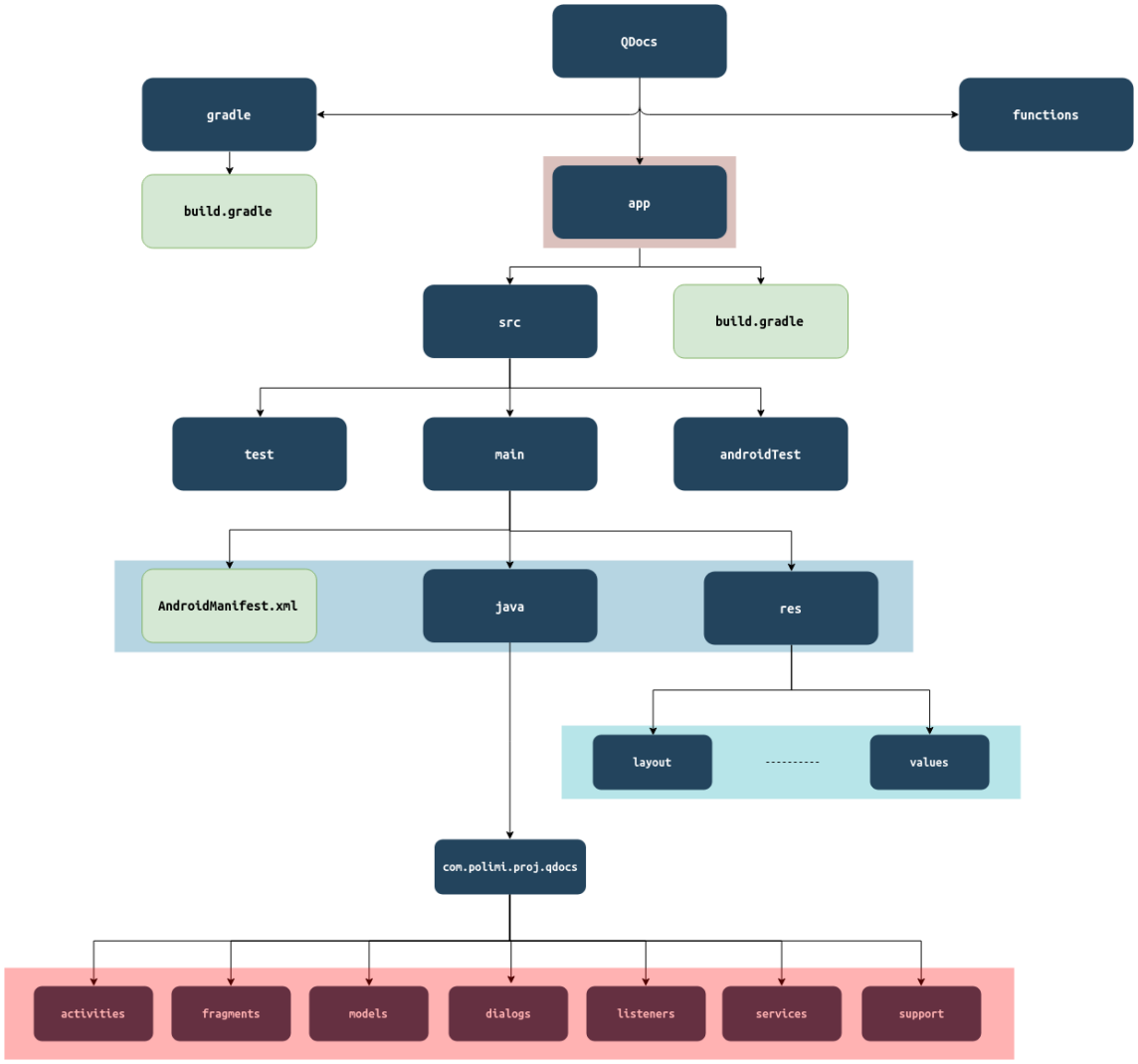

Figure 3.6: Directories structure of the *QDocs* mobile app project

the permissions required and some information about them, such as which Activity is the launcher one.

- **App Gradle:** defined by the *build.gradle* file, located in each project/module/ directory, allows you to configure build settings for the specific module it is located in. Configuring these build settings allows you to provide custom packaging options, such as additional build types and product flavors, and override settings in the main/ app manifest or top-level build.gradle file.

The **gradle** directory contains over all the *build.gradle* file, located in the root project directory, defines build configurations that apply to all modules in your project. By default, the top-level build file uses the buildscript block to define the Gradle repositories and dependencies that are common to all modules in the project.

Finally we have the **functions** directory that contains the backend logic that will have to run on the server through the Cloud Functions service of Firebase, more details on these functions can be found in section Server

3.4.1 Settings, Dependencies and Permissions

QDocs was developed defining some general settings about the application itself, like the API version, the supported Android version and so on. All these settings are reported in table 3.2.

Parameter	Value	Description
compileSdkVersion	28	Android version to use for compilation, this means your app can use the API features included in this API level and lower
applicationId	”com.polimi.proj.qdocs”	uniquely identifies the package for publishing
targetSdkVersion	28	Specifies the API level used to test the app
minSdkVersion	23	Defines the minimum API level required to run the app
versionCode	1	Defines the version number of your app
versionName	”1.0”	Defines a user-friendly version name for your app

Table 3.2: *QDocs* android settings

For the application development we have used several external dependencies that are listed in table 3.3.

Dependency	Description
com.google.firebase:firebase-database:18.0.0	Used for the Realtime Database APIs
com.google.firebase:firebase-storage:18.0.0	Used for the Cloud Storage APIs
com.google.firebase:firebase-auth:18.0.0	Used for the Firebase Authentication APIs
com.google.android.gms:play-services-auth:17.0.0	Used for the Google Auth APIs
com.facebook.android:facebook-android-sdk:4.38.1	Used for the Facebook Auth APIs
com.journeyapps:zxing-android-embedded:3.6.0	Used for the QR code generation and scanning
com.github.MikeOrtiz:TouchImageView:1.4.0	Used for our custom implementation of an image viewer
com.akexorcist:RoundCornerProgressBar:2.0.3	Used for the progress bar dialog (e.g. download and upload)
com.leinardi.android:speed-dial:2.0.1	Used for the upload floating action button
com.aurelhubert:ahbottomnavigation:2.3.4	Used for the bottom navigation view that allows switching among fragments
com.google.android.material:material:1.1.0-alpha06	Include other libraries compatible with androidx
com.github.bumptech.glide:glide:4.9.0	Used for loading preview images

Table 3.3: *QDocs* dependencies

QDocs requires some permission that the user has to grant in order to use the application, in table 3.4 we will list all required permission in association with the motivation of their requirements.

Permission	Usage
INTERNET	Required for the online files fetching and online operation
ACCESS_NETWORK_STATE	Required for the online files fetching and online operation
CAMERA	Required for the QR code scanning
WRITE_EXTERNAL_STORAGE	Required for storing files in the public storage directory of the device
READ_EXTERNAL_STORAGE	Required for reading files from the public storage directory of the device

Table 3.4: *QDocs* permissions

3.4.2 Architecture

The architecture of *QDocs* mobile application follows a FMVC (i.e. Framework-Model-View-Controller) that is an MVC's adaptation in the Android mobile development context, where the framework (i.e. Android OS) imposes its idea of component life cycle and related events, and in practice the *Controller* (e.g. Activity, Fragment etc.) is first of all responsible for coping with these Framework-imposed events, hence, there are two main differences between the MVC and the FMVC patters, the former is that in FMVC the *View* is simply a static view object that is inflated by *Controller* and the latter is that in the FMVC the events (e.g. hardware or user inputs) are triggered by the *Framework* (i.e. Android OS). Let's analyse component-by-component the FMVC pattern implemented in this application (figure 3.7):

- **Framework:** This component is represented by the Android Operating System that basically defines and imposes the components life-cycle and related events that they have to handle, in addition it captures users' inputs and forward them to the *Controller* objects that will have to manage them and update the UI accordingly.

- **Model:** This component is responsible for managing the business logic and handling network, authentication, storage and database API, it is clearly separated by the user interfaces and their related controllers.

- **View:** This is the visualization of the data, in this case a view is represented by an xml file (there can be more complex views that are defined by several xml files), which defines the structure and the composition of the view itself. Differently from the classic MVC, here, the view is a 'static' representation since it defines how to user interface appears when loaded (i.e. inflated by the controller) into the memory. All dynamics and updates to the view must be handled by its related *Controller* object.

- **Controller:** This component has to manage events (e.g. user o hardware events) triggered by the *Framework* and update the user interface (i.e. *View*), to which is associated to, accordingly. It communicates directly with the *Model* objects in order to achieve predefined operation on some events.

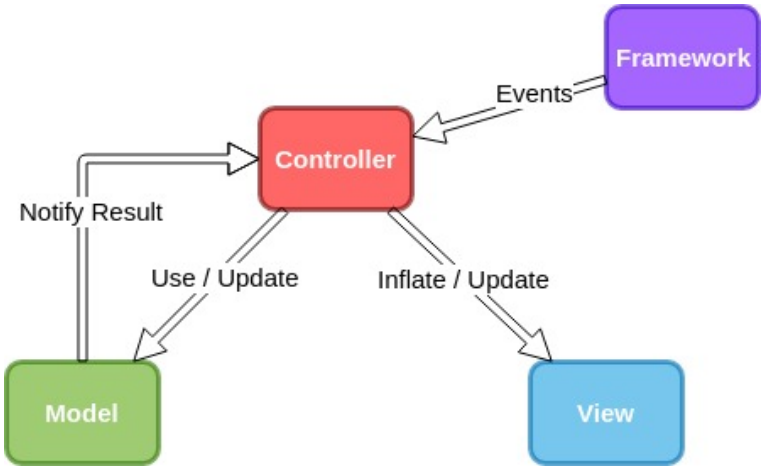

Figure 3.7: Model-View-Controller pattern

After this brief introduction on the components that compose the Framework-Model-View-Controller paradigm let's defined how it was realized in this project, hence, in table 3.5 you can find the realization of each FMVC component associated with some examples of Java objects. Notice that in this table there is no mention about the *Framework* since it is implicitly included, because it is the Operating System that is behind the mobile application, hence there is no code-realization of this component.

FMVC Compone-nt	**Java Objects**	**Examples**	**Description**
Model	All Java classes but Activities and Fragments	DownloadFileService, ShowFileReceiver, MyFile, MyDirectory, FirebaseHelper	All these objects represent the main client-logic that is behind the screens
View	XML files	activity_login, activity_registration, activity_main, fragment_scanner	Static layouts that define the user interfaces
Controller	Activity and Fragment classes	LoginActivity, RegistrationActivity, MainActivity, ScannerFragment	these objects are associated to a specific view which they will have to control and update in according to user's inputs

Table 3.5: Correspondence between MVC components and Java objects

3.5 Component Views

In according to the FMVC pattern discussed in section Architecture the package structure is organized as figure 3.8 defines.

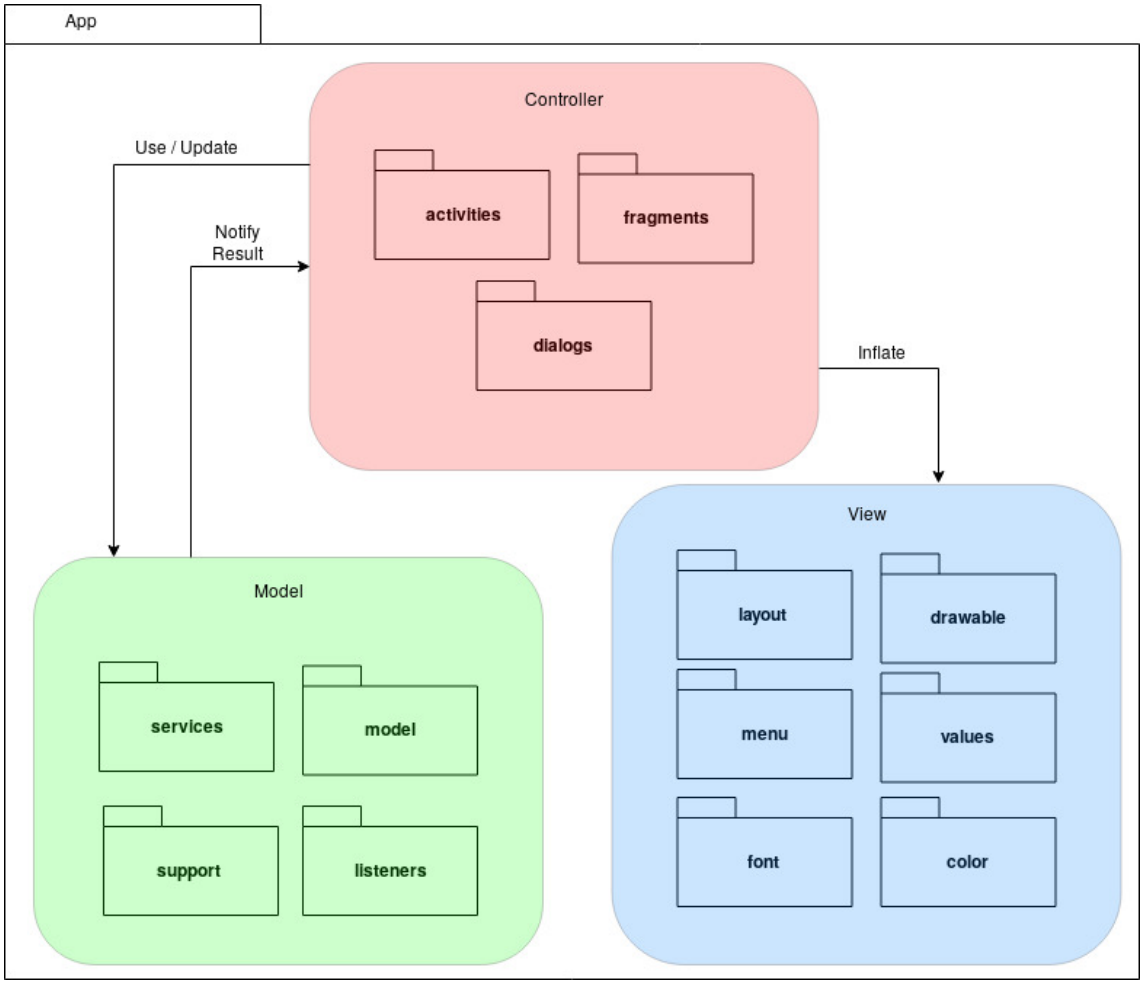

Figure 3.8: Packages organization and interaction

As you can see we can divide all packages into three main set: *model* (green box), *view* (blue box) and *controller* (red box), the table itself explain well how these three sets interact each other: the controllers objects will inflate the static layout contained in the view set, simultaneously the controllers use and/or update the model classes which will return results accordingly.

In the following figures we will provide some technical details about classes interaction and composition. Since the number of classes implemented is very high we decided to split the class diagram into two different ones each of one focusing on a specific kind of classes. For deeper class analysis refer to the next section, titled Component Descriptions.
The former diagram, represented in figure 3.9, focuses the attention on all classes that belong to the *controllers* logic set, as you can notice there are also colored box that define the packages in which the classes are organized (i.e. green box for activities, red box for fragments and blue box for dialogs).
The latter diagram (figure 3.10) focuses the analysis on the classes belonging to the *model* logic set, notice that we have reported only those classes, previously reported,

that are affected by these new classes.

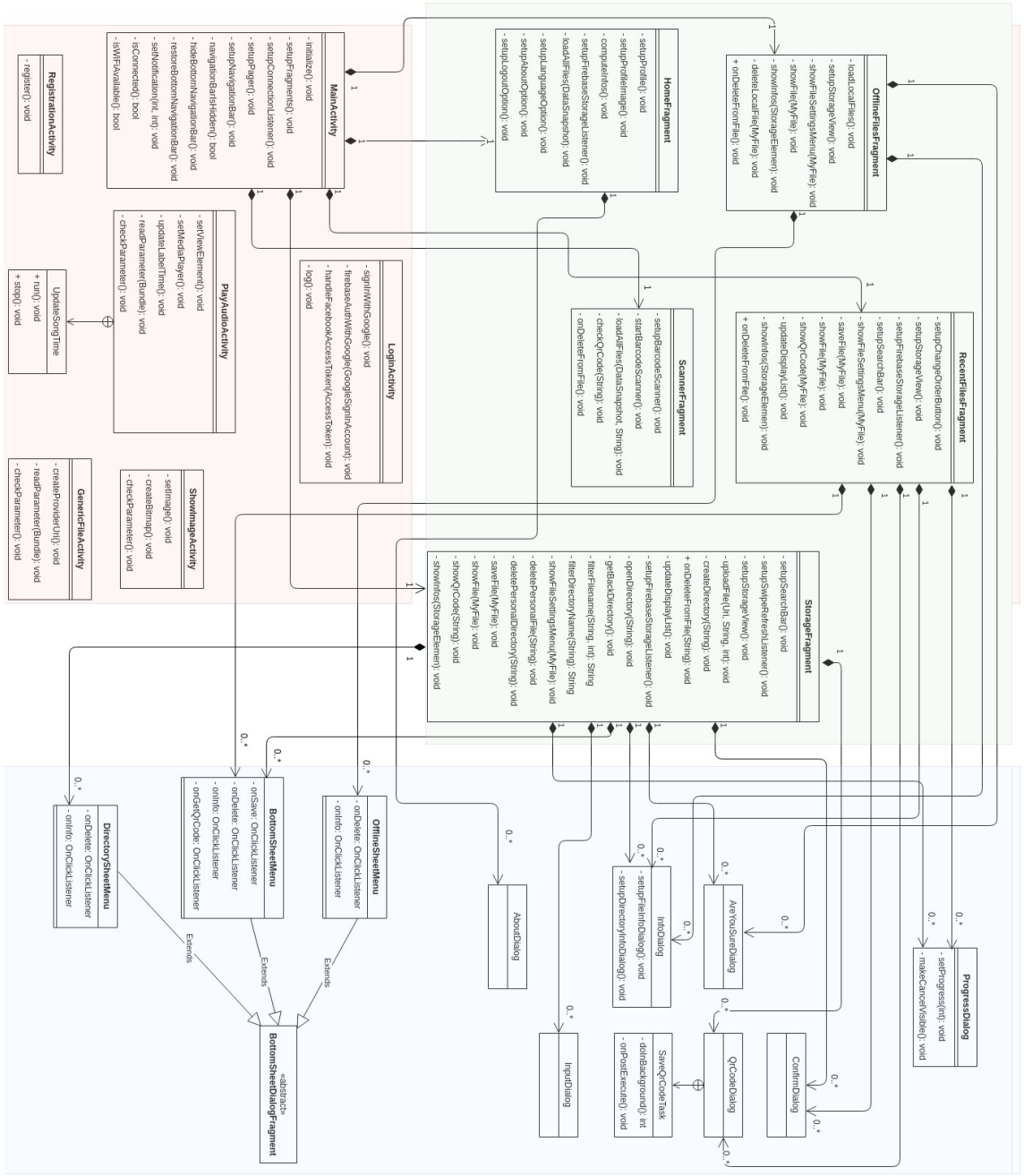

Figure 3.9: Class diagram focusing the attention on controller classes

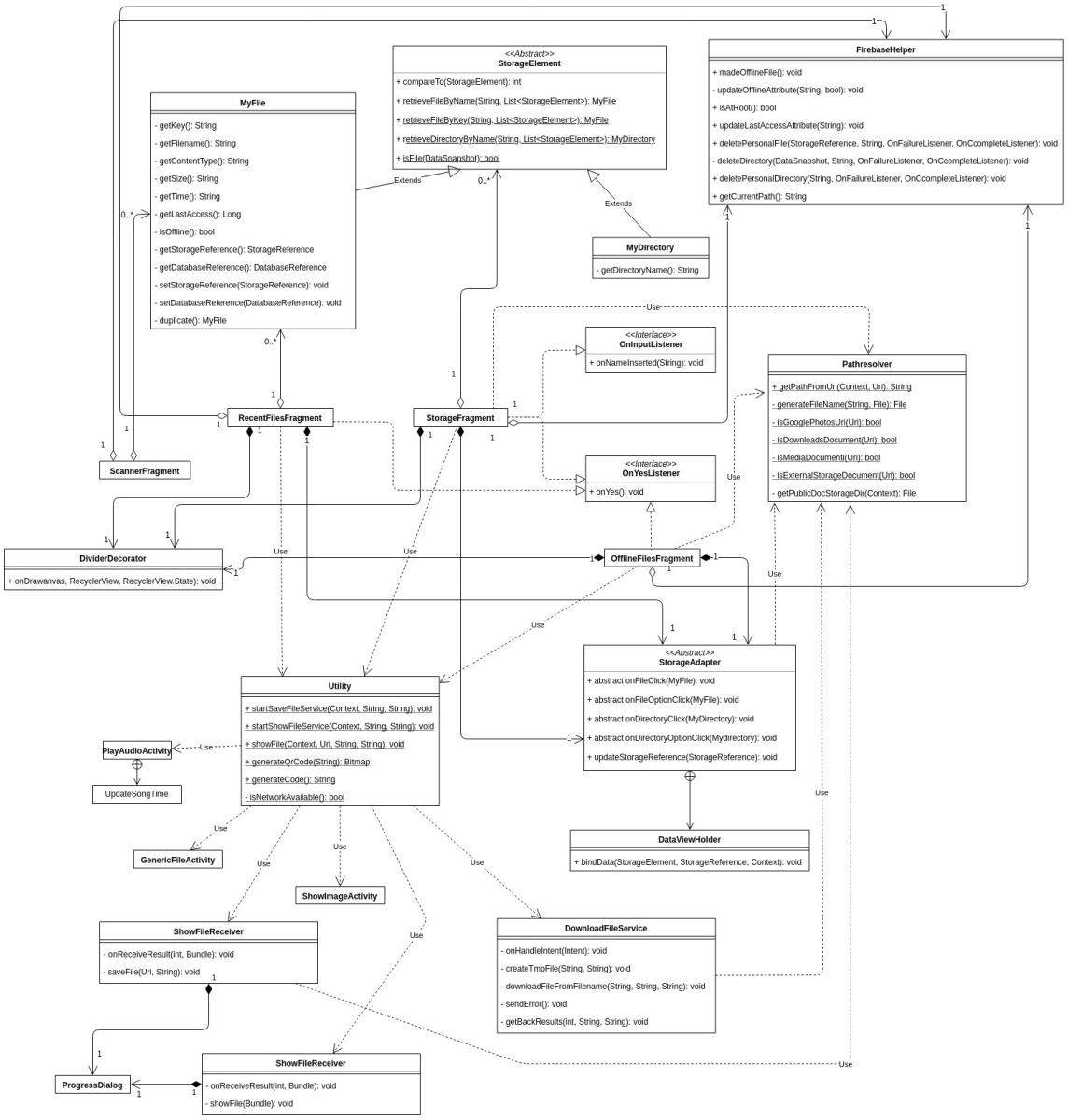

Figure 3.10: Class diagram focusing the attention on model classes

In figure 3.11 you can see the activities flowchart that focuses the attention only on the Activity objects and their relation. In this chart we have basically two elements:

- **Entity:** each entity represent an Activity object or Fragment object, where each of one is associated to a specific layout that will be displayed to the user.
- **Arc:** a directed edge that connects two Entity objects, it can be unidirectional o bidirectional. Its meaning is that if an arc connects A to B this means that the entity A at some point (or upon user's input) will start the entity B under some condition.

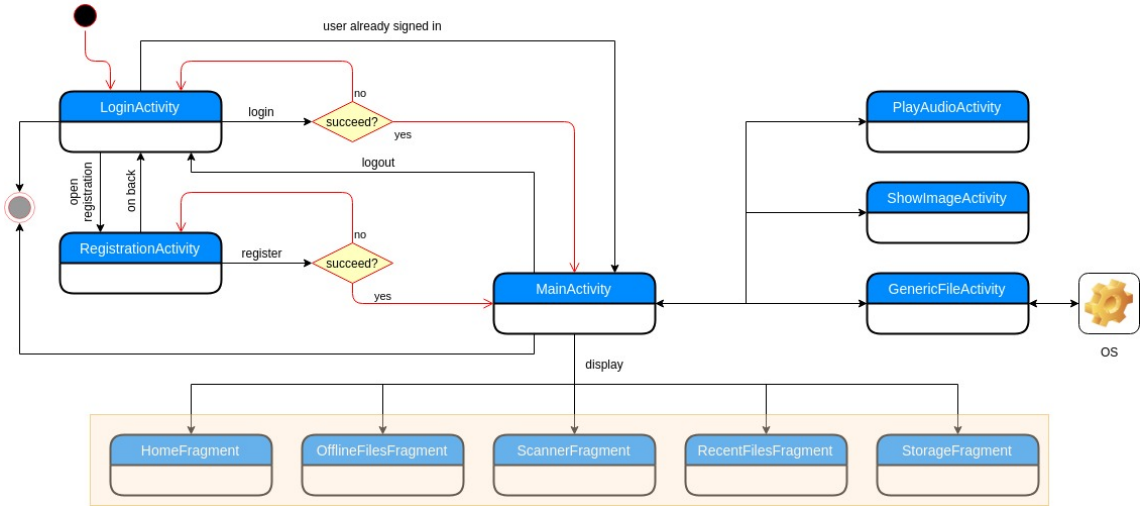

Figure 3.11: Activities flowchart

Let's start analysing the flowchart Activity-by-Activity:

- **LoginActivity:** this is the first Activity that will always be launched on application opening, on start it will check whether the user is already signed-in, if so it will directly start the MainActivity without showing its layout such that the user does not notice anything. If the user was not already signed-in it will display the login form and registration button, at this point the user have three possible action, the first is to exit, the second is to fill the form and try signing in, if the login operation succeed the activity will start the MainActivity. The last action is that the user click on *registration button* and now the activity will start the RegistrationActivity.
- **RegistrationActivity:** this activity is the simplest one since it basically displays a registration form that the user can fill, now it's time for user to make an action, it can come back to the LoginActivity (on back pressed) it fills the form and register itself to the system, if all goes well the activity will launch the MainActivity.
- **MainActivity:** this activity mostly acts as fragment container, such that it can dynamically show different layouts, without changing the Activity, in according to the user's inputs. From this activity the user can start the LoginActivity with the *logout* operation, exit the application on *back pressed button* or start one of the PlayAudio-/ShowImage-/GenericFile- Activity simply clicking on a file that it want to open.

- **PlayAudioActivity, ShowImageActivity and GenericActivity:** This three activities are in charge to show a specific kind of files, the third one (GenericFileActivity) differently from the others will start external application (activities) by launching an Intent object outside the application, which contains information about the file that has to be opened. From these activities that user can simply come back to the MainActivity by closing the opened file (on back pressed).

3.6 Component Descriptions

This section extend what has been discussed in Component Views section, in particular it is focused on the most important components implemented in the *QDocs* application, where for each of them there will be a detailed description, we focused on Activity/Fragments objects since they are strictly related to the screen views and of other important classes like services, receivers etc. The more important classes are described through tables that will include the name of the class, a brief description of the class and a list of methods, each of one is associated to a descriptions.

Class Name	LoginActivity
Brief Description: This activity is associated to the activity_login.xml file, it responsible to manage the login of the user, in according to the user's inputs it can start the RegistrationActivity or the MainActivity, the latter can only by started if the login succeed.	
Method	**Description**
signInWithGoogle	this starts a new activity retrieved from GoogleSignInClient which will have to provide google credential for the Firebase login
firebaseAuthWithGoogle	given a GoogleSignInAccount it retrieves the credential and it logs the user inside our system with Firebase
handleFacebookAccessToken	given the Facebook token, it retrieves the user's credential and it logs the user inside the system through Firebase
CredentialLogin.log	this is a method of an inner class that stores email and password provided by the user in the screen form, the method logs the user with Firebase authentication (i.e. email & password)

Table 3.6: LoginActivity object description

Class Name	RegistrationActivity
Brief Description: This activity is responsible to register a new user inside the system, following the registration mechanism provided by Firebase, hence, through email and password.	
Method	**Description**
register	this retrieves the email and password from the screen form and after it has checked all inputs, it registers the user inside the system through Firebase service

Table 3.7: RegistrationActivity object description

Class Name	PlayAudioActivity
Brief Description: This activity will basically play the audio file opened by the user.	
Method	**Description**
setViewElement	.
setMediaPlayer	.
updateLabelTiem	.
readParameter	it unpacks a Bundle object received as parameter, in order to retrieve information of the file that has to be opened, like Uri, MymeType etc.
checkParameter	checks whether there are all necessary file's information, unpacked from the Bundle in the readParameter method.

Table 3.8: PlayAudioActivity object description

Class Name	ShowImageActivity
Brief Description: This activity is in charge to display, internally, an image file allowing the user to zoom in and out the image or delete it directly from the image itself.	
Method	**Description**
setImage	given the Bitmap object created in 'createBitmap' method, it sets that into the TouchImage object.
createBitmap	create the Bitmap object from the file's Uri, that will be displayed on the screen.
checkParameter	checks whether there are all necessary file's information, unpacked from the Bundle in the readParameter method.

Table 3.9: ShowImageActivity object description

Class Name	GenericFileActivity
Brief Description: This simple activity has to unpack file's information from the received Bundle and the compute necessary data needed for launching an Intent outside the application in order to allow the OS to select an external application (if any) that will display the current file.	
Method	**Description**
createProviderUri	compute the provider uri of the specific file that has to be opened, required for starting external applciation.
readParameter	it unpacks a Bundle object received as parameter, in order to retrieve information of the file that has to be opened, like Uri, MymeType etc.
checkParameter	checks whether there are all necessary file's information, unpacked from the Bundle in the readParameter method.

Table 3.10: GenericFileActivity object description

Class Name	ScannerFragment
Brief Description: This fragment is in charge to show a camera layout that has to be used by the user for scanning its QR-codes, if the scanned code is associated to a cloud-stored files of the user it directly opens it.	
Method	**Description**
setupBarcodeScanner	setup the barcode scanner adding a click listener that will hide/restore the bottom navigation view
startBarcodeScanner	initialize the scanner, such that it will be able to scan QR-code formats images, it adds a barcode callback that will be invoked whenever a new QR-code is decoded.
loadAllFiles	retrieves all information about files, which are stored in the cloud storage by the user.
checkQrCode	this method checks whether a specific decoded QR-code text is associated to a specific file owned by the logged user, if so start a background service that will download the file (if needed) and which will show it in the screen.
onDeleteFromFile	this is called when a delete operation is triggered by the user on file directly opend after a scan operation. This will delete the file through the FirebaseHelper class.

Table 3.11: ScannerFragment object description

Class Name	MainActivity
Brief Description: This class is the principal component of the overall application, it is structured as a FragmentActivity that dynamically changes the fragments that has to be shown in according to the user's inputs, the fragment selection can be performed through a bottom navigation view or by left-right swiping. Each fragment will be in charge to show different aspects of the application. This activity has to manage all of them allowing communications among them. For more details about the fragments show their specific tables.	
Method	**Description**
initialize	checks whether the required permissions are already granted (i.e. camera and storage permissions), if no it requests them. It checks also that the user is really logged-in.
setupFragments	initialize all five fragments and put them inside the fragments' list.
setupConnectionListener	setup a listener on the Internet connection, disabling or enabling some operation, such as download, in according to the network state
setupPager	setup the ViewPager object that allows the user to change the fragment by left-right swiping.
setupNavigationBar	setup the bottom navigation view such that when a specific tab is selected by the user it will automatically show the correct fragment and update the viewPager object
hideBottomNavigationBar	hides the bottom navigation view, for instance when tapping on the scanner or scrolling down the files' list.
restoreBottomNavigationBar	restore the bottom navigation view, for instance when scrolling up the files' list
setNotification	set a badge notification on a specific tab of the bottom navigation view
isConnected	It check whether the device is connected to Internet
isWiFiAvailable	If the device is connected, it checks whether the WiFi is available or not.

Table 3.12: MainActivity object description

Class Name	StorageFragment
Brief Description: This fragment provides the directories structure of user's storage in the cloud, it displays all cloud-files allowing the user to retrieve information about them and performing some operation, like saving file offline, retrieve QR-code and so on. This is also the fragment where the user can upload new files, open them like any other file-storage application and browse among user-created directories.	
Method	**Description**
setupSearchBar	setup the search bat that allows user to find their files/folders by name, in the current directory.
setupSwipeRefreshListener	setup the swipe refresh listener that refresh the files' list on classic swipe down event occur.
setupStorageView	setup the container (i.e. RecyclerView) of the storage data stored in the user's Cloud Storage, it will contain all files and directories. It adds a custom StorageAdapter on the RecyclerView that includes all operation that can be performed on the files/directories, like settings, open and so on.
uploadFile	given the intent data of the file, obtained by external pick-file application, it uploads the file on the Firebase Cloud Storage. This operation may fail due to connection lost or on user's cancel input.
createDirectory	given a new name (which must not be already used in current directory), it creates a new directory with given name.
onDeleteFromFile	this is called when a delete operation is triggered by the user on file opened from this fragment. This will delete the file through the FirebaseHelper class.
updateDisplayList	update the files' list to display in according to the search bar text

Table 3.13: StorageFragment object description (pt. 1)

Class Name	StorageFragment
Method	**Description**
setupFirebaseStorageListener	setup the listener on the firebase database in order to keep updated the storage view that contains all files in the current directory.
openDirectory	open a new directory, it will refresh the files' list with the new ones.
getBackDirectory	come back to the previous directory in the hierarchy, reloading the correct files.
filterFilename	checks whether there already exist some file with the given name, if so it updates the current name with additional text (e.g. (1), (2) and so on)
filterDirectoryName	filter the name of a new directory checking whether this name is already used, if so it prevents the directory creation.
showFileSettingsMenu	show file's settings menu, which displays all operation that can be performed to that file.
deletePersonalDirectory	it displays an 'are you sure?' dialog listening for a user's confirmation of directory deletion, if so it deletes the directory through the FirebaseHelper.
deletePersonalFile	it displays an 'are you sure?' dialog listening for a user's confirmation of file deletion, if so it deletes the file through the FirebaseHelper.
showFile	show the selected file using Utility method.
saveFile	save the selected file offline using Utility method.
showQrCode	it displays the QR-code associated to the selected file inside a custom dialog that allows user to save it offline.
showInfos	it displays an InfoDialog that lists all information of that file.

Table 3.14: StorageFragment object description (pt. 2)

Class Name	RecentFilesFragment
Brief Description: This fragment provides all saved files ordered by last access, it allows the user to reverse the order as he want. For each file the user can perform the classic operation that can be performed in the StorageFragment (table 3.13, 3.14), like save, get information, get QR-code and so on.	
Method	**Description**
setupChangeOrderButton	setup the floating action button that, on click event, change order of displayed files.
setupStorageView	setup the container (i.e. RecyclerView) of the stored data in the user's Cloud Storage, differently from the StorageFragment this list will only contain files (all stored files). As in StorageFragment, it adds a custom adapter.
setupSearchBar	setup the search bat that allows user to find their files by name, in the whole storage.
setupFirebaseStorageListener	setup the listener on the firebase database in order to keep updated the storage view that contains all files in the storage.
onDeleteFromFile	this is called when a delete operation is triggered by the user on file opened from this fragment. This will delete the file through the FirebaseHelper class.
updateDisplayList	update the files' list to display in according to the search bar text.
showFileSettingsMenu	show file's settings menu, which displays all operation that can be performed to that file.
showFile	show the selected file using Utility method.
saveFile	save the selected file offline using Utility method.
showQrCode	it displays the QR-code associated to the selected file inside a custom dialog that allows user to save it offline.
showInfos	it displays an InfoDialog that lists all information of that file.

Table 3.15: RecentFilesFragment object description

Class Name	OfflineFilesFragment
Brief Description: This fragment is directly linked to a public storage directory inside the device that is using the application, this is quite similar to the RecentFileFragment (table 3.15) but instead of showing online file it shows all file in the local directory, which are generally all files saved inside the app. It also allows user to perform some operation on these files, notice that all these operation affect only the local copy of the files and not the online ones.	
Method	**Description**
loadLocalFiles	load all files object locally saved in the linked directory, which is located inside the internal storage of the device.
setupStorageView	setup the container (i.e. RecyclerView) of the storage data stored in the user's Cloud Storage, differently from the StorageFragment this list will only files saved locally, so there is no need to fetch data online. It adds a custom StorageAdapter on the RecyclerView that includes all operation that can be performed on the files, like open settings, open file and so on.
showFileSettingsMenu	show file's settings menu, which displays all operation that can be performed to that file.
deleteLocalFile	delete a local copy of the file, given its name. After this operation the online file's attribute, called 'offline' is set to false.
showFile	show the selected file using Utility method.
showInfos	show local file's information.

Table 3.16: OfflineFilesFragment object description

Class Name	HomeFragment
Brief Description: This fragment represents the home page of the user, here the user can change the default language of the application, logout from its account and sees information about how many file he has stored and what is their total size.	
Method	**Description**
setupProfile	setup profile, its image and its information.
setupProfileImage	setup the user's profile image. The image is downloaded with an asynchronous task.
computeInfos	compute all user's information (i.e. display name, email used, number of stored files, total space used).
setupFirebaseStorageListener	setup the listener on the firebase database in order to keep updated the storage view that contains all files.
loadAllFiles	load all stored files in order to compute the space used and the number of stored files.
setupLanguageOption	setup the language option buttons that allows user switching between English and Italian language.
setupAboutOption	setup the About option button, that when clicked open a dialog with *QDocs* legal information.
setupLogoutOption	setup the button that is in charge to logout the user.

Table 3.17: HomeFragment object description

Class Name	FirebaseHelper
Brief Description: This is an helper class used for performing some operation on the FirebaseDatabase and FirebaseStorage	
Method	**Description**
getUser	returns the FirebaseUser instance.
updateDatabaseReference	given a path, updates the database reference, by adding that path.
updateStorageReference	given a path, updates the storage reference, by adding that path.
updateOfflineAttribute	given a file, update the 'offline' attribute.
updateLastAccessAttribute	updates the lastAccess attribute of a given file.
deletePersonalFile	delete a specific file from the cloud storage.
deletePersonalDirectory	delete a whole directory, by deleting all files that is currently containing.
logout	logout the user from its own account.

Table 3.18: FirebaseHelper object description

Class Name	SaveFileReceiver
Brief Description: This is a ResultReceiver object that is linked to a service, in this case DownloadFileService, which is in charge to handle results sent by the service. This particular receiver will save the file in the local directory	
Method	**Description**
onReceiveResult	receive the result from the DownloadFileService and operates accordingly, for instance showing the progress or saving the file in the local directory when the download is complete
saveFile	given the URI of the temporary file creates a new file in the local directory and it preforms the copy of the file.

Table 3.19: SaveFileReceiver object description

Class Name	Utility
Brief Description: This is an utility class that exposes static methods that are used by multiple activities	
Method	**Description**
startSaveFileService	starts a save file service, that will download the file and save it in the local directory of the device, this operation is allowed only when the file is not already saved.
startShowFileService	starts a show file service, that will download the file (if it not already locally saved) and it directly opens it through other activities.
showFile	in according to the file's type, it starts the correct Activity instance.
generateQrCode	generate the QR-code bitmap and returns it, this operation is performed using external dependencies.
generateCode	generate a new code that has to be associated to a new file, and then encoded into QR-code format.

Table 3.20: Utility object description

Class Name	ShowFileReceiver
Brief Description: This is a ResultReceiver object that is linked to a service, in this case DownloadFileService, which is in charge to handle results sent by the service. This particular receiver will have to retrieve the file object and to start the correct activity necessary to show the file, this operation is performed using Utility class.	
Method	**Description**
onReceiveResult	receive the result from the DownloadFileService and operates accordingly, for instance showing the progress or opening the file when the download is complete
showFile	given the Bundle, containing the required data, shows the file, through the Utility functions.

Table 3.21: ShowFileReceiver object description

Class Name	DownloadFileService
Brief Description: This is a Service instance that operates in background asynchronously with respect to the main thread of the application. It performs the download of a file, that may require a lot of power and it may slowly the application, for this reason it is performed on background. This service has to be linked with a ResultReceiver object that will run on the main thread and that will handle the service result modifying the GUI.	
Method	**Description**
onHandleIntent	method used to invoke the service, it requires an Intent object where we have to put the action to perform and the data required for that action.
createTmpFile	it creates a temporary file, where that file that has to be downloaded will be stored. It also checks whether the file is already stored in the local directory, if so it send back result to the receiver notifying this situation, otherwise invoke the download method (see below).
downloadFileFromFilename	given the filename, it starts the download of that file into the temporary file previously created. It continuously notifies the progress of the download to the receiver.
getBackResults	method used to send result back to the receiver.

Table 3.22: DownloadFileService object description

3.7 Dynamic Model

Based on the operation defined in Use Cases section and the implementation design defined in Mobile Application Design, this section is focused on the analysis of the *QDocs* mobile application from the runtime perspective, this analysis is performed through sequence diagrams that explain how the implemented objects interact each other during the execution of defined operation (e.g. login, file scan etc.). All the following sequence diagrams, together, defined the Dynamic Model of the application, since it defines its dynamic behaviour.

Registration (3.12)

The registration is triggered by the LoginActivity, upon user request, which starts the RegistrationActivity that will handle the registration mechanism. If the registration succeed the activity immediately starts the MainActivity, signing-in the user automatically.

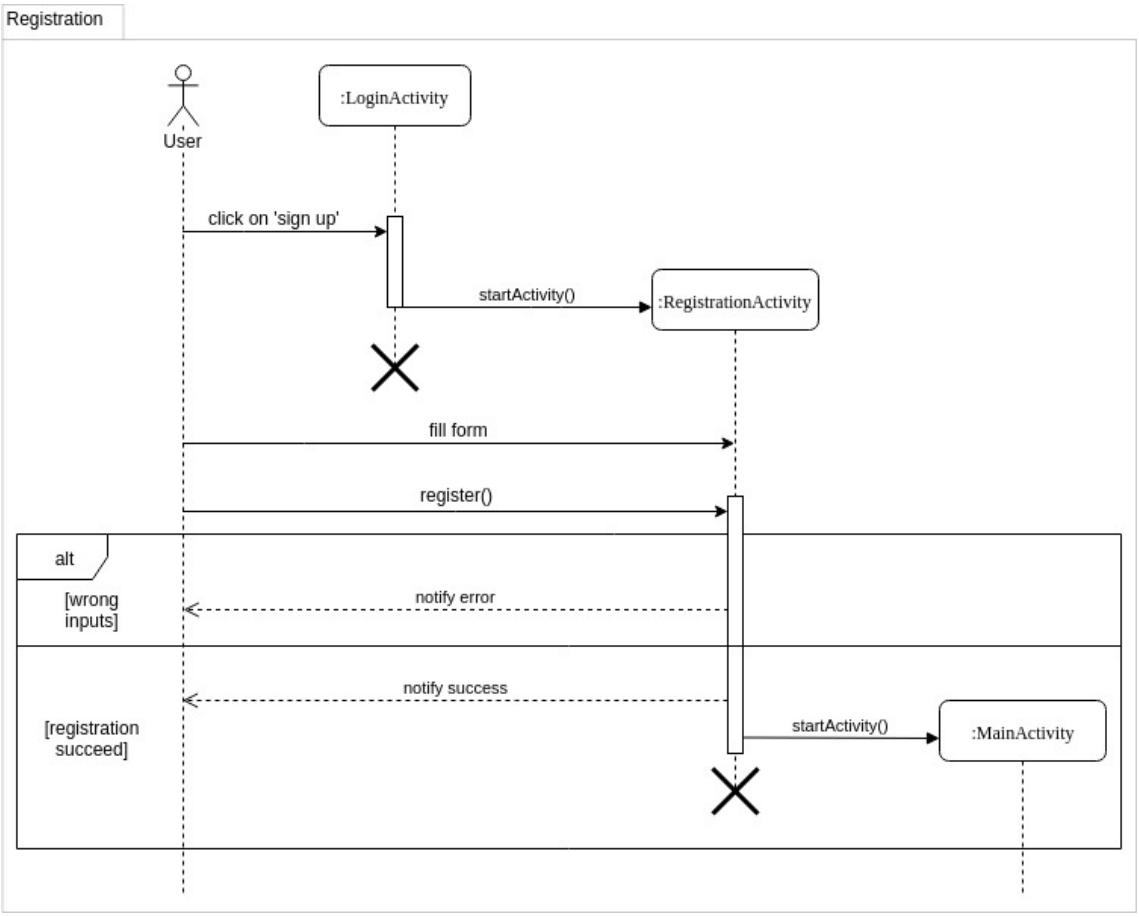

Figure 3.12: Sequence diagram of the user registration

Login (3.13)

This operation is quite all handled by the LoginActivity that interacts with external services (i.e. Google and Facebook) for specific cases.

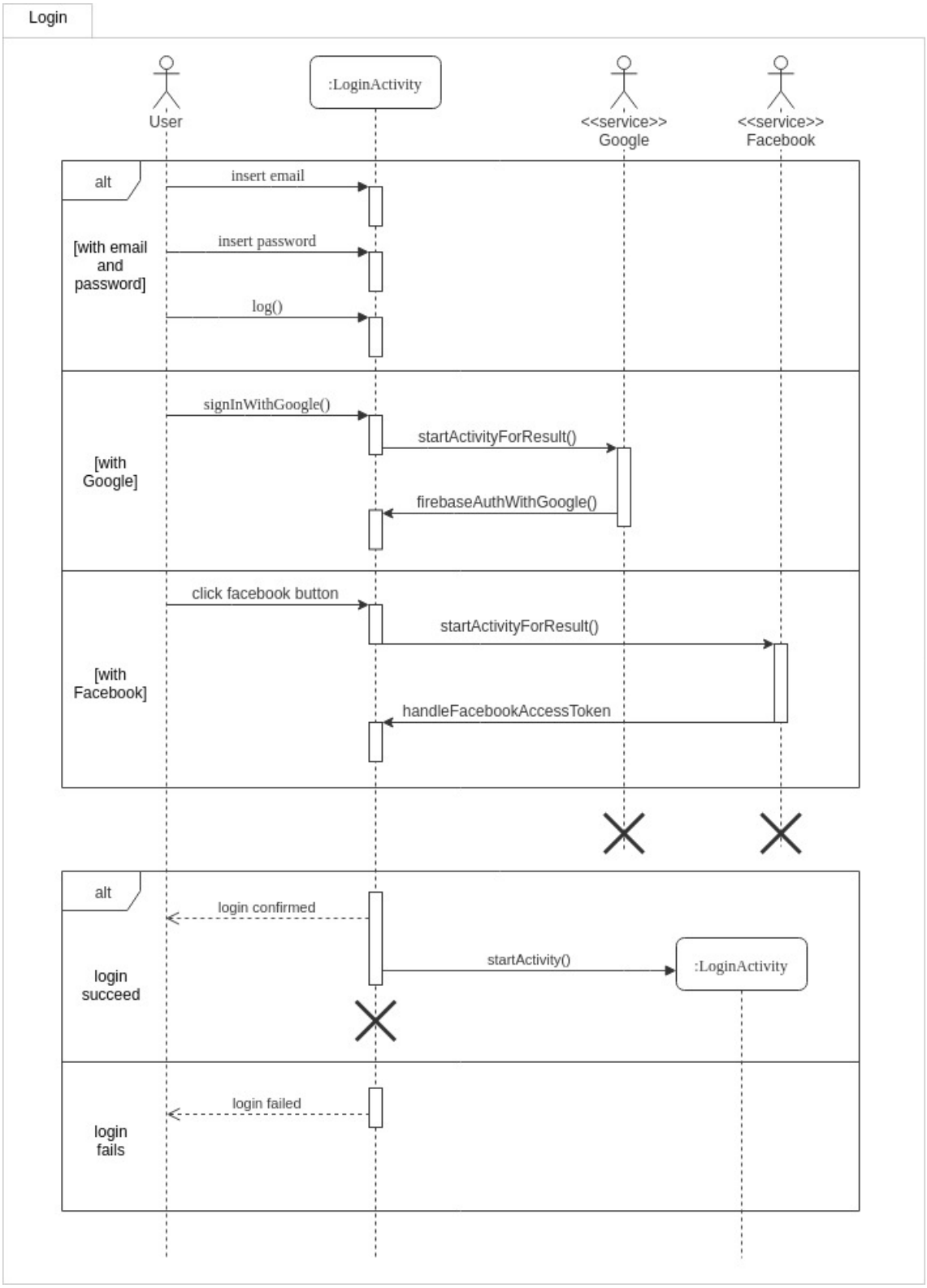

Figure 3.13: Sequence diagram of the login operation

Logout (3.14)

This operation is performed inside the MainActivity while showing the HomeFragment, the latter is responsible to use the FirebaseHelper utility and logout the user, at this point the HomeFragment starts the LoginActivity and it terminates.

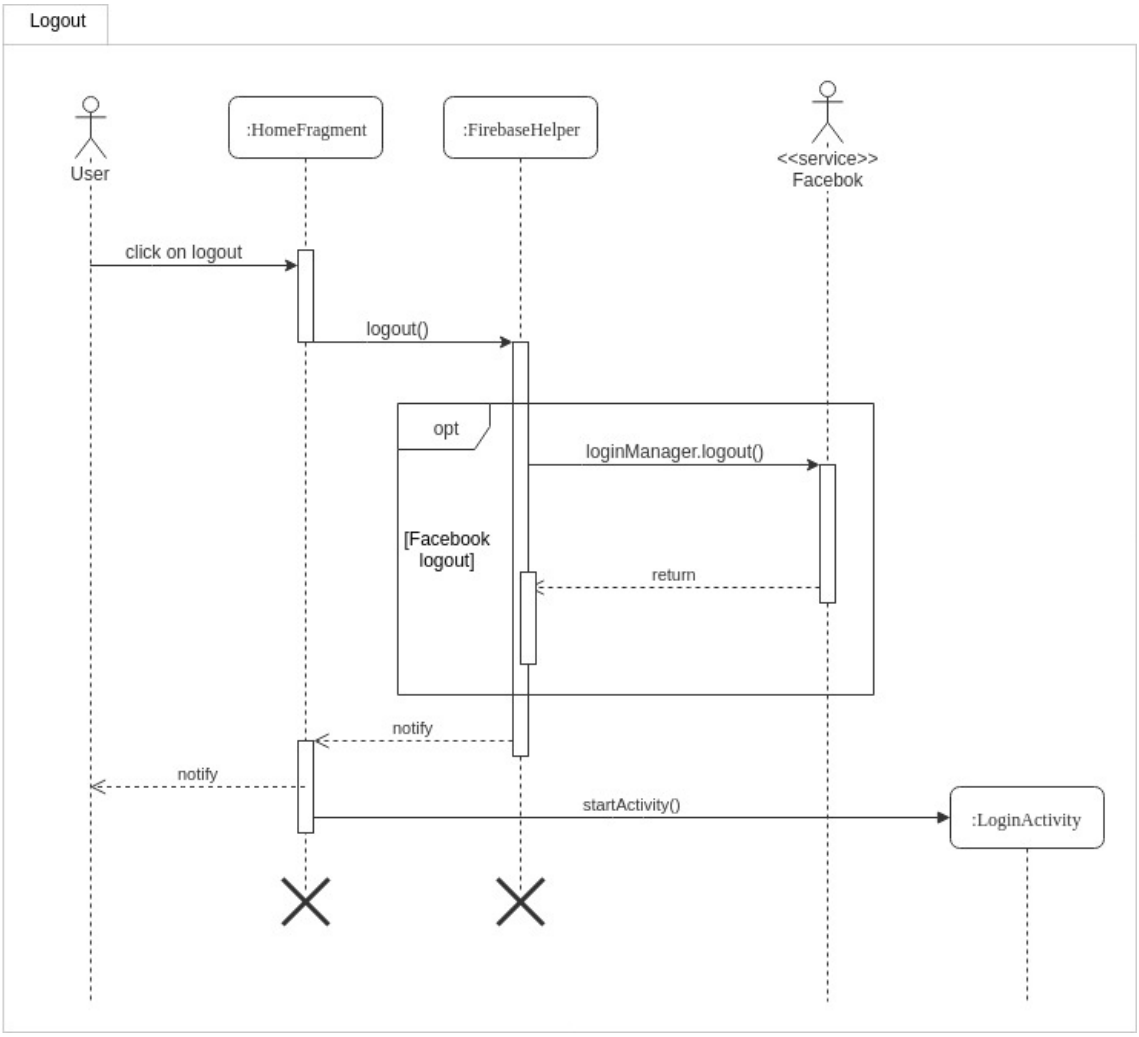

Figure 3.14: Sequence diagram of the logout operation

File Scan (3.15)

The file scan operation can be performed by the user inside the ScannerFragment (i.e. inside MainActivity), here the scan is executed through callback that reacts upon QR-code detection, at this point checks the decoded text by finding whetheer there exists a MyFile object that is associated tho that code. If the latter operation succeed start showing the corresponding file through the Utiliy class (see *Open File* diagram in figure 3.16 for more details) otherwise notify the user that there are no file associated to the scanned QR-code.

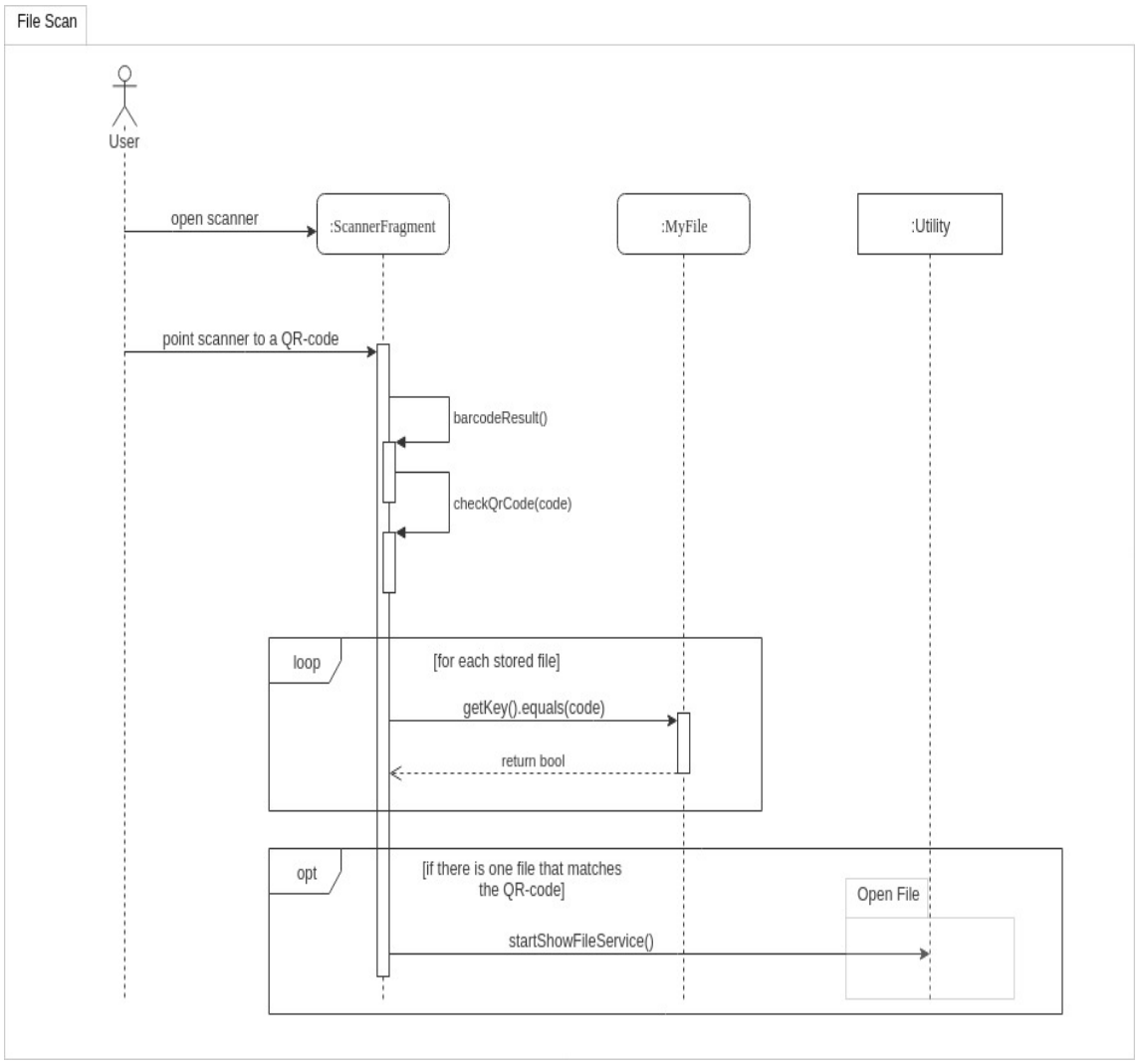

Figure 3.15: Sequence diagram of the file scan operation

Open File (3.16)

The following sequence diagram focuses the analysis on the file opening operation, abstracting from which point the file has been opened, specifically a file can be opened from four situation: from the StorageFragment, RecentFragment, Offline-Fragment and ScannerFragment (see figure 3.15). All of them follow this sequence diagram.

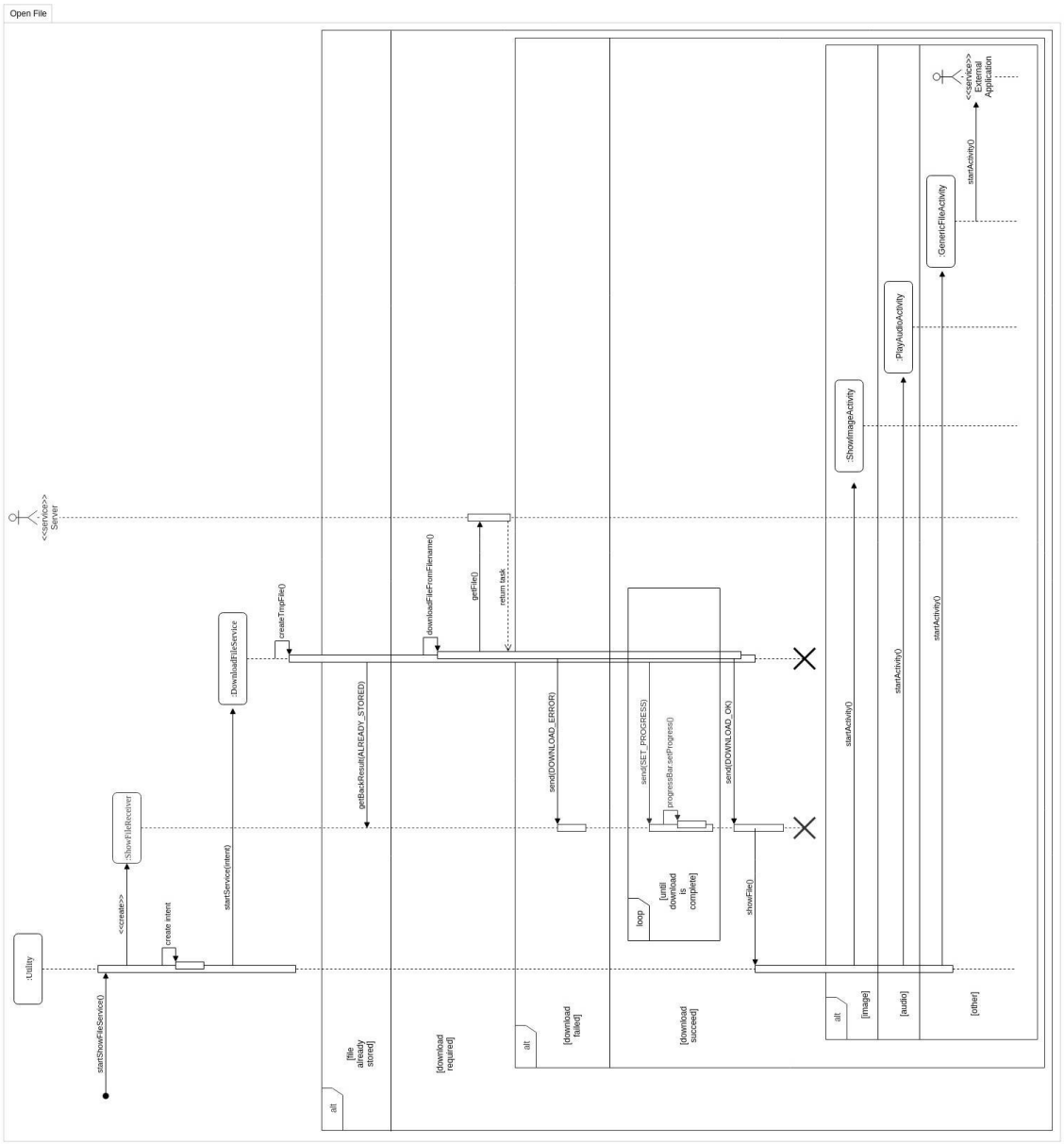

Figure 3.16: Sequence diagram of the file opening operation

Upload File (3.17)

This operation is quite all handled by the StorageFragment, which uses an external application (i.e. File Picker) in order to allow user to select the file that he want to upload, after that it starts uploading the file into the Cloud Storage.

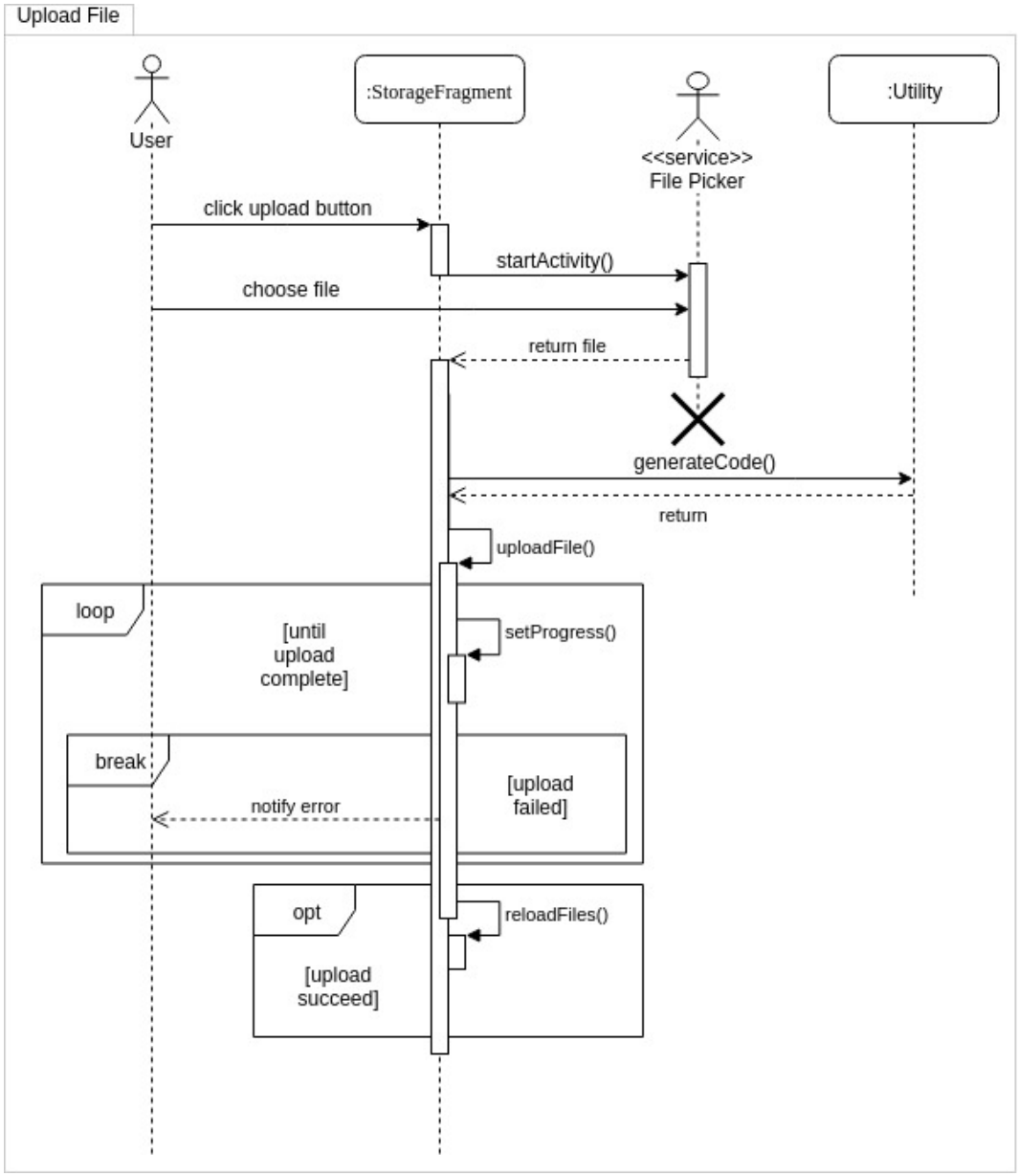

Figure 3.17: Sequence diagram of the file uploading operation

Print QR-Code (3.18)

This is one of the most important use cases since it allows user to automatically print the generated QR-code, using the print manager of the Android OS.

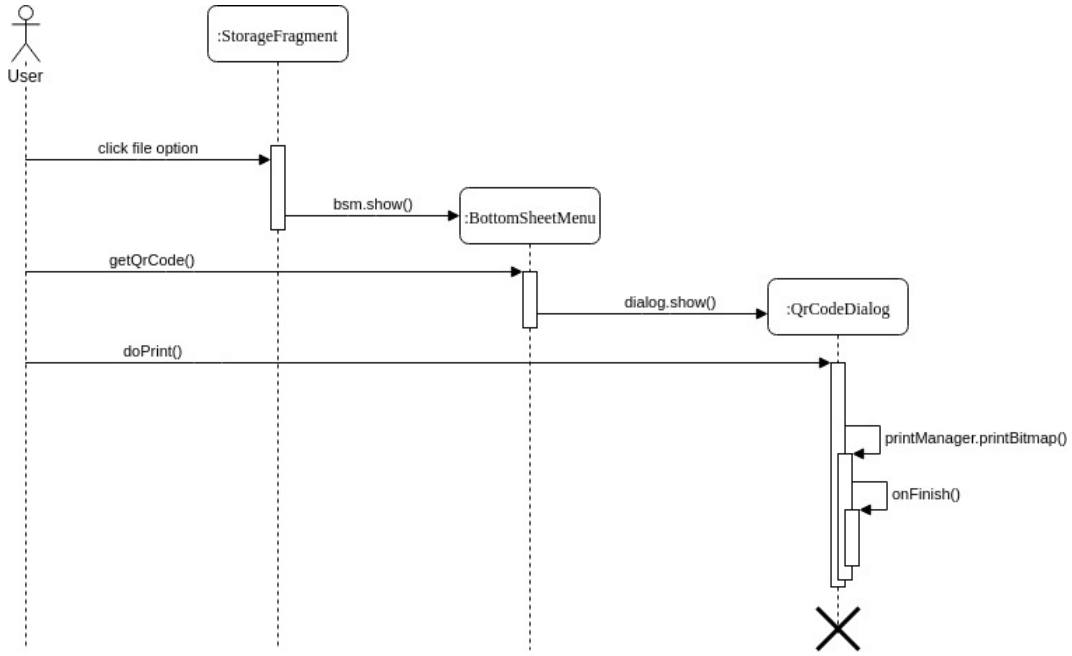

Figure 3.18: Sequence diagram of the QR-code printing operation

Chapter 4

User Interface Design

This chapter is focused on the UI Design, which is the process of making interfaces in software or computerized devices with a focus on looks or style, in particular this design represents how the user interfaces follow each other and upon which event/condition, providing a sort of user interfaces life-cycle. The whole UI design is represented in figure 4.2. Notice that, as already said, there exists a strong association between a user interface and an Activity object, hence there is a strong correlation between this UI design and the Activity life-cycle provided in figure 3.11 in Component Views section.

Initialization this paragraph focuses on what happened when the application is launched (figure 4.1), the first action that the application perform is to check whether the user is already signed-in, here we can have two different situation:

1. The user was already signed-in, so the application automatically starts the scanner screen (which is one of the principal screens of the application)

2. The user was not already signed-in, here the application shows a login screen, with a form. here the user can basically do two things:

   (a) try to login, filling the form then clicking on the *signin* button, or directly clicking on *Facebook* or *Google* button without filling the form. Now if the login procedure succeed the application starts the scanner screen, otherwise it notifies the user that something went wrong and it keeps the login screen active

   (b) switch to the registration, this operation starts a new screen (i.e. the registration screen) where the user can fill the form and try to register (on success start, again, the scanner screen) or come back to the login screen.

Main assuming that the user is arrived in the scanner screen (refers to figure 4.2), from this situation the user can switch among the following screens: home, offline files, recent files and storage screen, notice that from any of the previous screens (scanner included) you can switch to another of them. Let's analyse each screen:

- *home:* from this screen the user can change the language, which basically restart this screen or logout itself, and upon this latter operation the application come back to the login screen.

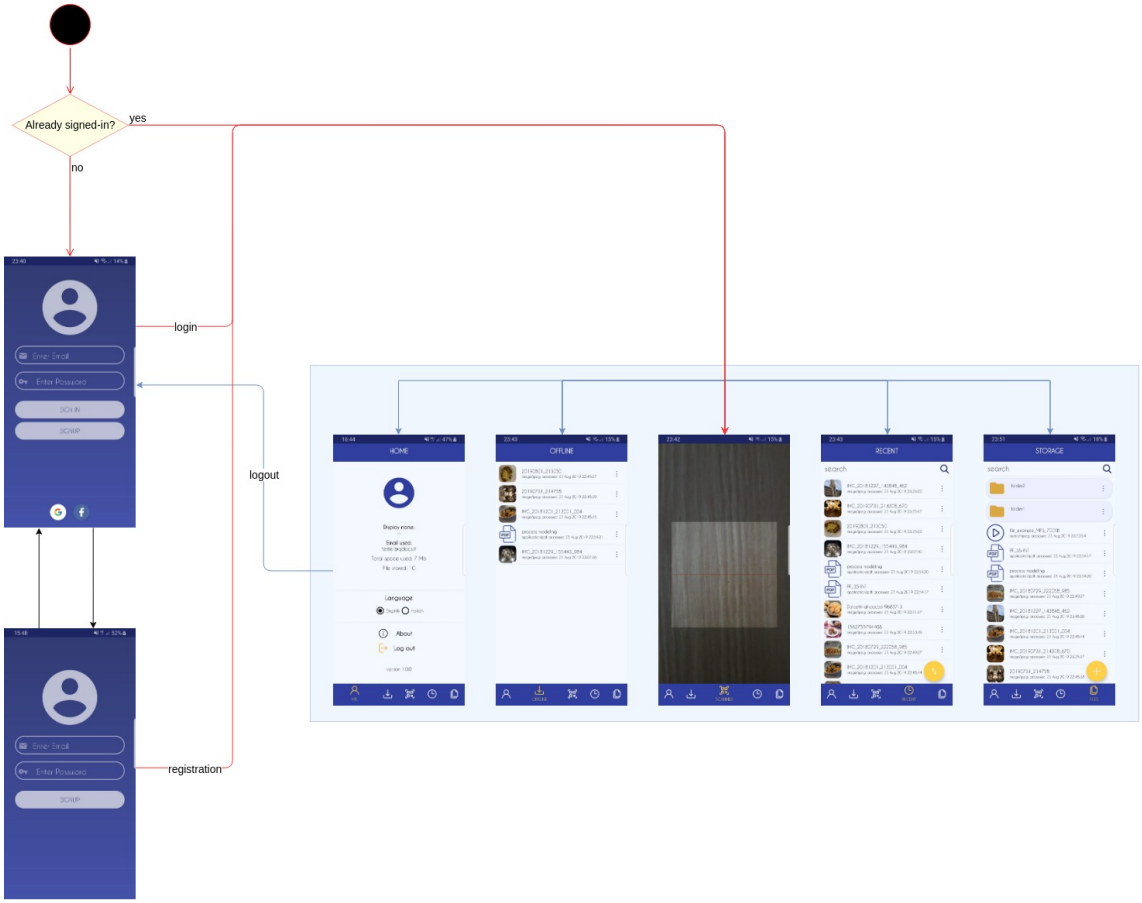

Figure 4.1: Application Start UI

- *offline files:* this screen provides a list of files stored offline, hence in the local storage of the device. Here the user can open a file (clicking on it) or open its settings (which show a simple dialog)
- *scanner:* here you can scan each QR-code you want, if it is associated to a file stored online, the application immediately, upon detection, starts a new screen showing the file.
- *recent files:* this provides a list of all online files, ordered by last access attribute, here the user can reverse the list as he want, open a file or open a file's settings and then performing some operations (i.e. save, delete, info, get QR-code)
- *storage:* this is quite similar to the previous one, here the online files are organized into directories (in according to what the user has done), hence here the user can also browse among them and upload new files into the cloud storage. As for recent files screen can open files, and their settings.

The *upload* operation initially opens a floating button menu where the user can choose which kind of file he want to upload, then it starts an external application (i.e. file picker) where the user can select his file, ones it was selected the external application ends, returning the file's data. Now the storage starts uploading the file on the cloud storage showing the progress into a dialog, where the user can perform the *cancel* operation , interrupting the upload. Ones the upload is complete the storage refresh its files' list. The *open file* operation differs in according to the type

of the file that has to be opened, if the file is an image or audio the application starts the corresponding screen that will show the file, instead, if the file's type is not recognized or it is different from audio/image it will start an external application (which one depend on the operating system).

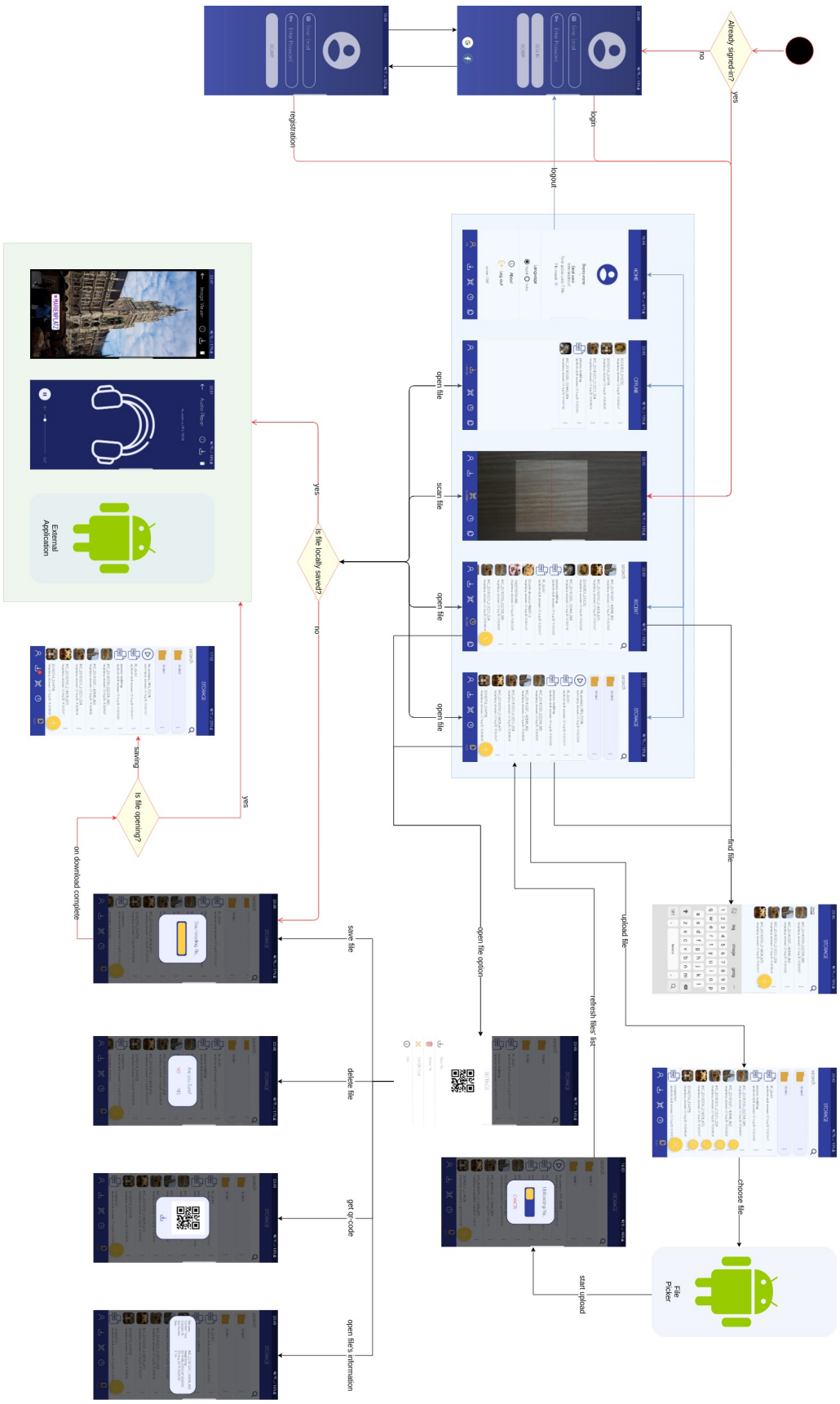

Figure 4.2: Complete UI Design

Chapter 5

Requirements Traceability

In according to the requirements defined in Use Cases chapter, we proceed providing which components, defined in Architectural Design chapter, are responsible to accomplish those requirements. The following table provides this traceability: the first column defines the use case, the second one the components that accomplish that use case and finally the third one provides the references to the class tables defined in Component Descriptions subsection.

Requirement	**Components**	**References**
Register	RegistrationActivity	3.7
Login	LoginActivity	3.6
Logout	HomeFragment, FirebaseHelper	3.17, 3.18
Find File	StorageFragment, RecentFilesFragment, OfflineFilesFragment	3.13, 3.15, 3.16
Open File	StorageFragment, RecentFilesFragment, OfflineFilesFragment, ShowImageActivity, PlayAudioActivity, GenericFileActivity, DownloadFileService, ShowFileReceiver	3.13, 3.15, 3.16, 3.9, 3.8, 3.10, 3.22, 3.21
Scan QR-Code	ScannerFragment	3.11
Upload File	StorageFragment	3.13
Create Directory	StorageFragment	3.13
Change Directory	StorageFragment	3.13
Save File	StorageFragment, RecentFilesFragment, OfflineFilesFragment, DownloadFileService, SaveFileReceiver	3.13, 3.15, 3.16, 3.22, 3.19
Save QR-Code	StorageFragment, RecentFilesFragment, OfflineFilesFragment, QrCodeDialog	3.13, 3.15, 3.16
Print QR-Code	StorageFragment, RecentFilesFragment, OfflineFilesFragment, QrCodeDialog	3.13, 3.15, 3.16
Open File Information	StorageFragment, RecentFilesFragment, OfflineFilesFragment	3.13, 3.15, 3.16
Delete File	MainActivity, StorageFragment, RecentFilesFragment, OfflineFilesFragment	3.12, 3.13, 3.15, 3.16
Change Language	HomeFragment	3.17

Table 5.1: Requirements traceability

Chapter 6

Implementation, Integration and Test Plan

This section identifies the order in which we planned to implement the sub-components of our system and the order in which we planned to integrate such sub-components and test the integration. So, in the following section we will explain all our decisions regard to the integration strategy, elements to be integrated and the testing sequence of components.

Integration Strategy Before starting the test phase, *QDocs* system has to satisfy the entry criteria, which is the set of condition(s) or requirements which are necessary and required to be met or fulfilled to create suitable circumstances and habitat favourable for testing. In our situation we identified the following entry criteria:

- Documentation in place, this means that the test started ones we have fully defined the application system, composed by all requirements that have to be satisfied (also defined by the use cases), system architecture and class interfaces (which defines classes behaviour and methods).
- After development phase, each module must pass through unit testing in order to outlines what faults have been found and which have been fixed. It gives the state of health about the system.

We have identified two different categories of tests, each of one focusing on a particular kind of classes:

- **Unit Test:** these are generally referred to as "local tests" or "local unit tests", they don't need a device or an emulator attached, since their scope is to test local classes' functionalities, for this reason they are not used to test UI but only local behaviour. These tests are implemented using Junit4 library, Mockito and Powered Mockito (in order to mock static methods)
- **Instrumentation Test:** Differently from the previous ones, these are tests that run on physical devices and emulators, and they can take advantage of the Android framework APIs and supporting APIs, such as AndroidX Test. Instrumented tests provide more fidelity than local unit tests, but they run much more slowly, hence we have applied these tests on the main important classes, that have to accomplish the more important use cases, like LoginActivity, RegistrationActivity, MainActivity and StorageFagment. They are mostly used to the the UI associated to those activities/fragments. These test are implemented using espresso freamwork and are runned on Firebase Test Lab

Unit Test From this category of tests we have basically considered those classes that have no relation with UI and Android classes (e.g. Activity, Fragment, Service, ResultReceiver etc.), in particular there are only four classes that have these preconditions:

- *MyFile:* this represent a file stored in the cloud storage, so we have tested its creation and its getter/setter method, in addition to this we have tested the 'duplicate' function that create an exact copy of the current file.
- *MyDirectory:* this is the simplest class implemented, since it acts as simple container of data. Here we have only tested getter/setter methods.
- *StorageElement:* from this class we have tested static methods that given a name/key and a list of StorageElements find whether there exist a file/directory with the given name/key.
- *Utility:* in this class we tested only those methods that does not affect/use Services or Activities, in particualr we have tested 'generateQrCode', 'generateCode' which generate the new code to associate to the new file. This class required PoweredMockito in order to mock static methods of the java Bitmap class.

Instrumentation Test As already said in this kind of tests we have focused on the UI test. All these test are based on the Espresso framework, which allows to write concise, beautiful, and reliable Android UI tests. Espresso tests state expectations, interactions, and assertions clearly without the distraction of boilerplate content. These test mainly follow the same architecture, they are based on retrieving some views from the screen (screen to which the Activity/Fragment is associated to), performing some actions on that views and checking graphical results. These tests are also be run on the Firebase Test Lab that allows executing the tests on different configuration devices. We focused on activity/fragment objects that accomplish the main important use cases:

- *RegistrationActivity:* this is required, in order to allow user to access *QDocs* services.
- *LoginActivity:* as for registration, the login is required and for this reason it is necessary and important to test.
- *MainActivity:* this is the activity in which the user stays for much time.
- *StorageFragment:* this is the more complex fragment, because it exposes more possible operation that the user can do (e.g. upload file, download file, create folder, change folder etc.)

Chapter 7

Tools Used

Server Application Firebase platform [2]

- *Authentication:* Firebase Authentication [3]
- *Database:* Realtime Database [4]
- *Storage:* Cloud Storage [5]
- *Backend Logic:* Cloud Functions [6]

Mobile Application

- *Screen mockup:* Marvel [8]
- *Source code:* AndroidStudio [9]
- *Code versioning and sharing:* Github [10]

Test

- *Unit testing:* JUnit [12], Mockito and Powered Mockito [13]
- *Instrumentation testing:* Espresso [14] and Firebase Test Lab [15]

Documentation

- *Text:* LaTeX [11]
- *Diagrams:* Drawio [7]

Bibliography

[1] Android Developers documentation.
https://developer.android.com/docs

[2] Firebase platform.
https://firebase.google.com/

[3] Firebase Authentication, which provides backend services, easy-to-use SDKs, and ready-made UI libraries to authenticate users to your app.
https://firebase.google.com/docs/auth

[4] Firebase Realtime Database, which is a cloud-hosted database. Data is stored as JSON and synchronized in realtime to every connected client.
https://firebase.google.com/docs/database

[5] Cloud Storage for Firebase, that is a powerful, simple, and cost-effective object storage service built for Google scale.
https://firebase.google.com/docs/storage

[6] Cloud Functions for Firebase, which let you automatically run backend code in response to events triggered by Firebase features and HTTPS requests.
https://firebase.google.com/docs/functions

[7] Drawio, online free diagram editor software.
https://www.draw.io/

[8] Marvel, the all-in-one platform powering design adopted for mobile application mockup.
https://marvelapp.com/

[9] AndroidStudio, Integrated Development Kit for Google's Android operating system, built on JetBrains' IntelliJ IDEA software and designed specifically for Android development.
https://developer.android.com/studio

[10] Github, company that provides hosting for software development version control using Git.
https://github.com/

[11] LaTeX, high-quality typesetting system; it includes features designed for the production of technical and scientific documentation.
https://www.latex-project.org/

[12] JUnit, a unit testing framework for the Java programming language.
https://junit.org/junit5/

[13] Mockito, an open source testing framework for Java released under the MIT License. The framework allows the creation of test double objects (mock objects) in automated unit tests.
https://site.mockito.org/

[14] Espresso, a framework to write concise, beautiful, and reliable Android UI tests.
https://developer.android.com/training/testing/espresso

[15] Firebase Test Lab, a cloud-based app-testing infrastructure. With one operation, you can test your Android or iOS app across a wide variety of devices and device configurations, and see the result.
https://firebase.google.com/docs/test-lab

[16] Unified Modeling Language.
https://www.uml.org/