České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra počítačů

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: Špatný Jakub

Studijní program: Otevřená informatika
Obor: Softwarové inženýrství

Název tématu: Mobilni aplikace pro správu a sběr dat z testů použitelnosti v reálném prostředí

Pokyny pro vypracování:

Proveďte analýzu potřeb při testech použitelnosti, zejména způsob sběru dat, jejich dostupnost a použití. Dále proveďte analýzu dostupných NoSQL databází a vyhodnoťte vhodnost jejich použití na základě analyzovaných potřeb. Výběr NoSQL konzultujte se zadavatelem. Na základě analýzy navrhněte vhodnou strukturu databáze a její použití jak pro textová tak pro binární data.
Implementujte prototypy dvou aplikací na platformě Android. První aplikace bude určena pro správu testů, účastníků testů a přehled dat v testech. Druhá aplikace se zaměří na sběr dat dle požadavků testů použitelnosti, tedy zejména textová data a video záznam, včetně možnosti náhledu.
Návrh uživatelského rozhraní aplikace ověřte ve vhodných testech použitelnosti. Kód aplikací testujte pomocí metod testování software.

Seznam odborné literatury:

[1] M. Jones, G. Marsden, Mobile Interaction Design, Wiley, 2006
[2] I. Malý, Analysis of Usability Tests with Context Model, Praha: 2012. PhD Thesis. České vysoké učení technické v Praze, Fakulta elektrotechnická.
[3] N. Marz, J. Warren, Big Data: Principles and best practices of scalable realtime data systems, 2015, Manning Publications

Vedoucí: Ing. Ivo Malý, Ph.D.

Platnost zadání do konce letního semestru 2017/2018

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta elektrotechnická
L.S.
4

prof. Dr. Michal Pěchouček, MSc.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 16.2.2017

Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science

Master's Thesis

Mobilní aplikace pro správu a sběr dat z testů použitelnosti v reálném prostředí

Mobile applications for management and data collections in field usability studies

Jakub Špatný

Supervisor: Ing. Ivo Malý, Ph.D.

Study Programme: Open Informatics

Field of Study: Software Engineering

May 23, 2017

Aknowledgements

I would like to thank Ing. Ivo Malý, Ph.D. for supervising this thesis, and my friends and family for their support during my studies.

Declaration

Abstract

The goal of this thesis is to create mobile applications for management and data collection in field usability studies. In the beginning, the usability testing process is discussed, followed by a requirement analysis of the new system. Later, an overview of available NoSQL databases is presented, and the Firebase platform is selected to be used. The mobile applications are first prototyped, and then implemented for the Android operating system. Finally, the system is verified using both unit tests and usability testing with real users.

Keywords Android, Mobile application, NoSQL database, Firebase, Usability testing

Abstrakt

Cílem této diplomové práce je vytvoření mobilních aplikací pro správu a sběr dat z testů použitelnosti v reálném prostředí. Práce nejprve představuje proces uživatelského testování použitelnosti, po nemž následuje analýza požadavků na navrhovaný systém. Následně je provedena rešerše dostupných NoSQL databází a platforma Firebase je zvolena jako datové úložiště nového systému. Dále jsou vytvořeny prototypy mobilních aplikací, na jejichž základě jsou implementovány aplikace pro operační systém Android. Nakonec je systém verifikován s využitím jednotkového testování a testování uživatelské použitelnosti se skutečnými uživateli.

Klíčová slova Android, Mobilní aplikace, NoSQL databáze, Firebase, Uživatelské testování použitelnosti

Contents

List of Figures

Chapter 1

Introduction

In software engineering, the development of a new software product usually follows a selected software development process methodology. Although, the concrete steps and their succession differs in each methodology, the following core actions can be identified in most: analysis, design, implementation, verification, and deployment. In this thesis, we will focus on only a small, essential, yet often overlooked part, which can be found within the design and verification steps: the usability testing.

Usability testing refers to an evaluation process of a product by testing it with representative users with a goal to identify possible usability problems and determine the participant's satisfaction with the product [59]. In the past, the implementation of new technical features often had priority over the user interface design. Nowadays however, the usability of application's user interface is an important competitive advantage on the market [13].

The testing of desktop applications, which are used in simple and stable environments, can be done inside a usability laboratory using one of many available tools for this purpose. However, due to the high adaption of smartphone devices among users in recent years, the development of mobile applications that are used in mobile environments and often make use of the user's context, such as his/her location, has also progressed. To reveal real usability problems within such applications, the test environment should resemble the designed context of use as much as possible [13]. One way to achieve that is to perform the tests in the real environment. Unfortunately, the currently available usability tools primarily focus on testing in laboratory settings and fail to sufficiently support usability testing in the field.

1.1 Goals and motivation

The goal of this thesis is to design and implement a system with client mobile applications for the Android operating system that would allow the management and collection of data during usability testing in the real context of application use. The system should be designed in a way to allow storage of wide range of data types, and be flexible to enable further extension of the supported types in the future if needed.

When finished, this system could provide a strong base for usability testing of projects within the Czech Technical University in Prague, resulting in better user experience of products used by the students as well as the staff.

1.2 Thesis structure

In chapter 2, the process of usability testing is discussed, followed by an overview of existing usability testing tools, and finally, the requirement analysis for the new system is presented. In chapter 3, the system's domain and use case models are discussed, available NoSQL database systems are researched, and prototypes of both mobile applications are created and tested. Chapter 4 deals with the configuration of a previously selected NoSQL database. In chapter 5, the implementation details of the client Android applications will be described. Afterwards, testing of the implemented system will be discussed in chapter 6. Finally, chapter 7 concludes this thesis.

Chapter 2

Problem description and requirement analysis

In this chapter, I will describe the process of usability testing in greater detail, point out the main problems and obstacles, present an overview of existing tools for usability testing, and finally, specify the requirements for the new system to be designed.

2.1 Usability Testing

Usability testing is a usability evaluation method, which employs users from selected target audience as testing participants to evaluate whether and to which extent an application meets specified usability criteria [21]. There are also other usability testing techniques that don't make use of representative users, such as expert evaluation, cognitive walk-through or heuristic evaluation. However these methods are outside the focus of this thesis and won't be further discussed.

Nielsen [18] defines five usability attributes which can be used in a systematic approach to evaluate and further improve the product under test as follows:

Learnability It should be easy to learn to work with the system in order to accomplish tasks the user wanted to perform in the first place.

Efficiency Once the user has learned the system, it should allow to carry out the tasks quickly and efficiently.

Memorability The user should be able to easily remember the system, so that after some period of time of not having used the system, he/she would recall the ways to carry out tasks without the need to learn the system again.

Few and Noncatastrophic Errors Users should make few errors while using the system. If an error is encountered, the users should be able to easily recover from it.

Satisfaction The users should be *subjectively* satisfied when using the system.

During the usability test, selected participants use the system in form of the application itself or just its prototype to perform a predetermined set of tasks [13, 18] while the participant and his/her actions are observed and/or recorded by test moderator or observer. These observations are later analyzed in order to get insights into user's usability problems [13].

2.1.1 Usability Test Workflow

The usability testing process consists of five sequential steps [13, 21]. Each of the steps contains a set of actions, which must be finished before proceeding onto the next step. However, unlike the process steps, these actions can be performed in parallel.

Planning

First of all, we must develop a test plan, which is the foundation for the entire test. At the end of this step, the purpose, goals, and objectives of the test should be defined. The test plan should also cover research questions, participant characteristics for participant recruiting, chosen usability method, and the test setup. At last, it must be clear what data is to be collected and what measures are going to be used for its evaluation [21].

Preparation

Once the test plan is complete, test practitioners can start setting up a testing environment. The test practitioners have to decide on a location and space for the test based on the advantages and disadvantages of several criteria, such controllability and observability of the environment, realism of the environment, participant availability, or the cost [13].

Meanwhile, the test participants can be selected from target audience based on the participant characteristics defined in the previous step using a screening questionnaire.

Finally, the remaining sub-step of test preparation is the development of test materials, which will be used to communicate with the participants, collect data, and fulfill possible legal requirements. The test materials vary from test to test, but the most commonly contained materials are an introduction script and debriefing guide, data collection instruments, pre-test and post-test questionnaires, task scenarios, as well as non-disclosure agreements (NDAs) and recording consent forms [21].

Execution

Having completed the planning and preparation, the test can finally be conducted according to the test schedule. While the participant is solving presented tasks, the test practitioners are observing his/her actions and recording data. To achieve further explanation of usability problems, the participant can be asked to complete pre-test and post-test questionnaires before, respectively after the test itself was performed. At last, the participant is debriefed and the session closed.

Data and Observation Analysis

After the test execution has been completed, it is time to begin compiling, summarizing and analyzing the data collected in the previous step. The data analysis can be divided into two distinct processes with two different deliverables [21].

First, a preliminary analysis is used to quickly discover the worst problems (also called *hot spots*), which can be fixed immediately by the application designers without needing to wait for the final report. Secondly, a comprehensive analysis is carried out during the 2 to 4-week period after the test to provide deeper analysis of possible usability problems.

Reporting

The last step consists of developing findings and recommendations, and presenting them in form of a written report, which usually consists of two parts. The first part is an output of the preliminary analysis, which covers only the most important usability problems. On the other hand, the second part thoroughly describes the test, as well as all of the identified usability problems and recommendations for improvement [21].

2.1.2 Test Environments

When planning and conducting a usability test, it is necessary to decide on the environment in which the test will take place. Each environment has its advantages and disadvantages in respect to the controllability of the environment, possibility to collect desired data, and realism of the environment (ecological validity of the test) [13].

Usability Laboratory

A usability laboratory setup consists of two adjoining rooms. One, designed as the testing room, contains a device running the application under test, an intercom and speakers for communication, as well as cameras and other devices needed for the test data collection. During the test, this room seats the participant and possibly the moderator, who can however be seated in the second room and communicate with the participant through the intercom system instead. The other room is intended for observation and controlling of the test and therefore contains equipment for data recording and test observation [21]. Possible setup of the usability laboratory can be seen in Figure 2.1.

Usability laboratory is a perfect example of highly controllable environment, which is suited for recording a large variety and amount of data. On the other hand, a laboratory is an artificial environment which can suffer from the lack of ecological validity that can lead to failure in finding real usability issues, which could have been found having conducted the test in the real context of use [13].

Field Testing

For specific context-sensitive applications, simulating complex context environment may be difficult inside the artificial environment provided by a usability laboratory. For this reason,

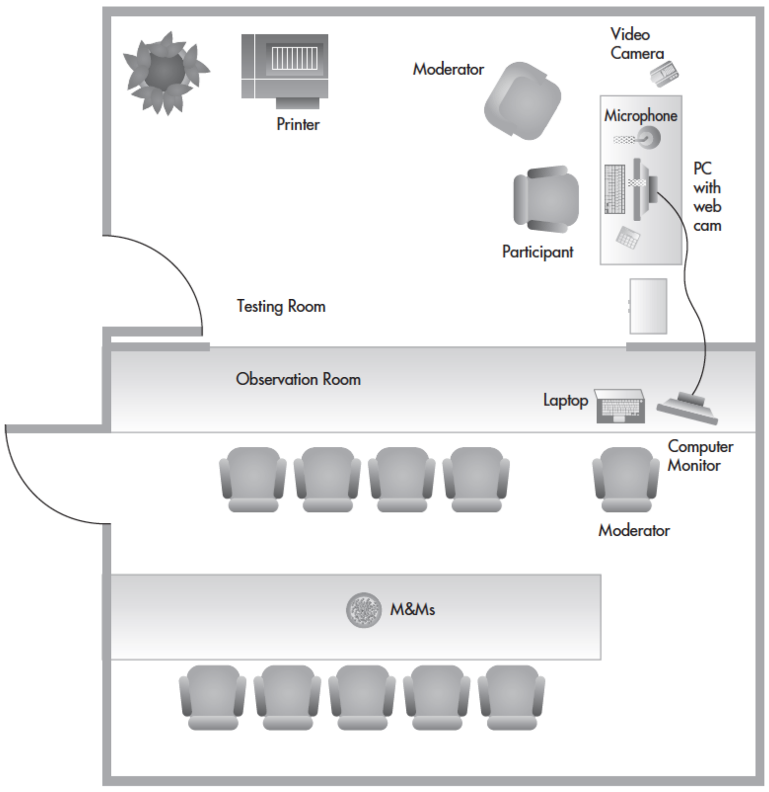


Figure 2.1: Classic laboratory setup [21]

it may be more fitting to execute the test in real context of application use by testing in the field. However, it is important to note that although the realness of the environment and the ecological validity of the test increases, the test controllability and the amount and types of collected data can decrease, leading to problems when identifying the cause of usability problems during data analysis [13].

2.1.3 Data Sources in Usability Testing

During a usability test, data of different types is recorded from various sources with each participants. Each data source produces one or more *data sets*. All data sets collected during a test with one participant can be referred to as *test participant data sets* [13].

The type of a data set usually falls into one of the two following categories. First, logs represent discrete data records for specific events in time, which can be recorded either automatically by a logging application or semi-automatically or manually by test practitioners. The second category, recordings, represents audio and video streams of data.

Audio/video recordings together with observer notes are the most popular data sources in most usability tests. These and other frequently recorded data sources will be further discussed in the following sections.

Audio/Video

In almost every usability test, audio and/or video is recorded, as it is usually the best data source for the analysis of usability problems [13]. The screen of the application is recorded in order to analyze user interaction within the application, as well as the application's current state. The participant can also be video recorded in order to analyze facial and body expression, which may reflect user's satisfaction. To record participant's comments an audio recordings can be used.

Observer Notes

Observer notes is a log of textual information, where test observers can describe events that happened during the test execution and might point out possible usability problems. Each event description typically contains a timestamp, which can later be used to synchronize the log with audio and video recordings.

Questionnaires and Interviews

Before and/or after the test, questionnaires and interviews can be used to further document information about test participant or the test itself. Before the test, the participant can for instance be questioned about his/her habits, background and experience. After the test, information about encountered usability problems and subjective evaluation may be collected.

Input Device and User Interface Events

Apart from recording video of the application's screen, input device events such as mouse clicks and movements, and keystrokes can be recorded using logging tools that are available for most testing platforms on operating system level [13]. User interface events within the application, such as clicking a button or filling out a form, are a result of the input device events and can therefore be derived from them.

2.2 Existing solutions

This section provides an overview of applications, which can be used for data collection and/or analysis during usability testing. For each application, its functionality, advantages, disadvantages, and the possibility of use for testing in the field were researched.

2.2.1 TestIT

TestIT was created as part of a bachelor's thesis [6], and consists of two Android client applications. The first (logger) application is meant to be used by test administrator to define basic test information such as test name, name of tested application, participants, and pre-test/post-test questionnaires. The admin application also enables to create a log for each test, where timestamped events with text, sketch or photo can be added, however the sketch and photo files are only locally stored and not synchronized to the remote server. Moreover, test data is only synchronized before and after the test, nothing is transferred while the test is running. On the other hand, the application can work without an Internet connection, making it useful for testing in the field. The second (data collector) application, which is to be run on test participant's device, records GPS locations and sensor information during the test execution.

Advantages

- Offline use

Disadvantages

- Data synchronized only before and after test
- Having to input database address to create a test is not very user friendly
- The system doesn't take into account the time differences of used devices

2.2.2 MAT

MAT (Mobile Application Tester) is an application, which targets the needs of usability testing in the field [10]. Within the application, a user can create a new test, which contains the name of test, and the name of tested application. Each test can have multiple test sessions, in which the user can define a participant, IP camera address, and enable or disable the use of GPS positioning. After the session has been started, events can be added to a log, either by choosing from a list of available markers (which utilizes the GPS location) or by choosing a location from the displayed map. To display the video stream and the map view, an Internet connection is needed. After the session, the recorded data can be exported to a XML file.

Advantages

- Real-time video stream
- Focuses on field testing

Disadvantages

- Limited data types (only log and its markers)
- No data synchronization with remote server
- Having to input IP camera address is not very user friendly
- Application designed only for tablet use

2.2.3 Morae

Morae is a commercial usability testing software by TechSmith. The software suite consists of 3 applications: Recorder, Observer, and Manager.

The Recorder application records the onscreen and keyboard activity of the participant's computer, as well as audio and video of the participant through the use of connected devices.

Multiple Observer instances can connect to the Recorder over the network, which allows all test practitioners to observe the participant's screen and his/her reactions on video. Moreover, each observer can use notes to annotate important test events using markers.

After the test, the recorded data can be further analyzed using the Morae Manager, where the screen recordings, video and audio recordings, and notes are synchronized to a single timeline. Furthermore, the Manager application automatically calculates usability measurements, such effectiveness, efficiency, and satisfaction, and creates appropriate graphs. Logged data can also be exported, so that it can for example be used in usability test reports.

Unfortunately, the system is focused on desktop use (namely supported only on Windows machines), making the use in field testing very limited. For additional information about Morae software see [60].

Advantages

- A complete usability testing solution

Disadvantages

- Price (1995 US Dollars)
- Focused on laboratory testing

2.2.4 IVE

The IVE (Integrated Interactive Information Visualization Environment) is a modular application, which allows importing data sets gathered during usability testing and its subsequent visual analysis [51]. The system was first created as part of a Master's Thesis by Jan Poživil [20], and currently is actively developed at the Czech Technical University in Prague by Ing. Ivo Malý, Ph.D.

The modular architecture makes use of two main parts: convertor plug-ins and visualization plug-ins. The purpose of convertor plug-ins is the import of recorded data sets and usability test context models. Implemented plug-ins allow importing a dialog model, CTT (Concur Task Trees) Task Model, Environment 2D and 3D models, generic XML log, or observer notes in XLS format. Visualization plug-ins are used to display imported data sets and usability test context models in one or more interactive views.

These plug-ins allow the selection of data sets for visualization. Moreover, the visualization plug-ins can communicate with each other to propagate changes. This for instance allows log timeline to be synchronized with playing audio/video sources.

Data collection is achieved through the use of external applications, which can export data to formats supported by the convertor plug-ins. The currently available logger applications (see [52]) focus on field testing and cover functionality such as video recording, event logging, taking observer notes and recording user positions during the test execution.

Advantages

- Modular architecture allows extensibility of supported data formats and visualization
- Interactive visualization

Disadvantages

- No data collaboration between several users
- External Applications have to be used for data logging

2.2.5 Conclusion

Out of the researched applications, the most complete usability testing tool is Morae, which covers the whole usability testing workflow from planning and preparation, over execution, to data analysis and reporting. However, the tool is not ideal for testing in the field. The second best solution, IVE, provides excellent means of interactive data visualization and its modular architecture allows further extensibility, but data collection depends on the use of external applications and data collaboration between multiple users could also be improved.

2.3 Requirements analysis

This section contains an overview of functional and non-functional requirements for the system to be designed. These requirements were either discovered during usability testing process analysis, were directly proposed by the client, represented by Ing. Ivo Malý, Ph.D. or emerged from interviews with the client, which helped to discover client's needs. Due to the size of the document, the full list of requirements is only available as an attachment to this thesis.

2.3.1 Functional Requirements

User Management The system must allow user to create a new user account within the system. Afterwards, the system must allow user to log into the system, respectively log out.

Test Definition Management The system must allow user to create, view, edit and delete a test definition. Additionally, the system must be able to list available test definitions for the currently signed in user. An instance is considered available, when the user is a member of the particular test definition.

Test Definition Members The system must allow user to add a another user as a test member with a specified role to test definition by specifying the user's email address. Also, the system must allow user to edit the role of previously added member, or remove the member altogether.

Test Definition Markers The system must allow user to add, edit and delete a marker within a test definition. Moreover, the system must allow user to view all the available markers, and change the order in which markers are displayed.

Test Definition Participants The system must allow user to add, view, edit, and delete a test participant within an existing test definition.

Test Definition Questionnaires The system must allow user to add a pre-test/post-test questionnaire to a test definition, if there is no pre-test/post-test questionnaire currently created.

Test Definition Task Lists The system must allow user to add, view, edit and delete a task list to a test definition. Also, the system must be able to display a list of available task lists within a test definition to a user.

Test Definition Files The system must allow user to add a new file from a list of supported file types to a test definition. Existing files can be viewed, edited or deleted. Moreover, the system must be able to display a list of available files within a test definition.

Test Definition Instances Within a test definition, the system must allow user to create a single new test instance, or create instances for all participants at once. Existing instances can be viewed, edited or deleted. Moreover, the system must be able to list available test instances within a test definition.

Test Instance Management The user can select a participant for the current test instance from the set of available participants in the test definition. Similarly, the user can select a task list for the current test instance from the set of available task lists in the test definition.

Test Instance Questionnaires If a pre-test/post-test questionnaire is created in the test definition, then the system must allow user to fill out its answers within its test instance.

Test Instance Files The system must allow user to add a new file from a list of supported file types to a test instance. Existing files can be viewed, edited or deleted. Moreover, the system must be able to display a list of available files within a test instance.

Test Instance Logging The system must allow user to select one test instance from available test definitions for logging. The system will then remember the selection even after the application had been restarted.

Unsupported files The system must correctly handle unsupported file types by displaying an error message.

2.3.2 Non-functional Requirements

Android OS The client applications will be implemented as applications for the Android operating system.

Supported Android Versions The client applications will support Android operating system versions from 4.1 to 7.1.

NoSQL Database The system must use a NoSQL Database for storage of textual data. The database system must enable access to data from Android and iOS mobile applications. Moreover, the database system must allow restricting access to stored data.

Offline use The system must be able to work temporarily without an Internet connection.

Chapter 3

Analysis and system design

This chapter describes an analysis of the problems, which were already discussed in the previous chapters. First of all, the application's domain model is presented, followed by an explanation of main use cases of the system to be designed. Next, based on the identified needs and requirements, available database systems are researched and a best suiting candidate is selected. Afterwards, the system architecture design is discussed, followed by the description of a client application prototyping process, which concludes the chapter.

3.1 Domain Model

Domain model is a kind of object model, which depicts problem entities, their attributes and relationships to other entities in the same domain [5]. Based on the knowledge gained from the requirements analysis, such model describing the main entities of the designed system was created (see figure 3.1).

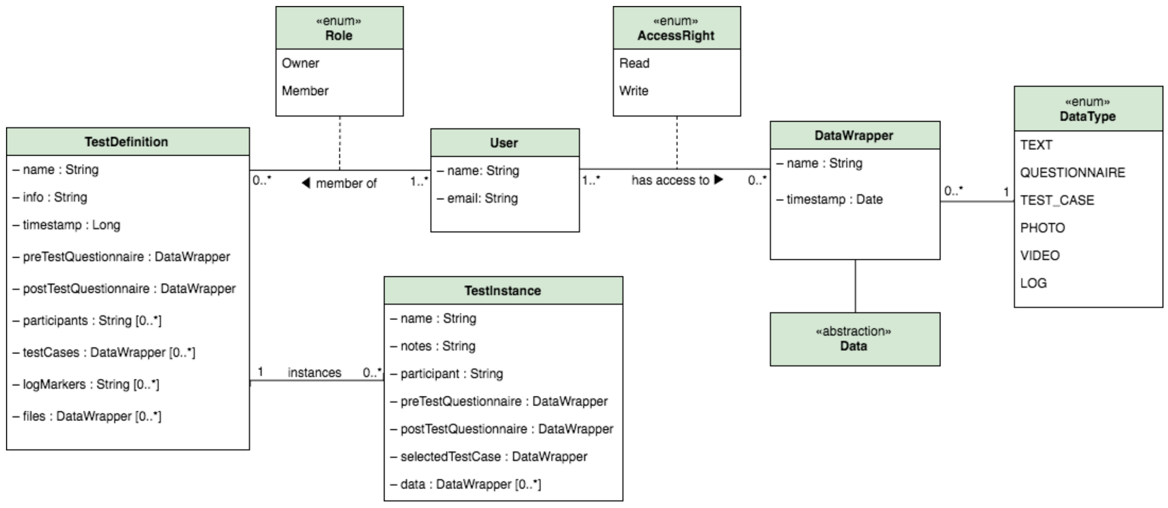


Figure 3.1: Application's domain model

3.1.1 User

Each user in the system provides his or her name and an email address, which acts as a unique identifier among all instances within the User domain. It might not be clear at the first sight, however the User entity is one of the system's most essential, for the reason that all other entities depend on it either directly (e.g. TestDefinition and DataWrapper) or secondarily (e.g. TestInstance).

A user can be a member of up to N test definitions (see 3.1.2), while each of these relationships is tied with a particular role, which defines user rights for actions within the test definition. Currently, the supported roles are an *Owner* and a *Member*.

Furthermore, a user can also have access to up to N data wrappers (see 3.1.2). Similarly to the relationship between user and test definition, the user's access to a data wrapper is bound to an access right, in this instance either *Read* or *Write*.

3.1.2 Test Definition

Test definition, which could also be referred to as a test plan, is comprised of all the information and data needed to perform a user test. Considering the usability test workflow as defined in section 2.1.1, the test definition entity is mainly used during planning and preparation phases.

Each test definition has at least one owner, however it can have up to N other members with either the *owner* or *member* role. Besides the customary attributes such as name and information, test definition also contains the pre-test and post-test questionnaires, a list of participants, and up to N test cases describing the tasks of a test. Next, a list of markers, which are to be used in log data files to define specific events during a test, is included. Furthermore, the entity can contain up to N files (data wrappers) of any type to be able to record any other previously undocumented information relating to the test. At last, the test definition has up to N test instances.

3.1.3 Test Instance

A test instance represents a particular test execution with a selected participant performed by following a selected test case from the set of available test cases in the parent test definition. The test instance entity is therefore used mainly during the execution, data and observation analysis, and reporting phases of the usability test workflow.

The test instance entity holds the user's answers to pre-test and post-test questionnaires, as well as all the other data used to record the execution of the test, such as photos, videos, logs and observer notes.

3.1.4 Data Wrapper

The data wrapper entity represent a file within the designed system. Apart of from a name and time of creation, the entity consists of three main parts.

First, the data type unambiguously defines the type of the file, which defines the structure of saved data. The second part is abstract data, whose contents are only limited by the defined structure of the entity's concrete data type. Currently, only text, questionnaire, test case, log, photo and video data types are allowed, however, the design supports easy extensibility by simply defining a new data type and its structure.

The last part, access rights, is used to describe user access restrictions to the file. Each instance must have at least one user with a *write* access right, but can contain up to N other user access records with either *read* or *write* rights.

3.2 Use Case Model

This section describes a use case model, which offers an alternative approach to requirements requirements description that helps to identify actors, system boundaries, user activities within the application and relationships between those actors and use cases [1]. As the documented use cases mostly represent simple create, read, update, and delete operations, no detailed use case scenarios were created.

The designed system will consist of two applications. The first (administration) application should mainly cover test management and test data management, whereas the second (logger) application should focus on data collection. However, as some functionality of these apps overlaps, the use cases that follow were not structured with respect to the functionality of a particular application. Instead, they are grouped in context of the domain entity they relate to the most. An overview specifically describing which use case groups are covered by each application can be found at the end of this section (see 3.2.9).

3.2.1 Actors

An actor represents a role played by a user or any other system, that interacts with the application to achieve a certain goal. Use cases then document such interactions [2]. The figure 3.2 describes the actors of the designed system and their relationships.

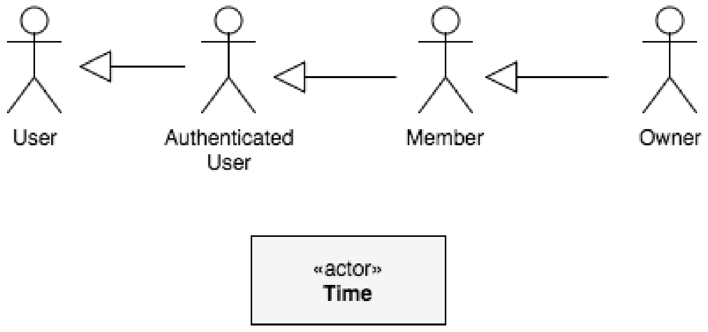


Figure 3.2: Diagram of use case actors

User A person who uses the system, while *not* being signed in (authenticated).

Authenticated User A person who uses the system, while being signed in (authenticated by one of the supported authentication providers).

Member An authenticated user who is a member in context of a particular test definition.

Owner A test definition member who also has full owner rights to the particular test definition.

Time An actor which is able to trigger time-bound actions.

3.2.2 User Management and Authentication

This section describes user management and authentication use cases that are shown in figure 3.3. An unauthenticated `user` can sign in to the system using his or her account credentials to become an `authenticated user`. If the user doesn't have an account yet, new account can be registered. Naturally, the `authenticated user` is then able to log out from the system if needed.

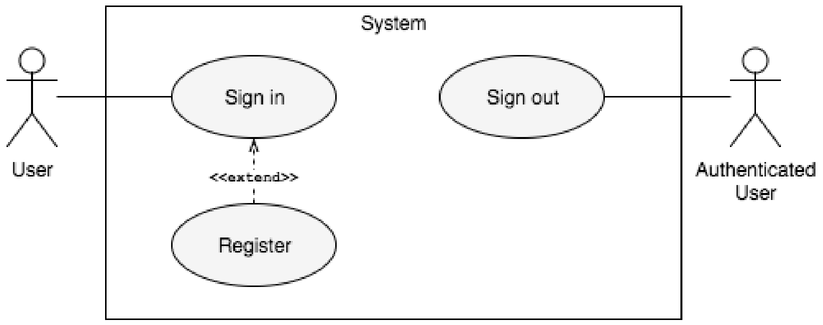


Figure 3.3: Use Case diagram describing user management and authentication actions

3.2.3 Test Definition Management

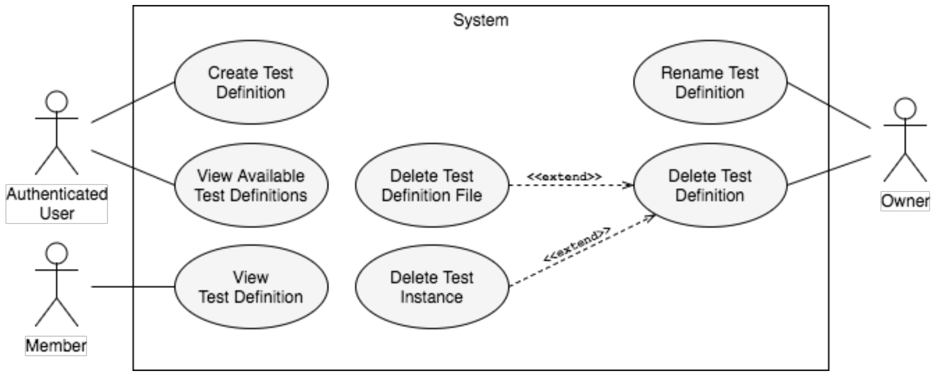


Figure 3.4: Use Case diagram describing test definition management

The uses cases related to the management of test definition itself are described in figure 3.4. Any `authenticated user` can create a new test definition and he/she automatically becomes the owner of this particular instance. `Authenticated user`s are also able to view a list of test definitions, which are available to them, i.e. the test definitions they belong to either as an owner or as a member. As previously shown in the diagram of actors (fig. 3.2), an `owner` is a generalization of `member`, which means `owner`s can do any action which is allowed to be performed by a `member`. With that said, a `member` is allowed to view a test definition's detail, while an `owner` can also rename or delete the test definition. Deleting a test definition also deletes all the test definition's files, as well as its test instances.

3.2.4 Test Definition Data Management

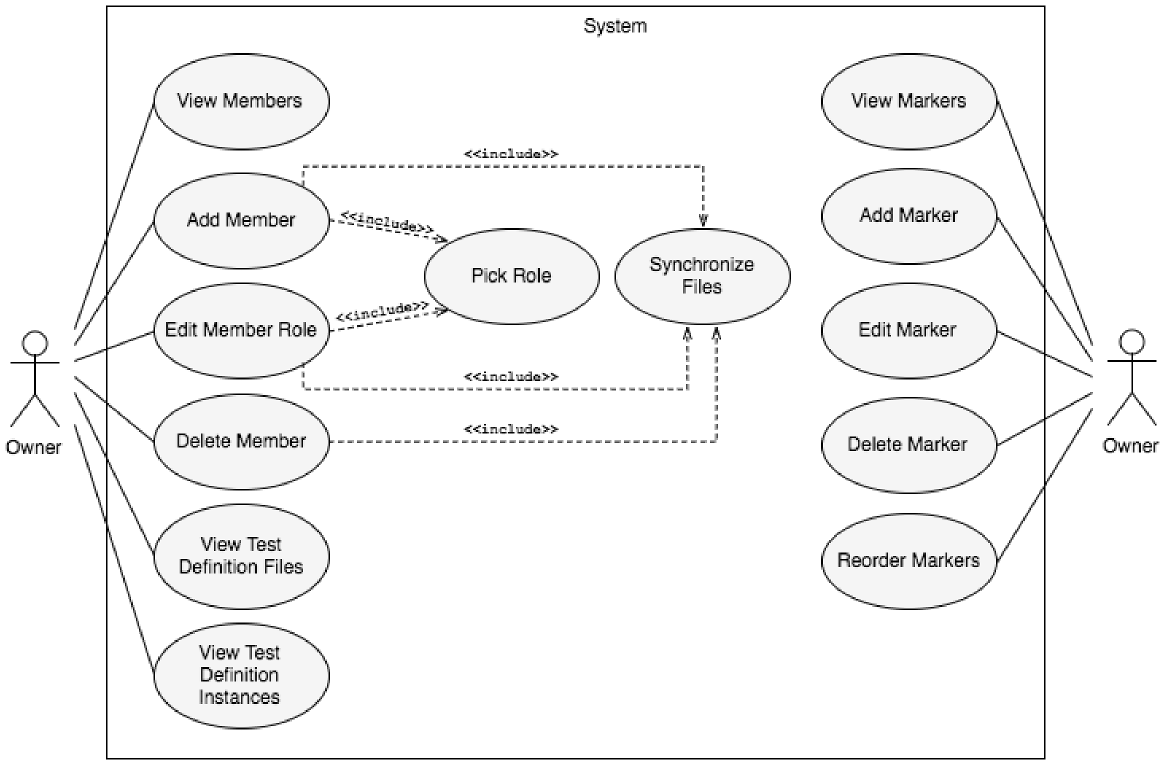


Figure 3.5: Use Case diagram describing test definition data management

In this section, the use cases of test definition data management are discussed. Figure C.1 contains actions, which can be executed by its `member`s, while figure 3.5 describes use cases which can be performed by the `owner`s only.

A `member` can view and edit the test definition's information in text format. Next, `members` are able to add new pre-test or post-test questionnaire, as well as view, edit, or delete these questionnaires. Furthermore, they can view the list of available test cases within the test definition and modify this list by adding new test cases or editing and deleting the existing ones. Of course, any available test case can also be viewed by the `member`. Last, `member`s can view, add, rename, and delete recruited test participants.

On top of the previous actions, `owners` can also view the test definition members, add new members, and edit or delete existing members. When adding or editing a member, a user role (member or owner) must be chosen. All member editing actions also evoke file synchronization to properly set user access rights of all available files within the test definition and all its instances. Next, the markers, which are to be used while adding events to log files (see 3.2.8), can be viewed, added, edited, deleted or reordered. `Members` can also display test definition files and existing test instances. The management of test definition files is later described in section 3.2.8, while the test instance management follows in the next section.

3.2.5 Test Instance Management

As shown in figure 3.6, use cases of test instance management can be executed solely by the `owners` of its parent test definition. Such `owner` is able to add a single new test instance to a test definition or create test instances for all test definition's participants at once. Existing instances can be viewed in detail, renamed or deleted. Similarly to deleting test definition, removing a test instance also results in deletion of all its files.

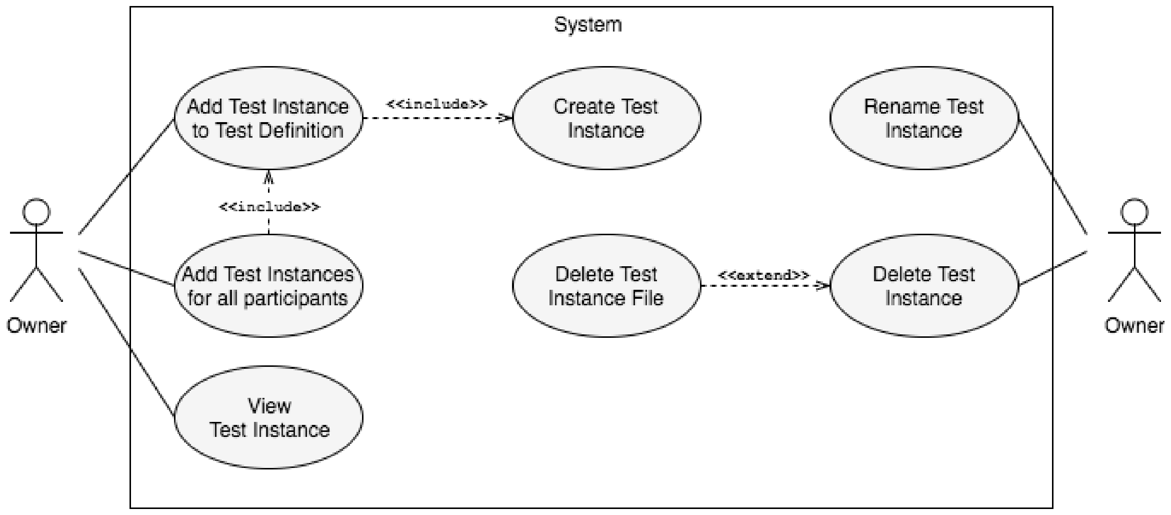


Figure 3.6: Use Case diagram describing test instance management

3.2.6 Test Instance Logging

For usability test execution purposes, `owners` can select an instance from available test definitions' instances for logging within the logger client application. This allows the test executioner to quickly access or edit needed test data and other information. Later, when the test execution has finished, the previous instance selection can be dismissed. The diagram for these use cases can be found in figure C.2.

3.2.7 Test Instance Data Management

As in previous section, test instance data can be managed only by the `owners`, as shown in figure C.3. First, the selected test participant for the current test execution can be displayed. If not yet selected, a participant can be chosen from the participants defined in the enclosing test definition. Next, the `owners` can view the pre-test and post-test questionnaires if present, as well as fill out the participant's answers to their questions. Before the test execution is started, a test case which is to be followed during the test can be selected. The selected test case then can be displayed or replaced at any later time. Furthermore, the `owner`s are able to view and edit the test instance notes to include information about the particular test. Lastly, test instance files can be displayed, added, modified or deleted as further described in section 3.2.8.

3.2.8 File Management

The file management use cases were divided into groups according to the create, view, edit and delete operations, and can be found in figures 3.7, C.4, C.5 and C.6, respectively.

When adding a file to either the test definition or the test instance, the `owner` has to choose a file type from the provided options: text, questionnaire, test case, log, photo or video. The system then creates the file according to selected type and invokes file synchronization to set up user access rights of the newly added item. In case the file was added while not being connected to the Internet, the files synchronization is also triggered periodically or after a network change to complete pending synchronization requests.

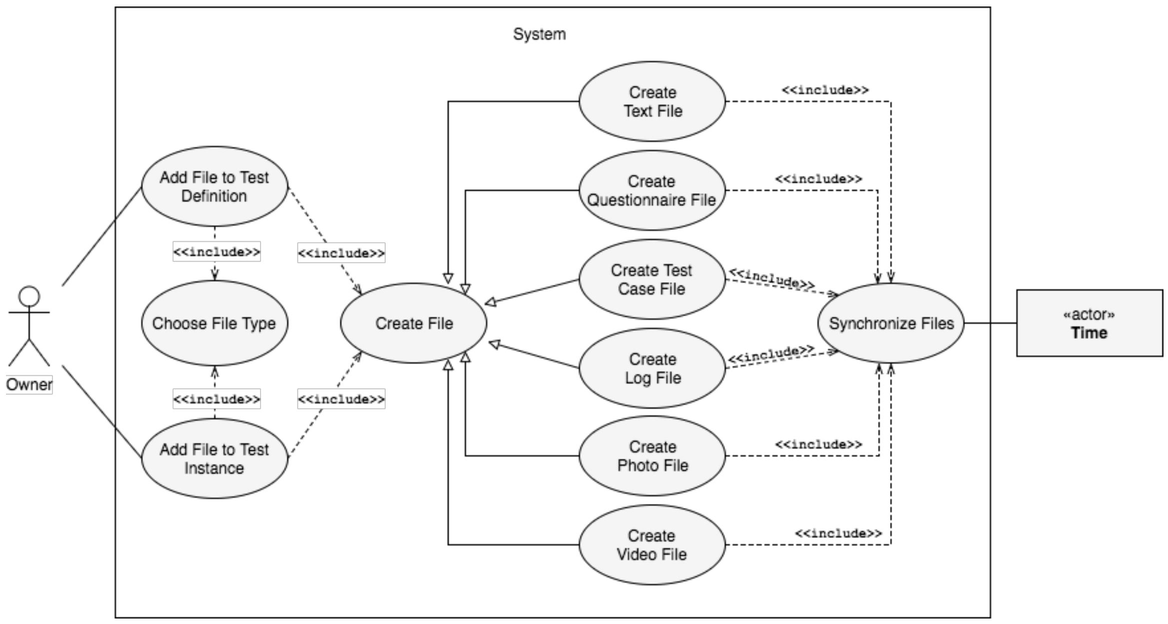


Figure 3.7: Use Case diagram describing create actions of file management

`Owner`s can also display, edit or delete any of the previously discussed file types. Text file displays its contents as a structured text, which can be edited by the user. A questionnaire file consists of questions and their corresponding answers. An `owner` can edit the questionnaire either by managing the questions or the answers. Questions can be added, edited, delete or reordered within the questionnaire, while answers to existing questions can only be added, edited or deleted.

A test case file contains a list of tasks, which can be added, edited, deleted or reordered. The log file displays a timeline of log events each consisting of a timestamp, marker and a optional comment. When adding a new log event, the `owner` can pick a marker signifying an important moment during the execution of the test from a list of existing markers or specify a custom marker. Markers and comments of existing events can be edited, or the event as a whole can be removed. As the name suggest, a photo file can hold a photograph, which can be either picked from the device's gallery or the user can take a new one with the built-in camera. Last, a video file either shows a video preview if the recording is currently in progress, or the recorded video itself, once the recording has successfully finished.

3.2.9 Applications' Functionality Overview

Admin Application

The administration client application covers all of the previously defined functionality groups, except test instance logging. Specifically, the following groups are included:

- User Management and Authentication
- Test Definition Management
- Test Definition Data Management
- Test Instance Management
- Test Instance Data Management
- File Management

Logger Application

The logger application focuses solely on data collection, for which the following functionality is needed:

- User Management and Authentication
- Test Instance Logging
- Test Instance Data Management
- File Management

3.3 Database Systems

One of the non-functional requirements for this project was the use of a NoSQL database for data persistence of textual data. This sections compares selected NoSQL database engines in respect to the requirements described previously in chapter 2. The most suitable database system is selected and used in demo applications for test administration and data collection, whose implementation is further described in chapter 5.

To see the advantages of using NoSQL database system for the designed applications, we'll first discuss the more traditional relational database systems and point out the main differences from the available NoSQL database systems.

3.3.1 Relational databases

The relational database management systems (RDBMS) are based on a simple, but robust mathematical concept of a relation, which is within the relational database systems most often implemented using tables with rows and columns. This fact dictates the use of highly structured data with predefined schema. A common objective is a normalization of the schema to Boyce-Codd normal form (BCNF) or the third normal form (3NF) in order to diminish data redundancy. This however results in high data granularity, and therefore the need to join the pieces of data back together when querying.

User data access and management operations within relational database management systems work on a transaction level, which is often referred to as a *pessimistic* approach as its goal is to prevent data inconsistencies in the first place. A transaction is a flat sequence of database operations, which transforms data from one consistent state to another while conforming to the so-called ACID properties [9]:

Atomicity A partial execution is not allowed, i.e. the transactions either executes as whole or it doesn't execute and leaves no changes at all.

Consistency A transaction transforms the database from one valid state into another valid state.

Isolation Uncommitted effects are not visible to other currently running transactions.

Durability Committed effects of a transaction are permanent.

3.3.2 NoSQL databases

NoSQL databases represent a new generation of database systems, which are geared towards persistence of high volume of mostly non-relational data, data distribution and replication, as well as horizontal scalability of the system. Often, these systems work without a predefined schema (schema-less) and offer a simple interface for data access and management.

Unlike the relational database systems, the NoSQL systems usually don't guarantee the strong ACID consistency for its operations, instead they settle for a BASE model as an *optimistic* alternative used in distributed environments, which offers an *eventual consistency.* These are the characteristics of the BASE model [9]:

Basically Available The system work basically at all times, i.e. partial failures can occur, but without total system failure.

Soft State The system is in flux, non-deterministic state, where changes occur all the time.

Eventual Consistency The system will sooner or later be in some consistent state, but consistency is not guaranteed at all times.

3.3.2.1 Advantages

The main advantages of using NoSQL databases over the traditional relational database management systems are as follows:

Flexible scalability Unlike the classic relational database management systems, which rely on vertical scalability, i.e. upgrading to a more powerful hardware, the NoSQL databases are scaled horizontally by adding new nodes of cheap commodity hardware to an existing cluster. Besides being cheaper, horizontal scaling can be done without the need to turn off the system for maintenance.

Flexible data model NoSQL databases do not enforce a definition of a data schema. In practice however, the data naturally has an implicit schema, it's just not explicitly defined at the database level, but the responsibility of schema management is transferred to the application logic level instead.

Effective reading NoSQL databases guide users to structure data in such way that facilitates the read operations of frequently performed queries. Such queries can then be executed very swiftly, resulting in higher system throughput [14].

3.3.2.2 Disadvantages

In comparison to the relational database management systems, the NoSQL systems is a much younger technology, so naturally, the users can run into some of the following challenges whilst using it:

Lack of maturity In comparison to the relational database management systems, the existing NoSQL databases lack the robustness, which was proven for RDBMS over the years of their use in production systems. Furthermore, being a new technology, the amount of available NoSQL database experts in the market is also low in comparison to the RDBMS.

Administration What comes from the fact that NoSQL databases are mostly distributed systems, is the increase in complexity of installation as well as maintenance.

No standardized data access Unlike relational database systems which offer widely standardized data access and management through SQL, the existing NoSQL systems each have its own non-standardized application programming interface (API), which makes switching between systems quite cumbersome.

3.3.3 File Storage

For textual data, which needs to be indexed and queried based on its inner structure and values, databases are a perfect solution. However, when it comes to storing large static binary files, a database might not be the most optimal data store.

Even though most database systems provide a data type for storing BLOBs (Binary Large Object), using a dedicated file system or object store to serve the large static files might be more favourable in both performance and price.

A research paper [7] comparing the performance of NTFS file system and SQL Server database system showed that BLOBs smaller than 256KB are more efficiently handled by the SQL Server, while the NTFS file system is more efficient when working with BLOBs larger than 1MB. In distributed database systems, which replicate its data over the network to other nodes in the cluster, storing large binary files could also hinder the database performance due to the increased amount of data that needs be replicated after a write operation. The price difference of the per-GB cost of storage of both solutions can be can be portrayed using the Microsoft Azure calculator [1], which shows that the cost of the same amount of storage in a NoSQL database (DocumentDB) is approximately ten times the cost of storage in a file system (Storage).

3.3.4 Researched Database Systems

This section analyzes available NoSQL database systems, which were considered for use within the designed system with respect to the usability testing needs and requirements as discussed in chapter 2. Namely the following characteristics were taken into account during the evaluation:

- Supported NoSQL store models.
- Data access and management options for considered platforms and programming languages.
- Availability of integrated access control to stored data.
- Support of offline capability for mobile client applications.
- Provided means of BLOB storage.

3.3.4.1 Amazon DynamoDB

DynamoDB is a cloud NoSQL database fully managed and hosted by the Amazon corporation as part of the Amazon Web Services (AWS). This database supports both the document and key-value store models without the need to define a database schema.

For data access and management the system provides a REST API, or Software Development Kits (SDKs) for various programming languages and platforms such as Java, .NET,

[1]https://azure.microsoft.com/en-us/pricing/calculator

C++, JavaScript, Node.js, Android or iOS. However, these SDK libraries only act as a wrapper for the REST API, which means that for the client to be able to write or access data, an Internet connection is always necessary.

The DynamoDB system also allows a fine-grained data access control. Rules can specified to restrict access to a table, specific items in that table, or even specific attributes of an item.

As part of Amazon Web Services (AWS), an object storage Amazon S3 for files up to 5TB in size is also offered. Further information about the S3 can be found at [24]. For information about the DynamoDB see [23].

Advantages

- Document and key-value store models.
- REST API for data access and management.
- SDKs for variety of programming languages.
- Fine grained data access control.
- Availability of object store, which can be accessed and managed using the same REST API or SDKs.

Disadvantages

- No offline capability.

3.3.4.2 Microsoft Azure DocumentDB

DocumentDB is a NoSQL document database service managed and hosted by Microsoft as part of the Microsoft Azure suite of products. As the name suggests, DocumentDB database supports only the document store model. However if needed, Microsoft Azure also offers a key-value store named Table Storage.

Data can be accessed and managed using a provided REST API or an SDK. SDKs are only available for Java, .NET, Node.js and Python. Platforms such as Android and iOS are not officially supported. Moreover, as the SDKs seem to be geared towards enterprise and web application, an Internet connection is necessary for data access and management.

Data access control in DocumentDB is based on resource tokens, which map the relationship between a user and the permission this user has for a specific resource, such as a collection or a document.

The Microsoft Azure suite also offers a storage for unstructured binary data, aptly named Blob storage. More information about Blob storage can be found in [27]. Further information about DocumentDB is in [36].

Advantages

- REST API for data access and management.
- SDKs for variety of programming languages.
- Data access control.
- Availability of object store.

Disadvantages

- No offline capability.
- No official Android and iOS support.

3.3.4.3 Firebase Realtime Database

Firebase Realtime Database is a cloud-hosted NoSQL document database managed and hosted by Google as part of the Firebase platform. This database stores its data as one large JSON document.

Data can be accessed and managed through a provided REST API or using the SDKs, which are available for Java, Node.js, C++, Unity, Android and iOS. When using the SDKs, all the connected clients automatically receive updates with the newest data within hundreds of milliseconds every time data changes. Moreover, when a client goes offline the SDK persists updated data locally to disk. Once connectivity is reestablished, the local client data is synchronized with the current server state, merging any conflicts automatically.

A fine-grained data access control is provided by the Firebase Realtime Database Security Rules, which define who can read or write to the database, to a particular object, or even to a particular object's attribute. These rules can also be used to define what a correctly formatted value will look like, whether it has child attributes, and its data type.

For binary content, the Firebase platform offers Firebase Storage which can be accessed directly from the previously mentioned SDKs. Moreover, the platform offers a complete authentication solution, which supports email and password authentication, as well as popular federated identity providers like Google, Facebook and Twitter. Other integrated solutions, primarily aimed at mobile applications, include Cloud Messaging for notifications and Crash Reporting for error data collection. Further information about Firebase can be found in [39].

Advantages

- REST API for data access and management.
- SDKs for variety of programming languages and platforms.
- Fine-grained data access control.
- Availability of object store.
- Offline capability.
- Other integrated solutions (Authentication, Cloud Messaging, Crash Reporting).

Disadvantages

- Key-value store model not supported.

3.3.4.4 Couchbase

Couchbase Server is an open-source NoSQL database, which supports both the document and key-value store models.

SDKs for various programming languages such as Java, .NET, and Node.js enable data access and management. To be able to use a REST API, an additional component called Sync Gateway is needed to be integrated. This component is also needed for remote data synchronization, when using Couchbase Lite. Couchbase Lite is an embeddedable NoSQL database, which enables offline data management and automatic synchronization between the Couchbase Server and the mobile device when the network is restored using the Sync Gateway tier.

The Sync Gateway also enables the configuration of user or role based data access control by defining security rules, which are used to determine who has read and write access to a particular document in the database. Further information about Couchbase Server and Couchbase Lite can be found in [33], respectively [61]

Advantages

- Document and key-value store models.
- SDKs for variety of programming languages.
- Offline capability.
- Open sourced.
- Fine-grained data access control.

Disadvantages

- Additional components needed for REST API and integration with Couchbase Lite.
- No integrated object store.

3.3.4.5 Conclusion

Based on the research provided above, the Firebase platform was selected for the design and implementation of the system, as it best meets all the proposed requirements. For web applications a REST API can be used to access and manage data, while desktop and mobile applications can use provided SDKs. The system allows a fine-grained level of data access control, which can be edited dynamically. The Android and iOS SDK libraries offer a fully integrated support for offline use, as well as automatic synchronization with remote server. On top of it all, the Firebase platform provides other beneficial solutions, such as Authentication, Cloud Messaging or Crash Reporting.

3.4 System Architecture Design

The designed system's architecture could be described as a two-tier architecture, often referred to as the client/server architecture [3].

3.4.1 Clients

The system contains 2 thick client applications that include both the presentation and business logic. Both of these clients are to be run on mobile devices with the Android operating system. The communication of client applications with the server tier is carried over HTTPS through the use of provided Firebase SDK for Android [22].

3.4.2 Server

The server tier is represented by the Firebase platform. Unfortunately, as a commercial product, the internal Firebase architecture is not available. However, of all the provided features [43], only Authentication, Realtime Database, and Cloud Storage will be used.

Authentication component allows the implementation of a complete authentication system that supports social identity providers (Facebook, Twitter, GitHub and Google), and email and password based authentication. The NoSQL Realtime Database is to be used to store application's textual data in JSON format, while Cloud Storage will be used for storage of binary files such as photos and videos.

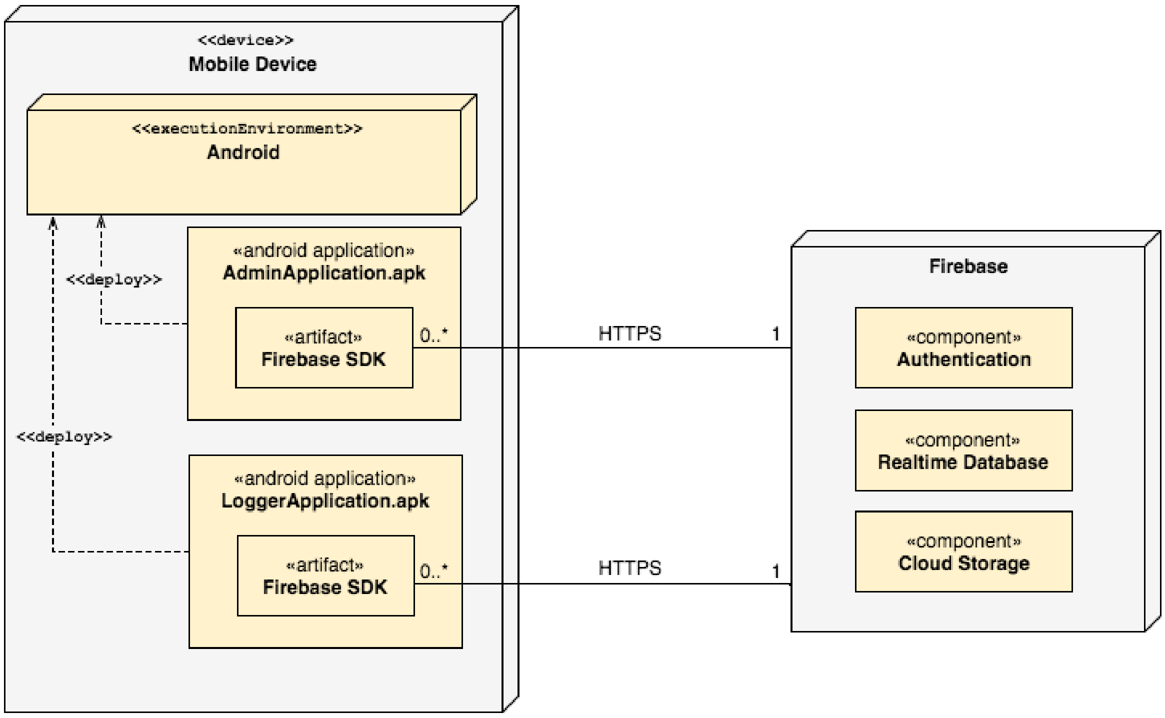


Figure 3.8: Deployment diagram showing the designed system's architecture

3.5 Client Application Prototyping

Before implementing the Android client applications, the user interface (UI) of both applications was first developed as a low fidelity prototype. To create the prototypes, phone templates were printed out on paper, and the user interface was drawn over it by hand. The first iteration of prototypes was tested with users using mobile application POP[2] (Prototyping on Paper), which enables to take photos of UI sketches to create an interactive clickable prototype.

The testing helped to discover usability problems within the prototype, which were analyzed and proposed solutions to these problems were used to create the second iteration of the prototype that was used as a base during implementation.

3.5.1 Test Scenarios

This sections describes test scenarios, which were used during user testing of the prototype of the application for administration, respectively the application for logging.

Admin Application

1. Create a new test definition named “My Test 01”.
2. Edit the information of the new test definition to “This is some text”.
3. You would like to add a pre-test questionnaire.
4. Now you would like to add a new question to the post-test questionnaire and delete its last question afterwards.
5. A new participant has been recruited. Add him to the test as "Participant 1".
6. You just finished up setting up the test and want to take a photo of it for later reference. So take a new photo [from camera] and rename it to “Photo File”.
7. Now it’s time for testing with a participant. So add a new test instance for participant “Participant 003”, which is going to follow test case “Test Case 002”.
8. Now that you’ve executed the test, fill out the post-test questionnaire.

Logger Application

1. It’s time to execute a new test. Select test instance “Instance 003” of “Test 002” for logging.
2. The test is going to be executed with Participant 003.
3. The test is going to follow “Test Case 002”.
4. Now that the test has finished, fill out the post-test questionnaire.
5. This testing is now over, and we want to start logging for “Instance 002” of “Test 002”.

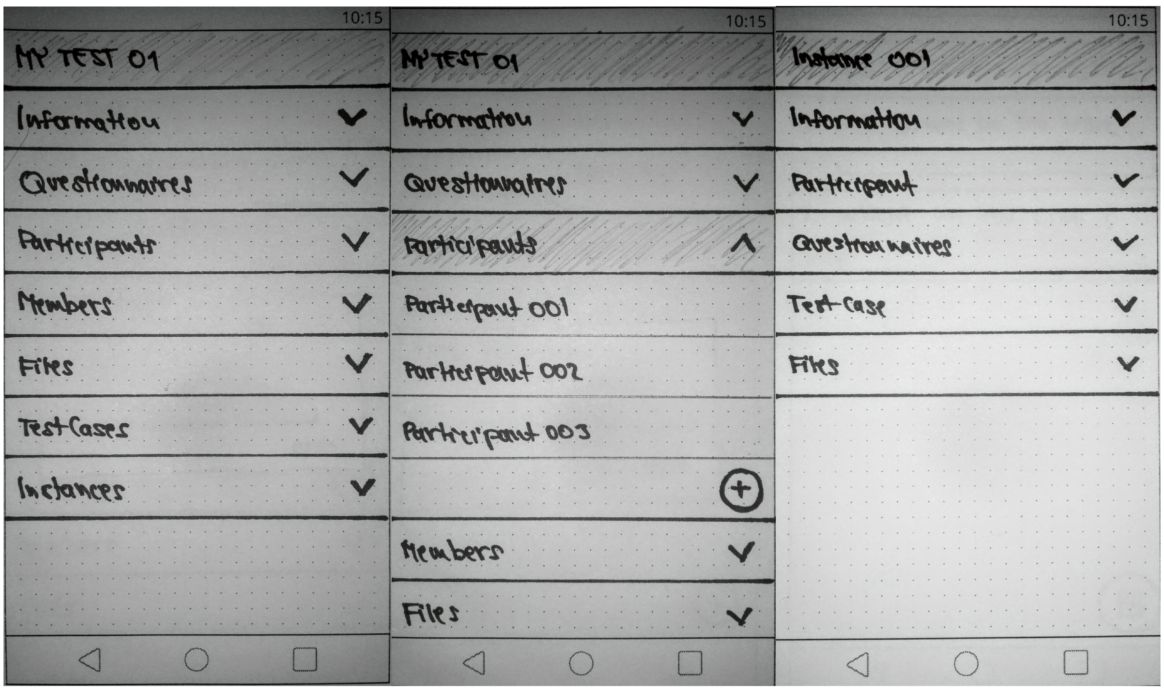
[2]https://marvelapp.com/pop/

3.5.2 First Prototype

Based on the previously defined domain and use case models, the main application screens were identified as *main screen of administration application*, *main screen of logger application*, *test definition detail*, *test instance detail*, and *file detail.* Detailed description of these screen is discussed later in context of the final prototype in section 3.5.4.

As shown in figure 3.9, the first iteration of the prototype heavily relies on the use of expendable list items [53] for both the *test definition detail* and *test instance detail.* The reasoning behind this was limiting the visual depth of the menu so that all the important information is quickly accessible.

All screens of the prototype are either available as an attachment to this thesis, or an interactive version of the admin application prototype[3], respectively the logger application prototype[4] is available.



(a) Test definition detail (b) Test definition detail with expanded participants (c) Test instance detail

Figure 3.9: First version of the low fidelity prototype

3.5.2.1 Testing

The only requirement for the participants was at least basic knowledge of the usability testing process and its terminology, therefore CTU students who were previously enrolled in either TUR (Testing of User Interfaces) or NUR (User Interface Design) courses were chosen.

[3]invis.io/4CBJEDEGD
[4]invis.io/XZBJFN7N5

Participant 1 Within the admin application, the first participant had no trouble adding a new test definition, as well as editing its information. He successfully added a questionnaire, and edited it according to the scenario. However, when adding a participant, he noted that he had problems orienting within the menu and found the expanded list of participants confusing as they looked very similar to the menu itself. Moreover, when renaming a photo, the participant noted there wasn't a way to cancel the rename operation (to cancel the action, a tap outside the dialog was needed).

In the logger application, the participant solved all the tasks successfully, but commented that he wasn't at first sure whether the row with selected test case was clickable to display its contents.

Participant 2 The participant had successfully added a test definition, edited its information, and added and edited questionnaires. When adding a participant the user was looking for a rectangular button at the top (as when adding a questionnaire), but successfully finished the task after noticing the circular button at the bottom. When trying to rename a photo, the participant accidentally clicked *delete* instead of *rename* and got confused as to what exactly has happened.

Within the logger application, no problems were encountered.

3.5.3 Usability Problems of the First Prototype

3.5.3.1 Lists within a list

Problem Participant was confused by the detail screen of test definition, where the list of participants, members, files or instances is shown within the list of available options (Information, Questionnaires, Participants, ...) as shown in figure 3.9b.

Solution Display the detail of each option in a separate screen instead of the current expandable list view.

3.5.3.2 Different ways of adding files

Problem Participant was confused by the variety of ways of adding different files. To add a questionnaire a normal rectangular button was used, whereas participants and files used circular FAB [5] (Floating Action Button) at the bottom of concrete section.

Solution Unify the buttons used for adding files within the whole system.

3.5.3.3 Action cancelling

Problem Participant didn't know how to cancel an editing action (such as when renaming). The correct way was to click outside the displayed dialog.

[5] https://material.io/guidelines/components/buttons-floating-action-button.html

Solution Add a negative (cancel) action to the editing dialogs.

3.5.3.4 Delete confirmation

Problem Participant accidentally deleted a file when trying to rename it.

Solution When deleting a file, a confirmation should be displayed allowing either to confirm the deletion, or cancel it.

3.5.3.5 Slow Menu orientation

Problem The participants had trouble quickly navigating though the "menu" in the test definition detail screen (3.9a) and test instance detail screen (3.9c).

Solution A common way to enhance menus is using icons. Displaying text and icons together helps to create an association in the user's mind [11], and afterwards, icons are fast to recognize at a glance [8].

3.5.4 Second Prototype

The solutions to the usability problems in previous section were used to create a second iteration of the prototype. As before, all prototype screens are either available as an attachment to this thesis, or as an interactive prototype of the admin [6], respectively the logger application [7].

In the following sections, the most important screens of both prototypes and the functionality they cover is described.

3.5.4.1 Home screen of Admin app

The home screen of admin application displays a list of test definitions, which the currently logged in user has access to either as a member or an owner. In the bottom right corner, a floating action button (FAB) allows adding a new test definition. After clicking the button, a dialog is shown, prompting the user to input a name of the new test definition. Pressing any row of the list displays a detail of selected test definition.

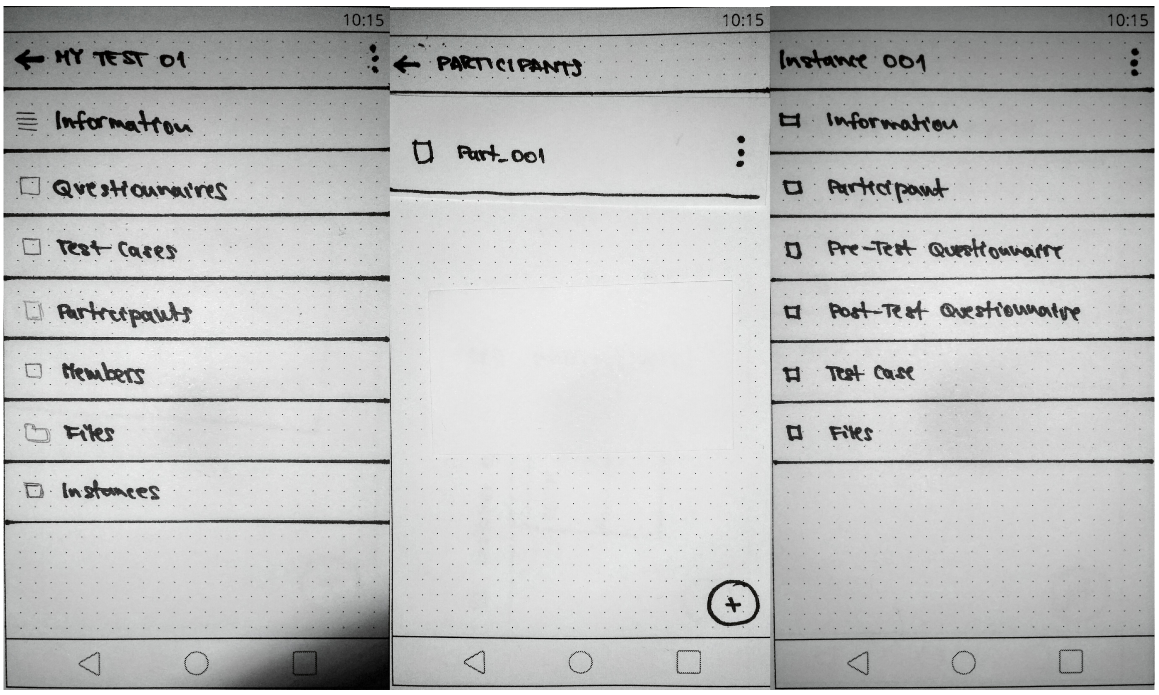
3.5.4.2 Home screen of Logger app

The home screen of logger application allows the user to select a specific test instance for logging. The selection can be made by clicking a FAB in the bottom right corner, which displays a dialog that requires the user to first select a test *definition*, and afterwards one of it test *instances*. Once an instance is selected, the test instance detail is shown.

[6]invis.io/QUBJROQ9B
[7]invis.io/S7BJRYD6Q

3.5.4.3 Test Definition Detail

As shown in figure 3.10a, the test definition detail screen consists of two parts: a toolbar [31], and a list of menu items. The toolbar displays the test definition's name as its title, and provides a context menu with options to rename or delete the test definition. The menu contains links to test definition's *information*, *questionnaires*, *test cases*, *participants*, *members*, *files*, and *test instances.* Unlike the first prototype, clicking a menu item now results in displaying a separate screen with a detail of the selected option, as for example shown in figure 3.10b.



(a) Test definition detail (b) Test definition participants (c) Test instance detail

Figure 3.10: Second version of the low fidelity prototype

Information In the information detail screen, the definition's information in text format is displayed. An *edit* button allows the user to switch to an editing mode, where the text can be edited. Afterwards, changes can be saved by clicking the *save* button, which is shown in place of the *edit* button.

Questionnaires In the second version of the prototype, the questionnaires detail also uses a FAB in the bottom right corner, which displays a dialog allowing the user to select either the pre-test or the post-test questionnaire to be added. Available questionnaires are then displayed in a list, where clicking a list row opens up a new *file detail* screen.

Test Cases As in previous screen, available test cases are displayed in a list, which can be used to access the detail of a particular test case. In the bottom right corner, a FAB allows adding a new test case to the test definition.

Participants Once again, based on the previously used principle, participants are displayed in a list. Here however, no detail is shown after clicking a list row, as all the available information is already shown within the list row. Instead a context menu can be used to edit or delete a participant. New participants can be added using the button in the bottom right corner of the screen.

Members Test definition's members are shown in a list, where each row contains member's email address, his/her role within the test definition, and a context menu that allows changing member's role or deleting him/her from the project altogether. When adding a new user using the FAB, a dialog, where email address and role type of the new member has to be specified, is shown.

Files Making use of the already familiar principles, the test definition's files are also displayed in a list. This time, each row contains the name of the file and an icon representing the file's type. Clicking a list row results in showing a *file detail* as described in section 3.5.4.5.

To add a new file, the user first has to click the FAB in the bottom right corner, which is followed by a bottom sheet[30] sliding up from the bottom of the screen to reveal available file types. After the user chooses the desired type, a new file is created and its detail displayed.

Instances As well as in most previous screens, a list is used to display available test instances and a button in bottom right corner allows adding new ones. Clicking a particular test instance forwards the user to its *test instance detail* screen.

3.5.4.4 Test Instance Detail

The test instance detail screen is very similar to the test definition detail, as shown in figure 3.10c. In the toolbar, the name of test instance is shown, as well as a context menu allowing to rename or delete the test instance. The menu below the toolbar contains links to test instance's *information*, *selected participant*, *pre-test* and *post-text questionnaires* (if available), *selected test case*, and at last its *files*.

The *information* and *files* screens are identical to those previously discussed within the test definition detail screen. The *pre-test questionnaire* and *post-text questionnaire* options only display the questionnaire in its *file detail* view. Therefore, these screens will not be further described.

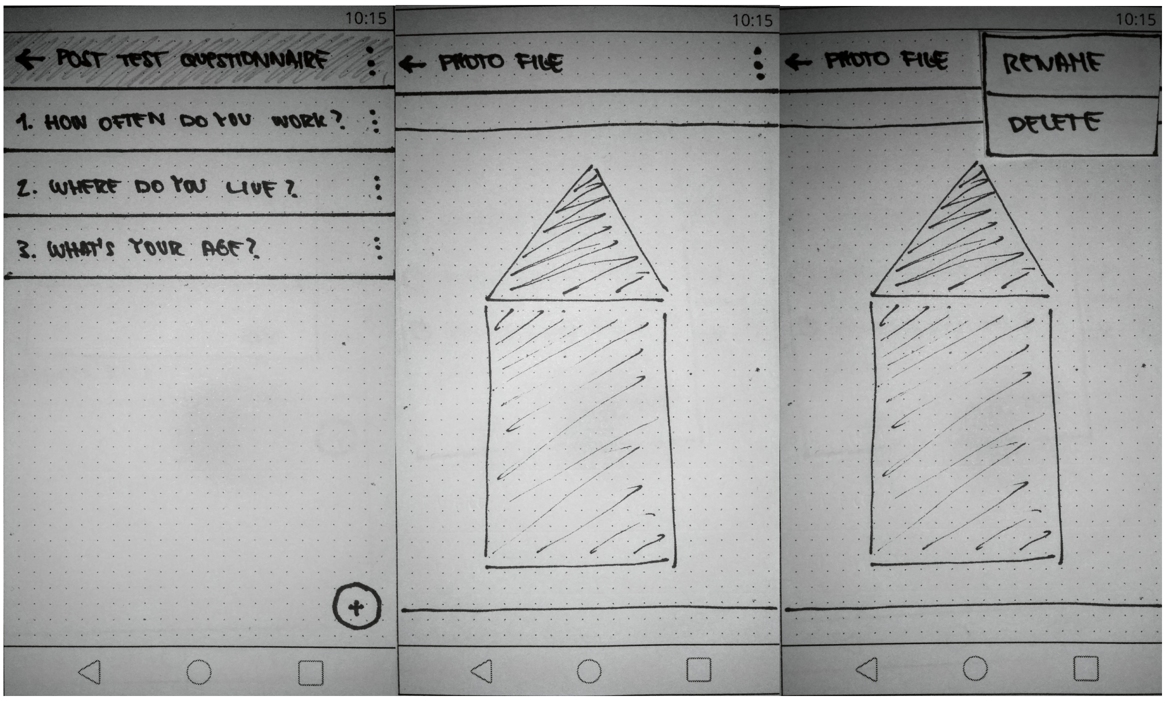
Participant If selected, the screen displays the name of the participant. In the opposite case, the user can select a participant by clicking the FAB in the bottom right corner, which forwards the user to a new screen with a list of test definition's participants. After selecting a participant, the participant detail is shown again.

Test Case This screen shows the tasks of the selected test cases. If no test case was yet selected, the user can do so by using the FAB, which shows a new screen with a list of available test cases, where the desired test case can be selected.

3.5.4.5 File Detail

The contents of file detail screen are dependent on the particular type of selected file. As shown in figure 3.11, a questionnaire file for example contains a list of its questions, whereas a photo file displays a photo that was previously taken.

However, one component that is shared between all file types is a toolbar, which shows the name of the displayed file, and allows the file to be renamed or deleted through the actions in the toolbar's context menu.



(a) Questionnaire file (b) Photo file (c) Photo file with context menu

Figure 3.11: File detail screens within the second version of the low fidelity prototype

Chapter 4

Database Configuration

In section 3.3, available NoSQL database systems were researched and the Firebase platform was selected as best suiting option for the designed system.

In this chapter, the configuration of both the Firebase Realtime Database and Cloud Storage is discussed. In both environments, data structure design is described, followed by a description of the techniques used to secure stored data.

4.1 Firebase Realtime Database

4.1.1 Data Structure

As already discussed in chapter 3, the Firebase Realtime Database is a *schema-less* document NoSQL database. But even though the database is schema-less (i.e. no database schema has to be defined as for instance required in relational database systems), it doesn't mean we can disregard the database structure altogether. In fact, it is still essential to think about how the stored data will be accessed and to structure it accordingly, otherwise the system's performance and throughput would be at risk [44].

4.1.1.1 Best Practises and principles

This section goes over recommended practises of structuring data within the Firebase Realtime database, however most of the discussed principles are applicable to any document-oriented NoSQL database system.

Avoid nesting data The whole database in Firebase is stored as one large JSON tree, which is allowed to be up to 32 levels deep. However, when you fetch data at a location in the database, all of its child nodes are retrieved as well. This could result in downloading unnecessary amounts of data, which is undesirable especially in context of mobile client applications, which often work on mobile networks with limited bandwidth dictated by the FUP (Fair Usage Policy).

Flatten data structures The previously discussed problem can be solved by keeping the data structure as flat as possible. In practise, a nested data structure can be split into separate paths instead, so that it can be efficiently downloaded in separate calls, as it is needed [57].

Data duplication One of recommended ways for increasing the read performance of the system is data duplication. For example, in the designed system, there can be a huge number of test definitions, while a user can be a member of only few of them. In such large data scenario, a query of all definitions would be too slow to get only those available to a single user [4]. Instead, for each user a list of available definitions is kept, resulting in duplication of data such as test definition's name, but also improving the performance of an operation, which is expected to be performed often.

4.1.1.2 Designed Database Structure

This section describes the database structure that was designed with respect to the previously recommended best practises and principles. For each part of the data structure, examples of real data (with a bit more meaningful ids) are included.

User and User mapping Each user's information is stored as a child of a `users` node. For quick verification of user's registration within the system, a `user_mapping` structure was added. Because the email address is unique, it can be used as a key. However, as dots are not allowed within Firebase keys, these characters are replaced by underscores.

```
{
  "users": {
    "user_id_001": {
      "name": "James Bond",
      "email": "james.bond@google.com"
    },
    "user_id_002": {...}
  },
  "user_mapping": {
    "james_bond@google_com": "user_id_001",
    "john_smith@google_com": "user_id_002"
  }
}
```

Listing 4.1: User and user mapping database structures

Data Wrapper As already mentioned when discussing the domain model in section 3.1, each file consists of metadata (`data wrapper`) and its contents (`abstract data`). Each data wrapper contains basic attributes such as id, name, file type, and time of creation, but also a map of user access rights, a link to the contents of the file within `abstract data`, and a list of links that reference this file, as described in the next section.

```
{
  "data_wrapper": {
    "data_wrapper_id_001": {
      "id": "data_wrapper_id_001",
      "name": "Pre-Test Questionnaire",
      "timestamp": "1492980949082",
      "initialized": true,
      "metadata": {
        "type": "QUESTIONNAIRE"
      },
      "rights": {
        "user_id_001": "write",
        "user_id_002": "read"
      },
      "links": {
        "link_id_001": "path/to/datawrapperlink/location",
        "link_id_002": "user_tests_data/user_id_002/test_definition_id_001"
      },
      "data": "/abstract_data/data_wrapper_id_001"
    }
  }
}
```

Listing 4.2: Database data structure of data wrapper

Data Wrapper Links In order to improve the read performance, data wrapper links are widely used thourough the system. Using the previously discussed *data duplication* principle, some data wrapper's information such as the name and type is duplicated, which enables to easily display a link to said file within the UI without the need to perform another query for these attributes.

```
{
  "id": "data_wrapper_id_001",
  "name": "Pre-Test Questionnaire",
  "type": "QUESTIONNAIRE"
}
```

Listing 4.3: Database data structure of data wrapper links

Abstract Data File's abstract data is the part, which most benefits from the schemalessness of the used database. The structure of children of the `abstract_data` node is only dependent on the type of the particular file.

As shown in listing 4.4, the first child contains a list of questionnaire questions, whereas the second child contains a timestamp and a link to the photo stored in Cloud Storage.

```
{
  "abstract_data": {
    "data_wrapper_id_001": {
      "question_id_001": {
        "id": "question_id_001",
        "order": 0,
        "text": "How often do you use a computer?"
      },
      "question_id_002": {
        "id": "question_id_002",
        "order": 1,
        "text": "When was the last time you watched cat videos on YouTube?"
      }
    },
    "data_wrapper_id_009": {
      "timestamp": "1492980949082",
      "photo_link": {
        "downloadUrl": "https://image-url.com/image.jpg",
        "fileName": "e21e0e4c-fcfb-4ad8-9eda-ea6c691943ad.jpg",
        "progress": 100,
        "status": "FINISHED"
      }
    }
  }
}
```

Listing 4.4: Database data structure of abstract data

Test Definition Test definitions are stored as children of a `tests` node. Each definition is further divided into several sections: `info`, `members`, `markers`, `instances`, and `data`. There are two reasons for this structure. First, it allows to setup user access rules for each of the nodes separately. Secondly, the flattening of the structure enables loading only relevant information (i.e. it's rarely needed to load the whole test definition, more often only part, such as its instances, is needed).

```
{
  "tests": {
    "test_definition_id_001": {
      "info": {
        "id": "test_definition_id_001",
        "name": "Firebase UX Testing",
        "info": "Some basic information",
        "timestamp": "1492980949082",
        "pre_test_questionnaire": {...},
        "post_test_questionnaire": "",
        "participants": {
          "participants_id_001": {
            "id": "participants_id_001",
            "name": "Participant Name #1"
          }
        },
        "test_cases": {
          "data_wrapper_id_003": {...}
        }
      },
      "members": {
        "user_id_001": {
          "role": "owner",
          "name": "james.bond@google.com",
          "id": "user_id_001"
        }
      },
      "markers": {
        "marker_id_001": {
          "id": "marker_id_001",
          "order": 0,
          "text": "Usability Problem"
        }
      },
      "instances": {
        "test_instance_id_001": {
          "id": "test_instance_id_001",
          "test_definition_id": "test_definition_id_001",
          "name": "Test Instance 1",
          "timestamp": "1492980949082"
        }
      },
      "data": {
        "data_wrapper_id_006": {...}
      }
    }
  }
}
```

Listing 4.5: Database data structure of test definition

As already mentioned in data duplication principle, querying *all* test instances in the database to get only those that the currently logged in user has access to could be slow and it's definitely not a very scalable approach. Instead, each user has its own list, with links to available test instances, as seen in listing 4.6.

```
{
  "user_tests": {
    "user_id_001": {
      "test_definition_id_001": {
        "name": "Firebase UX Testing",
        "timestamp": "1492980949082",
        "role": "owner",
        "id": "test_definition_id_001"
      }
    }
  }
}
```

Listing 4.6: Database data structure of test definition links

Test Instance Test instances aren't stored directly under the `test_instances` node. Instead, the `test_instances` node contains children each identified by test a *definition*'s ID, which then contains all of the test definition's test instances. This allows loading all data of all instances of a definition using a single query, which is for example useful when deleting a test definition. The code example of this section can be found in listing D.1.

4.1.2 Securing Data

Firebase provides a way to limit data access and define data structure validation. These rules are stored and enforced by the Firebase server, independently from client application logic. This separation ensures data security even in cases when the client's data access logic is improperly implemented.

4.1.2.1 Rule Types

Three distinct rules types are available to be set for a database location. As the names suggest, `.read` and `.write` rules can be used to define when a database location is allowed to be read, respectively written to. The third rule type allows data structure validation, such as value type or parameter names [45]. For instance, the validate rule in figure 4.7 ensures that newly added data is a string.

```
"rules": {
    "foo": {
      ".read": true,
      ".write": true,
      ".validate": "newData.isString()"
    }
}
```

Listing 4.7: Read, write and validate rule types

It is important to note that in a case a `.write` or `.read` rule isn't present, then the evaluation at that database location defaults to `false`. If a `.validate` rule isn't defined, then new data is only not being validated before writing.

4.1.2.2 Predefined Variables

Firebase provides a set of predefined variables (see table 4.1), which can be used within security and validation rules. This allows referencing database data outside the location of the currently executing rule, as shown in listing 4.8.

root	The current data at the root of the database.
data	The current data at the location of the currently executing rule.
newData	The data that will exist at the location of currently executing rule, if the write operation is allowed.
$wildcard	A variable, which can be used to reference a key value that was used earlier in the rule structure.
auth	Authenticated user's token payload.

Table 4.1: Predefined varibles in Firebase [45]

```
"rules": {
    "users": {
      "$user": {
        ".read": "root.child('admins').hasChild(auth.uid),
        ".write": "auth.uid === $user",
        ".validate": "newData.hasChildren(['name', 'age'])"
      }
    }
}
```

Listing 4.8: Example usage of the predefined variables in Firebase rules

4.1.2.3 Rules Cascade

Another key concept of Firebase security is a so-called *rules cascade*, which defines the evaluation process of access rights. Evaluation of `.validate` rules is done only in context of current data location without any cascading. On the other hand, `.read` and `.write` rules evaluation follows a top-down model. If a parent node grants a permission, then this permission is recursively granted to all its children, even though the evaluation of a particular child could deny that permission [45].

Furthermore, the access right evaluation is an atomic operation, meaning that if access hasn't been granted at the queried location, then the read/write operation is denied immediately, even if for every child of this node the access would be granted. As a consequence, the read rules cannot for instance be used for query filtering, as it might at first glance seem.

4.1.2.4 Designed Security Rules

The security rules of the implemented system were designed according to the previously discussed principles and are as a whole available in listing D.2.

To limit user access to test definitions and its instances, the `member` map of the test definition was used to either deny or grant access based on the current user's id passed through the `auth` variable.

For access to individual data wrappers, its `rights` data, which specifically defines read or write access to a user, has been used. This structure was also referenced in rules protecting data wrapper's abstract data.

Security rules of user-specific data (such as the list of user's test definitions) took advantage of the fact that such data is stored within a node whose key is equal to user's id. Then, a path wildcard (`$userId`) could be used for comparison with current user's id.

4.2 Firebase Cloud Storage

4.2.1 Data Structure

Cloud Storage for Firebase stores data within a Google Cloud Storage [49] bucket, where the files are presented in a hierarchical structure, similar to the Firebase Realtime Database

[40] with the difference that the hierarchy nodes are referred to as directories. Directories can recursively contain other directories or files. Whereas files consist only of their data and metadata.

4.2.1.1 Designed Storage Structure

In the implemented system, all files are represented using the data wrapper structure within the database, which references its abstract data.

In case of textual data, all the data can be stored directly in the database. Binary files, such as photos and videos, are to be saved in the Cloud Storage, and only a link to such file should be stored within the abstract data structure in the database.

As each binary file in Cloud Storage is directly tied to a single data wrapper, the data structure can be very simple - the root bucket directory contains a list of directories, each representing a single data wrapper. All of data wrapper's binary files are then stored within the respective data wrapper directory.

4.2.2 Securing Data

Similar to the Realtime Database Security Rules, the Storage Security Rules can be used to determine who has read and write access to files stored in Cloud Storage, as well as to ensure the structure of a file and its metadata.

Unfortunately, the Realtime Database data is not directly accessible from the Storage rules, which means it's not possible to read the map of user rights stored in the data wrapper within the database. Luckily, there are other ways to achieve a fine-grained user access control to a file.

4.2.2.1 Custom Authentication Token

One way to pass additional information to Storage rules is the use of custom authentication tokens, which allow to add custom parameters that can be used during the authorization phase. This concept is shown in an example in listing 4.9, where a `groupId` passed in the token is being compared with a corresponding path wildcard. However, there is one great limitation of this approach, as the generated token's total payload must be less than 1KB in size [42].

```
match /files/{groupId}/{fileName} {
  allow write: if request.auth.token.groupId == groupId;
}
```

Listing 4.9: The user of custom authentication token in Storage rules

4.2.2.2 File Metadata

Another approach to have all the necessary information for user-based security available in Storage rules is to use the metadata, which is stored alongside each file within the Cloud Storage. Besides information such as the file's size or content type, metadata can also store custom key/value pairs with additional data [47].

As seen in listing 4.10, this approach was used in the implemented application, and is further discussed in the following section.

4.2.2.3 Designed Security Rules

In the implemented system, the security rules were simply configured to allow reading to users with `read` or `write` access, and writing only to users with `write` access. For this to be possible however, the map of user rights that is stored within the data wrapper in the database has to copied over to the metadata of all its files *and* updated each time the user rights change.

```
match /b/uxtester-8e3c5.appspot.com/o {
    match /{dataWrapperId}/{file} {
        allow read:  if resource.metadata[request.auth.uid] == 'read' ||
                        resource.metadata[request.auth.uid] == 'write';
        allow write: if resource == null ||
                        resource.metadata[request.auth.uid] == 'write';
    }
}
```

Listing 4.10: Cloud Storage Security Rules of the implemented system

Chapter 5

Client Applications Implementation

In this chapter, the implementation of client applications for test administration (Admin app) and data collection (Logger app) is described. First, the Android operating system is briefly introduced, followed by a description of the applications' architecture. After that, the implementation details of the UI layer are explained. Finally, the implementation of data layer, which is connected to both the Firebase Realtime Database and Firebase Storage is discussed.

5.1 Android

Android is a Linux-based mobile operating system actively developed by Google. Since the launch in 2008, the support of the Android OS had been expanded from just mobile phones to a range of other devices, such as tablets, televisions, wearables, or even cars [16]. As of the end of 2016, Android claims a 81.7% market share in global smartphone sales [48].

For implementation of native Android applications, developers can use the freely available Java SDK, which provides fundamental building blocks such as Activities, Fragments, Services, or Views. In this thesis however, these basics of Android development will not be discussed. To learn more about Android and development fundamentals see [50] and [35].

5.2 Architecture

The architecture of the implemented client applications follows a Model-View-Presenter architectural pattern. As can be seen in figure 5.1, the MVP pattern is a derivation of the MVC (Model–View–Controller) architectural pattern. The main difference between MVC and MVP is the absence of direct communication between the View and the Model components in MVP.

Model In MVP, Model represents a data layer, which is responsible for handling the business logic and communication with the database layers [17].

View The responsibility of the View is the presentation of data, but also notifying the Presenter about user actions [17]. In the context of Android applications, the View can be implemented by Activities, Fragments or custom Android views.

Presenter The Presenter communicates with the Model to retrieve necessary data, but also handles the UI logic, reacts to user input actions from the View, and manages the state of the View [17].

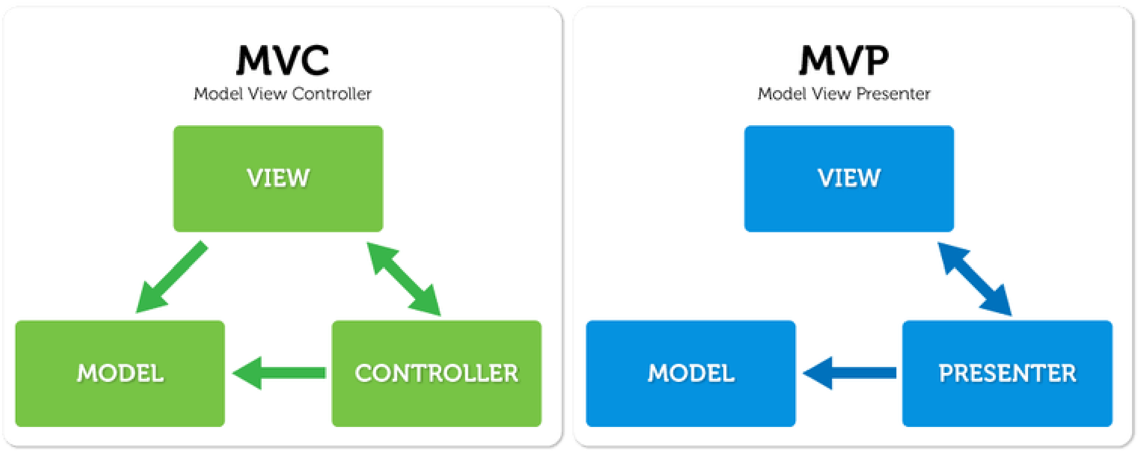


Figure 5.1: Comparison of MVC and MVP architectural patterns [15]

As the View and the Presenter need to communicate with each other, they need to store a reference to one another. To make the components loosely coupled and more testable, both the View and the Presenter are abstracted through the use of a `IView`, respectively `IPresenter` interface. The model is implemented by a class `FirebaseSync` and its subclasses.

5.3 Sharing Code

As some functionality of the Admin and Logger applications overlaps, it was necessary to configure the project in such way that allows code sharing in order to avoid duplication of code and resources.

```
productFlavors {
    admin {
        applicationIdSuffix "admin"
    }

    logger {
        applicationIdSuffix "logger"
    }
}
```

Listing 5.1: Project build script

The project is divided into three folders: admin, logger, and main. The admin and logger folders contain java source files and resources (e.g. icons, strings, layouts, styles) only used within the admin application, respectively the logger applications. The main folder contains source files and resources, which can be used within both applications.

Next, the created gradle build script uses product flavors to define the available types of applications to be build. For instance, the `admin` flavor will use the code and resources within the admin folder as well as the default main folder. Furthermore, the product flavor settings allow to create a unique application ID for packaging and distribution of each flavor without the need to modify the source code. For example, as shown in listing 5.1, the admin flavor appends `admin` to the base package name `cz.cvut.uxtester`, resulting in an ID of `cz.cvut.uxtester.admin`. More information about build configuration is available at [32].

5.4 RxJava

In Android, possibly time intensive operations, such as data loading or internet requests, have to be done on a thread separate from the main UI thread, otherwise the whole application would become unresponsive. One of the recommended ways is the use of AsyncTasks, which allow to perform an operation on a background thread and publish the result back to the UI thread [26]. For simple tasks, using AsyncTasks is completely sufficient. However when dealing with complex logic, which for instance needs to chain or combine asynchronous operations, using AsyncTasks leads to equally complex code, which is difficult to read, let alone maintain.

Another popular and much more versatile solution to background processing in the Android world is called RxJava. RxJava is a Java implementation of Reactive Extensions (Rx) library, which combines ideas from the Observer pattern, the Iterator pattern, and functional programming [55].

```java
ClockSkewSync.getClockSkew()
            .map(clockSkew -> System.currentTimeMillis() + clockSkew)
            .map(timestamp -> getFormattedTimestamp(timestamp))
            .subscribeOn(Schedulers.io())
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe(new SingleObserver<String>() {
                @Override
                public void onSuccess(String time) {
                   mView.showTime(time);
                }

                @Override
                public void onError(Throwable e) {
                   mView.showError();
                }
            });
```

Listing 5.2: Loading current server time with RxJava

The library provides three main components: observables, subscribers, and operators. As the source of data, an observable can emit zero or more items, and can finish its data flow either successfully or with an error [12]. A subscriber consumes the items emitted by the observable. And finally, operators can be used to create, combine, filter, or transform observables. For instance, the `filter()` operator receives items from an observable but only forwards items matching a given condition. The `map()` operator transforms each received item into another object, which allows extracting, enriching, or wrapping the original items [19].

Another operator, which is thoroughly used in the implemented applications is a `zip()` operator, which creates a new observable from 2 other observables by combining the items they emit in pairs. For example, when initiating file access rights, both the test definition's members and the file itself need to be fetched. By zipping the two observables (one observable for members, one observable for the file), we receive all the data at once, or in case of error none at all. Naturally, all of the operators can be chained after one another, as shown in figure 5.2. The full list of implemented operators is available at [56].

5.5 UI Implementation

This section discusses details of the user interface implementation, and the libraries used make the application's UI follow the Material Design Guidelines [54] or to reduce the lines of boilerplate code.

5.5.1 Data Binding Library

In Android, the view layout is usually defined in an XML file, which is then inflated by an Activity or Fragment. To access a particular view, the `findViewById()` method has to be used with the view's ID as defined in the XML layout file, and the result cast from base View class to the proper type, as depicted in listing 5.3. In practise, this creates a lot of seemingly unnecessary boilerplate code. To avoid it, the Data Binding library was used.

```java
setContentView(R.layout.activity);

Toolbar toolbar = (Toolbar)findViewById(R.id.toolbar);
setSupportActionBar(toolbar);

FloatingActionButton fab = (FloatingActionButton)findViewById(R.id.fab);
fab.setOnClickListener(new InnerOnClickListener());
```

Listing 5.3: Layout inflation without the use of Data Binding library

As shown in listing 5.4, the Data Binding library's method for layout inflation returns an instance of a binding class that is automatically generated from the XML layout file. Using this object, all views of the used layout can be accessed as public fields.

The Data Binding library offers a lot more functionality, such as binding variables to views, which can be useful when implementing an application with a MVVM (Model-View-ViewModel) architecture [34]. However, none of this functionality was used in this project.

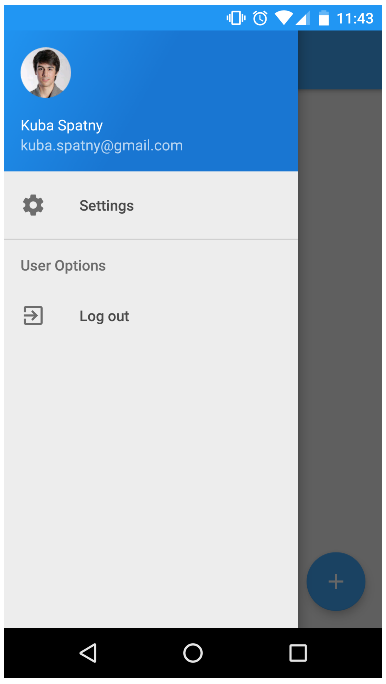
```java
ActivityBinding ui = DataBindingUtil.setContentView(this, R.layout.activity);

setSupportActionBar(ui.toolbar);
ui.fab.setOnClickListener(new InnerOnClickListener());
```

Listing 5.4: Layout inflation using the Data Binding library

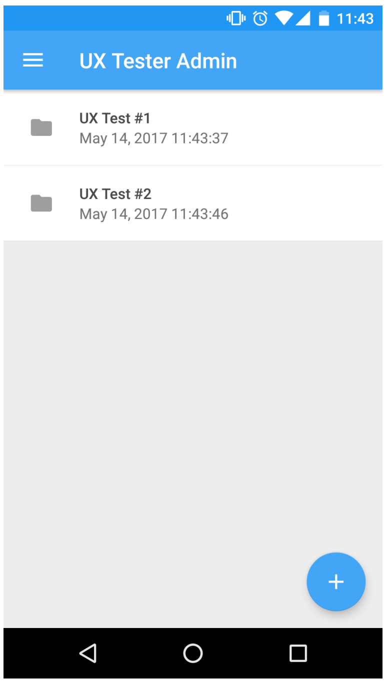
5.5.2 Design Support Library

Material design is a visual language created by Google in order to unify visual, motion, and interaction design across platforms and devices [54]. To make the implementation of material design in Android applications easier, the system provides the newly designed components in Android 5.0 and above. However, these components are not available in earlier (still supported) versions of Android. So, in order to achieve unified style across all supported versions, the Design Support Library [58] was used.

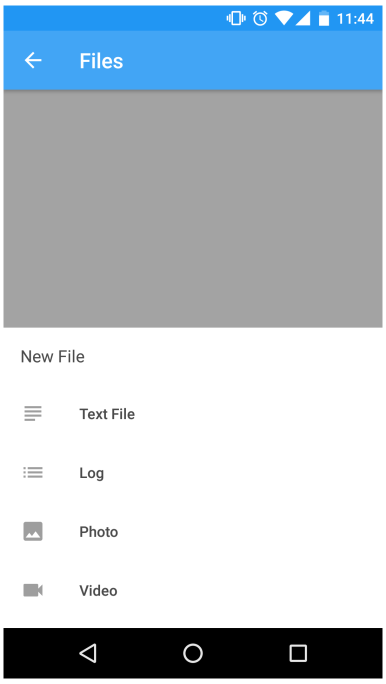
Not only this library provides all the components available in the newer Android systems, it also implements other components, which were added to the Material Design specification at a later time. Some of the used components, such as the Navigation Drawer, Floating Action Button, or Bottom Sheet, can be seen in figure 5.2.



(a) Navigation Drawer



(b) Floating Action Button



(c) Bottom Sheet

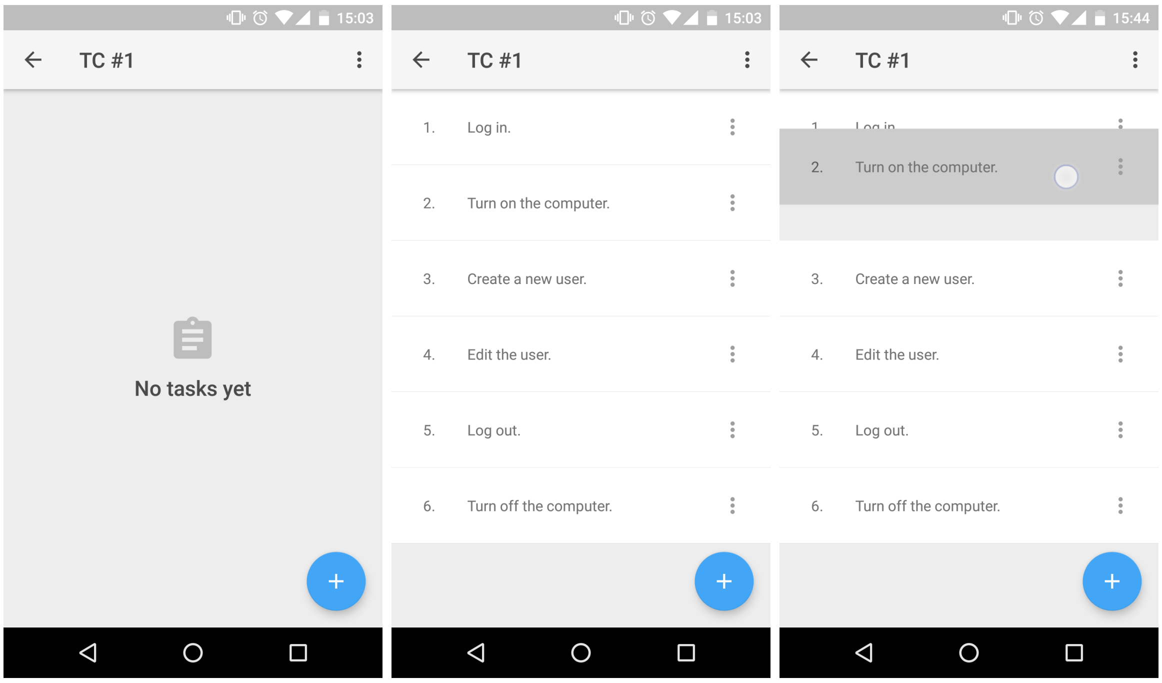
Figure 5.2: Screenshots from the Admin application

5.5.3 Firebase UI

FirebaseUI is a part of the provided software development kit by Firebase, which focuses on binding standard Android UI elements to data stored in the realtime database [46].

For instance, the `FirebaseRecyclerAdapter` class allows displaying a collection of items, which are automatically loaded based on a database reference or query passed during initialization. After loading the initial data, the adapter keeps listening to changes within the displayed collection and refreshes the lists as items are added, edited or removed.

This adapter was further extended to provide missing functionality that was needed. First, it was impossible to tell whether the adapter is still loading data, or whether the collection simply doesn't contain any items. In such case, an empty state view should be shown in place of the loading progress bar. By also registering on the passed database reference or query, the new adapter is notified when loading has finished and uses the item count of the underlying adapter. The implemented functionality can be seen in figure 5.3a. Another added feature, shown in figure 5.3c, allows the user to reorder the list items using a drag and drop gesture.



1. Log in.
2. Turn on the computer.
3. Create a new user.
4. Edit the user.
5. Log out.
6. Turn off the computer.

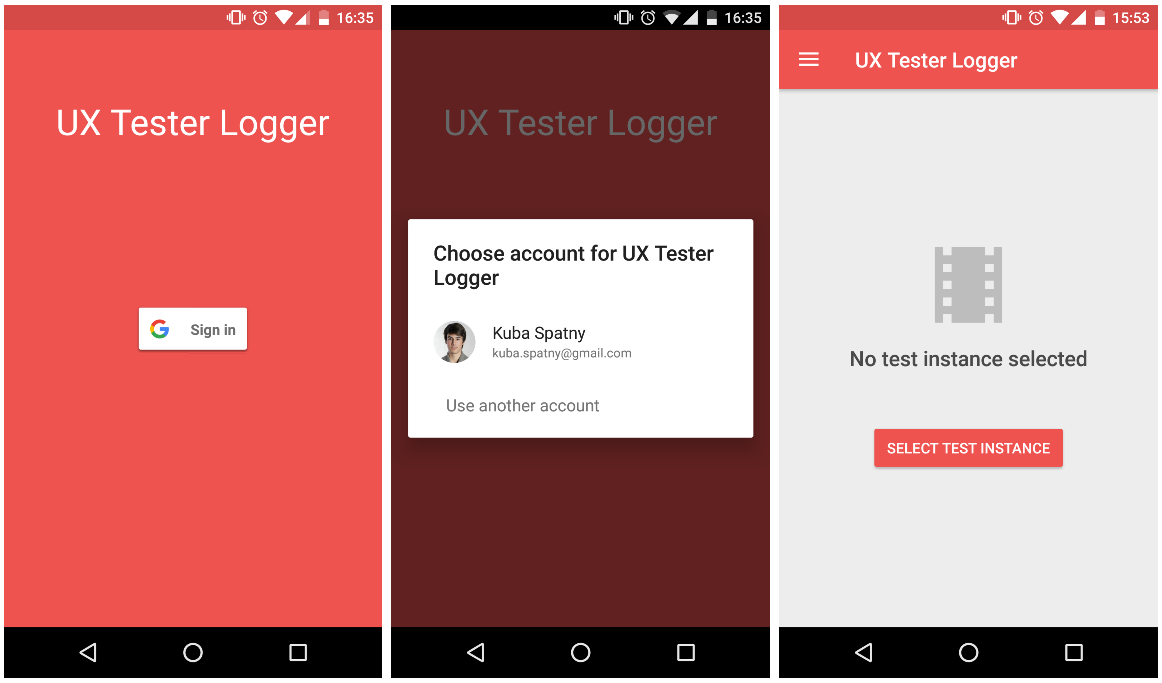
(a) Empty State (b) Loaded Items (c) Drag and Drop

Figure 5.3: Screenshots showcasing the use of FirebaseUI library

5.5.4 Styling

As already described in section 5.3, the project was divided into 3 subfolders containing the source code and resources. This structure allowed easy styling of both implemented applications separately.

The easiest way to visually distinguish the applications, while still maintaining a unified look, was differentiating the applications' primary colors. The primary color of the Admin application is blue as already shown in figure 5.2. The Logger application uses the color red, as can be seen in figure 5.4. File detail, which can be accessed from both applications, uses a neutral grey color (see figure 5.3).



(a) Login Screen (b) Choosing Account (c) Empty State

Figure 5.4: Screenshots of the Logger application

5.6 Authentication and User Management

Authentication within the implemented applications makes use of the Firebase Authentication SDK and its ability to use federated identity providers like Google, Facebook, Twitter and others. As each Android device is tied to at least one Google account, the use of Google as an identity provider was a clear choice.

The figures 5.4a and 5.4b show the authentication workflow in the Logger application. After clicking the *Sign in* button, a dialog is shown, and it allows choosing either an account, which is already registered within the Android system, or specifying another Google account. After an account is selected, the user is registered within the implemented system and is forwarded to the application's main screen. Even though only one authentication provider is supported at the moment, the system was configured to prevent users from creating multiple accounts using the same email address with different authentication providers.

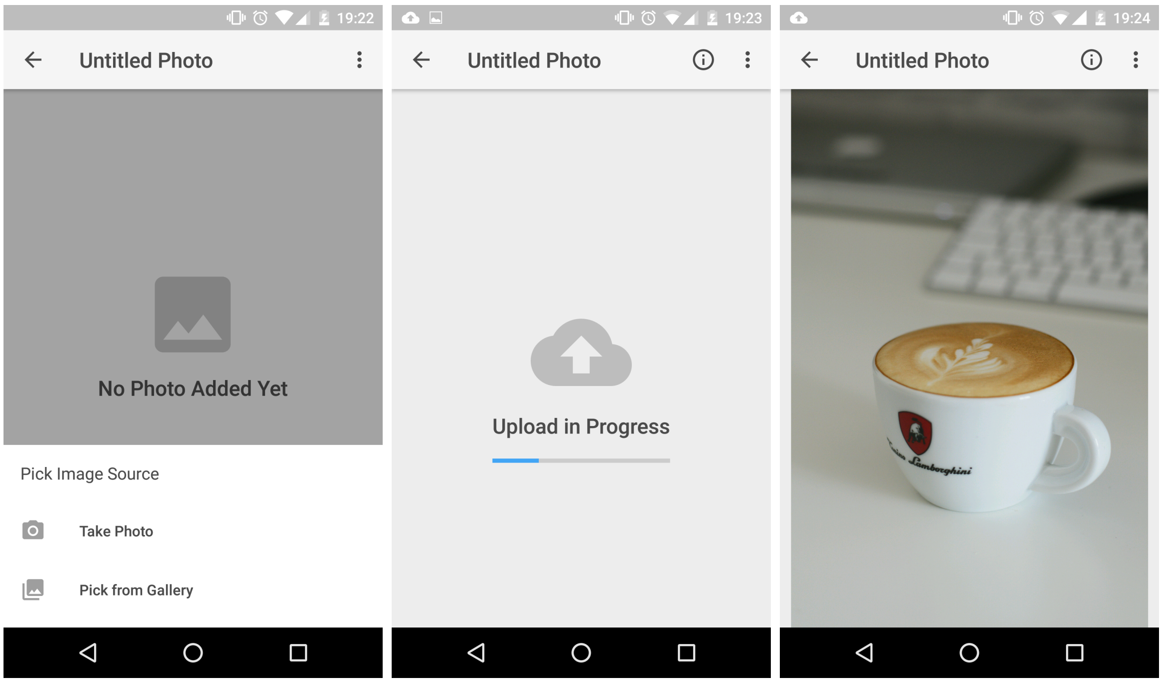
5.7 Files Implementation

This section describes the implementation of some of the non-trivial file types such as log, photo, and video, but also the general principles, which were used to ensure that the system will be prepared for further extensions.

5.7.1 Data viewer resolution

As already discussed in section 3.1, each file consists of its abstract data and a data wrapper, which unambiguously defines the type of the file. When the user selects a file, its data wrapper is passed to the `DataViewerActivity`, which uses the file type to resolve corresponding data viewer instance implemented by a Fragment. The data wrapper is passed on to the data viewer Fragment, which is displayed within an Activity. It is then the data viewer's responsibility to load, parse and display the abstract data.

In the future, a support for additional data types may be added. However, not all users will update the application at the same time, and some may never update it at all. So, it is possible that the application will encounter an unknown file type. Therefore, if no corresponding data viewer is found, an `UnsupportedDataTypeFragment` is shown instead.



(a) Available Image Sources (b) Upload in Progress (c) Uploaded Photo

Figure 5.5: Data Viewer of Photo file

5.7.2 Photo

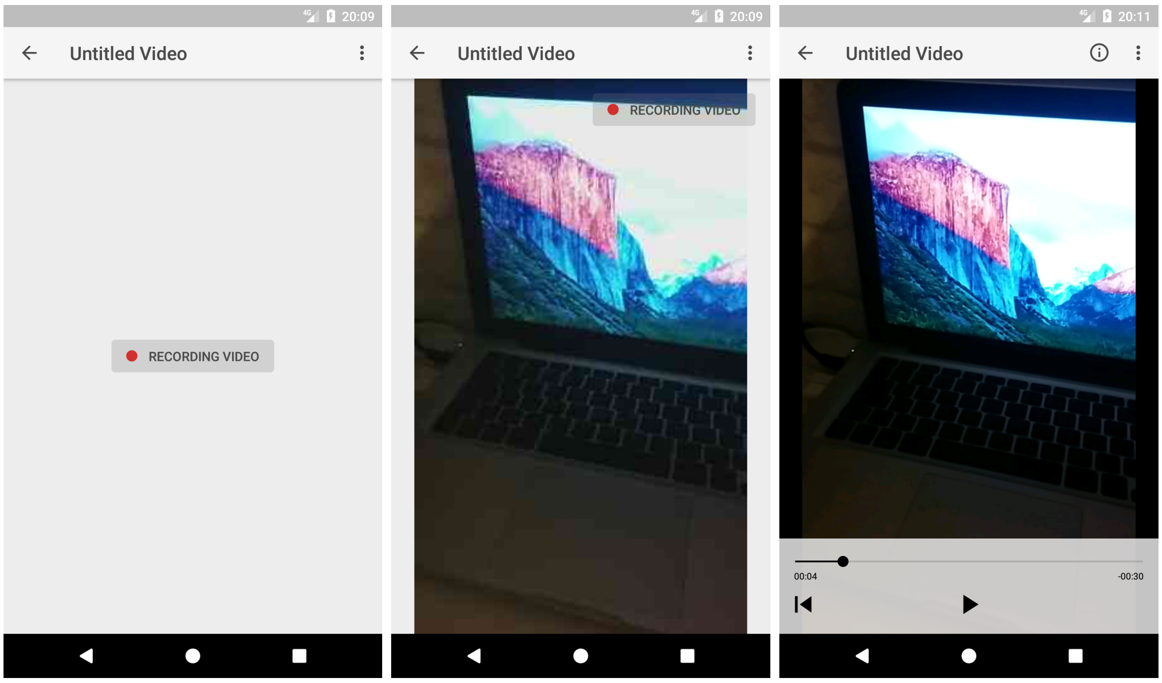
After creation, the photo file allows the user to add a new photo from two sources. The first supported source is an image gallery, which enables to import a photo which was already taken earlier. The second option forwards the user to a default camera application, where he/she can take a new photo.

When the photo is picked or taken, it gets copied to the application's internal memory and an upload request is created. During the upload, current progress is shown, as depicted in figure 5.5b. After the image had been successfully uploaded, it is displayed in a view, which also supports the pinch and pan gestures. The uploading process of binary files will be discussed in greater detail later in section 5.10.

5.7.3 Video

One of the requirements for a video file was the ability to show video previews during the video recording. Because of this, the default camera application couldn't be used. As an alternative, an open-sourced library Annca [25] was used internally within the implemented applications.

The library worked well out-of-the-box, however it still didn't allow to show the required video previews. Being open-sourced, the library wasn't added as a gradle dependency, but included in the project as a git submodule. This way, it was possible to extend the library with all the missing functionality.



(a) No Preview Available (b) Video Preview (c) Uploaded Video

Figure 5.6: Data Viewer of Video file

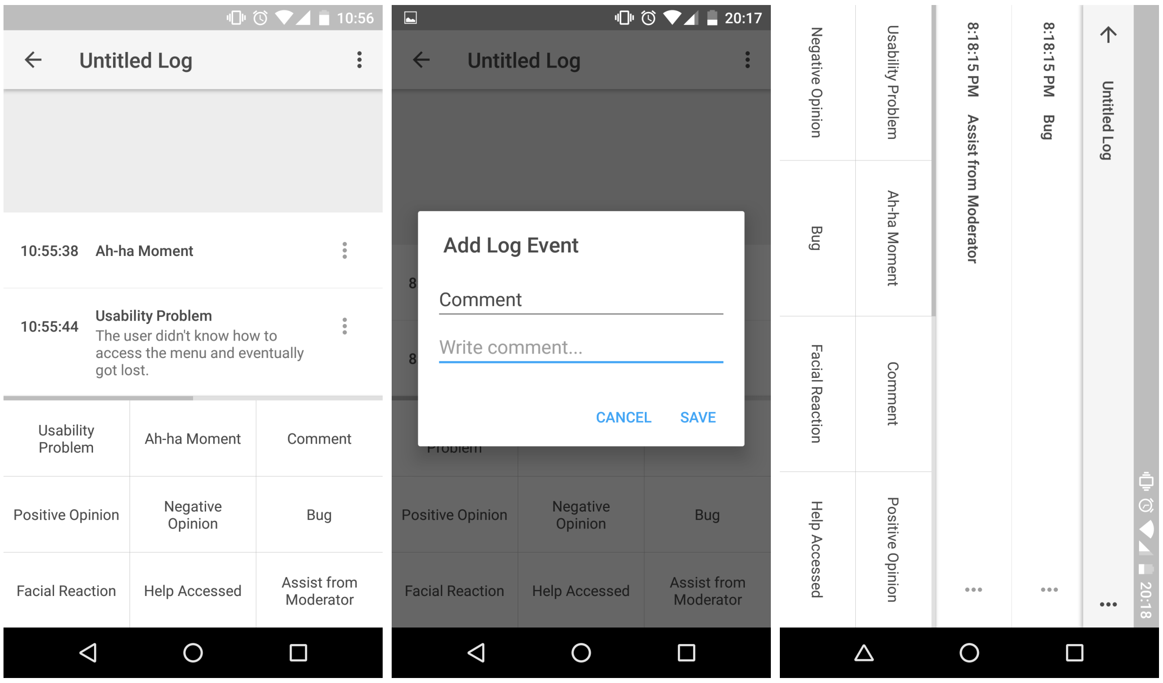
During video recording, the data of camera's canvas view is periodically copied to a bitmap, which is further resized and compressed to minimize the size of the preview file. Each preview file then proceeds to be uploaded. In case the previous file is still being uploaded, the current preview is skipped. When the video file is accessed by a different user on a different device, the video preview is displayed as shown in figure 5.6b. If no preview is available, the user is at least notified that a video recording is in progress (figure 5.6a).

Similarly to the photo file, after the video was recorded, an upload request is issued. When the video is successfully uploaded to the Firebase Storage, a video player allowing to replay the video is displayed (figure 5.6c).

The library was also edited to be able to specify the destination of the video recording. By default, the files were added to the Documents folder, however it was needed for the video to be stored in the application's internal memory for it to be safely uploaded. Of course, it would be possible to copy the file after the recording has finished, but this operation could fail due to insufficient storage space.

5.7.4 Log

Another data source in usability testing is observer notes, which is a log, where each entry represents a real-life event that happened during the test execution and might be an indication of usability problems.



10:55:38 Ah-ha Moment

10:55:44 Usability Problem
The user didn't know how to access the menu and eventually got lost.

8:18:15 PM Bug

8:18:15 PM Assist from Moderator

(a) Log (b) Additional comment (c) Landscape mode

Figure 5.7: Data Viewer of Log file

As shown in figure 5.7a, each log event consists of a timestamp, marker, and possibly an additional comment. Timestamps, which are set automatically by the application when a user adds a new event, could later be used to synchronize the log with audio and video recordings. Markers, which symbolize the nature of the problem, are loaded from the test definition of the current test instance. To simplify the process of adding a new event, the user can select, whether a dialog allowing to write a comment should be shown (5.7b), or whether the event should be added right away. Either way, both the marker and comment can be edited at a later time.

The number of available markers is not limited, which means that all the markers might not fit on the screen. For this reason, the grid of items is horizontally paginated. Moreover, the size of the grid is dynamically adjusted to the screen dimensions. For instance, in portrait mode (5.7a) the grid page consists of 3 rows and 3 columns, whereas when rotated to landscape mode, the grid transforms to a layout with 2 rows and 4 columns.

5.8 Clock Skew

After the test, timestamps of photos, videos and log events should form a single timeline. However, the local time on different devices can vary, which could result in synchronization errors, when for instance trying to match log events to a recorded video. To solve this, the Firebase platform provides a functionality called clock skew.

The database reference `.info/serverTimeOffset` was used to obtain a value that estimates the client's clock skew with respect to the Firebase Realtime Database's servers [37]. When this value is added to the client's local time, the server time estimate is obtained. This way, we can be sure that the recorded timestamps will be correct regardless the local time settings. However, it's important to note that the offset's accuracy can be affected by networking latency, which makes it only useful for discovering time differences greater than 1 second.

5.9 FirebaseSync

In context of the MVP architecture, the model is implemented by a `FirebaseSync` class and its extensions. The base class facilitates access to the Firebase database instance and the currently authenticated user. Each subclass (e.g., `TestDefinitionSync`, `LogSync`, `QuestionnaireSync`) is responsible for data access and management of the particular model entity.

The Firebase SDK allows to read, set or remove the value of a single node, which might suffice in simple use cases. However, sometimes it's necessary to update data of several database locations at the same time. For instance, when adding a new test definition, the test definition itself has to be saved, but also a new reference to it has to be added to the user's personal list of available test definitions. Two `set()` operations could be used to achieve this, however, one of the operations could possibly fail leading to inconsistencies in our data.

The correct way of doing an update of multiple database locations in Firebase is called client-side fan-out, example of which can be seen in listing 5.5. Unlike the previous solution, the fan-out is an atomic operation, which can still success or fail, but only as a whole [4].

```
public static Single<Boolean> setDataSort(DataWrapperLink link,
                                          Map<Question, Integer> sort) {
  HashMap fanout = new HashMap();

  for (Map.Entry<Question, Integer> e : sort.entrySet()) {
    fanout.put("/abstract_data/" + link.id + "/" + e.getKey().id + "/order",
               e.getValue());
  }

  Task task = getDatabaseReference().updateChildren(fanout);
  return Single.create(new TaskSuccessObserver<>(task));
}
```

Listing 5.5: Client-side fan-out used for data consistency

The methods of `FirebaseSync` return the queried values either as a database reference or as an RxJava Observable. The database reference is used in cases, when a list of values is to be displayed and the reference is passed to the previously explained `FirebaseRecyclerAdapter`. In other cases, Observables are preferred as it allows to make use of the advantages of RxJava, such as filtering, mapping, chaining or thread configuration.

5.9.1 Offline Persistence

One of great advantages of Firebase Realtime Database is its ability to work even when the device is not connected to the Internet. As shown in listing 5.6, enabling this feature is a matter of few lines of code.

```
public class UXTesterApplication extends Application {
    @Override
    public void onCreate() {
        FirebaseDatabase.getInstance()
                        .setPersistenceEnabled(true);

        super.onCreate();
    }
}
```

Listing 5.6: Firebase offline persistence configuration

With offline persistence enabled, the Firebase Realtime Database client writes the data locally to disk, so all changes are available even after the user or the operating system restarts the application [38]. Once the connectivity is regained, all local changes are automatically sent to the Firebase Realtime Database server.

In offline mode, the Firebase client transparently loads data from the local cache. In most cases, this behaviour would be considered an advantage. However, during the implementation it was often needed to load the most up-to-date data from the server. Unfortunately, the only way to achieve this was through the use of transactions, which lack most of the advantages of the standard value listeners, such as notifications about data changes in real time.

Firebase Storage, which was used for storage of photos and videos, doesn't support offline use at all. Therefore, the offline persistence of binary files had to be implemented differently, as described in the following section.

5.10 File Upload Service

As previously stated, the Firebase Storage cannot be used without an Internet connection. However, one of the requirements for the implemented applications was the ability to record data while offline, which can often happen when testing outside lab conditions in the field.

Therefore, a binary file that should be uploaded to the Storage is first copied into the application's internal memory to a `pending files` folder. After that, an upload request is created and added to a requests queue, which is stored in the Realtime Database.

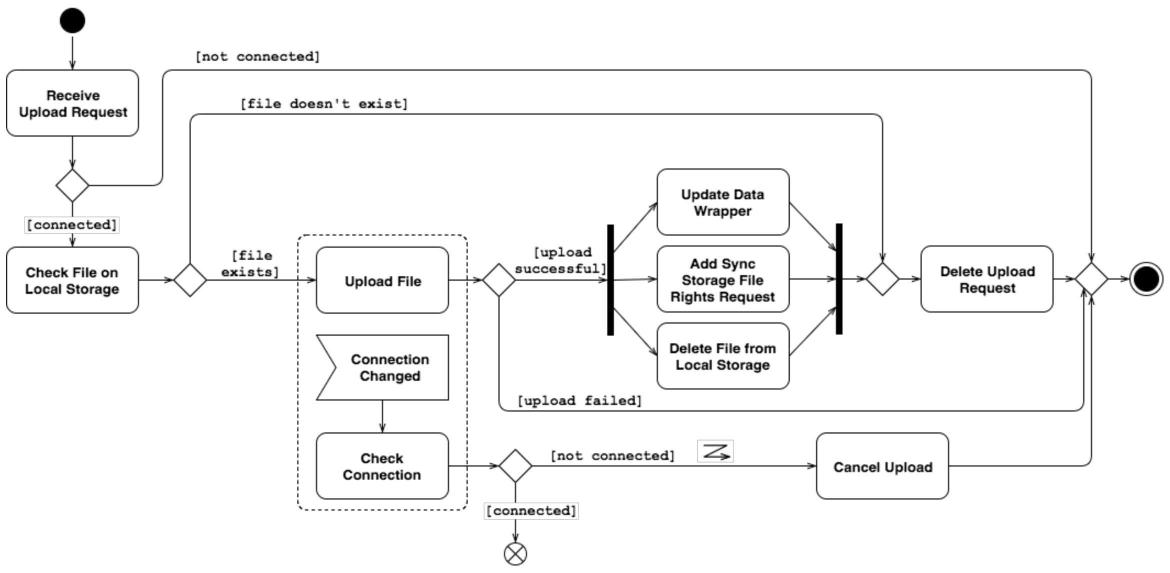


Figure 5.8: Upload request processing

Meanwhile, a background service is started, which then proceeds to check the requests queue. If an upload request is found, the service starts to process it as depicted in an activity diagram in figure 5.8. As the first step, the Internet connection is checked. Due to the size of photo and video files, mobile network connections are denied, unless it was specifically enabled by the user in the application's settings. If the connection doesn't meet the requirements, the request is left in the queue to be processed at a later time. In the opposite case, the binary file specified in the request is checked.

If the referenced file doesn't exist, then the upload request is deleted. If it does, the file is uploaded to the location described in the request. During the upload, the connection may change to a network, which no longer meets the previously checked criteria. In such case, the upload is canceled.

If the upload succeeds, the enclosing data wrapper is updated with an URL to the file, the locally stored file is deleted, access rights are requested to be synchronized, and the upload request is removed. On the other hand, if the upload fails, the upload request is left in the queue to be processed again later.

When discussing the security rules of Firebase Storage in section 4.2.2, it was explained that it is not possible to access data stored in the Realtime Database from the Storage security rules. Instead, each file in the Storage will use its metadata to store the information about user access rights. However, this means that the user rights as defined in the binary file's metadata have to be kept in sync with those in its data wrapper. This functionality is implemented by a second type of request, which is also handled by the service.

As shown in a diagram in figure 5.9, when a *sync* request is received, the connection is checked. In this case however, any connection will suffice as we only require it to load the most up-to-date version of the data wrapper directly from the server to avoid using an obsolete cached version. If the fetched data wrapper is properly initialized, all its storage file links are obtained and corresponding Storage files are updated with the current access rights. After the update has successfully finished, the sync request is deleted. If a failure occurs, the request is left in the queue, and will be processed again in the future.

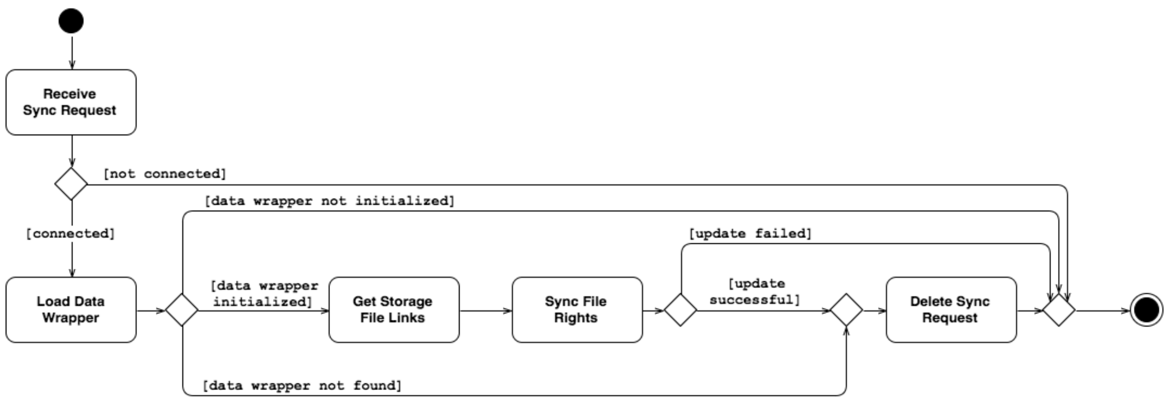


Figure 5.9: File rights sync request processing

Chapter 6

Testing

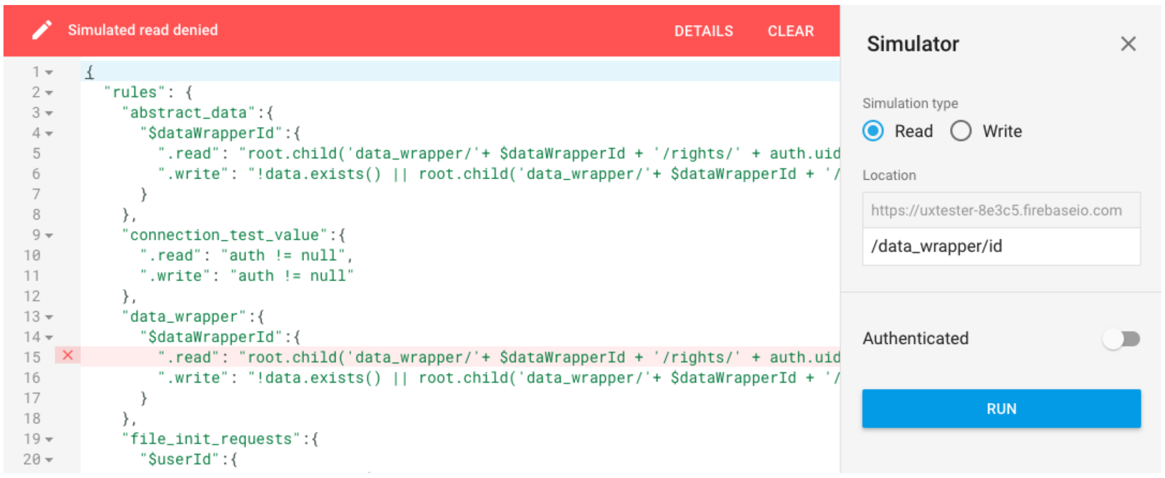
This chapter describes the testing methods, which were used during and after development of the system to ensure its quality. First, testing of database security rules is described, followed by the testing of implemented Android applications.

6.1 Database Security Rules Testing

Within the Database Rules section of the Firebase console, the access rules for the project can be defined, but they can also be tested using the provided Simulator tool.

In this tool, both read and write operations can be simulated on any specific location in the database. For each request the authentication provider and UID can be specified, or the request can be run as an unauthenticated user. The simulated operations are evaluated against the real database data, but of course without making any changes.

During the testing, the tool helped to discover several errors, where the rules were too restrictive. Moreover, the simulations were also useful in gaining full understanding of the evaluation concepts, such as the rules cascade.



```
{
  "rules": {
    "abstract_data":{
      "$dataWrapperId":{
        ".read": "root.child('data_wrapper/'+ $dataWrapperId + '/rights/' + auth.uid
        ".write": "!data.exists() || root.child('data_wrapper/'+ $dataWrapperId + '/
      }
    },
    "connection_test_value":{
      ".read": "auth != null",
      ".write": "auth != null"
    },
    "data_wrapper":{
      "$dataWrapperId":{
        ".read": "root.child('data_wrapper/'+ $dataWrapperId + '/rights/' + auth.uid
        ".write": "!data.exists() || root.child('data_wrapper/'+ $dataWrapperId + '/
      }
    },
    "file_init_requests":{
      "$userId":{
```

Simulation type: [✓ Read] Write

Location: https://uxtester-8e3c5.firebaseio.com /data_wrapper/id

Authenticated

Figure 6.1: Simulator of Firebase Realtime Database security rules

6.2 Compatibility testing

Android devices come in a variety of hardware and software combinations. To ensure that the client applications are working as expected, both real devices and software emulators were used during development and testing.

The implemented applications were tested on devices running the following version of the Android operating system: 4.1.2, 4.4.4, 5.1.1, 6.0.1, and 7.1.2. No problems connected to the use of different API versions were discovered, as the applications use the Android Support libraries, which provide backward-compatibility of the newer APIs to the older system versions.

Considering screen sizes and densities, both phones and tablets were used. Namely, tested densities range from `mdpi` to `xxxhdpi`, and screens in sizes 4.7", 5.0", 5.2", 7.0", and 10.2" were covered.

6.3 Component testing

The code of the implemented applications was tested using the standard JUnit tests, as well as Instrumented Unit tests, which are specific only to Android. Both types of tests were only run locally, however it would be possible configure a CI (Continuous Intergration) server to execute them periodically or with each new commit to the repository. This way, these tests could also be used to discover regression defects.

6.3.1 Standard Unit tests

Plain Java methods, which make no use of the Android APIs, were tested using the standard JUnit tests. The advantage of these tests is the fact that they can be executed locally on the development machine without the need for an Android device. On the other hand, it is not possible to easily test code that uses the Android framework APIs.

6.3.2 Instrumented Unit Tests

Unlike the previous type, the instrumented unit tests can take the advantage of the Android framework APIs [28]. However, to be able to run these tests, a physical device or an emulator is required. Therefore, the instrumented unit tests were only used to test code, which accesses the application's `Context` or other framework components, such as the `Parcelable` or `SharedPreferences` objects.

6.4 User testing

The user interface of client applications was based on the second version of the prototype that was presented in section 3.5. The prototype itself was tested with users, however the implemented applications also provide some additional functionality, which wasn't covered by the original prototype. In order to verify that the Android clients are still intuitive to use, another round of user testing was performed.

6.4.1 Test Scenarios

The test scenarios are based on usability testing data, which is used in a TUR (User Interface Testing) course at the CTU. The provided data addresses the testing process of Ashampoo Burning Studio program and contains information such as an application description, test goals, screener, questionnaires, task list, participants, found usability problems and their solutions, and lastly a video recording from one test execution.

In its original form, the data was a bit too long to be used in the user testing. Therefore, some repetitive parts (e.g., task list, participants) were shortened. The reduced version, which is referenced within the test scenarios, is available as an attachment to this thesis.

Test Administration

1. Log in to the application using the provided account [uxtester.user001@gmail.com].
2. According to the provided document[1], create a new test definition and fill all its information.
3. A new colleague joined our testing team. Add him to the project as a member.

   [kuba.spatny@gmail.com]
4. When logging, we want to have the following markers available: Comment, Positive Opinion, Bug, and UI Problem.
5. We are about to test with the participants, prepare everything needed to be able to start testing [filling questionnaires, creating observer notes, etc].
6. Log out.

Data Collection

1. Log in to the application using the provided account [uxtester.user001@gmail.com].
2. We are going to be testing with participant 1 now. Start by filling out the pre-test questionnaire.
3. Before the test starts, take a photo of our test setup.
4. We are starting the test now. Start telling the participant the tasks he should do.
5. Here is a video[2] of an executed test. Create a log of important events.
6. The participant finished all his tasks. Complete the test by filling out the post-test questionnaire.
7. The photo of our test setup seems a bit blurry. Take a new one instead.

[1]Document available as an attachment: test-data.pdf
[2]Video available as an attachment: test-data.mp4

8. Now we want to start the test with the next participant.
9. This time, you want to record a video of the participant while he's trying to accomplish the tasks.
10. Rename the video to "test 1".
11. You're going to be testing out in the field with no WiFi available there. Make sure the application is setup to upload files on a mobile network.
12. Log out.

6.4.2 Test Organization

Participants

To select the participants, no screener was used. The test participants were only required to have some basic knowledge of the usability testing process. Therefore, five students who were previously enrolled in the TUR course were chosen as participants. What's more, these students were also already familiar with the data used during the test.

Setup

The testing was performed in an empty classroom at the Karlovo Náměstí campus of the CTU. The participants used a Nexus 5 device, which had both the Admin and Logger applications installed. A second device (LG L9 P760) was used by the test moderator to observe previews of the video being recorded by the participants. Both phones were connected to the eduroam WiFi network. To play the video in step 5 of the Data Collection test scenario, a 13-inch MacBook Pro laptop was used. During the test execution, moderator took notes on paper.

6.4.3 Usability Problems and Solutions

6.4.3.1 Confusing naming of Files section

Problem Some of the participants didn't know how to start collecting test data, such as creating a log. The correct way was to create a file within the Files section of a test instance. Previously, the participants had no trouble using the Files section inside a test definition to create documents to record information such as the test goal or test setup.

Solution One way to solve the problem would be to rename the Files section to a more fitting name. Another solution would be to leave the Files section as is for other documents as used without a problem within test definition and add a whole new section strictly for test data collection.

6.4.3.2 Deleting a photo/video

Problem When a photo file is created, it contains no photo. After a photo is added from camera or gallery, only the whole file can be deleted. This confused participants, as they only wished to replace the existing photo. Although not encountered during the tests, the same problem applies to a video file as well.

Solution Allow an existing photo or video to be replaced or deleted without removing the file itself.

6.4.3.3 Inaccessible navigation drawer

Problem Within the Logger application, when a test instance is selected, the navigation drawer with settings and log out actions is not accessible. The participants had trouble finding out, that they had to first exit the selected test instance to be able to access the menu.

Solution Make the menu accessible from the screen with selected instance, as it acts as the main screen when an instance is selected.

6.4.3.4 Accidental log event clicks

Problem Some of the participants accidentally added log events while holding the phone and watching the video.

Solution Change the gesture used to add a new event from a simple click to a long click.

6.4.3.5 Test Case Selection

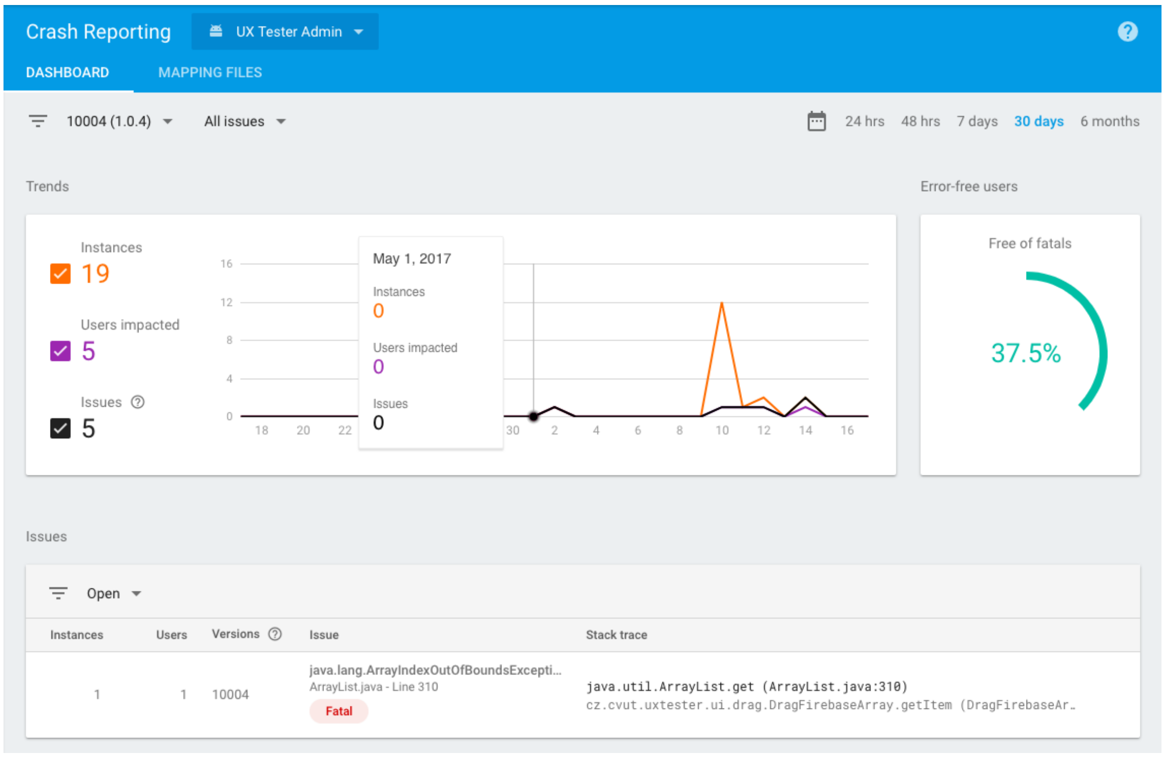
Problem When selecting a test case for a test instance, only the file names are displayed. One participant noted, that if there were multiple test cases available, it would be hard to pick one without seeing its tasks first.

Solution When selecting a test case, allow the user to display its contents before confirming the selection.

6.4.4 Firebase Crash Reporting

Although the applications were thoroughly tested, it is expected that a few bugs might still be hidden somewhere. To be able to detect these errors when the application is already deployed, the Firebase Crash Reporting was connected.

Unlike the crash reporting within Google Play Store, where users have to manually report individual crash events, the Firebase Crash Reporting automatically records fatal errors and their stack traces, but also collects additional data, such as device characteristics, performance data, and user circumstances when the error took place [41]. The recorded crash reports are then available in the Crash Reporting section of the Firebase console, as shown in figure 6.2.



Instances	Users	Versions	Issue	Stack trace
1	1	10004	java.lang.ArrayIndexOutOfBoundsExcepti… ArrayList.java - Line 310 Fatal	java.util.ArrayList.get (ArrayList.java:310) cz.cvut.uxtester.ui.drag.DragFirebaseArray.getItem (DragFirebaseAr…

Figure 6.2: Firebase Crash Reporting console

Chapter 7

Conclusion

The goal of this thesis was to create a system with client mobile applications that would allow the management and collection of data during usability testing in the real context of application use. In order to do so, the process of usability testing was analyzed. First, the steps of a usability test workflow were described, followed by a discussion of different test environments with an emphasis on testing in the field. In contract to testing in a usability laboratory, field testing doesn't provide a highly controllable environment, which for instance means that we cannot count on an Internet connection to be available at all times. These factors had to be later taken into account during the requirement analysis. Last, an overview of most common data types that are recorded in usability tests was presented.

Afterwards, existing applications, which can be used for data collection and analysis during usability testing, were researched. For each application, its functionality, advantages, disadvantages, and the possibility of use for testing in the field were considered. Many applications covering the whole usability testing workflow are available, however they usually focus on laboratory testing and are not ideal for testing in the field.

The previous analysis and client interviews were used to formulate requirements for the designed system. After the system's domain model was documented, a use case model was created to better describe the system's actors and the functionality each application covers. Next, available NoSQL database systems were researched, and the Firebase platform was selected as the most suitable option. Using recommended practices, a database schema was designed and data security configured.

Before implementation, a low fidelity prototype of both client applications was created and tested with users. Found usability problems were solved in a second version of the prototype. Based on the improved prototype, the client Android applications for test administration and data collection were implemented. Finally, the implementation of the system was verified by unit tests, whereas the system's usability was tested again with real users.

7.1 Contribution of Implemented Applications

Unlike most applications already available on the market, the created system offers an alternative, which was designed with testing in the field in mind. Thanks to focusing on field testing, the system can be used to record data with an unreliable Internet connection or even no connection at all. When a connection is available however, the system allows an easy collaboration of multiple test observers as recorded data is shared between all connected devices in real time.

Being built upon a schema-less NoSQL database, the system can be easily extended in the future with additional data types and other functionality as needed. Moreover, the possibility to use other new technologies, such as RxJava, helped to create code that is readable and maintainable.

Admittedly, using the Firebase Realtime database was difficult at times during development. After all, in comparison to most database systems, it is a relatively new technology. However in the end, I still stand by the decision of choosing this platform, as the Firebase team constantly works on improving the existing functionality and even keeps adding new features.

7.2 Future work

While the implemented applications fulfill the required functionality, many opportunities for extending the system remain. This section presents some of the possible improvements.

7.2.1 Firebase Cloud Functions

When the implementation of the applications was nearly done, the Firebase platform was extended with a new feature called Cloud Functions, which enables to run server code in response to events triggered by Firebase [29].

The perfect use case for this feature within the implemented system is the initialization of the data wrapper's access rights. Right now, it's up to the client applications to do this work. However, the client application can be offline at times and has to wait till the connection is regained in order to use the most up-to-date data stored at the server. Using the Cloud Functions, the access rules of a data wrapper could be automatically initialized on the server side when it's uploaded to the cloud.

7.2.2 Test and Production Environments

At the moment, the application uses only one database for both development and release builds. Once the system is deployed, it would be right to set up separate environments for development and production.

7.2.3 Archiving Test Definitions

After using the application during development for some time, the number of created test definitions was rather large. These definitions were just for testing and could be deleted without a problem, however in production, it would be more appropriate to be able to only archive these instances. Additional features could also include test definition sorting and searching.

7.2.4 Solving Found Usability Problems

The usability testing in section 6.4 discovered a couple of usability problems. Solutions to these problems were proposed, but they were not verified nor implemented. These problems are expected to be fixed during the future collaboration with the supervisor, after the solutions had been properly tested.

Bibliography

[1] J. Arlow and I. Neustadt. *UML 2 a unifikovaný proces vývoje aplikací: Objektově orientovaná analýza a návrh prakticky.* Brno: Computer Press, 2nd edition, 2007.

[2] D. Burke. Use case actors - primary versus secondary. `https://blogs.oracle.com/oum/entry/use_case_actors_primary_versus`, seen on 15. 4. 2017.

[3] M. Cade and H. Sheil. *Sun Certified Enterprise Architect for Java™ EE Study Guide.* Prentice Hall, 2nd edition, 2010.

[4] D. East. Client-side fan-out for data consistency [online]. `https://firebase.googleblog.com/2015/10/client-side-fan-out-for-data-consistency_73.html`. [Accessed: 2017-03-11].

[5] M. Fowler and D. Rice. *Patterns of Enterprise Application Architecture.* Boston: Addison-Wesley, 2003.

[6] E. Füzesséry. *Aplikace pro sběr dat z testů použitelnosti v mobilním prostředí.* 2015. Bachelor's thesis.

[7] J. Gray. To blob or not to blob: Large object storage in a database or a filesystem. Technical report, April 2006.

[8] A. Harley. Icon usability [online]. `https://www.nngroup.com/articles/icon-usability/`. [Accessed: 2017-02-01].

[9] I. Holubová, J. Kosek, K. Minařík, and D. Novák. *Big Data a NoSQL databáze.* Grada Publishing, a.s., 2015.

[10] V. Hotový. *Aplikace pro záznam mobilních testů uživatelské činnosti na zařízení typu tablet.* 2012. Bachelor's thesis, `https://dspace.cvut.cz/handle/10467/10745`.

[11] M. Jones and G. Marsden. *Mobile Interaction Design.* Wiley, 2005.

[12] D. Lew. Grokking rxjava, part 1: The basics [online]. `http://blog.danlew.net/2014/09/15/grokking-rxjava-part-1/`. [Accessed: 2017-04-03].

[13] I. Malý. *Analysis of Usability Tests with Context Model.* PhD thesis, Czech Technical University in Prague, 2012.

[14] N. Marz and J. Warren. *Big Data Principles and best practices of scalable realtime data systems.* Manning Publications, 2015.

[15] T. Megali. Model view presenter (mvp) in android, part 1 [online]. `http://www.tinmegali.com/en/model-view-presenter-android-part-1/`. [Accessed: 2017-03-27].

[16] R. Meier. *Professional Android 4 Application Development.* Wiley/Wrox, 2012.

[17] F. Muntenescu. Android architecture patterns part 2: Model-view-presenter [online]. `https://medium.com/upday-devs/android-architecture-patterns-part-2-model-view-presenter-8a6faaae14a5`. [Accessed: 2017-03-27].

[18] J. Nielsen. *Usability engineering.* Academic Press Boston, 1993.

[19] T. Nurkiewicz and B. Christensen. *Reactive Programming with RxJava.* O'Reilly Media, Inc., 2016.

[20] J. Poživil. *Integrated Visualization Environment for Interactive Analysis of User Activity Logs.* Master's thesis.

[21] J. Rubin and D. Chisnell. *Handbook of Usability Testing: How to Plan, Design, and Conduct Effective Tests.* Wiley Publishing, Inc., 2008.

[22] Add firebase to your android project [online]. `https://firebase.google.com/docs/android/setup`. [Accessed: 2017-02-20].

[23] Amazon dynamodb product details [online]. `https://aws.amazon.com/dynamodb/details/`. [Accessed: 2017-01-14].

[24] Amazon s3 product details [online]. `https://aws.amazon.com/s3/details/`. [Accessed: 2017-01-14].

[25] Annca [online]. `https://github.com/memfis19/Annca`. [Accessed: 2017-04-09].

[26] Asynctask [online]. `https://developer.android.com/reference/android/os/AsyncTask.html`. [Accessed: 2017-04-06].

[27] Blob storage [online]. `https://azure.microsoft.com/en-us/services/storage/blobs/`. [Accessed: 2017-01-14].

[28] Building instrumented unit tests [online]. `https://developer.android.com/training/testing/unit-testing/instrumented-unit-tests.html`. [Accessed: 2017-05-10].

[29] Cloud functions for firebase [online]. `https://firebase.google.com/docs/functions/`. [Accessed: 2017-05-20].

[30] Components – bottom sheets [online]. `https://material.io/guidelines/components/bottom-sheets.html`. [Accessed: 2017-02-24].

[31] Components – toolbars [online]. `https://material.io/guidelines/components/toolbars.html`. [Accessed: 2017-02-24].

[32] Configure build variants [online]. `https://developer.android.com/studio/build/build-variants.html`. [Accessed: 2017-04-02].

[33] Couchbase server [online]. `https://www.couchbase.com/nosql-databases/couchbase-server`. [Accessed: 2017-01-14].

[34] Data binding library [online]. `https://developer.android.com/topic/libraries/data-binding/index.html`. [Accessed: 2017-04-06].

[35] Developing android apps by google [online]. `https://www.udacity.com/course/new-android-fundamentals--ud851`. [Accessed: 2017-05-03].

[36] Documentdb [online]. `https://azure.microsoft.com/en-us/services/documentdb/`. [Accessed: 2017-01-14].

[37] Enabling offline capabilities on android - clock skew [online]. `https://firebase.google.com/docs/database/android/offline-capabilities#clock-skew`. [Accessed: 2017-05-01].

[38] Enabling offline capabilities on android - disk persistence [online]. `https://firebase.google.com/docs/database/android/offline-capabilities#section-disk-persistence`. [Accessed: 2017-05-02].

[39] Firebase [online]. `https://firebase.google.com/features/`. [Accessed: 2017-01-14].

[40] Firebase - cloud storage [online]. `https://firebase.google.com/docs/storage/`. [Accessed: 2017-03-24].

[41] Firebase crash reporting [online]. `https://firebase.google.com/docs/crash/`. [Accessed: 2017-05-04].

[42] Firebase - create custom tokens [online]. `https://firebase.google.com/docs/auth/admin/create-custom-tokens`. [Accessed: 2017-03-24].

[43] Firebase features [online]. `https://firebase.google.com/features/`. [Accessed: 2017-02-21].

[44] Firebase realtime database [online]. `https://firebase.google.com/docs/database/`. [Accessed: 2017-03-10].

[45] Firebase - securing your data [online]. `https://www.firebase.com/docs/security/guide/securing-data.html`. [Accessed: 2017-03-11].

[46] Firebaseui [online]. `https://opensource.google.com/projects/firebaseui`. [Accessed: 2017-04-07].

[47] Firebase - use file metadata on web [online]. `https://firebase.google.com/docs/storage/web/file-metadata`. [Accessed: 2017-03-26].

[48] Gartner says worldwide sales of smartphones grew 7 percent in the fourth quarter of 2016 [online]. http://www.gartner.com/newsroom/id/3609817. [Accessed: 2017-05-03].

[49] Google cloud platform - cloud storage [online]. https://cloud.google.com/storage/. [Accessed: 2017-03-24].

[50] Introduction to android [online]. https://developer.android.com/guide/index.html. [Accessed: 2017-05-03].

[51] Ive tool [online]. http://dcgi.felk.cvut.cz/home/malyi1/IVE/index.html. [Accessed: 2017-02-01].

[52] Ive tool - loggers [online]. http://dcgi.felk.cvut.cz/home/malyi1/IVE/loggers.html. [Accessed: 2017-02-01].

[53] Lists: Controls [online]. https://material.io/guidelines/components/lists-controls.html. [Accessed: 2017-02-24].

[54] Material design [online]. https://material.io/guidelines/material-design/introduction.html. [Accessed: 2017-04-06].

[55] Reactivex [online]. http://reactivex.io/intro.html. [Accessed: 2017-04-06].

[56] Reactivex - operators [online]. http://reactivex.io/documentation/operators.html. [Accessed: 2017-04-06].

[57] Structure your database [online]. https://firebase.google.com/docs/database/android/structure-data. [Accessed: 2017-03-10].

[58] Support library packages [online]. https://developer.android.com/topic/libraries/support-library/packages.html#design. [Accessed: 2017-04-07].

[59] Usability testing [online]. https://www.usability.gov/how-to-and-tools/methods/usability-testing.html. [Accessed: 2017-02-01].

[60] Usability testing with morae [online]. https://www.techsmith.com/morae.html. [Accessed: 2017-02-01].

[61] What's couchbase mobile? [online]. https://developer.couchbase.com/mobile. [Accessed: 2017-01-14].

Appendix A

Used Libraries

CWAC-Security https://github.com/commonsguy/cwac-security

Easy Video Player https://github.com/afollestad/easy-video-player

EventBus https://github.com/greenrobot/EventBus

Firebase SDK https://firebase.google.com/docs/android/setup

Firebase UI https://github.com/firebase/FirebaseUI-Android

Glide https://github.com/bumptech/glide

Google Play Services https://developers.google.com/android/guides/overview

PhotoView https://github.com/chrisbanes/PhotoView

RxJava https://github.com/ReactiveX/RxJava

Support Library https://developer.android.com/topic/libraries/support-library

Appendix B

List of used abbreviations

3NF Third Normal Form

API Application Programming Interface

AWS Amazon Web Services

BCNF Boyce-Codd normal form

BLOB Binary Large Object

CI Continuous Intergration

CTT Concur Task Trees

FAB Floating Action Button

FUP Fair Usage Policy

IVE Integrated Interactive Information Visualization Environment

MAT Mobile Application Tester

MVC Model–View–Controller

MVP Model-View-Presenter

MVVM Model-View-ViewModel

NDA Non-disclosure agreement

NUR User Interface Design

POP Prototyping on Paper

RDBMS Relational database management system

Rx Reactive Extensions

SDK Software Development Kit

TUR Testing of User Interfaces

UC Use Case

UI User Interface

Appendix C

Use Case Diagrams

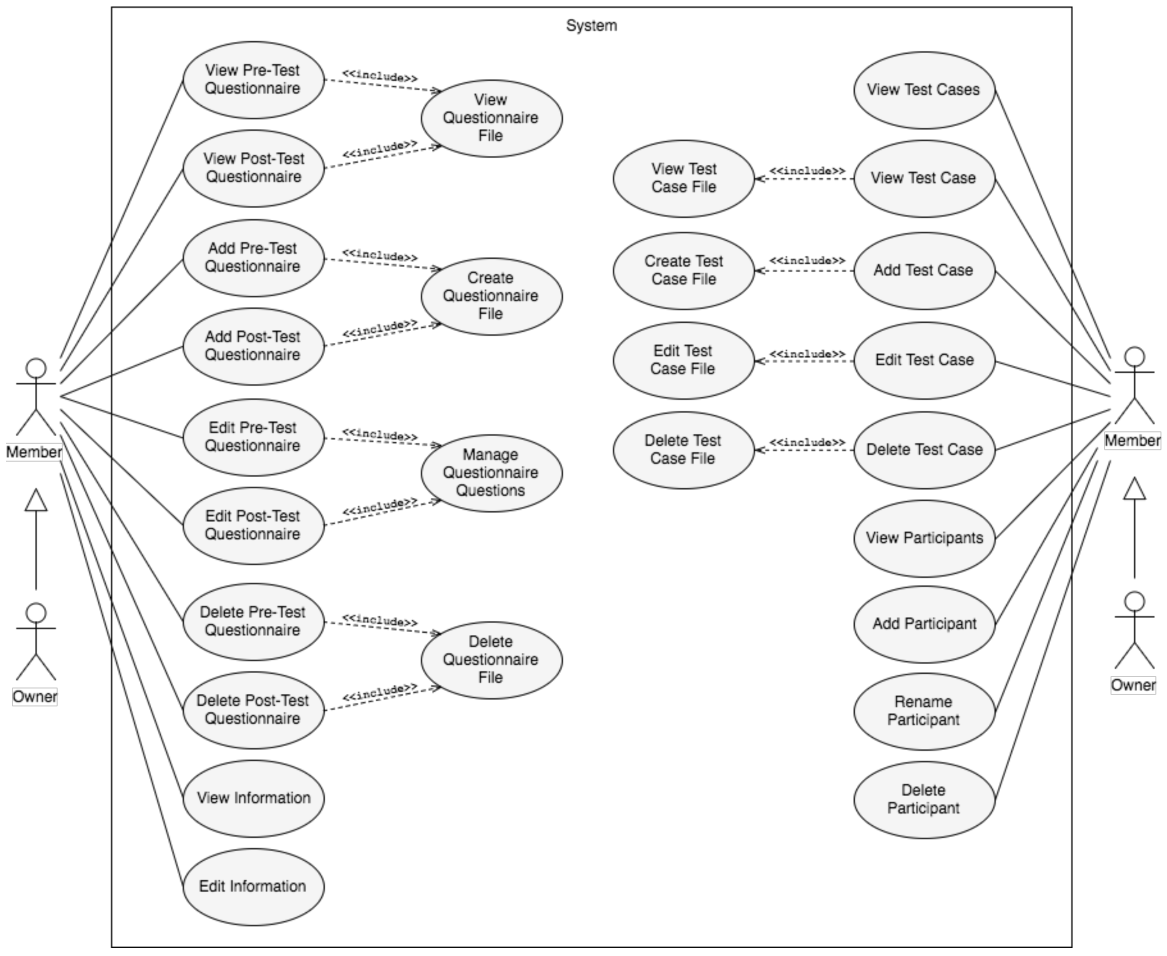


Figure C.1: Use Case diagram describing test definition data management actions that can be executed by members

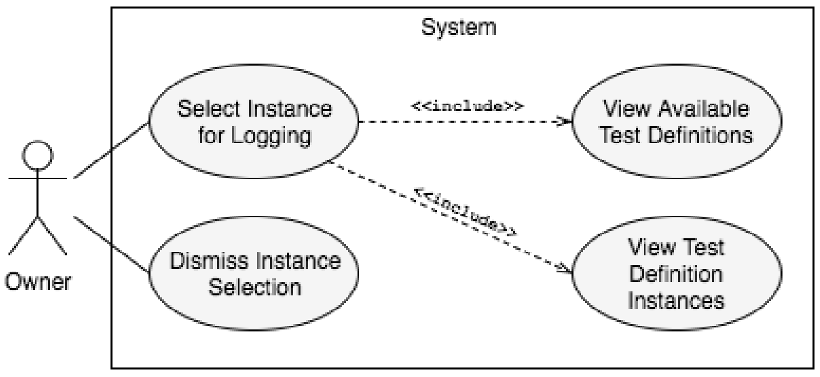

System
Owner
Select Instance for Logging
Dismiss Instance Selection
View Available Test Definitions
View Test Definition Instances
<<include>>
<<include>>

Figure C.2: Use Case diagram describing test instance logging management

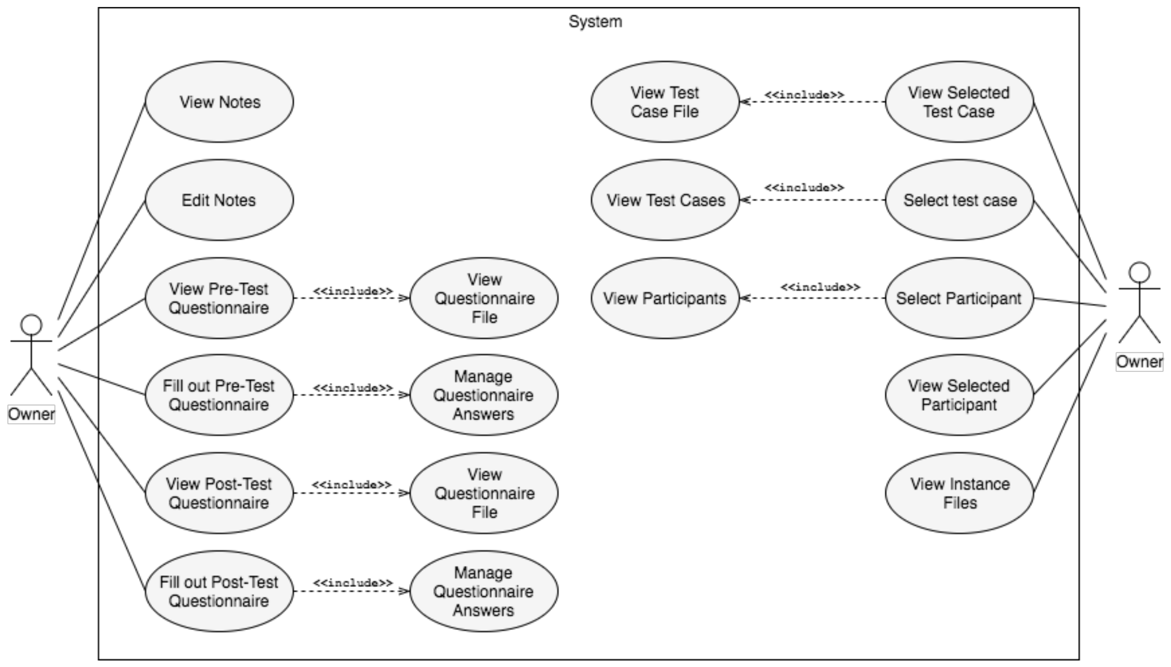


Figure C.3: Use Case diagram describing test instance data management

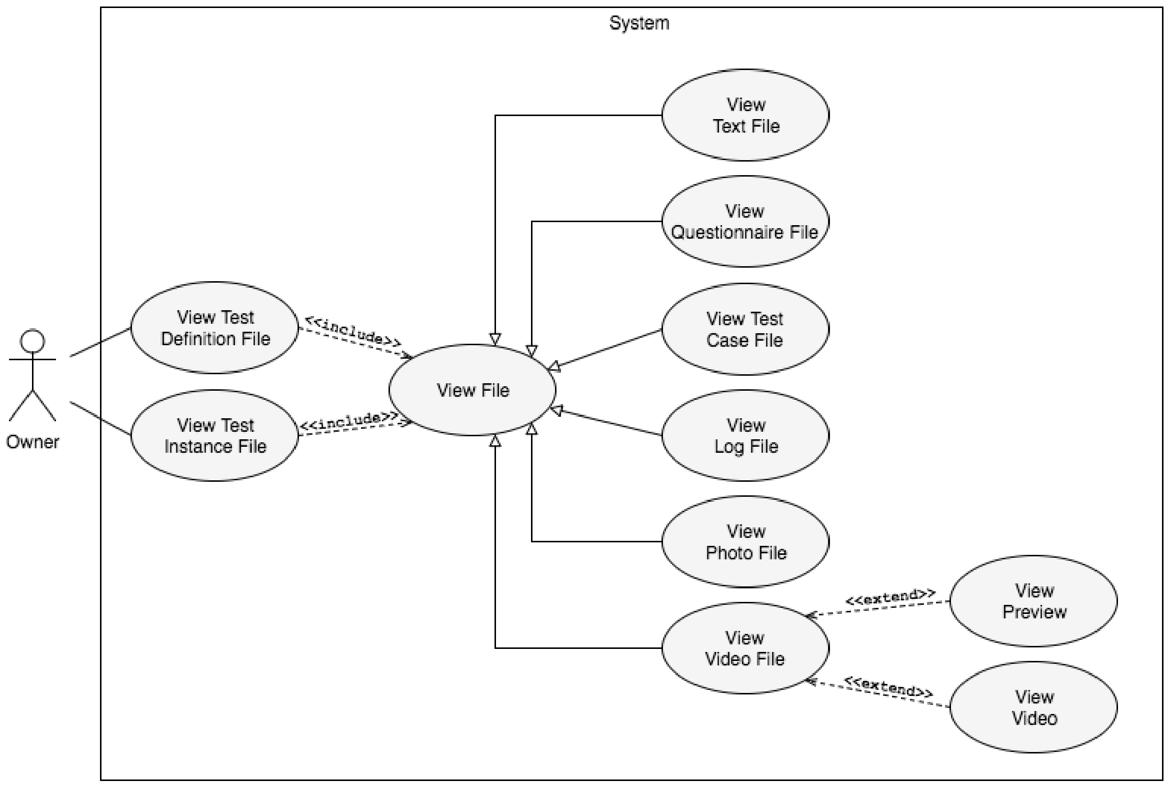


Figure C.4: Use Case diagram describing view actions of file management

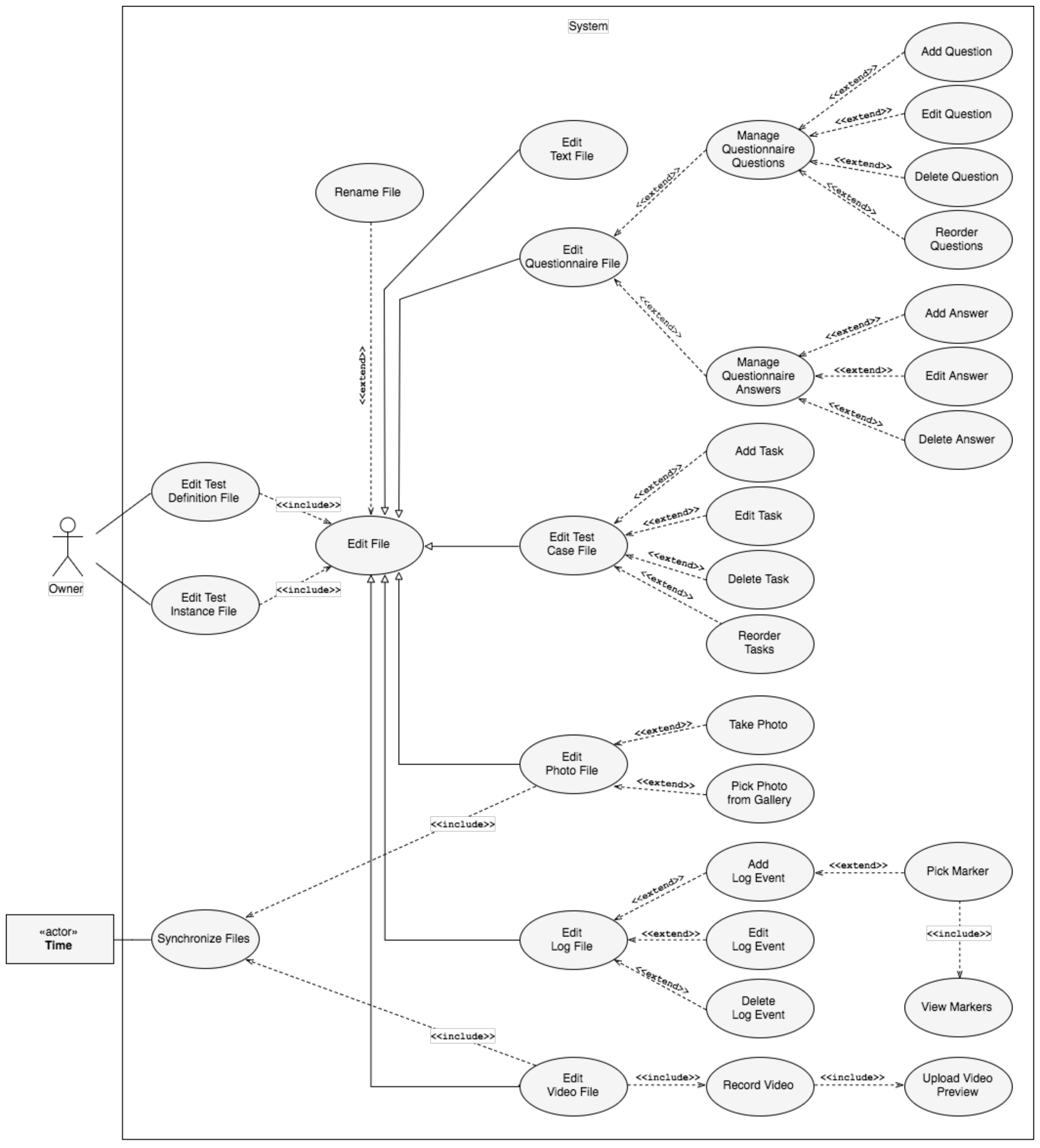


Figure C.5: Use Case diagram describing edit actions of file management

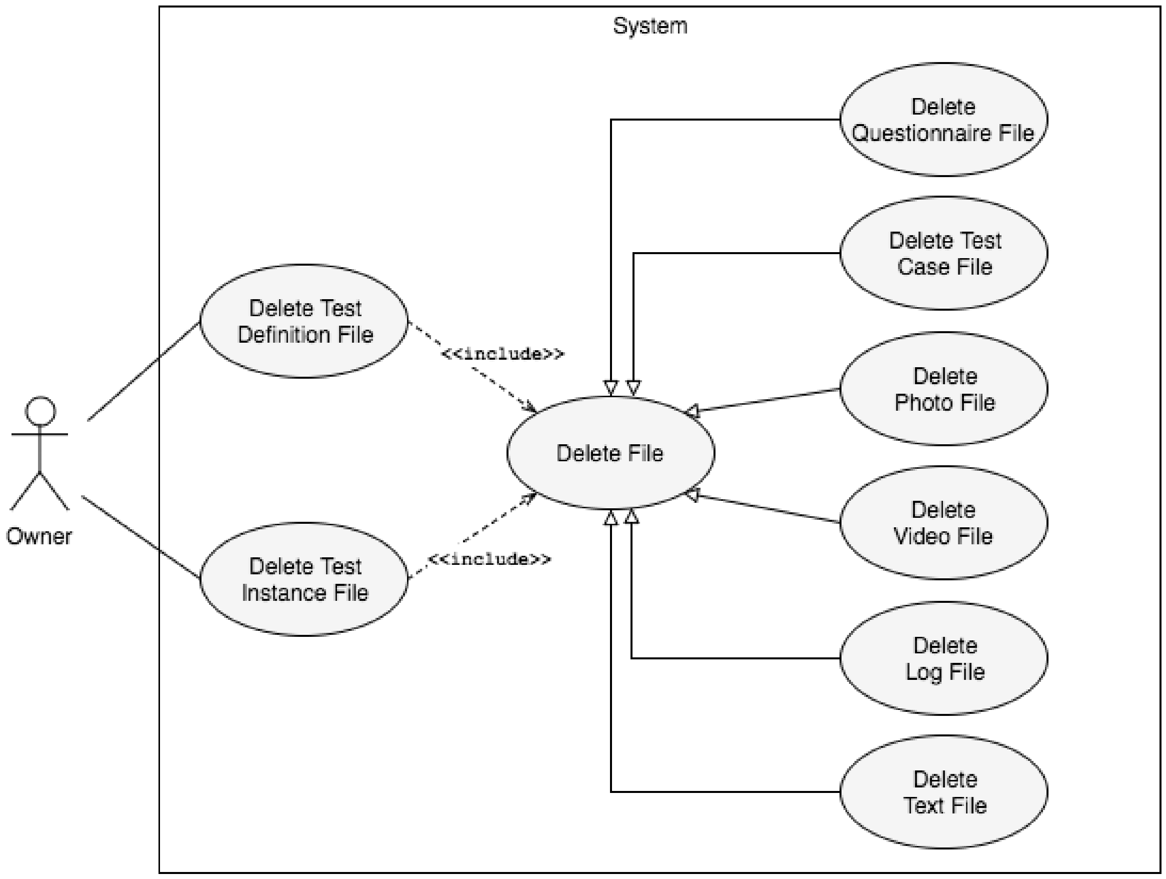


Figure C.6: Use Case diagram describing delete actions of file management

Appendix D

Code examples

```
{
  "test_instances": {
    "test_definition_id_001": {
      "test_instance_id_001": {
        "id": "test_instance_id_001",
        "test_definition_id": "test_definition_id_001",
        "name": "Test Instance #1",
        "notes": "This is a note.",
        "participant": "Participant Name #1",
        "pre_test_questionnaire": {...},
        "post_test_questionnaire": "",
        "test_case": {...},
        "data": {
          "data_wrapper_id_009": {...}
        }
      }
    }
  },
}
```

Listing D.1: Firebase database data structure of test instances

```
{
  "rules": {
    "abstract_data": {
      "$dataWrapperId": {
        ".read": "root.child('data_wrapper/'+ $dataWrapperId + '/rights/' ...
            + auth.uid).val() === 'write' || root.child('data_wrapper/'+ ...
            $dataWrapperId + '/rights/' + auth.uid).val() === 'read'",
        ".write": "!data.exists() || root.child('data_wrapper/'+ ...
            $dataWrapperId + '/rights/' + auth.uid).val() === 'write'"
      }
    },
    "connection_test_value": {
      ".read": "auth != null",
      ".write": "auth != null"
    },
    "data_wrapper": {
      "$dataWrapperId": {
        ".read": "root.child('data_wrapper/'+ $dataWrapperId + '/rights/' ...
            + auth.uid).val() === 'write' || root.child('data_wrapper/'+ ...
            $dataWrapperId + '/rights/' + auth.uid).val() === 'read'",
        ".write": "!data.exists() || root.child('data_wrapper/'+ ...
            $dataWrapperId + '/rights/' + auth.uid).val() === 'write'"
      }
    },
    "file_init_requests": {
      "$userId": {
        ".read": "auth.uid === $userId",
        ".write": "auth.uid === $userId"
      }
    },
    "file_upload_requests": {
      "$userId": {
        ".read": "auth.uid === $userId",
        ".write": "auth.uid === $userId"
      }
    },
    "storage_files_rights_sync_requests": {
      "$userId": {
        ".read": "auth.uid === $userId",
        ".write": "auth.uid === $userId"
      }
    },
    "instance_user_data": {
      "$userId": {
        ".read": "auth.uid === $userId",
        "$testInstanceId": {
          ".write": "!data.exists()",
          "$dataWrapperId": {
            ".read": "root.child('data_wrapper/'+ $dataWrapperId + ...
                '/rights/' + auth.uid).val() === 'write' || ...
                root.child('data_wrapper/'+ $dataWrapperId + '/rights/' + ...
                auth.uid).val() === 'read'",
            ".write": "!data.exists() || root.child('data_wrapper/'+ ...
                $dataWrapperId + '/rights/' + auth.uid).val() === 'write'"
          }
        }
      }
    },
    "test_instances": {
      "$testDefinitionId": {
        ".read": "root.child('tests/'+ $testDefinitionId + '/members/' + ...
            auth.uid + '/role').val() === 'owner'",
        ".write": "root.child('tests/'+ $testDefinitionId + '/members/' + ...
            auth.uid + '/role').val() === 'owner'"
      }
    },
    "tests": {
      "$testDefinitionId": {
        ".read": "root.child('tests/'+ $testDefinitionId + '/members/' + ...
            auth.uid + '/role').val() === 'owner'",
        ".write": "!data.exists() || root.child('tests/'+ ...
            $testDefinitionId + '/members/' + auth.uid + '/role').val() ...
            === 'owner'",
        "info": {
          ".read": "root.child('tests/'+ $testDefinitionId + '/members/' ...
              + auth.uid + '/role').val() === 'member'",
          ".write": "root.child('tests/'+ $testDefinitionId + '/members/' ...
              + auth.uid + '/role').val() === 'member'"
        }
      }
    },
    "user_mapping": {
      "$userEmail": {
        ".read": "auth != null",
        ".write": "!data.exists() && newData.val() === auth.uid"
      }
    },
    "user_tests": {
      "$userId": {
        ".read": "auth.uid === $userId",
        "$testDefinitionId": {
          ".read": "root.child('tests/'+ $testDefinitionId + '/members/' ...
              + auth.uid + '/role').val() === 'owner'",
          ".write": "!data.exists() || root.child('tests/'+ ...
              $testDefinitionId + '/members/' + auth.uid + '/role').val() ...
              === 'owner'"
        }
      }
    },
    "user_tests_data": {
      "$userId": {
        ".read": "auth.uid === $userId",
        "$testDefinitionId": {
          ".write": "!data.exists()",
          "$dataWrapperId": {
            ".read": "root.child('data_wrapper/'+ $dataWrapperId + ...
                '/rights/' + auth.uid).val() === 'write' || ...
                root.child('data_wrapper/'+ $dataWrapperId + '/rights/' + ...
                auth.uid).val() === 'read'",
            ".write": "!data.exists() || root.child('data_wrapper/'+ ...
                $dataWrapperId + '/rights/' + auth.uid).val() === 'write'"
          }
        }
      }
    },
    "users": {
      "$userId": {
        ".read": "auth != null",
        ".write": "auth.uid === $userId"
      }
    }
  }
}
```

Listing D.2: Firebase database security rules of the implemented system

Appendix E

CD Content

The following contents are to be found on the attached CD:

```
CD
|-- android
    |-- UXTester       Source code of the Android applications
    |-- apk            Generated apk files
|-- documentation
    |-- requirements   Document with the system's requirements
    |-- usecases       Use Case diagrams in full resolution
|-- screenshots
    |-- android        Screenshots of the Android applications
    |-- prototype      Scanned sketches of the prototypes
|-- testing
    |-- test-data.pdf  Document used in usability testing
    |-- test-data.mp4  Video used in usability testing
|-- thesis
    |-- src            Source files of this thesis in Latex
    \-- thesis.pdf     PDF version of this thesis
```