

Contents

[INTRODUCTION 3](#_Toc13828558)

[PURPOSE 3](#_Toc13828559)

[SCOPE 3](#_Toc13828560)

[GLOSSARY 4](#_Toc13828561)

[Definition 4](#_Toc13828562)

[Acronyms 4](#_Toc13828563)

[DOCUMENT STRUCTURE 5](#_Toc13828564)

[ARCHITECTURAL DESIGN 5](#_Toc13828565)

[OVERVIEW 5](#_Toc13828566)

[SELECTED ARCHITECTURAL STYLE AND PATTERNS 6](#_Toc13828567)

[Server 7](#_Toc13828568)

[Storage 8](#_Toc13828569)

[Client 8](#_Toc13828570)

[STRUCTURE LOGIC 11](#_Toc13828571)

[COMPONENT VIEWS 15](#_Toc13828572)

[RUNTIME VIEWS 23](#_Toc13828573)

[Login 24](#_Toc13828574)

[Logout 24](#_Toc13828575)

[Registration 24](#_Toc13828576)

[Upload 24](#_Toc13828577)

[Download 24](#_Toc13828578)

[Create Directory 24](#_Toc13828579)

[Scan File 24](#_Toc13828580)

[ALGORITHMS DESIGN 24](#_Toc13828581)

[USER INTERFACE DESIGN 24](#_Toc13828582)

[REQUIREMENTS TRACEABILITY 25](#_Toc13828583)

[IMPLEMENTATION, INTEGRATION AND TEST PLAN 25](#_Toc13828584)

[REFERENCES 25](#_Toc13828585)

# INTRODUCTION

## PURPOSE

The purpose of the Design Document (DD) 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 entire document.

The technical aspects of the **QDocs** implementation start in the second chapter, titled << ARCHITECTURAL DESIGN >>.



Figure : QDocs logo

## SCOPE

The **QDocs** [Figure 1] app 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:

1. << 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. >>
2. << 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 you 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 separated such that you cannot access other user’s files.

[[ LOGO QDocs ]]

## GLOSSARY

### Definition

* **QDocs:** Mobile application.
* **Cloud**: Physical storage located somewhere that can be accessed through a network connection.
* **Client/User:** Person that uses the mobileapplication.
* **Firebase:** Mobile development platform.
* **Internal Storage:** This refers to the internal storage of the device that is using the application.

### Acronyms

* **DD:** Design Document.
* **API**: Application Programming Interface.
* **DB:** Database.
* **UML:** Unified Modeling Language.
* **MVC:** Model View Controller pattern.

## DOCUMENT STRUCTURE

The structure of this document is the following:

* **INTRODUCTION,** in this part we basically introduce the application itself specifying, which is its goal, and providing some simple, real-world, usage examples.
* **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 only on the client-side (mobile app).
* **ALGORITHM DEISGN,** this chapter provides a more detailed analysis about some interesting algorithms implemented client-side.
* **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 lifecycle of the application from the point of view of the screens.
* **REQUIREMENTS TRACEABILITY,** this chapter provides the list of requirements that this application was thought to satisfy providing also which are the specific classes that are in charge to satisfy them.
* **IMPLEMENTATION, INTEGRATION AND TEST PLAN,** in this part we focus the attention on the testing part of the development, analysing how the tests were performed.
* **REFERENCES,** this is the last part of the document, which describes the programs and tools used for developing the application system and this document.

# ARCHITECTURAL DESIGN

## OVERVIEW

The goal of this chapter is to analyse and describe the architecture implemented in the **QDocs** application. This chapter is structured as follow:

* SELECTED ARCHITECTURAL STYLE AND PATTERNS: this first paragraph introduces the general architecture used for developing the application, analysing all parts involved and their connection/communication.
* APPLICATION STRUCTURE: in this part we focus the attention on the graphic representation of the whole architecture, with different levels of granularity, through UML diagrams.
* COMPONENT VIEWS: this paragraph provides a more detailed analysis on the class structure of the mobile application, providing all necessary class diagrams and highlighting all the classes’ interactions.
* RUNTIME VIEWS: this paragraph provides a more detailed analysis on the interactions-flows for the more important action that can be performed in the application, such as login, registration, scanning, etc.
* ALGORITHMS DESIGN: this paragraph provides a description of the more significant algorithms implemented in the developing of the application, such as the update list algorithm, etc.
* USER INTERFACE DESIGN: the goal here is to provide the whole lifecycle of the application’s screens through apposite diagrams
* REQUIREMENTS TRACEABILITY: this paragraph provides all **QDocs’s** requirements analysing in detail which are the activities that are in charge to guarantee them.
* IMPLEMENTATION, INTEGRATION AND TEST PLAN: here all information about how the testing is performed are provided.

## SELECTED ARCHITECTURAL STYLE AND PATTERNS

The **QDocs** application was developed using the 3-tier client-server architecture [Figure 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: Occupies the top level and displays information related to services that the system can provide. This tier only communicates with the application tier through internet request/response. 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: Realtime Database and Cloud Storage.

Immagine che contiene screenshot

Descrizione generata automaticamente

Figure : General 3-tier architecture

In the specific case of **QDocs** application the Presentation Tier corresponds to the Android mobile application, the Application Tier to the Firebase server and the Data Tier to the Realtime Database and Storage.

The following sub-paragraphs will provide more details about how each tier was implemented for the **QDocs** application.

### Server

The server represents the main backend logic, it allows multiple users to interact with it simultaneously and keeping them separated. From the technical point of view, the software was developed using the Firebase development platform providing a set of APIs that hides to the client all the backend logic.

The following list provides all the APIs included in the server:

* AUTHENTICATION API: the server can directly handle the authentication of the users or forward the authentication mechanism to other external services like Facebook and Google, in both cases provides a set of APIs that allows users to make request for registering, logging-in, etc.

It stores all user’s information such as usernames, passwords, personal image and so on.

* CLOUD STORAGE API: since in the server a Cloud Storage service is included, it provides a set of APIs that allows users to interact with their own cloud filed such as upload, download, create new folders and so on.
* REALTIME DATABASE API: since making request directly to the storage may require too much time, the idea was to provide also a Realtime Database service that is queried by the client whenever it want to retrieve static information about the stored files in the cloud, so without updating anything. For this reason, the server provides a set of APIs used for listening on database information.
* CLOUD FUNCTIONS API: backend logic used to react on files upload/delete operation and keeping the Realtime DB updated.
* LOGGING: It logs all the interaction between users and the server.

[[ Firebase image ]]

### Storage

The storage layer is provided by the Firebase infrastructure (Google cloud) and in our case is basically composed by two main parts:

* CLOUD STORAGE: this is basically the cloud location where all the users’ files are stored, this is provided by the Google cloud.

[[ storage structure image ]]

* REALTIME DATABASE: not relational database provided by Firebase that is directly associated to the cloud storage and kept congruent with it, it allows a faster access to the stored data for read-only operation.

[[ database structure image, tree ]]

### Client

The client is represented by the Mobile Application itself [**QDocs**], it is a Native mobile app implemented in Java for the Android platform.

From the architectural point of view the app was developed following the MVC pattern (Figure 2), as you can see this pattern is composed by 3 main parts:

* MODEL: The central component of the pattern. It is the application's dynamic data structure, independent of the user interface. It directly manages the data, logic and rules of the application.

In this specific case contains the model of files, directories, users and so on, and the main logic used for some processes like authentication, downloading, uploading and so on, that are kept separated from the part responsible of manage the graphic of the screens.

* VIEW: The view means presentation of the model in a particular format. In this case represents the screen appearance coded in xml files. An xml file is static representation of the app’s screen or of a simple component, that can be reused more times.

For each screen of this app there is at least one xml file representing it, in some case more than one file are used for a clearer representation and reuse.

* CONTROLLER: The controller responds to the user input and performs interactions on the data model objects. The controller receives the input, optionally validates it and then passes the input to the model. These elements are represented by all the Activity/Fragment objects where each of them is responsible to inflate a specific xml file.



Figure : General Model-View-Controller pattern

Differently from the general MVC pattern, here, there is not so great distinction between view and controllers because in this case it is the controller itself that listen to user’s action/event and react accordingly. In this case the view is basically the static part inflated when the screen is loaded in the memory and all the changes that may occur to the screen are directly handled by the controllers themselves. So for this reason you can see the true pattern implemented in **QDocs** application in [[ FIGURE ]] .

[[ MVC applied to QDocs ]]

#### Controllers

In this specific context the controllers are represented by Activity/Fragment objects. Each controller is a class responsible of creating the screen associated to it, the invocation of lifecycle-methods is directly controlled by the OS through a callback model. In the following [Figure 3] and [Figure 4] you can see the lifecycle of Activity and Fragment objects respectively.

The following diagrams show the important state paths of an Activity/Fragment. The square rectangles represent callback methods you can implement to perform operations when the Activity/Fragment moves between states. The coloured ovals are major states the Activity/Fragment can be in.

Immagine che contiene testo, mappa

Descrizione generata automaticamente

Figure : Activity lifecycle

Immagine che contiene screenshot

Descrizione generata automaticamente

Figure : Fragment lifecycle

Quite all the interaction between the mobile application and the Firebase Database are managed in according to an event-listener model: the application basically retrieves the database reference through the Firebase API and attach to it a listener, this listener provides some methods that are directly called in that context when the event occur (i.e. when the user upload a new file on the storage, its data is simultaneously added in the database and if you have the listener “onChildAddedEventListener” on the DB reference, its method will be execute).

You can find a more detailed analysis of the code in this section: ALGORITHMS DESIGN

[[ example of event-listener ]]

## STRUCTURE LOGIC

In this paragraph we will provide some diagrams that try to explain how the whole system was developed, providing all information about how the communication among objects inside the system are performed and handled.

Immagine che contiene mappa, testo

Descrizione generata automaticamente

Figure : Whole system component diagram

From the implementation point of view the structure of the application is a simple Client-Server architecture [fig. ]:

* CLIENT: that is represented by the mobile application (**QDocs**) that basically communicates with other parties through external/internal interfaces, in specific it is able to store and retrieve data from the internal storage of the mobile device through the FileProvider interface provided by the OS, it is able to perform cloud operation (toward the server) using custom interfaces provided by the server itself (Authentication API, Cloud Storage API, Realtime Database API)
* SERVER: this part is logically represented by the firebase server that provides all the services needed by the mobile application, such as authentication, storage and database services. In order to provide these services, it leans on the usage of other external interfaces like Facebook and Google. All the interaction between the server and the database/storage are handled internally in the server. The Cloud Function service is used by the server itself to keep database and storage consistent with each other.

T the following diagram focuses the attention on the internal structure of the **QDocs** mobile application, highlighting which are the activities in charge to communicate with external interfaces.

Immagine che contiene screenshot

Descrizione generata automaticamente

Figure : QDocs composite structure

As described in the [Client] paragraph the **QDocs** application was developed following the MVC pattern, for this reason, as you can see in the [fig. ] the application structure is split into 3 main parts: Views, Controllers and Model:

* VIEWS: This part is represented by a collection of xml files representing all the screens layout of the application itself, they represent the static part of the screens. They are not in charge to handle dynamic operations.
* CONTROLLERS: This part contains all the Activity objects implemented in **QDocs**, these activities are in charge to inflate the layouts and to handle all the user-actions triggered on the screens, modifying the interfaces accordingly.

The MainActivity is not a simple Activity object but a FragmentActivity one since it handles different screens (layouts) allowing to the user a more interactive usage of the application. For a more detailed analysis see [fig. ].

* MODEL: this part contains the main logic of the application, in the [fig. ] you can see an extract of used classes (for a more detailed analysis refer to the [COMPONENT VIEWS] paragraph). These classes are in charge to communicate with external interfaces provided by the server and ‘internal’ ones provided by the OS.

Now a more detailed analysis of the Controller Activities is provided, describing what each class is design for.

**LOGIN ACTIVITY**

This Activity object allows the user, if already registered to this application, to login int the application. The user can decide to login in three way: with Google account, with Facebook account or with email-password.

**REGISTRATION ACTIVITY**

This is a simple Activity that allows user to register themselves to the application, providing personal email and password.

Immagine che contiene screenshot

Descrizione generata automaticamente

Figure : MainActivity structure

**MAIN ACTIVITY**

The [fig. ] provides the internal structure of the MainActivity, it is basically structured as a container that shows a different fragment in according to what the user has to do, keeping the state of all five fragments. When the Activity is created it instantly create all the fragments and it starts showing the scanner one. In the following list we will provide what each fragment is used for:

* HomeFragment: This fragment represents the user’s home, where some account’s information is stored, such as display name, email used, cloud space used and so on. It provides also some option like language to use, logout and app information.
* OfllineFilesFragment: This fragment is in charge to show to the user all the files stored I the internal storage (so called offline files). All the files interaction in this section affect only the internal storage, for instance if we remove a file here, the file will be removed from the local storage of the mobile device.
* ScannerFragment: This is the most important fragment; it is the one opened each time the application is restarted. This fragment provides a scanner tool used to scan the QR codes and if the scanned one matches one in the user’s storage it directly opens the associated file.
* RecentFilesFragment: The implementation of this fragment is quite like the OfflineFilesFragment one, in this case the fragment shows all the files stored in the cloud, independently from their directories, ordered by last access. The user can select between ascending and descending order.
* StorageFragment: This represent the cloud storage of the user where all the files are stored, here the user can interact with all the files, retrieving their information and so on, he can upload new files and create new directories.

**PLAY AUDIO ACTIVITY**

This Activity is in charge to handle recognized audio files, it can play them.

**SHOW IMAGE ACTIVITY**

This Activity open images inside the application, it provides some operation on the images themselves like delete, it supports the zoom image.

**GENERIC FILE ACTIVITY**

This Activity handles the opening of not recognized file and other files that are not neither images and audio. It redirects the opening operation to external application, if any.

## COMPONENT VIEWS

This paragraph will provide a more detailed analysis on the mobile application structure, providing information about the classes used and how these classes interact together.

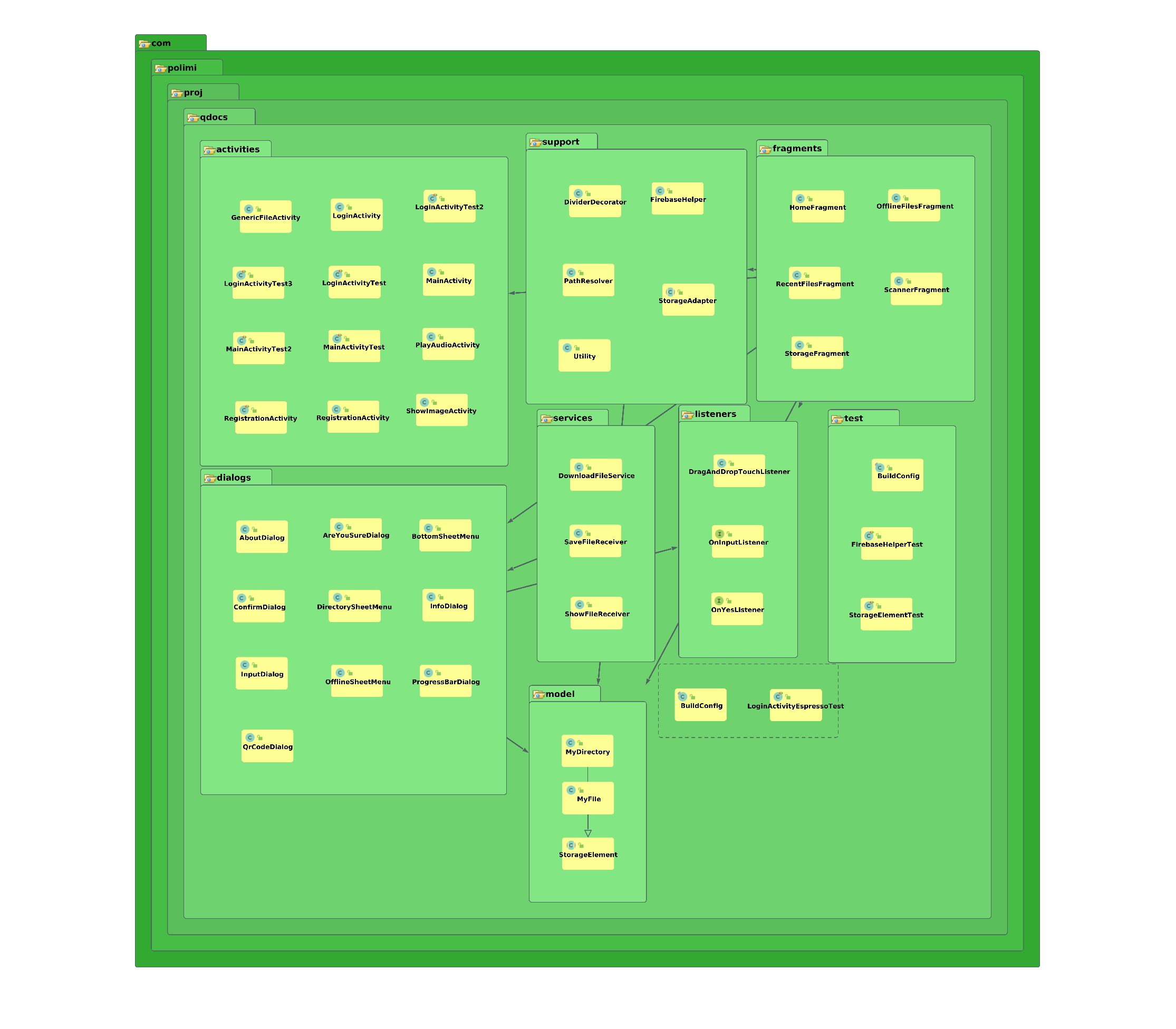


Figure : Package organization of the QDocs classes structure

The [fig. ] provide a diagram describing how all the classes are organized in the project: the project is split into 7 packages:

* Activities: this package contains all the Activity classes used for development of **QDocs** application, for every Activity an xml file is associated except for the MainActivity that, as already described in the STRUCTURE LOGIC section, manages different Fragments, so it indirectly manages more layouts.
* Fragments: This package contains all fragments that are managed by the MainActivity
* Services: This includes a Service class and two Receiver classes, the former represent a background service used for download file through the internet and the latter are receivers used to capture and process the data generated by the service, allowing Activities to modify the layouts.
* Dialogs: This is the container of all the Dialog classes, here each class is associated to a dialog layout, so basically all the classes manage different dialogs.
* Support: This package contains all support classes, which are classes used by Activities that basically separate some work from the Activities themselves.
* Models: This package contains classes used to model the data stored I the database, in this case Files and Directories. The StorageElement class is the superclass of both files and directories
* Listeners: This includes some interfaces used by Activities

The [fig. ] represent the whole class diagram showing the more important relation among them.

In the following diagrams we will show the relationship established by the main important activities in the project.

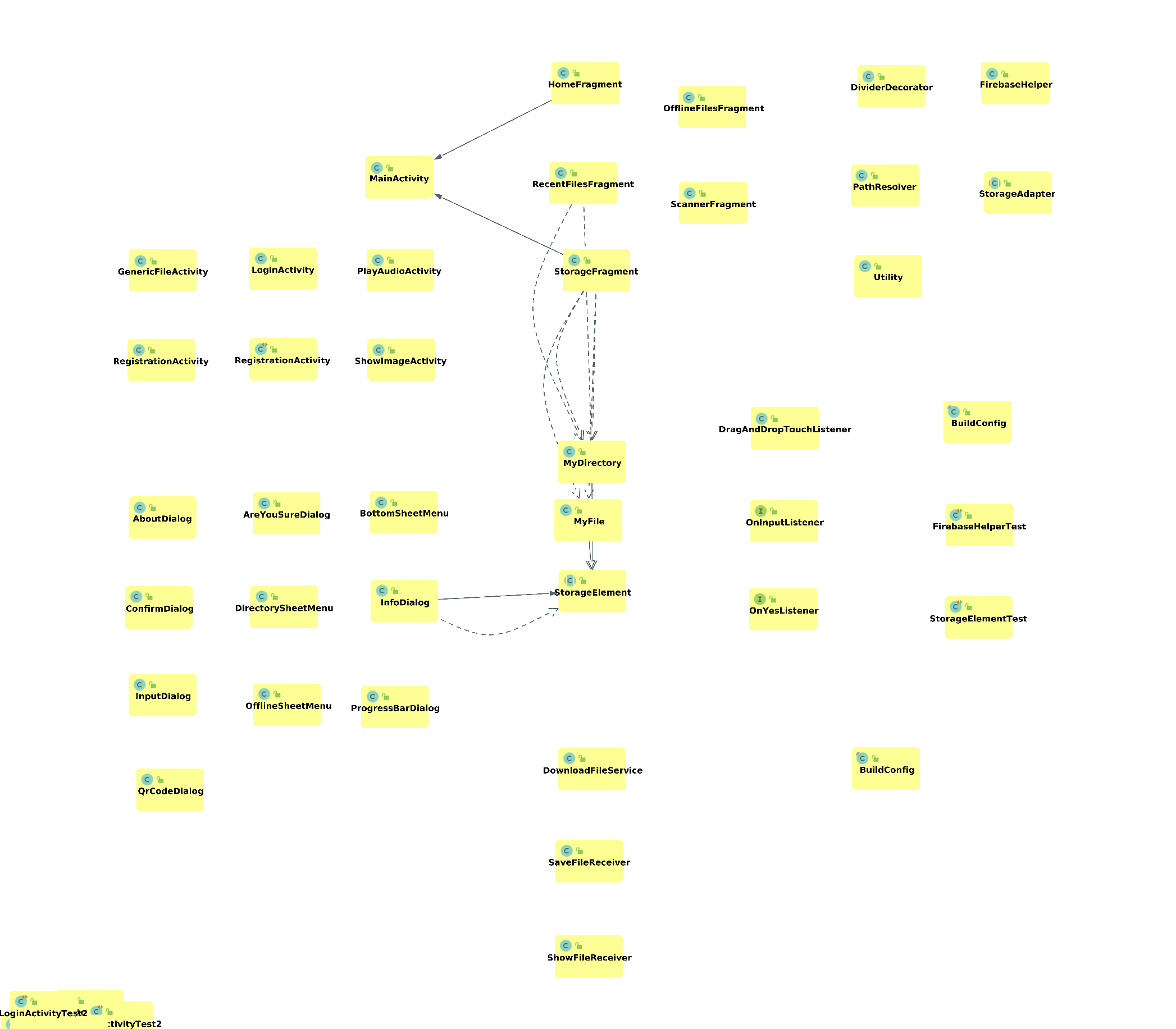


Figure : QDocs class diagram

[TODO: change image ^ ]

Immagine che contiene mappa

Descrizione generata automaticamente

Figure : MainActivity interactions

Immagine che contiene testo, mappa

Descrizione generata automaticamente

Figure : ScannerFragment interactions

Immagine che contiene testo, mappa

Descrizione generata automaticamente

Figure : StorageFragment interactions

Immagine che contiene testo, mappa

Descrizione generata automaticamente

Figure : MyFile interactions

Immagine che contiene mappa, testo

Descrizione generata automaticamente

Figure : MyDirectory interactions

## RUNTIME VIEWS

This paragraph, through sequence diagrams, simply depicts interaction between objects in a sequential order i.e. the order in which these interactions take place. In order to show these interactions some **QDocs’** functionalities are analysed (i.e. login, registration, upload and so on.)

### Login

### Logout

### Registration

### Upload

### Download

### Create Directory

### Scan File

# ALGORITHMS DESIGN

[[ provide examples of algorithm implemented ]]

# USER INTERFACE DESIGN

In this section we provide a less technical analysis of the **QDocs** application flowchart, from the screens point of view:

As you can see in [fig. ] when the app is launched the first step [1] is to check whether the user is already logged in or not, suppose that the user has never logged in, he/she directed into the login page. Here [2] the user can decide to login, if he/she has already an account on **QDocs** or he/she want to login through facebook or google,otherwise can register to it by clicking on the ‘register’ button and providing the email and password that he/she will later use to login!

Suppose now that the user has the account and open the application, he/she is directed into the Scanner page, in this page he/she can scanner their own QR codes or can switch among all pages simply by swiping left and right the screen. Among these pages there is the Home one [4], that includes some information about your account (e.g. display name, email, total space used, number of stored files) and some options (e.g. choose language, about and logout), by clicking on the logout button the user is directed into the login page. From Storage, Offline, Recent and Scanner pages you can open your files, in different ways in according to the pages [5], this action can open 3 different pages in according to the format of the file: if audio opens the Play Audio page, if image the Show Image page otherwise it opens external application that are able to open that file.

Immagine che contiene screenshot

Descrizione generata automaticamente

5

3

4

2

1

Already logged in.

Not yet logged in.

Figure : QDocs' screens flowchart

# REQUIREMENTS TRACEABILITY

[[ use case diagram ]]

[[ describe which activity is associated to each requirement ]]

# IMPLEMENTATION, INTEGRATION AND TEST PLAN

[[ describe how tests are performed ]]

# REFERENCES