Ulm University | 89069 Ulm | Germany

Faculty of Engineering Sciences, Computer Science and Psychology
Institute of Databases and Informationsystems

Data Management in Large-Scale Data Collection Scenarios

Bachelor Thesis at Ulm University

Submitted by:
Philipp Jung
philipp.jung@uni-ulm.de

Reviewer:
Prof. Dr. Manfred Reichert

Supervisor:
Johannes Schobel

2018

Version February 26, 2018

Satz: PDF-LaTeX 2_ε

Abstract

Despite the progress of digitization and constant innovations in almost all industries, data collection through questionnaires is still often paper-based. The innumerable disadvantages and the unnecessary burden on the environment are obvious. And that is precisely why it is worth asking, why nothing has happened in this sector. This can be due to several reasons, for example, the lack of flexibility compared to paper is one reason why paper questionnaires are often preferred. Also costs are a factor, because especially when flexibility in the execution is required, the implementation of such a questionnaire is often expensive.

This is exactly where the QuestionSys framework should help. With its configurator it offers the possibility to create complex questionnaires without previous knowledge. In addition, the web application is a good platform for managing and organizing these. And for the execution of the questionnaires there is the questionnaire application, which has been developed in the context of this bachelor thesis. Concerning this, the focus is especially on the associated data management. Therefore, it is demonstrated how the data flows can be implemented in an application like this, in order to guarantee the full range of functions also on a large scale. This also applies to the storage, management and external communication of data within the application and is always under the condition of cross-platform development and therefore being compatible to the majority of smart mobile devices.

The implementation of these requirements is explained by code samples and graphical illustrations.

Acknowledgment

At this point I would like to thank all those who supported and motivated me in carrying out and preparing this work.
Especially my family is to be mentioned here, which made it possible for me to study without any worries. In addition, I would like to thank my girlfriend Alana and her family for always supporting me.
A very special thanks also goes to my fellow students and friends Dimi, Robin and Ralph without them I probably would never have been able/allowed to write this work.
And last but not least, I would like to thank my mentor, Johannes Schobel, who has always been able to motivate and challenge me with his enthusiasm for this topic and at the same time was an excellent advisor.

Contents

1 Introduction

Nowadays data collection has grown to a constant companion in everyday life. Already in 2010 the consumption of electricity used for storing data in data centers was comparable to the electricity production of Australia [1].
Due to this enormous increase, it is therefore no longer far off that the collection and management of data is becoming increasingly important. However, it should be noted that the collection of data is usually not active. In the rarest of cases, the user is asked explicitly for information, but they are automatically transmitted by his usage behavior.
In contrast, this work deals with the active collection of data, more precisely by collecting data with questionnaires. In this sector, too, the potential for efficiency is not yet exhausted and a framework has been created in the form of *QuestionSys* framework, that is created to make the collection of data even simpler, more intuitive and more effective for both the questioner and the respondent.
A few years ago, "the only choices available were between personal interviews (face-to-face interviews), telephone interviews, and mail surveys, all using paper questionnaires. [...] Nowadays, each of these modes of data collection can also be computerized by computer-assisted personal interviewing (*CAPI*), computer-assisted telephone interviewing (*CATI*), and computer-assisted self-interviewing (*CASI*) or Web surveys." [2] Hence, QuestionSys is primarily concerned with *CASI* and *CAPI*. This is because it should be possible for the questioner to keep the device and enter the answers collected during an interview, or to leave the device completely to the interviewee.
And although a lot of digitisation has already taken place in this segment, a questionnaire, in general, is still on paper. This can be due to multiple reasons: First, so far digital questionnaires have always been linear. However, if one wants to continue with

a specific question as a result of a given answer, one has to either break through the linearity or ask a question to all persons that concerns only a few. Another reason is, for example, changes to coded questionnaires, where a change to the questionnaire often means a change to the implementation, which is particularly deterrent for inexperienced users. Therefore, an easy and flexible solution is needed to bring those paper based questionnaire users to the (mobile) devices using an application that gives them no restrictions on their handling with questionnaires [3]. Within the scope of this bachelor thesis, an application has been created, that is able to execute graph based and externally generated questionnaires. In particular, the associated data management is described in detail to this extent. The important aspect in this context is that it is integrated in such a manner that the application and its use cases are almost arbitrarily scalable.

1.1 Outline

Since the aim of this work is to describe the data management of the application created in this context, it is useful to initially define a course of action.
In this regard all fundamentals will be clarified in Chapter 2 in order to enable a better understanding of the thesis in the subsequent stages. This includes an introduction to *Angular* (Section 2.1), *Ionic* (Section 2.2), an overview of what data management actually means (Section 2.3) and an introduction to the *QuestionSys* concept (Section 2.4).
Based on the latter, use cases (Chapter 3) will then demonstrate the necessity and benefits of QuestionSys. Therefore, a comparison is made between paper-based questionnaires and those of the QuestionSys framwork (Section 3.1). Afterwards, two possible scenarios of application, especially in the large scale, will be presented (Section 3.2).
Once the basics and goals of the application should be established, its implementation can be approached. For this purpose, the concept regarding the mobile application is presented (Section 4.1) in order to define the requirements of the application afterwards (Section 4.2). These requirements are then in turn used to develop an application

structure that serves as an orientation for the implementation in Section 4.3.
Next, Chapter 5 discusses the implementation of the Questionnaire application. In this context, the basic structure of data storage and its communication structure is examined in Sections 5.1 and 5.2. The individual functionalities of the application are examined in detail in Sections 5.3 to 5.8 afterwards.
In the following, a general overview of the application and the achieved goals with regards to the requirements will be provided. In conclusion, an outlook on the future and the potential of the Questionnaire application will round off and complete this thesis (Chapter 6).

2 Fundamentals

In order to ensure a well comprehensible investigation of the subject of this thesis, it is important to clarify and define some basic concepts in advance. In the context of the application described, this concerns both the technical background in form of Angular and Ionic, as well as the theoretical background like the *QuestionSys* framework and of course the data management in general. Once these terms are sufficiently defined, the thematic approach can be approximated to the implementation of the application.

2.1 Angular

In recent years, a new dynamic has emerged in the front-end of web applications, as more and more JavaScript web frameworks and libraries are entering the market in this sector. So the impression can be aroused that new frameworks are created every day and front-end developers are constantly urged to decide whether the new toolkit brings enough value to justify the additional workload. With *Backbone*, *Ember*, *Knockout* and *Angular 1* as an example, client-side frameworks that have more relevance than others could emerge early on. [4]

Furthermore, Angular is not an library but a JavaScript framework [5]. It can, therefore, be concluded that Angular is a JavaScript based front-end framework which can be used to create web applications. With this information it is now possible to take a closer look at the historical background of this framework, with a particular focus on technical development over time.

Angular formerly known as AngularJS, is a client-side JavaScript framework for *single page web applications* created and maintained by Google in 2009 [5]. In case of the

Questionnaire application, AngularJS is not that important, mainly because Angular 2 respectively Angular 4 were used to create the later explained program.
Angular 2 is a complete rewrite of AngularJS, allowing developers to decide, whether to write in JavaScript, using the syntax of ECMAScript 5 or 6, TypeScript or Dart. Before compilation, the code is transpiled to JavaScript. The decision what programming language to use, is mostly up to the developers preferences. Apart from this flexibility there has been many more improvements from AngularJS to Angular 2, concerning the performance or the code simplicity. The latter is mainly due to the possibility of using TypeScript, that will be helpful for anyone who is into object related programming languages. [6]
The most recent version of Angular is Angular 4.[1] The difference between Angular 2 and Angular 4 is not as big as the difference between AngularJS and Angular 2. This is most probably due to the announcement that the angular team wants to avoid breaking changes like they had from AngularJs to Angular 2. Angular 4 is a bit faster in performance and smaller in memory size than Angular 2 and has several new features while it's still backwards compatible to Angular 2 [7].

2.2 Ionic

Ionic is an Angular based, open source framework for creating cross platform mobile applications. To understand the power of the Ionic framework, one has to take a closer look on the problem Ionic tries to solve. Currently, there are three main operating systems to consider when developing mobile applications: Android, iOS and Windows for Mobile. The three of them come together with a market coverage of 99.9% (Q4 2016) [8]. As one can see, there are at least three different operating systems with different SDK's and different programming languages(Table 2.1).

[1]Because of one npm-package already having version number 3 they skipped 3 in enumerating the angular versions.

Operating System	**SDK**	**Programming Language**
iOS	iOS SDK	Objective-C / Swift
Android	Android SDK	JAVA
Windows for Mobile	Windows SDK	.NET

Table 2.1: Mobile Operating Systems Overview [9]

Applications developed for one of these specific platforms with the corresponding framework and programming language are called native applications. As already mentioned, they work under the aspect of different development environments, technologies and APIs (**A**pplication **P**rogramming **I**nterfaces) for each mobile platform. This inevitably leads to a waste of development time and effort, as well as increased maintenance costs. [10]
In contrast to this, however, Ionic tries to create a mobile application with only one code base for nearly all mobile devices. A very important part to reach this goal is *Apache Cordova*, which "is a technology that lets any web application be packaged as a native mobile application while also providing access to device features.“ [9] This results in a browser-based application that uses HTML and SCSS and is complemented with plug-ins and device APIs. This construct is then transpiled to a native application using the device-specific APIs (cf. Figure 2.1).

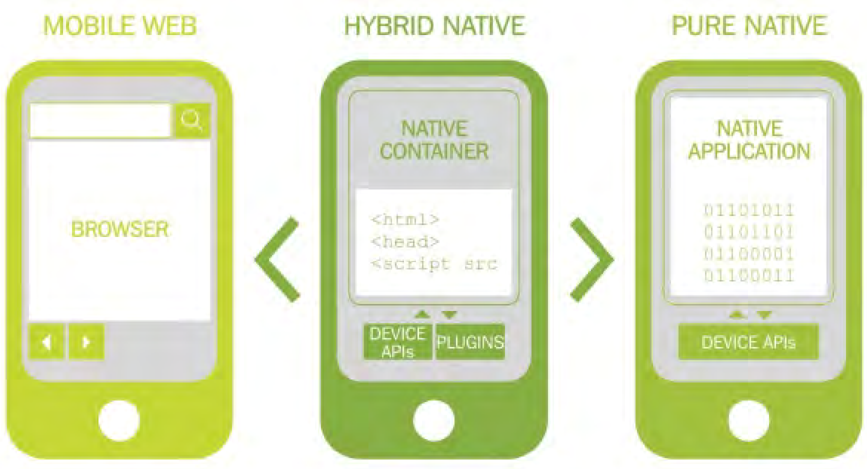


Figure 2.1: Native Applications with Cordova [9]

Therefore, one has the possibility to design user interfaces that rely on HTML and SCSS, that most likley look like native ones for each device. With Cordova as an addition, this results in not only having a native look, it also supports native features [9]. Together with Angular as a base, Ionic builds up to a powerful hybrid framework that allows to create mobile applications that nearly look like native ones.

2.3 Data Management

Especially in the context of this thesis, it is important to define the term *data management*. This section will give a short introduction to what data management means, and how it is used in the Ionic, respectively, the Angular framework.
Data management is all about handling data within an application. In this context, two types of data management are particularly interesting. Firstly, storing data in databases or on the file system is a big part and secondly, one has to transfer data to be able to handle it extensively, therefore, data transfer is the other outstanding part of data management.

2.3.1 Data Storage

General

In order to store data, hardware is always required. In the earlier days this could be stone, papyrus, wood and paper that stored data until punch cards were invented in 1884 by Herman Hollerith. Since then a lot of different storage media like magnetic tapes, hard drives, floppy disks, CD/DVDs, flash drives, blue-rays or cloud based storage solutions were invented. However, data is always persisted on hardware. [11] From a programmatic perspective, there is another decision to make regardless of what kind of hardware was chosen to store the data. Especially in mobile application development, one has to decide to either store data in databases, the file system or to upload it to some kind of server.

Framework Based

Concerning the general statements made about data storage, one should have a closer look at what data storage means in the context of Ionic, respectively, Angular applications. The first possibility to store data with Ionic is simply called *storage*. This enables storing both `key/value` pairs and `JSON` objects.
In order to ensure a platform-independent and consistently functioning storage, the storage process is subject to different storing engines. This means that the saving process in the code is always the same, but is interpreted differently on a native level after the compilation. In a native mobile application, for example, storage will prioritize `SQLite` to store input, as it is a safe place for the data when the device gets into low disk-space situations. In a browser context, however, storage will firstly attempt to use `IndexedDB`. If the letter isn't applicable, it will try `WebSQL` or localstorage.[12]
Another way of storing data is saving data in databases which is possible using the *cordova-sqlite-storage plugin*. It uses the native `SQLite` database on the specific device and comes with a bunch of functions that make it quite easy to read and write data from/to the database.[12]
The third way of storing data is saving, copying and deleting files in the device specific file system. This is a complex procedure, as this concerns storage locations of android, windows phone and iOS. Cordova eliminates this issue, by providing default variables holding the correct paths. This results in having the possibility to go through a case distinction and assign correct writing and reading paths to the specific operating system, while keeping one code base (for any platform) [12].

2.3.2 Data Transfer

General

The second part of data management is transferring the data. This can be done in two ways, within a certain device (e.g., between UIs or components) or from one device to another. In general, there is always a sender and a receiver that transfers data with the help of a messaging service or a fixed protocol.

Framework Based

It is probably more interesting to take a closer look at the framework to see how data transfer is managed in angular. Starting at the `front end`, the application has to transfer user input to the `back end` in order to process it. The HTML input fields, therefore, have a *two-way* binding to a variable at the `back end` connected by Angulars' `ngModel`. This allows to adapt changes at the `front end` in close to real time to the variable in the `back end`. This results in a fast and easy data transfer [13]. In most cases, a component is not the final destination of the input data and it has to be transferred to a storage possibility as explained in section 2.3.1. To understand the process of an angular data transfer in a better way Figure 2.2 should give an orientation. It shows the bindings as a connection between template and component and shows the manner in which data could get out of the component. The services shown in Figure 2.2 inside the injector are also known as providers in the context of angular and ionic and represent the standard way of data communication between components. Regardless of whether data is to be stored on a local database, storage or an external server, the data transfer within the application is always carried out via providers. [14]

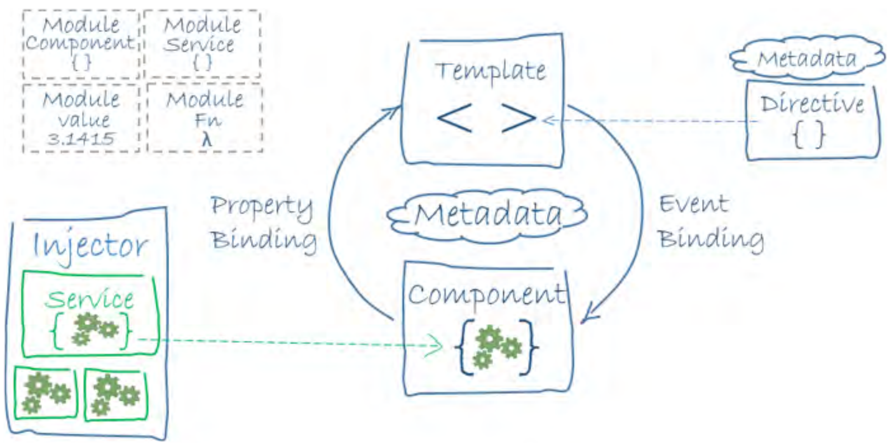


Figure 2.2: Angular Data Transfer [14]

How data transfer and management happens in detail on ionic/angular will be shown in chapters later on. but first of all, it is necessary to explain in which environment the described data management should take place so that the detailed implementation can be dealt with.

2.4 The *QuestionSys* Concept

This section gives an introduction into the *QuestionSys* framework to provide a better understanding for the created *Questionnaire* application.

The problem is that “[n]owadays, most psychological studies are performed based on specifically tailored and validated *pen & paper* questionnaires. Usually, such a paper-based approach results in a massive workload of the actors involved when collecting data as well as aggregating, analyzing and evaluating it afterwards.” [15] In order to reduce workload and increase efficiency at working with questionnaires a framework that allows “a process-driven approach for defining, validating, deploying, processing, and analyzing questionnaires based on generic tools” [15] has been developed. As shown in Figure 2.3 the framework consists of three parts, a questionnaire configurator, also known as *Questioneer*, to create questionnaires as shown on the left side. A middleware ensures a secure data exchange and a possibility to deploy, run and log questionnaires on mobile devices. [16] This thesis deals with on the most right part. Which is an application that is able to execute questionnaires. More specifically, the thesis focuses on data management within this application.

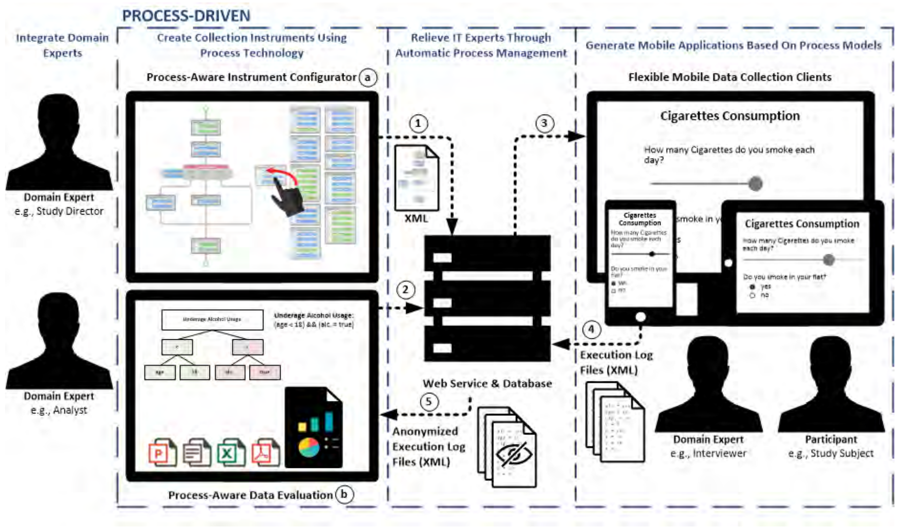

PROCESS-DRIVEN
Integrate Domain Experts
Create Collection Instruments Using Process Technology
Relieve IT Experts Through Automatic Process Management
Generate Mobile Applications Based On Process Models
Process-Aware Instrument Configurator (a)
Flexible Mobile Data Collection Clients
Domain Expert e.g., Study Director
Domain Expert e.g., Analyst
Cigarettes Consumption
How many Cigarettes do you smoke each day?
Do you smoke in your flat?
yes
no
1
2
3
4
5
XML
Underage Alcohol Usage
Underage Alcohol Usage: (age < 18) && (alc. = true)
Web Service & Database
Execution Log Files (XML)
Anonymized Execution Log Files (XML)
Domain Expert e.g., Interviewer
Participant e.g., Study Subject
Process-Aware Data Evaluation (b)

Figure 2.3: *QuestionSys* architecture [17]

3 Use Cases

Now that the technical basics have been explained and the *QuestionSys* framework has been defined, the strengths of it will be demonstrated in the following, by the help of use cases. For this purpose, a comparison is made between paper-based questionnaires and those provided in the *QuestionSys* framework. Subsequently, use cases in large-scale scenarios are shown to demonstrate the power of QuestionSys.

3.1 Paper based Questionnaires vs. *QuestionSys*

Starting from scratch, one has to ask if *QuestionSys* has a big enough advantage to be preferred over paper based questionnaires. According to [16] studies usually rely on specifically tailored paper based questionnaires. But a digital questionnaire is a useful alternative. That was demonstrated with psychological questionnaires that already exist and usually are performed in a traditional fashion (e.g., KINDEX). Those questionnaires were taken and transformed into equivalent ones that were rendered and executed on smart mobile devices, like smartphones or tablets. This resulted in better data quality, shorter evaluation cycles, and significant decreases in workload [16]. To find out why digital questionnaires are more effective than paper-based questionnaires, one should consider the different phases of a study. At first the questionnaire has to be created. Nowadays, this is typically done on a computer, therefore, there is no difference in this phase considering workload or time. After creating the questionnaire the distributing has to be prepared: On a non paper based framework, the questions are uploaded with a few clicks, the paper based variation ends up in a huge amount of printed pages, that costs time, paper and money, resulting in an advantage for *QuestionSys*.

The next step is the data collection by using either the *QuestionSys* application or the paper based questionnaire. Having a digital solution comes with the possibility to guarantee data validation, by using field restrictions, that only allow correct inputs [18]. This is probably the most important advantage *QuestionSys* has over the physical questionnaires, because incorrect data is a waste of time, and resources. After the data is collected, evaluation is needed. This usually happens on computers, regardless of the used framework (physical or digital). To get the data in a statistical program from the *QuestionSys* framework, one just has to download the collected data which takes time in the minute or second range. The paper based questionnaires instead have to be transcripted separately, resulting in another possible source of errors. At this stage, the paper-based approach has a higher error rate and takes longer, which again puts the digital process at an advantage. Summing up the results of the discussion, there are cost, time and resources saved and a higher data correctness reached by using a digital questionnaire in combination with the *QuestionSys* framework. That shows a use case for the framework in standard questionnaire scenarios is absolutely existing.

3.2 Large-Scale Scenarios

Tthe suitability of *QuestionSys* for standard studies or questionnaires is very good, but of course the ambition is that the concept can also be applied to large scale scenarios. For this purpose, the quantity of collected or to be collected data as well as the quantity of persons to be interrogated, have been selected as scalable parameters, which shows that *QuestionSys* can also master these (large scale) scenarios.

3.2.1 Large-Scale Concerning Collected Data

The first scenario matches probably more intuitively to the definition of large-scale data collection. Imagine one has to collect a huge amount of data as seen on the global sharpers survey of the world economic forum, where over 31,000 young people of 186 countries have been asked questions of 5 different topics in 14 languages which is a tremendous size of a survey [19]. The interesting question now is: Would this large scale

example be feasible with the *QuestionSys* framework and is it an use case for it. The first point to look at is the 31,000 asked people. Because of the central data storage at the middleware of the *QuestionSys* framework, it would be no problem to store this data centrally. In addition, the flexibility and the possibility to let multiple persons work on one project, would be feasible as well. It would even be possible to ask each of the 31,000 participants with a different device. Therefore, the amount of the people would have been no problem for the *QuestionSys* framework, either the 186 countries the people live in. Because that challenge is solved by the default connection and central storage system integrated in the questionnaire application of the *QuestionSys* framework.
The last point to check is the multilingualism factor of the Global Sharpers survey. This is another challenge that emphasizes the strength and versatility of the framework. Because in this case one has several options for multilingualism. It is possible to create questionnaires in any wanted language with the configurator and, in addition, have an independent system language of the mobile application, that can be extended to any number of languages, too.

3.2.2 Large-Scale Concerning Involved Persons

Another kind of large-scale data collection scenario where the *QuestionSys* questionnaire mobile application could have an impact on, is when a lot of data from multiple people is needed to be collected. For example a doctor's office, that has about 25 patients per physician a day [20]. That means if there are 3 physicians in a joint practice there are about 75 patients a day or 1500 patients per month. Each patient has an average time with his physician of about 17 minutes, assuming each physician is working 7 hours a day. Because of insurances not paying that much these days, it is in the interest of doctors to serve as much patients as possible with no lack of quality and this is where the *QuestionSys* framework could be a tremendous help. As one can see in Figure 3.1 an average doctor's office has no good work distribution between the two main phases of a doctor visit. At first the patient is waiting (doing nothing; on average 19 minutes [21]). Afterwards he gets into the doctor's office and the complete procedure with asking questions, comparing those answers to symptoms of diseases, coming to a diagnosis and starting the treatment starts. It's obvious that this approach can't be the most efficient one. If a doctor's office would use *QuestionSys*, it would be possible to separate this workload of one phase to three phases. This happens by giving the patient a tablet with a preinstalled questionnaire that handles the same questions like the doctor did in the first scene. After the patient answered those questions in the waiting room the application will compare the answers with the symptoms of typical diseases, make a suggestion for a diagnosis with the help of predefined diagnostic rules and upload the results. Subsequently, after the patient already entered the doctor's office the physician can check those results on his/her computer and after a short debriefing he/she can immediately start the treatment.

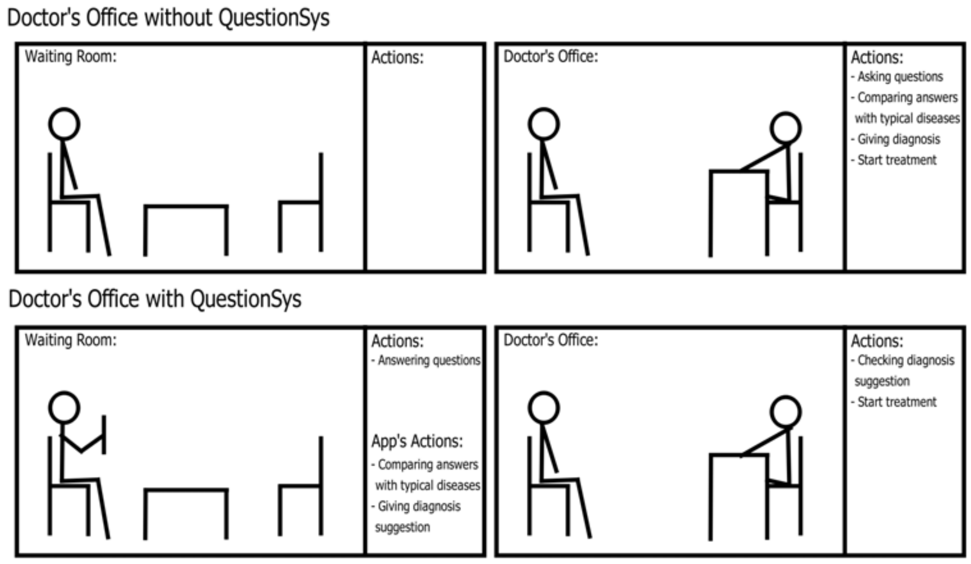

Doctor's Office without QuestionSys
Waiting Room:
Actions:
Doctor's Office:
Actions:
- Asking questions
- Comparing answers with typical diseases
- Giving diagnosis
- Start treatment
Doctor's Office with QuestionSys
Waiting Room:
Actions:
- Answering questions
App's Actions:
- Comparing answers with typical diseases
- Giving diagnosis suggestion
Doctor's Office:
Actions:
- Checking diagnosis suggestion
- Start treatment

Figure 3.1: Large-Scale Data Collection Scnenario

Additionally the results of the questionnaire can be added to the patients medical report and also be tagged for ongoing actions on base of this document. This scenario can scale up to any size and therefore describes a large-scale scenario, concerning the involved persons.

3.3 *Google Forms* vs. *QuestionSys*

However. it seems reasonable to ask whether programs such as *Google Forms* could not have mastered these scenarios as well and why *QuestionSys* should be such a big improvement.
If one tries to reconstruct the first use case with the *Global Sharpers Survey* with *Google Forms*, there is no problem at the beginning. The creation of surveys is straightforward and possible in teamwork. Sharing is also done in a very short time and the execution is even possible on mobile devices. Also storing and organizing the results is no problem due to the enormous capacities of *Google Drive* [22]. However, if one spends a little time in the help forum of the application one can quickly identify a problem. *Google Forms* does not support multi language support, which is insured by *Google*'s own community experts several times [23]. Accordingly, in order to do this with *Google Forms*, a separate questionnaire would have to be created for each required language (14 in the case of the Global Sharpers Survey) and its results would have to be merged afterwards. Thus, this is an use case in which *QuestionSys* outshines a big competitor.
But even in the second use case *QuestionSys* is full of advantages over *Google Forms*. Therefore, for example, only one of them has the possibility to react to given answers by continuing the questioning in a different way and that is *QuestionSys*. *Google Forms* instead only provides the possibility to run a questionnaire in a linear sequence. However, a doctor's office usually has a large number of diagnoses that it must be able to give, which in turn can be based on a large number of symptoms. And in order not to overwhelm the user with too many questions, an internal logic that excludes certain questions on the basis of given answers, is necessary.

4 Concept and Structure

After designing a framework concept with *QuestionSys* and demonstrating realistic use cases and areas, it is now a question of limiting the whole concept to one part.

4.1 Concept

This part is the *Questionnaire* application, which forms the core of the thesis and was created in this context. In the *QuestionSys* environment, it corresponds to the input medium and should therefore be able to collect valid data in different scenarios.
Furthermore, it is important to be able to offer the application on as many devices as possible. Accordingly, also different operating systems, even the change between smartphone and tablet, should not be a problem. This is for the purpose of being able to rule out compatibility problems, as a reason for deciding against the *QuestionSys* Framework.
Within the application, the arbitrary adaptation possibility of different elements is a central point. This should enable to provide the customer with a tool that can be perfectly adapted to the respective intended use.
At the same time it was still important to maintain good usability and clarity, so that even customers with little technical knowledge can handle the application well. And of course, the ability to be integrated into the *QuestionSys* framework was one of the biggest factors contributing to the creation.
With these core statements as a basic framework, an application was developed whose structure and requirements are explained below.

4.2 Requirements

Despite a found basic concept and identified core requirements, further, more detailed requirements, have to be defined in order to be able to guarantee correct functionality even in large-scale. For this purpose, both *functional* and *non-functional* requirements are defined below, with which it is then possible, in turn, to sketch a structure of the application.

4.2.1 Functional Requirements (FR)

Starting with the functional requirements, it is first advisable to define the notion of functional requirements before the requirements themselves are defined. Functional requirements are “[t]he necessary task, action or activity that must be accomplished.“[24] In the application context that leads to defining following functional requirements:

#FR1 Login: The application must be password-protected at least with the first login, while the administrative area must always be password-protected.

#FR2 Administation area: An administrative area should exist in which sensitive data can be managed and viewed. Furthermore, settings that have an influence on data acquisition can also only be changed from this area.

#FR3 Presenting downloadable questionnaires: The user should only be offered the questionnaires which he is entitled to download. The goal of this is to reduce the load on the application and to displaying all existing questionnaires could obstruct the user.

#FR4 Download questionnaires: In addition to displaying downloadable questionnaires, it should also be possible to transfer them to the mobile device. Once this has been done, it should be made visually recognizable and the questionnaire should be executable.

#FR5 Fill in a questionnaire: It is supposed to be possible to start a questionnaire, fill it out until the end and finally complete it.

#FR6 Pause a questionnaire: To prevent a user from being forced to work through a questionnaire in one piece, it should be possible to interrupt the filling of the questionnaire. This is to avoid the situation that a user gives false or inaccurate information in a hectic process or that data is lost if a survey has to be interrupted due to external circumstances.

#FR7 Continue paused questionnaires: After a questionnaire can be interrupted, it should also be possible to resume this paused questioning.

#FR8 Finnish questionnaires: Although the completion of a questionnaire is a very logical requirement for an application that focuses on the execution of questionnaires, this refers to the entire termination process. This means that the answered questions should be correctly combined into a coherent data record, provided with a time stamp and an identifier and archived without any additional loss of data.

#FR9 View results of completed questionnaires: It should also be possible to view the results of completed questionnaires. For this purpose, the results are to be presented in a folder system in order to demonstrate a clearer affiliation. If one has navigated to the folder of a questionnaire, the individual results should be recognizable by their identifiers and sorted according to the creation date.

#FR10 Upload results to a server: Uploading several questionnaire results to a server should be one of three export options to transfer results from the application. If a result has already been uploaded, this should be recognizable by an optical indicator.

#FR11 Export single results to the device: The second export type supported by the application is the export of a result as an `HTML` file. This can then be stored on the device-specific memory and opened using a browser. For economic reasons, this export method only allows the export of one questionnaire at a time.

#FR12 Send single results per email: The last possibility of transferring the results out of the application is sending them by email. To do this, the application should access the device-specific mail program and attach the data as an `HTML` file. It should also be possible to configure a default recipient address in the settings, so that in an ideal case only the send button has to be pressed. With this export option, again only individual results can be removed from the application.

#FR13 Questionnaires can be resumed after an application breakdown: A crash of the application should cause as little data loss as possible. Especially, during the execution of the questionnaires, the completed questionnaire should not be lost. Although application crashes should not be frequent, precautions should have been made in case of their occurrence.

#FR14 Invalid data input is prevented and marked: In order to increase data validity, all incorrect entries should be optically marked and excluded from storage where they are not permitted. The purpose of this is to ensure that a decisive advantage over paper questionnaires can be guaranteed, because when filling in a paper questionnaire, only an indication of the expected input can be stored, whereas the Questionnaire application should not allow this input to be made.

#FR15 Questionnaires can be executed in a test environment: If the user of the application only wants to fill in questionnaires on a test basis, it should be possible to put the application into a test mode. As a result, all results obtained in this mode are marked as a test, so that they cannot be mistaken for the original valid results.

#FR16 Results can be flagged manually as "test": It should also be possible to mark results as a test afterwards. This is done in the results management and serves the purpose that test results can be identified more quickly. Furthermore, it is no longer necessary to delete results just because they have been created on a test basis and there is a risk of mixing them with the real results. The marking helps to identify them and thus greatly reduces the likelihood of incorrect treatment.

#FR17 Results can be tagged: However, marking a result as a test should not be the only way of marking in this application. Individual results of questionnaires can also be marked with colors. This should make it possible to simplify later running processes with a result, in which the user assigns meanings to the available colors.

#FR18 Themes can be switched: Since the application also claims to be able to handle psychological questionnaires, it is important to make the color design of the application alterable, as individual color settings could possibly influence the respondent. This functionality should therefore also be easy to extend in order to be able to meet customer requirements promptly.

#FR19 Results with errors in the generation are marked as such: In order to avoid that an originally valid result is affected by errors in the questionnaire execution, the responsible engine should mark the result of the execution as erroneousThis makes it possible to check for lost data, or sort them out.

#FR20 Personalized imprint: Since the mobile device containing the application could be available for general use, legal functionalities are also necessary. In Germany, for example, it is obligatory for every provider of information or services that are not purely private or family-related to provide an imprint [25]. And since the provided questionnaires are usually not created by QuestionSys itself but by other users or organizations, they have to be labelled in the form of an imprint.
Therefore, as a conclusion it can be formulated that a personal imprint, i. e. a user customizable one, should exist.

#FR21 Customizable dashboard for users: And for the same reason, i. e. the complete possibility of handing over the device to respondents, it should also be possible to determine elements of the homepage completely freely. These elements can be the various available questionnaires which can be filled in and, if necessary, a field from which interrupted questioning can be resumed.

4.2.2 Non-functional Requirements (NFR)

The complexity of software is mainly defined in two ways, the functionality and the global requirements. *Non-functional* requirements cover the global aspect while the functionality is described in the functional requirements. The global or *non-functional* requirements play a critical role during software development, as they define its development or operational costs, performance, reliability, maintainability, portability, robustness and the like [26]. By applying this knowledge to the *Questionnaire* application, following *non-functional* requirements can be formulated:

#NFR1 Application should work online: As the title of the description already implies, the application should be able to connect to the Internet. This in turn allows any server communication concerning the questionnaires or their results.

#NFR2 Application should work offline: If an Internet connection is not available or possible, the application should still be able to perform its core tasks without any problems. Hence, questionnaires that have already been downloaded should still be executable and results or started questionnaires can still be managed (and resumed). Only the communication interfaces towards the server should be affected by the missing connection.

#NFR3 Manageable device memory load: The use of the application's device memory should be clear and manageable. This means that downloaded questionnaires, which burden the devices memory, can be removed from it, this also applies to results and started questionnaires.

#NFR4 The application should provide breakdown robustness: In case of an application crash, the loss of data should be as low as possible, this applies primarily to the situation when the crash occurs during the completion of a questionnaire. However, the loss of data should also be kept to a minimum for other use cases.

#NFR5 Multi language support: Since the questionnaires, which can be executed with the help of the application, are configured externally, their multilingualism is not the same as the multilingualism of the application. This means that the language of a questionnaire should not always be the same as the language of the application. Nevertheless, it is the objective to ensure a maximum of flexibility, which certainly underlies the principle of multilingualism. The questioner can now use the application in his or her native language, while the interviewee can have the questions displayed in his or her native language. To enable scenarios like this, it should be easy to change these languages. The questionnaire language should therefore be selectable directly before the start and the application language, contrary to a large number of applications which adapt to the system language, should be selectable in the settings.

4.3 Structure

Once the requirements have been defined, the *functional* requirements can now be grouped according to their tasks. From these groups, a structure can be worked out, which corresponds to the final application structure. Starting with the first two functional requirements, it can be quickly determined that both, a login page and an administrative section, are required to fulfill them. These are linked to a start page to serve *functional* requirement 21 and in turn, offer a good link to the imprint page, which satisfies #FR20. In order to be able to display a personal imprint, there must also be a possibility to deposit the required data, this leads to the first page in the administrative area to add: The settings. This enables besides the settings for the personal imprint also settings for #FR 7,15 and 18. The only requirements not yet covered are all directly related to the core functionality of the application, the processing of questionnaires. This includes downloading, resuming and result management. For each of these functions, an additional page is provided in the administrative area to meet requirements 7, 9-13, 17 and 21. The last remaining group of functional requirements is the one, which relates directly to the execution of a questionnaire and consists of #FR 5, 6, 8 and 14. To fulfill these requirements, another page is added to the application structure. However, in this case the challenge is that the page should be accessible both from the start area as well as from the administrative area. Accordingly, a concept was developed to integrate the execution view into the different pages without causing permission violations. This led to the result that the same functionality now takes place on a component instead of a page, which can therefore be displayed on the respective pages. To summarize, the application structure consists of 8 pages whose links can be seen in Figure 5.2.

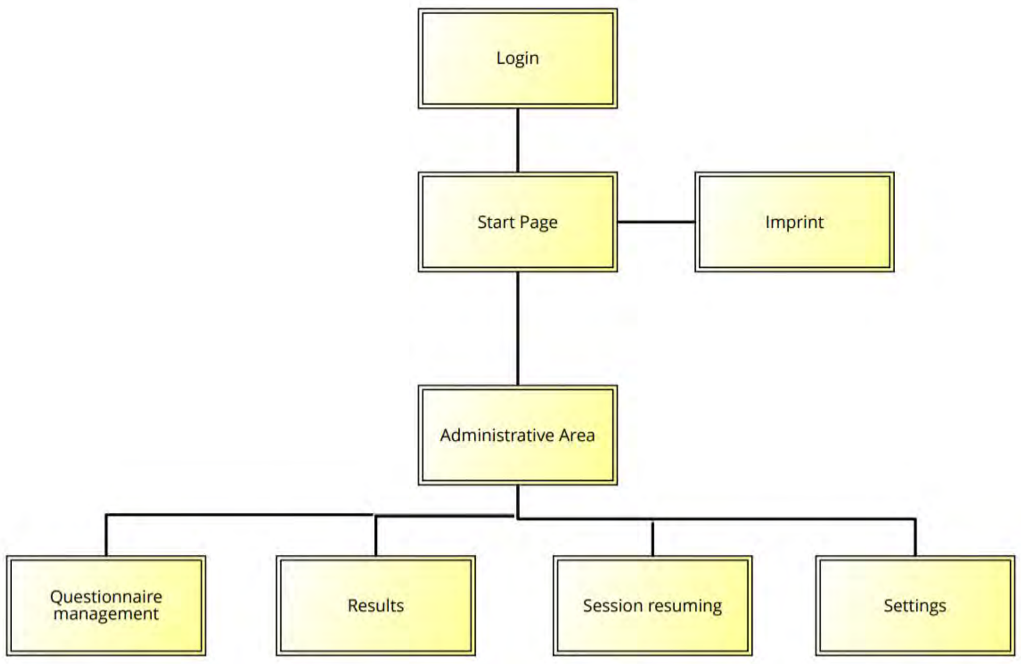


Figure 4.1: Navigation Structure of the *Questionnaire* Application

Starting from this as a base, the following describes the exact functionality and the implementation of the individual components, always with the focus on the existing data management. These components should, in turn, guarantee fulfillment of the above requirements.

5 Implementation

After the previous chapter explained and defined the framework structure of the application in terms of its concept, requirements and structure, this chapter will now be a functional analysis of the application with regard to the implementation of data management. In this case, the procedure is always that the respective function itself is first presented and subsequently examined from the application's perspective and finally from the data management point of view. Furthermore, the creation and evolution of a functionality will be explained with the help of code snippets and screen-shots, to simplify the understanding of it.

5.1 Database

As already explained in Section 2.3.1 there are three main ways presented to store data: 1. the database, 2. the file system or 3. a server. This section introduces the `sqlite` database respectively the database scheme used on the device to store local data.
Therefore, three tables were used to store all offline data of the questionnaires on the device as shown in Figure 5.1. The central and most important is the `questionnaire` table. It provides all the necessary status information, such as synchronized or pinned, as well as all the data needed to complete a questionnaire, such as graphics and labels. Moreover, there is some meta information that supplies the user with supplementary data.
Further, there are two additional tables: `result` and `instances`. They contain the questionnaire id as a foreign key and are therefore linked with each other. An element of the `result` table stores all the data of a completed questionnaire, such as the answers

given and whether it is a test in addition to if errors occurred during the execution of a questionnaire. Also, whether a result has already been sent to the server is stored in this table, as well as the specified flags, which will be explained later.
The last of the three tables is the `instances` table that keeps all questionnaires that were started but not completed. This is necessary to provide the possibility of resuming questionnaires on a later moment. Therefore, the table keeps the whole engine instance with some meta information and is also connected to the questionnaire table with a foreign key.

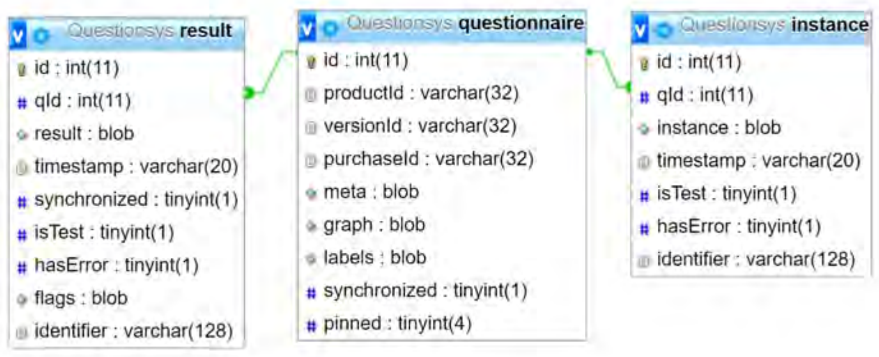


Figure 5.1: Database Scheme

Introducing this database will help on further explanations of several features and give a closer understanding of the data storage within the application.

5.2 Services

As mentioned in Chapter 2.3.2, services in an Angular environment primarily serve the purpose of data transfer, allowing for parallel transformation processes. They are therefore absolutely necessary to supplement the structure of the application with communication paths shown in Figure 5.2. In terms of the *Questionnaire* application, 5 services have been integrated. These are: `authentication-service`, `file-system-service`, `online-service`, `preferences-service` and `sql-storage-service`. The goal in creating these was, to imply the respective function already at the naming, in order to minimize error sources in the coding. Nevertheless, in the following, the individual services are explained in detail, how they work, and how they are integrated into the overall application.

5.2.1 Authentication Service

The `authentication-service` is, as the name implies, responsible for the authentication of the user. The administration of the associated data is also handled by this service. Thus, passwords are encrypted here before they are stored with the user data, both on the device, as well as for the first login to the server. To encrypt the passwords, the JavaScript library `crypto-js` is used, which encrypts the password using a salt and 1000 hash iterations. Subsequently, the encrypted user hash is stored together with the other user data in the data storage of the device.
In the case of a login attempt in the administrative area, the entered password is then compared with this stored data in order to be able to ensure offline access to this area. Also stored on the device is the data of the service which is presented next, the `file system service`.

5.2.2 File System Service

This service supports the application when it comes to storing files on the device. In contrast to the previously mentioned user data, i. e. type script objects, which can be stored with the ionic storage, files have to be managed in a file system. For this purpose, a folder infrastructure must be created that meets the requirements of the application and allows a clear and structured management of the files. The files that are involved in this case are `HTML` files containing the result of a questionnaire or images of the personal imprint. The former involves the task of initially creating a managing folder.

Concerning the `HTML` files, it should be noted that these files exist because they are or have been exported. They are therefore irrelevant within the application and are copied or forwarded by the user immediately after they have been provided, which could lead to unnecessary overcrowding of the associated folder. Especially since the user, after creating, has no further access to these files using the application. Therefore, on every application boot, this folder is flushed by the `file-system-service`.
The management of images of the personal imprint, however, is less complex, since there is always only one image that is either replaced or removed. The used storage space does not scale but stays within the scope of the usual image file sizes. However, the text for a possibly existing image is not managed by the `file-system-service`, this is the `preferences-service` in charge, because the personal imprint itself falls under the range of preferences.

5.2.3 Preferences Service

Of course, settings can also be made in the *Questionnaire* application, and in order to manage the data flow of this category the `preverences-service` has been created. This part of the application is often underestimated, since it often contains only small values that can be changed. However, what is often overlooked here is the fact that settings are set centrally, but are used in many different places. This also defines the main task of the `preferences-service`, namely to collect, store and keep the settings available. The following concept was developed during the implementation process: First, default settings are set for each value, and then, like the authentication data, they are placed in the application's local storage. This data is then placed in the RAM every time the application is started. Keeping the settings in RAM is not too much of a burden for the device, as the settings are only a few bytes big. If a setting is therefore changed, it is adjusted in two places of the back-end: The local storage and the RAM, because if it would only be stored in the RAM it would be lost in case of an application reboot. Now all this leads to the question why the settings are kept in the RAM at all. This is because settings usually have to be available very quickly. Obtaining the data at any time from the local storage would cost time unnecessarily. Instead retrieving data from RAM is a very fast process. For this reason, it was decided to release a part of the RAM for the settings, because the bottom line is that it saves more time, than it costs resources. This presupposes, however, that, as already mentioned, images are not saved directly in the settings but only the information about whether an image is present and if so, where. In summary, the structure can be formulated as follows: The settings are stored in local storage and are available in RAM during the use of the application.

5.2.4 SQL Service

Another way to store data is, the database mentioned in the previous chapter. In the case of the *Questionnaire* application, a `SQLite` database was integrated here. The `sql-storage-service` is responsible for organizing the data transfer to and from the database. As usual with SQL databases, this is done via SQL queries that either retrieve data from the database or deposit it and change it there. In the context of the application, the `sql-storage-service` is primarily limited to organizing questionnaire-specific data. Thus, the instances of a started questionnaire, the results of a completed questionnaire and the questionnaire itself are processed via this service to the database. However, synchronizing the results with the server is done by the `online-service`, so the `sql-storage-service` passes the results to it before uploading.

5.2.5 Online Service

The last service to be examined is the already mentioned `online-service`, that covers all communication with the server. This primarily involves the synchronization of results and the account management relating to the login. The downloadable questionnaires are also obtained from the server via the `online-service`. In order to ensure communication with the server, the routes for the desired REST interface are stored, only the authorization token is procured via the `preferences-service`.
If then server communication is required, it always runs the same, only the corresponding routes change. First, the *Header* is generated, which contains the API token. Next, this may be supplemented with data, which should be sent to the server. Then the request is

sent to the respective route as `GET` or `POST` request, with the help of the `HTTP` package of *Ionic*. Finally, the returned Promise is processed according to the situation.

Adding these services to the database and completing the page structure of Figure 5.2 results in an aggregation of different elements on which the *Questionnaire* application is built.

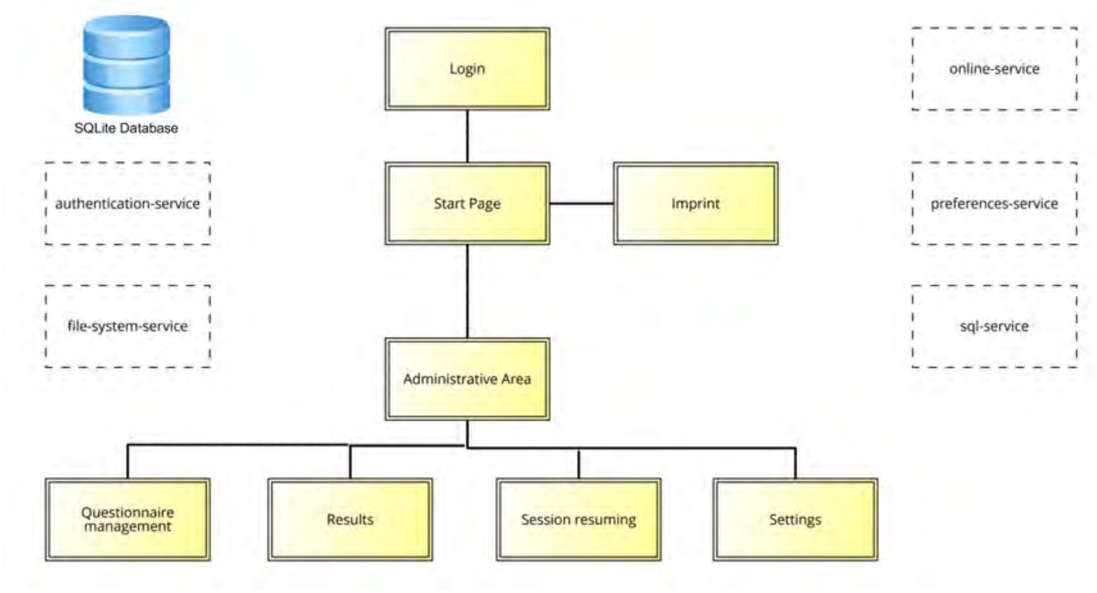


Figure 5.2: Navigation Structure of the *Questionnaire* Application Supplemented with the Services and the Database

5.3 Login

After explaining the basic structure of the application, it is now feasible to focus on the implementation of special functionalities. Since one has to login first, before using the application, the login procedure is described first.
The functionality of the login is trivial after the layer system is defined and explained. As one can see in Figure 5.3 there are different ways to organize login layers. A layer, thereby, is an access level reachable with certain rights. In Figure 5.3 “A” refers to the

public layer. Anyone can access this system area and there is no special permission needed. Second group (“B”) are the users with access rights to the software functionality and the third and last group (“C”) are the accounts with administration rights.
Differing the types of layer constellations one can see that the first layer of Figure 5.3 is a simple login with just a user group and a public group. The second one shows a standard account based software. Where all admins can access the administrative area and the user area, but not any user is an administrator and has access to the administrative area. The third login constellation is a depiction of the *Questionnaire* login. In this case, user and public groups are the same. This is because in order to access the user area, the admin has to enter the password after installation first. This means that, before this login, no one gets into the user area which is not admin and after that everyone. However, the administrative area is reserved for the administrator only.

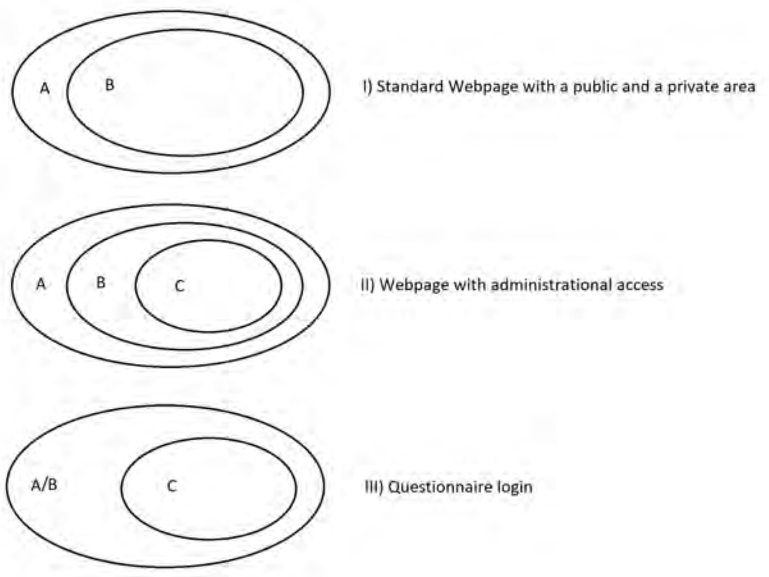

A
B
I) Standard Webpage with a public and a private area
A
B
C
II) Webpage with administrational access
A/B
C
III) Questionnaire login

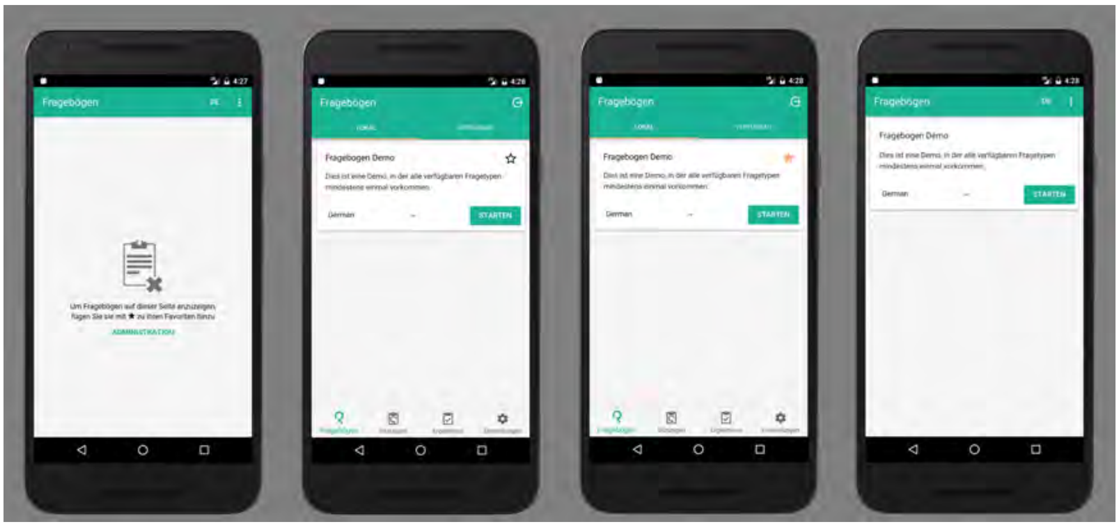
Figure 5.3: Access Levels in Different Web Applications Under the Aspect of the Number of Authorized Users (not to scale)

In detail, the login is a two layer login using the same password. If the user installs the application and starts it for the first time the initial page will show a login screen where the user is asked for the credentials. If this is done he will never be asked for that login

again until he changes it in the *QuestionSys* web application. After the first login has been passed successfully the public home view of the application is shown. There, the user has the ability to login to the administration area by using the same password he used for the first login. To come back on the explanation of Figure 5.3 it means that public people and users are the same group as they don not need to login anywhere.
Integrating a double login like it has been explained, comes with the big advantage, that there is no internet connection needed while accessing the admin panel within the application. However, the security aspect of a protected admin area is still present. The only moment an internet connection is needed, while a log in takes place, is at the very first time after a (re-)installation.
If one considers this from a technological point of view, the login procedure is also split into two parts: saving userdata to provide an offline login and the, therefore, needed communication. Starting with the saving part, one has to define data that has to be saved, in this case the user name and the password. Both of them are stored on the device after the first login took place. To get a better understanding of how the first login is structured, the code example A.1 provides a good orientation. The online login takes place at line 6 and after this is completed, the new user will be stored as a verified account and has now access to the administration panel of the application. The storage of the data happens in line 10 and is part of the `loginNewUser` function of the `authentication service`.

5.4 Creating an Individual Home Screen

Accessing the application the first time leads the user to a nearly blank screen. This is due to the functionality of customizing the home screen with questionnaires to be filled in. To achieve a better knowledge of how this is realized, Figure 5.4 shows the process of adding a questionnaire to the home screen from left to right. While the first screen shows the home page, the two screens in the middle show the admin area, mentioned in 5.3. By clicking the *star icon* on the upper right of a questionnaire it is pinned to the home screen, as one can see at the last screenshot.



Um Fragebögen auf dieser Seite anzuzeigen, fügen Sie sie mit ★ zu Ihren Favoriten hinzu

Dies ist eine Demo, in der alle verfügbaren Fragetypen mindestens einmal vorkommen.

Figure 5.4: Pin a Questionnaire to the Home Screen

This creates the possibility of individually designing the application, by offering specific questionnaires to participants.
Concerning the functionality and for a deeper understanding of the process of pinning a questionnaire, a more detailed explanation is needed. At first one has to log into the administration area. If the access has been granted, the next page on the screen is a page listing up all offline available questionnaires. To pin a questionnaire clicking on the *star icon* is required. After logging out from the administration area, one will now see the homescreen with the previously selected questionnaire, pinned at the next available position. This pin stays active until it is removed (by clicking the icon again).
Regarding the chain reaction from pressing the *star icon* to the view of a pinned questionnaire at the home screen, the first action is setting the questionnaire pinned. This happens on two different places, at first the questionnaire is set pinned in the device own database, before the questionnaire itself, is set pinned what mainly provides the checked *star icon* as one can see at Figure 5.4. Because of the database access being an asynchronous function, it is important to keep the order of the two actions as explained to avoid checked *star icons* although database transactions were not completed successfully. However, if the transaction was successful, the questionnaire will be in

the returned list of pinned questionnaires, every time the home screen is accessed. This happens with the help of the `ionViewDidLoad` function as seen in A.2. After the `sqlService` delivered all pinned questionnaires, they will be stored and collapsed[1]. The `selectLanguage` function assigns the correct language to the questionnaireThe application language is picked, if this language is provided by the questionnaire or english if not. If neither of those are available, the first possible language is picked.

5.5 Downloading a Questionnaire

A questionnaire must be downloaded first, before it can be shown on the home screen. To get a deeper understanding of what downloading a questionnaire means, it is necessary to remember the origin of a questionnaire. At first, a questionnaire is created with the *Questioneer* application and is then published on the store (Section 2.4). The crucial criteria why someone is able to download this published questionnaire is that one has got the permission to do so. So if an application user is requesting all available questionnaires, he will only see those where he has the permission for downloading.
Additionally, downloading a questionnaire requires administrator rights, therefore, it is only possible at the administration area. When this area is accessed, the first visible tab is the same as in Figure 5.4. This is also the view where questionnaires can be downloaded. But instead of clicking the *star icon* to pin the questionnaire to the start page, one has to click on the “Available” button to get an overview of all the questionnaires the user has the permission to download. By clicking the download button of a questionnaire, the recent version of a questionnaire will be downloaded automatically. It is also possible to select a different version by clicking the displayed version number and select another. By clicking on download then, the selected version of the chosen questionnaire will be downloaded. Downloaded questionnaires can be viewed vy accessing the “Local” tab (cf. Figure 5.4).
Considering, the entire process from the perspective of data management within the Questionnaire application, the following procedure description results: Before actually downloading a questionnaire one has to get a list of downloadable questionnaires. To

[1]A collapsed questionnaire is a questionnaire where the detail view is hidden.

get this, one has to send a request to the server as seen at code sample A.3. There at first a `Get-Request` to the `my_purchases` route is made and after the response arrived and was successful each of the received purchases is transformed into a local manageable meta object. It should be noted that this meta object created at line 16 and following is not a meta object described in Section 5.1 because the created object here is not stored for a longer time.

After the transformation is completed the available purchases are transferred to the *front end* view to be displayed. This is the same view where the user has the possibility to press the download button and initiate the process of copying a questionnaire from the server to the local device which happens in several steps. First step is contacting the server and getting the data. That works essentially like the `Get-Request` already explained but using another request route and receiving different data. The data received from the server, is in this case `JSON`-object which is transformed into three different *TypeScript* objects and stored in the database of the device. Figure 5.5 gives a visual impression of the changes made in this phase. Out of the resulting three objects, two of them were adopted from the original date. The meta and the labels property of the `JSON`-object became "obj1" and "obj3" while the model attribute was fully transformed into a new object by the `GraphTransformation` ("obj2").

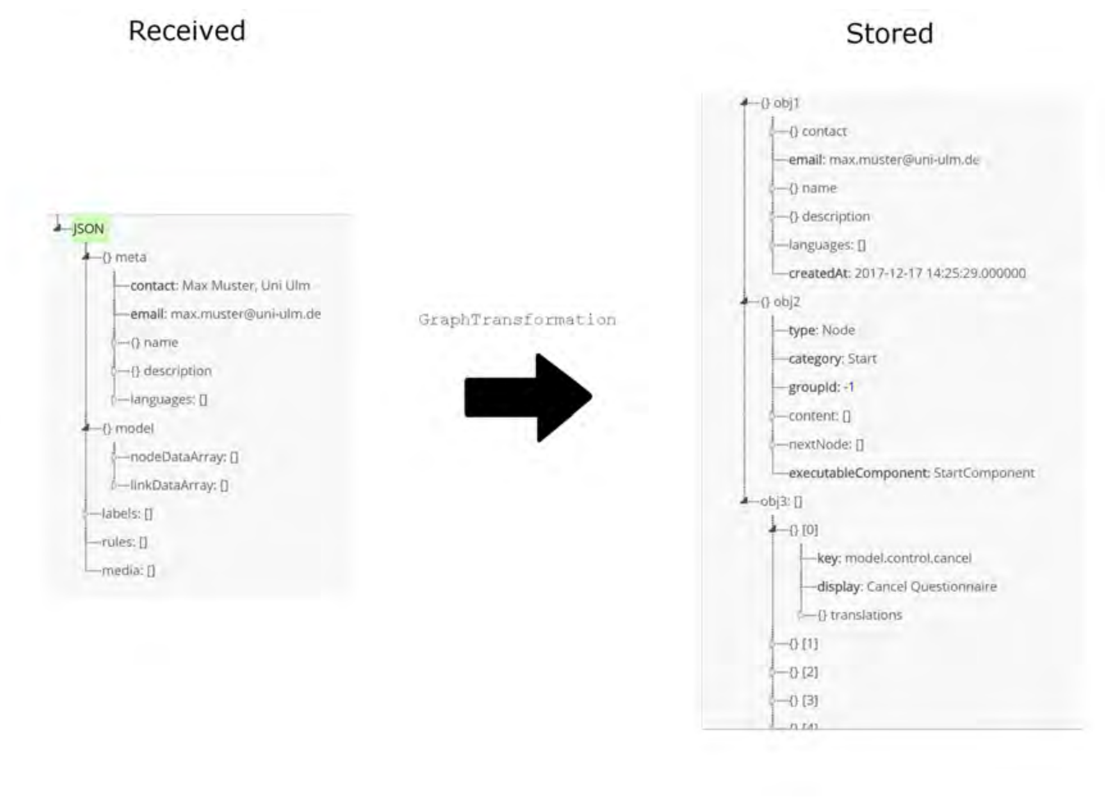


Figure 5.5: Data transformation after the download

5.5.1 Graph Transformation

To gain a deeper understanding of the complexity of the `GraphTransformation`, one should first consider the data structure and representation of a questionnaire before and after transformation. A good first orientation, therefore, is the questionnaire created with the configurator and also represented below (cf. Figure 5.6). This shows the structure of a questionnaire, which corresponds to the graph structure. The exemplary questionnaire serves as an illustration and should be a guide to better understand the individual steps of the transformation.

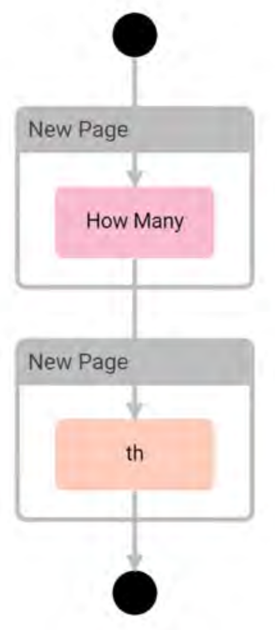


Figure 5.6: A Questionnaire During Creation Within the Configurator

A questionnaire like this is a set of nodes and links connecting the latter. `AND`, `XOR` or `LOOP` semantics are provided, in order to describe respective control flow. Apart from the `split` type, there are additional categories that a node can belong to. Listed below are all categories provided by the server:

- Start: The node every questionnaire starts with.
- Invisible: Acts as a separator of pages respectively groups.
- PathLabel: A description of a path that is used as additional information during execution.
- Split: `AND`, `XOR`, `LOOP` and other non defined variations possible.
- Join: The end of a split section.
- Page: The node initiating a new group of questions that are displayed on the same page.

- Media: Media elements that can be placed on pages like questions.
- Question: A question of any type.
- Custom: Not defined yet.
- Headline: A headline that can be placed on pages.
- Text: This is like the headline node but displayed as normal text and therefore usually longer than headlines.
- End: The node every questionnaire ends with.

In addition to the nodes already described connectors are also necessary to build a graph. These connectors appear as a `linkDataArray` on Figure 5.5, in which each of them keeps the `id` of the starting and the ending node in addition to its own `id`.
The next step is to examine is the behaviour of the `GraphTransformation` which gets the node and the link list as input value and transforms it to a graph that is executable by the *Questionnaire* engine. The exact composition of the `GraphTransformation` can best be seen in Code Sample A.4, which is a good way to trace the steps explained in the following.

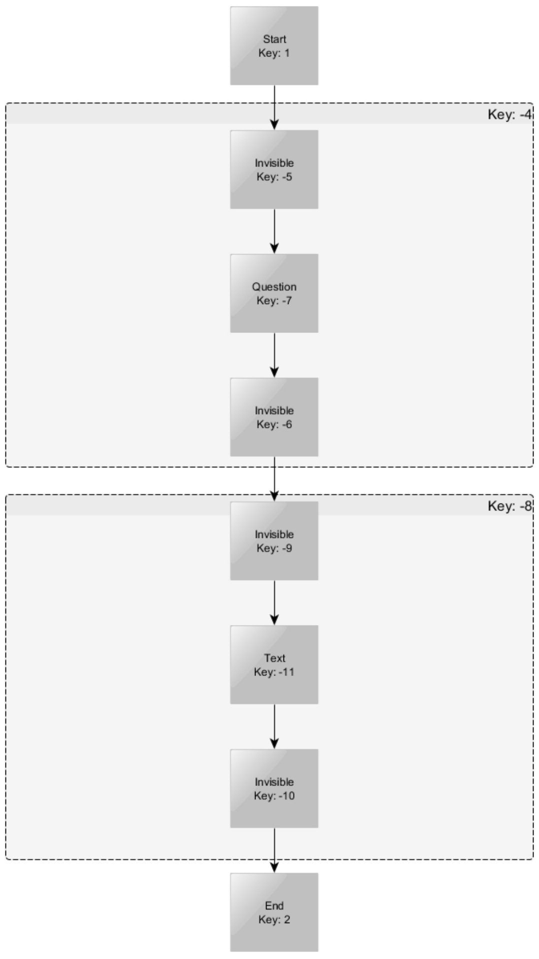
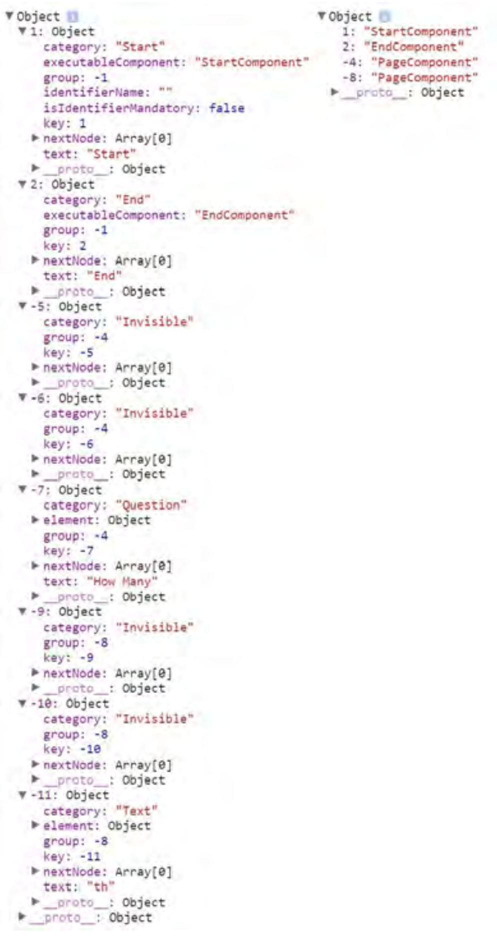


Figure 5.7: The Server Sided Structure of a Questionnaire

The nodes and the links of the received graph are arrays. The transformation combines them into only one object, the start node. Starting from that node, the graph structure will be represented by the `nextNode` attribute that replaces the links. By storing the next Node inside the `nextNode` property, no additional links are required.
Removing this links will therefore be the first task of the graph transformation, as seen at the `transformGraph` function, on code sample A.4. The transformations order is: Generating `Key-Value` pairs of the nodes and their `id`, followed by the `assignLinks` function that generates the `nextNode` assignment and reduces `Key-Value` pairs and

links to just one start node. The last step is grouping the graph, by removing unnecessary nodes and assigning application specific data types.
Starting with the creation of the `Key-Value` pairs the responsible function is the `_generateKeyValuePairs` function, which at first defines an empty `map` and `groupMap` property. It is important to keep these types of maps apart to ensure the layered structure of the graph (cf. Figure 5.7). Taking the page node and the question node as an example, it is obvious that page and question can not be on the same level because a question is always part of a page, otherwise it wouldn't be displayed and make the question purposeless. On the other hand there are nodes on the level of a page where a visual presentation is not absolutely necessary for example the "Start" and "End" node. Hence, the structure of a group node is different to the structure of a standard node, which leads to a different treatment and storage. While the `groupMap` is stored as a value of the class attribute `groupMap`, the `map` is returned and handed over together with the link array to the next function, `_assignLinks`[2].

[2]Both maps are visible on Figure 5.8, with the map on the left and group map on the right.



```
▼Object
 ▼1: Object
    category: "Start"
    executableComponent: "StartComponent"
    group: -1
    identifierName: ""
    isIdentifierMandatory: false
    key: 1
  ▶nextNode: Array[0]
    text: "Start"
  ▶__proto__: Object
 ▼2: Object
    category: "End"
    executableComponent: "EndComponent"
    group: -1
    key: 2
  ▶nextNode: Array[0]
    text: "End"
  ▶__proto__: Object
 ▼-5: Object
    category: "Invisible"
    group: -4
    key: -5
  ▶nextNode: Array[0]
  ▶__proto__: Object
 ▼-6: Object
    category: "Invisible"
    group: -4
    key: -6
  ▶nextNode: Array[0]
  ▶__proto__: Object
 ▼-7: Object
    category: "Question"
  ▶element: Object
    group: -4
    key: -7
  ▶nextNode: Array[0]
    text: "How Many"
  ▶__proto__: Object
 ▼-9: Object
    category: "Invisible"
    group: -8
    key: -9
  ▶nextNode: Array[0]
  ▶__proto__: Object
 ▼-10: Object
    category: "Invisible"
    group: -8
    key: -10
  ▶nextNode: Array[0]
  ▶__proto__: Object
 ▼-11: Object
    category: "Text"
  ▶element: Object
    group: -8
    key: -11
  ▶nextNode: Array[0]
    text: "th"
  ▶__proto__: Object
 ▶__proto__: Object
```

```
▼Object
   1: "StartComponent"
   2: "EndComponent"
   -4: "PageComponent"
   -8: "PageComponent"
  ▶__proto__: Object
```

Figure 5.8: A questionnaire while creation within the configurator

This function is responsible for the link replacement which was implemented to loop over the links and push the node owning the `link.to` id into the `nextNode` list of the node owning the corresponding `link.from` id. This results in the start node now representing a coherent graph, but still without group and data type assignment.

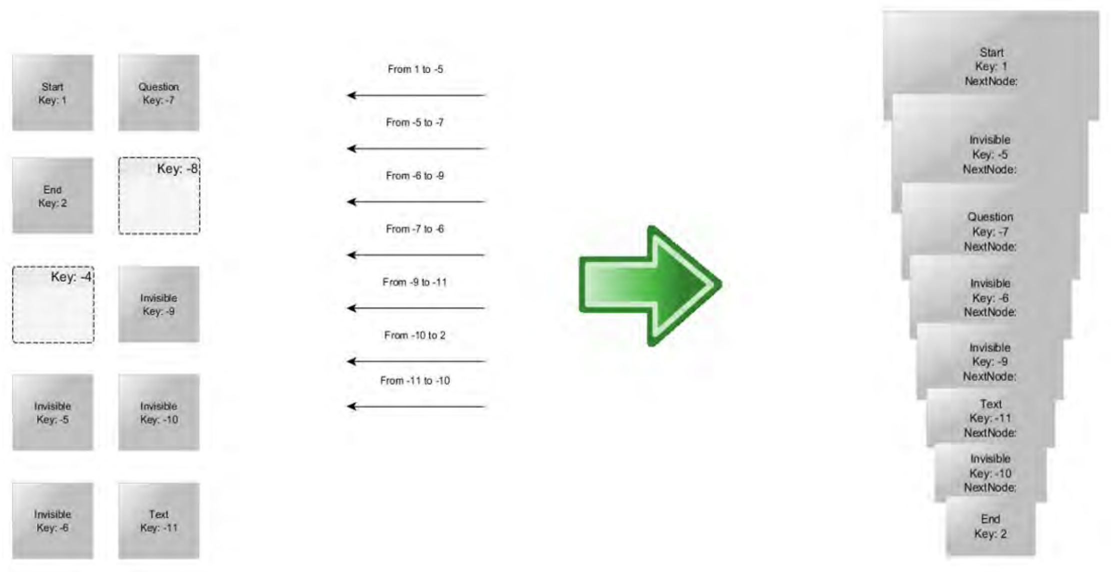


Figure 5.9: The Effect of `_assignLinks`

This is the purpose of the last function needed to transform the graph, the `_groupGraph` function.
`_groupGraph` has two given parameters: `graph` and `stopAtJoin`. While the first variable names the currently handled node, the second variable declares if the walk through has to stop at the next join. As the variables already indicate the `_groupGraph` is a recursive function that runs through the structure created before.
A rough overview:
The currently handled node gets the class it belongs to, is managed into its group (if it has one) and the function is triggered again with its next node. The `stopAtJoin` variable is important in case of a split, because without that one would have multiple branches running through until the end node, becoming more parallel processes on every split. But with this variable, in case of a split, the first path is running through (with `stopAtJoin=false`) while all the other branches have this variable on `true` and therefore terminate after the `join` has been reached.
To find out what is happening to a node within the `_groupGraph` method, the code is

a good guide. It can be seen that first a case distinction is made, using the already mentioned categories of a node. After that there is a specific handling, concerning the category that has the aim to put group members into groups and only link those groups with splits, joins and other top level elements.
This leads to a layered graph consisting of pages connected with splits, joins and the start/end node to simplify the graph itself. Within the groups are, in turn, smaller linear graphs chaining the questions together in a linear manner as they are displayed on the page later on.
However, after splits or joins can not occur within a page, there is no need to maintain the `nextNode` structure and is, therefore, replaced by a list which ensures the correct order of the elements. After all, this processes were executed and successfully completed, the questionnaire is in the structure represented in Figure 5.10. Th transformed questionnaire can then be stored into the database, like it is apparent in Section 5.1 and the download process is completed.

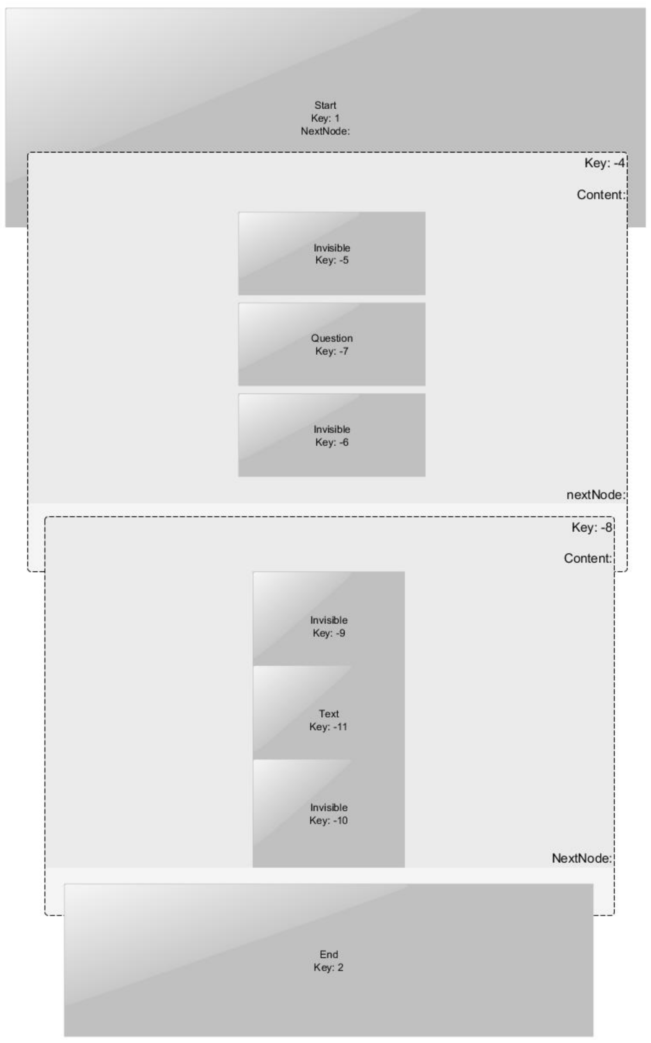


Figure 5.10: The graph structure after the last recursion of `_groupGraph` was made

5.6 Result Management

After a questionnaire is completed the engine initiates the storage of the answers a result at the database.[3] Having its focus on the result management this chapter will give an explanation on how results are synchronized with the server and marked. Additionally,

[3]The exact structure of this object is visible at Figure 5.1.

the challenge of managing results when being offline is demanded.
Therefore, one must examine the result life cycle from generation to the server transmission and/or the deletion from the device.
After the engine creates the result it is stored in the *SQLite* database of the mobile device, and presented within the result section of the *Questionnaire* application. The presentation of the results happens in three different views (cf. Figure 5.11).
First, all questionnaires are listed, from which results are available. Once one of these questionnaires has been selected, all results are displayed with their identifiers. Selecting a specific result, shows the collected data for this instance. Each view comes with additional features: View one and two can be switched into a selection mode, where multiple questionnaires (respectively results) can be selected and subsequently synchronized or deleted. The third view enables marking and exporting single results. As seen on the right screen of Figure 5.11, different colors are available to highlight a result. Further, results may be marked as *test*, in order to easily filter them from the valid results. The bottom right of the screen shows other features, like switching languages or exporting results. The latter is either a `HTML` export so it can be printed directly from the browser or an email.

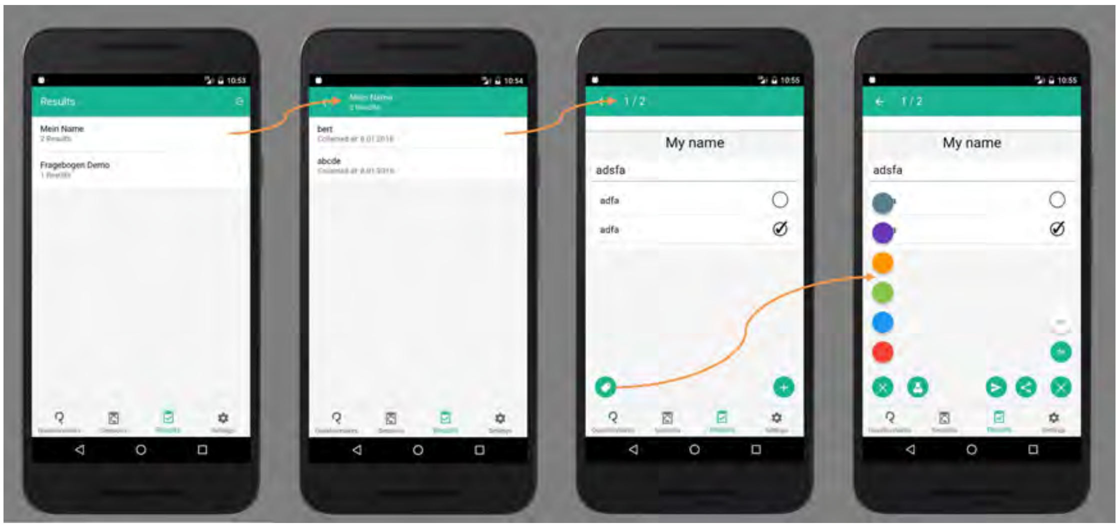


Figure 5.11: Result Management of the *Questionnaire* Application

To examine the result management under the aspect of data management, the data model of the database (cf. Figure 5.1) is integrated into the already explained functions and examined again. Therefore, each of the results attributes, will be described and its use and effects will be expounded. The `qId` attribute is the first attribute investigated which, as the name suggests, is the ID of the associated questionnaire. This allows for easy filtering and listing of results of a respective questionnaire using a `SQL` statement. This statement is executed on each page load and the result is displayed at the list as one can see at the first screen of Figure 5.11.

The second attribute of the database scheme is the result blob (a stringified `JSON` object of the result generated by the engine). It is used for visualizing a single result shown on the third and fourth screen of Figure 5.11 and is furthermore used for exporting the data[4].
The `timestamp` keeps the date and time of the moment the result was stored into the database. That provides time information for debugging and is used to sort the results within a questionnaire. The `timestamp` is displayed beneath the identifier, visible at the second screen of Figure 5.11.
Also to be recognized on this figure are the different marking possibilities that are there `synchronized`, `isTest` and `hasError` that can easily be summarized because of their identical functionality, even if they differ in their meaning. As the names of the variables imply, they describe if a result has either already been synchronized to the server, is just test data or an error occurred during execution. These values can be switched from `true` to `false` and this switch will be examined next. In the `synchronized` scenario this is already implied by the name. If a result is sent to the server it is marked as synchronized. This process of synchronizing is visible at A.5 of the “Source Code“ section. After the synchronization has been initiated, the next step is, to get all the result data needed from the database of the device and to subsequently have it sent to the server by the `onlineService`. If the data transmission was successful, all results selected are marked as synchronized. Hence, their `synchronized` attribute at the database is set to `true` and it is visible within the result view that these results don’t need further synchronization.

[4]Export as `HTML` and send as email.

Results that may not be sent could be the test results. Identifying these results is possible via the `isTest` variable. This attribute is set to `true` if the test mode has been activated in the settings[5] while executing a questionnaire. Another way to set `isTest` to `true` is by clicking the test icon at the bottom left of screen four in Figure 5.11. This also allows to set the test mark on `false` by clicking the icon again.
The last value, the `hasError` attribute, is true if the engine recognizes any errors while executing a questionnaire. If this attribute is true it is not possible to set it false again. Therefore marking errors on answer validity is not possible with the `hasError` attribute, but flags can be supportive on this task.
The `flags` attribute has similar functionality as the tags mentioned earlier, but differs in the back end data handling. Flags at the *Questionnaire* application are colors, as it can be seen on the bottom left of Figure 5.12. These colors can be activated or deactivated by clicking them. If a result is marked with a color, the mark will appear at the different result view layers. Taking the data management of this marking in concern one will see that the `flags` attribute is a list of colors. More precisely, it is a list of strings representing the hexa decimal code of the color, as seen on Code Sample A.6. This has several advantages. First, the server does not have to offer any predefined colors, since every transmitted result already includes the colors it was marked with. Second, there is no specified amount of colors to be provided, if a client only wants three colors instead of six (as shown in the examples) one just has to adjust the source of the colors.
To put it all together, the source of the selectable colors is a `JSON` document stored in the assets and the selections of this colors is stored as a `string` array [6] containing the colors as hex code in the database.

[5]See middle screen on Figure 5.13
[6]The array is stringified and, therefore, stored as a `string/blob` at the database.

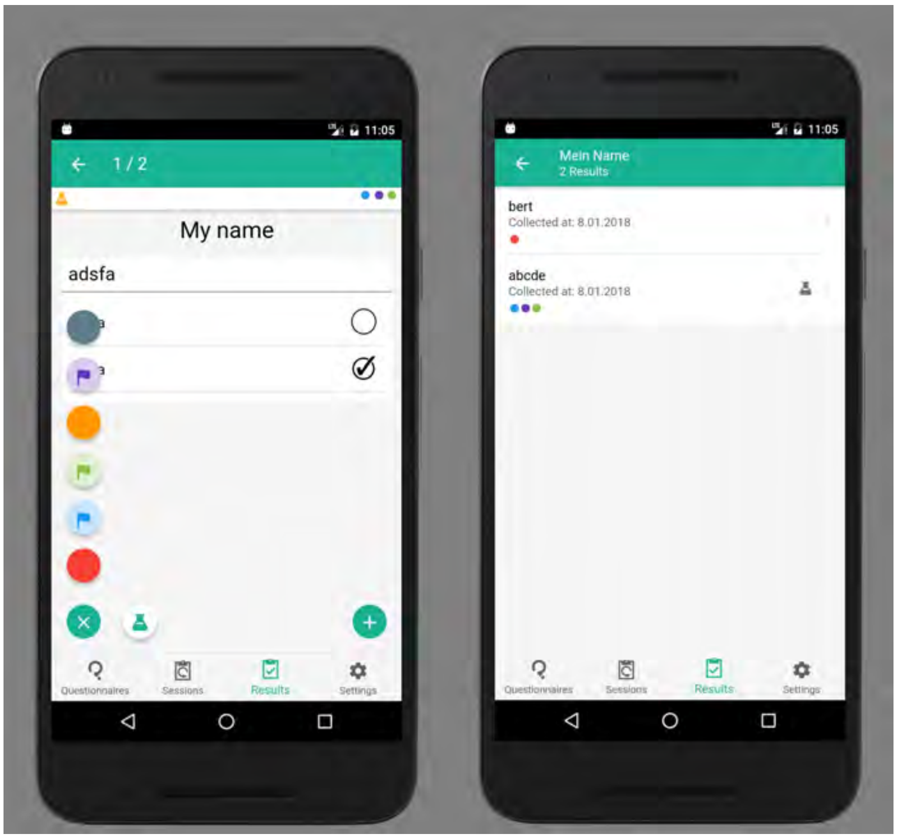


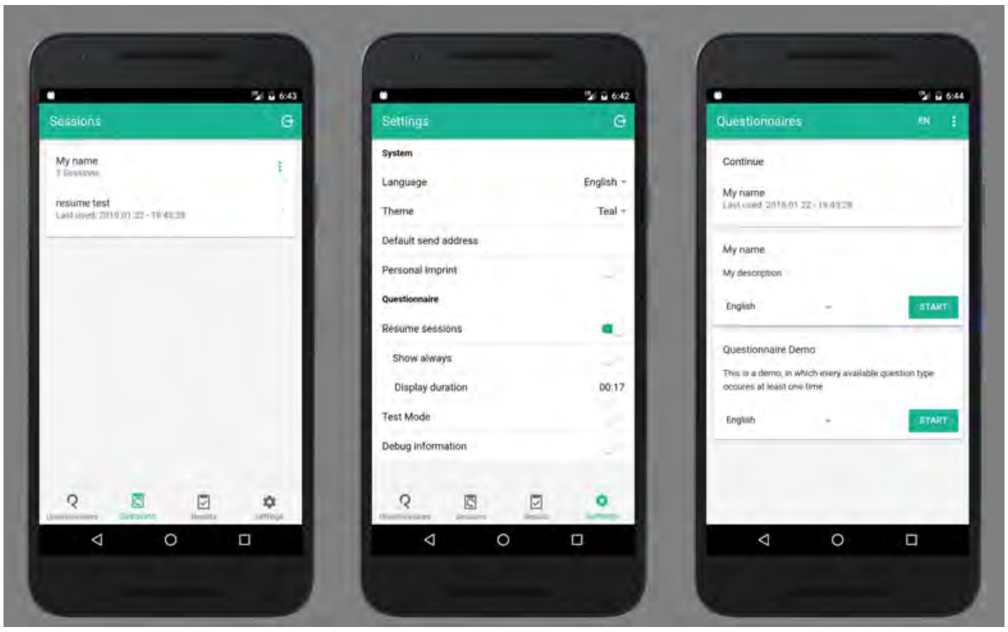
Figure 5.12: Marking a Result of a Questionnaire

Finally, if one takes another look at the database graphics (Figure 5.1), one notices that a last attribute, the `identifier`, has not yet been examined. The `identifier` attribute acts as the title of a result and is the first value specified when the questionnaire is executed. It supports identifying results and reduces time, while searching for a specific data set.

5.7 Resuming a Questionnaire

Managing results of a questionnaire requires the completion of a questionnaire. However, it is also possible that a user has to interrupt filling in the questionnaire or an application break down happens. If the loss of data in these cases is to be minimal, the possibility of

resuming a currently stopped instance is essential.
Therefore, the *Questionnaire* application has a session resuming functionality implemented. If a questionnaire is interrupted it is possible to resume it on two different ways. Either by activating the `resuming` option in the settings and use it from the home screen or resume a questionnaire from within the admin area (the *Sessions* tab). The difference of these methods are that a resumption out of the admin area is always possible as long as an interrupted questionnaire exists, the resumption from the home screen, however, has some requirements to fulfill.
To get a deeper understanding, it is helpful to include the illustration 5.13 on which the possibility to resume a questionnaire from the admin-area can be seen (on the first screen on the left side). All identifiers of the started questionnaires are listed there under the title of the corresponding questionnaire. A similar display of the resumeable questionnaires is shown on the third screen, where a questionnaire can be continued from the start page. The above mentioned `resuming` option is displayed on the middle screen. This screen shows the settings page and contains a point called *Resume session*. If this switch is enabled, two additional options pop up. One of them is the *Show always* toggle. If this is activated all resumable questionnaires will always be shown on the home screen. If this is not activated the second option gets into the fore. This option is a time in format `hh:mm` and represents the maximum age of a questionnaire session. That means if an interruption of a questionnaire is older than the given time, it does not appear in the list of resumable questionnaires. However, the list within the admin area doesn't have any restrictions, therefore this list will always show all interrupted and resumable questionnaires.



My name
1 Sessions
resume test
Last used: 2018.01.22 - 19:43:28

System
Language English
Theme Teal
Default send address
Personal Imprint
Questionnaire
Resume sessions
Show always
Display duration 00:17
Test Mode
Debug information

Continue
My name
Last used: 2018.01.22 - 19:43:28
My name
My description
English
Questionnaire Demo
This is a demo, in which every available question type occures at least one time
English

Figure 5.13: Resuming a Questionnaire

Concerning the data management this functionality has complex background processes that already start on creating the instances, which is done by the engine that processes a questionnaire. Everytime the engine switches to another page of the questionnaire, the instance object will be updated to the latest engine state.
That creates the possibility of resuming a questionnaire by an engine restart, as it is stored in a working state. Therefore the only attribute of the instances object stored in the database having specific data of a questionnaire is the instances attribute, as it holds the engine state, while the other attributes mainly contain meta data. In the next step, one can now examine more precisely how the instances are handled after they have been created. Thus, it was already mentioned in before that there are two ways of describing the representation possibilities. The straight way is visible at the administrative area where all the existing instances are listed. This at first came by a simple `SQL` query of the `sql-storage-service` that fetches all instances existing. But for usability reasons the administrative area was changed to a single page with different tabs instead of four pages as it was at the beginning. Although this has many advantages,

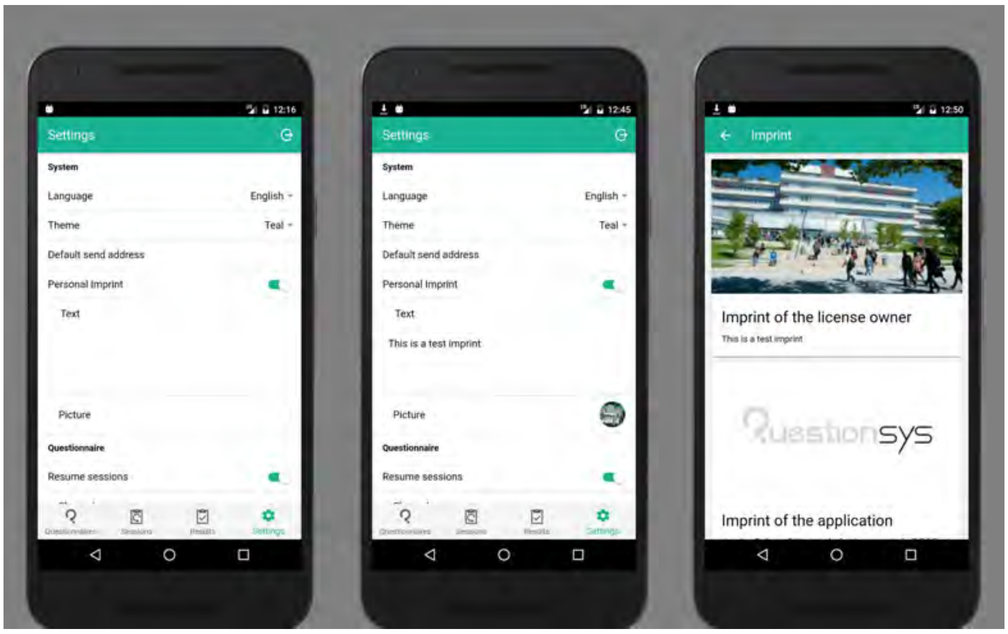
it also has the disadvantage that questionnaires can be started and continued on the same page. Therfore, if a questionnaire is started and interrupted the list of resumeable questionnaires has to contain this questionnaire without a page reload.
To realize this described behavior, an `observable` was created and subscribed from the administrational area. In order to clarify in advance the functionality of an `observable`, the following definition can be used as an explanatory reference: “Observables are data structures that trigger events when their internal state changes. The Angular eventing infrastructure extensively uses observables to communicate the components’ internal state to the outside world” [13]. But in this case the observable is more precisely a behaviour subject wich is a special kind of observable. Because “[w]hen an Observer subscribes to a BehaviorSubject, it receives the last emitted value and then all the subsequent values” [27]. Therefore, an observable usually delivers data the first time to its subscriber on the first change after the subscription, while a behaviour subject returns the last nexted[7] data already on subscribe. This means that a correct presentation of the questionnaires is guaranteed in the instances of the administration area. Because with the Behavior Subject, a download delay of the subscription, i. e. the list of instances, can no longer cause errors. Furthermore, the standard functionality of an observable is still given, which means that when a questionnaire is interrupted, the list of resumable questionnaires within the *sessions* tab is updated immediately. This structure and functionality is showen in the code sample A.7 and is the base of the session representation on the home screen. This requirement is, as already mentioned, that the apperance of a session at the startpage can be limited by time. If the time limit is, for example, set to 5 minutes and a session has been created ten minutes ago it should not be in the list (cf. A.8). Focusing the code sample step by step the first point is the subscription to the engine instances of the `sqlService` what is already known from the adminstrative area, `take(1)` means that this is a one time subscription and a manual cancelation of the subscription is not necessary and the third important part is the `repeatWhen` funtionality. This “[r]eturns an Observable that emits the same values as the source Observable with the exception of an onCompleted“ [28]. This means that every time the `notificationHandler`, i. e. the given function that is executed by

[7]In this case, *nexting* means adding a value to the value sequence from which the observables obtain their values [27].

`repeatWhen`, the value of the parent-observable is also passed to the map and filter functionality, which in turn guarantees a filtered list that meets the requirements of the given restriction. Limitations that are initialized at the settings page, where another functionality that should be discussed can be defined.

5.8 Personal Imprint

Due to the numerous possibilities of the application to perfectly adapt content to customer needs, it was also important to create a personal imprint. In this context, personal means that there should be the possibility of creating an imprint consisting of text and image as an addition to the imprint of the application provider.
To setup this personal imprint it is necessary to switch into the administrative area and the *settings* tab. There one can see a adjustment option called *Personal Imprint* and a toggle button next to it (cf. Figure 5.14). If this button gets activated additional input fields for custom text and image are displayed. The picture is optional but contributes to the the customizability of the application and, therefore, offers users the ability to keep their corporate appearance.



System
Language English
Theme Teal
Default send address
Personal Imprint
Text
Picture
Questionnaire
Resume sessions

System
Language English
Theme Teal
Default send address
Personal Imprint
Text
This is a test imprint
Picture
Questionnaire
Resume sessions

Imprint of the license owner
This is a test imprint
Questionsys
Imprint of the application

Figure 5.14: Personal Imprint

Concerning the personal imprint one has four variables that are important: If the imprint is activated, the text, the picture and a variable that indicates that a picture has been

saved. These variables are stored with the help of the preferences service that handles them as attributes of an object. This object is created at the settings page where any value is stored or changed immediately. Hence, the value of the imprint text needs to be safed immediately, too. This can be made after each key press on the keyboard but this causes a method execution on each key press and is therefore inefficient.
To avoid those multiple executions one can integrate a `debouncer`. This debouncer is a timer that executes a function after a given period of time. To avoid multiple executions of functions on multiple keypresses one insert a delay of a few seconds. Now the timer will execute a function after the given amount of time, but he will again do this for every key press. This leads to a trick used to avoid this unnecessary effort namely, only the first key press initiates the timer and each of the following just resets it to the first given period of waiting. After the typing ended and the timer expired the function will be called and in this case the text will be stored by the preferences service.
Less complex is the storage of the picture that can be added to the imprint. This picture is fetched with the cordova camera plugin that makes it possible for the user to select a picture from his/her library. Subsequently it is stored in the file system of the device as it was mentioned in section 2.3.2. Hence, the exact location of storing the picture differs concerning the operating system the application is running on. This also means that not the whole picture is saved by the *preferences service* but only the path and a boolean variable indicating if a picture was added to the imprint. These four values build a construct that gives the imprint page the possibility to quickly generate the personal imprint view that can be seen on Figure 5.14. The loading process of the imprint, shown in the diagram below (Figure 5.15), happens in several stages. These steps are dependent upon the variables that indicate the existence of a personal imprint and a therefore related image. The first step is the decision whether a personal imprint is loaded at all. If this is the case, there is definitely a text that can be loaded. If this has been loaded, it is decided in the next step whether or not an associated image has to be fetched from the data storage. If this is not the case, an imprint will be created without an image, but if an image exists, it will be loaded using the saved path and included in the imprint view.

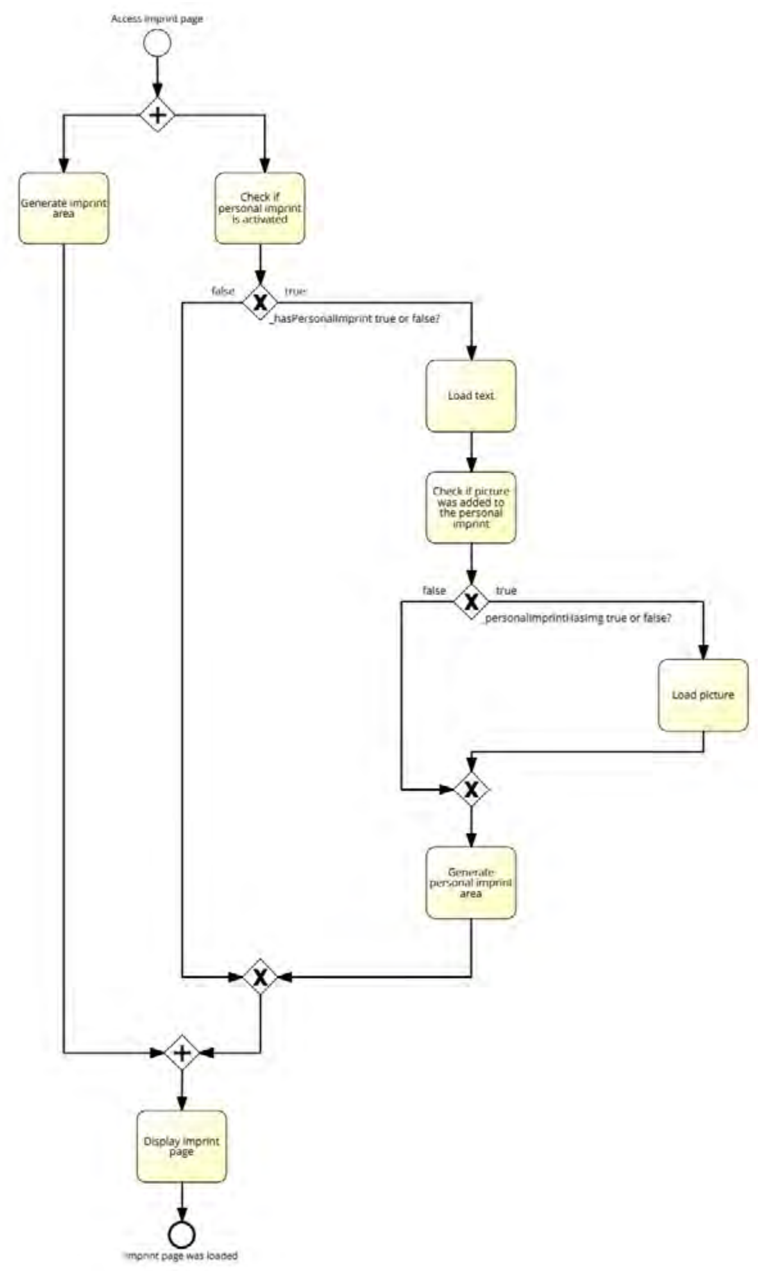

Access imprint page
Generate imprint area
Check if personal imprint is activated
false
true
_hasPersonalImprint true or false?
Load text
Check if picture was added to the personal imprint
false
true
_personalImprintHasImg true or false?
Load picture
Generate personal imprint area
Display imprint page
Imprint page was loaded

Figure 5.15: Loading Process of the Imprint

Regardless of the configuration, one will receive a fully customizable imprint page that allows the user to create a reference to the questionnaire creator.

6 Summary

To sum up, the mobile application is able to meet the requirements mentioned in chapter 4.2 under consideration of the data management.
For example, a login and an administrative area are made possible by the, therefore, created `authentication-service`. However, until a downloaded questionnaire can be displayed there, the `online service` must first procure this, then it has to be transformed by the *GraphTransformation* in order to be stored by the `sql-storage-service`. For the final display on the start page, the has to be pinned to it. If one of these questionnaires is interrupted during execution, it belongs to the resumable questionnaires. These instances are created regularly during execution, so that the questionnaire can be resumed even if the application crashes. If, on the other hand, it ended correctly, the result can be found on the results page. From here they can be managed and organized. In addition, results can be marked with colors as a test and possible errors during the execution are pointed out in this view.
Furthermore, the results page also represents the output interface of the application, as a wide variety of export variants are available from here. With the synchronization to the server, sending them via e-mail or exporting them as `HTML`-page, the user has sufficient possibilities to handle collected data.
The complete handing over of the application to the interviewee, is no problem either, since the administrative area protects sensitive data and the personal imprint can be used to store the information of the distributor.
This is an ideal function to fulfill one of the already introduced use cases, because in the case of the doctor's office the device can remain in the waiting room and be operated by patients. The other patient data remains secure, due to the protected administrative

area and there is no need for an employee to intervene between the interviews. With the marking options, which are also synchronized to the server, the physician can record further steps. For example, the colour orange can indicate that a follow-up appointment will be arranged and green means that immediate healing procedures have been initiated. Therefore, the conclusion can be drawn that an application has been created to collect and process data in a very structured way. At the same time, however, it allows the customer to make a large number of adjustments so that numerous use cases can be covered. Although the focus is on editing questionnaires, any protocol or document that can be presented in graph structure is executable.

6.1 Outlook

This means that there is a very broad outlook on the future possibilities of the *Questionnaire* application. Due to the early stage of development, there is still potential for expansion within the application. For example, media in the form of pictures or videos could still be integrated. Due to the modular structure of the application, extensions of this type are easily possible. Immediate feedback in the form of direct evaluations after completing a questionnaire is also conceivable. This feature could then become interesting for medical purposes.
However, extensions to the field of use are possible. The basic principle of the application is to run through a process structure in the form of graphs and to store the knowledge gained in this process. Therefore, the possible field of use of the application will be further expanded, e. g. in education. In the future, entire examinations could be made with this application. The answers are then either compared directly on the device with previously defined solutions and corrected at the moment of delivery or collected centrally in a server where they are then forwarded for correction.
It could also be used as a companion system in a work environment with a structured workflow. In these cases, the application can be used to document different incidents and at the same time recommend the further procedure due to its non-linearity.
It can therefore be said that the application presented and investigated here is only the beginning of something that is practically unlimited in its use cases. And with a variety of advantages ranging from environmental protection to increased efficiency, the *QuestionSys* framework is very promising for the future.

Bibliography

[1] Garimella, S.V., Persoons, T., Weibel, J., Yeh, L.T.: Technological drivers in data centers and telecom systems: Multiscale thermal, electrical, and energy management. Applied Energy **107** (2013) 66 – 80

[2] Saris, W.E.: Design, Evaluation, and Analysis of Questionnaires for Survey Research. 2nd ed (online-ausg.) edn. J. Wiley & Sons, Hoboken, NJ (2014) Bibliotheks-Zentrale.

[3] Schobel, J., Pryss, R., Schickler, M., Reichert, M.: Towards Flexible Mobile Data Collection in Healthcare. In: Computer-Based Medical Systems (CBMS), 2016 IEEE 29th International Symposium on, IEEE (2016) 181–182

[4] Deeleman, P.: Learning Angular 2. Packt Publishing Ltd (2016)

[5] Jain, N., Bhansali, A., Mehta, D.: AngularJS: A modern MVC framework in JavaScript. International Journal of Global Research in Computer Science (UGC Approved Journal) **5** (2015) 17–23

[6] Fain, Y., Moiseev, A.: Angular 2 Development with TypeScript. (2016)

[7] Minar, I.: Igor Minar - Opening Keynote - Version 4 Announcement - NG-BE 2016. `https://www.youtube.com/watch?v=aJIMoLgqU_o` (2016) Accessed: 2017-09-20.

[8] Gartner: Mobile OS market share 2017. `https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/` (2017) Accessed: 2017-09-21.

[9] Yusuf, S., ed.: Ionic Framework By Example: Build amazing cross-platform mobile apps with Ionic, the HTML5 framework that makes modern mobile application development simple. Online-ausg. edn. Packt Publishing, Birmingham, UK (2016) Bibliotheks-Zentrale.

[10] Xanthopoulos, S., Xinogalos, S.: A Comparative Analysis of Cross-platform Development Approaches for Mobile Applications. In: Proceedings of the 6th Balkan Conference in Informatics. BCI '13, New York, NY, USA, ACM (2013) 213–220

[11] Trikha, B.: A Journey from floppy disk to cloud storage. International Journal on Computer Science and Engineering **2** (2010) 1449–1452

[12] Ionicframework: Docs. `https://ionicframework.com/docs/` (2017) Accessed: 2017-09-22.

[13] Arora, C., Hennessy, K.: Angular 2 By Example: Discover everything you need to know to build your own Angular 2 applications the hands-on way. Second edition (online-ausg.) edn. Packt Publishing, Birmingham, UK (2016) Bibliotheks-Zentrale.

[14] Angular: Authors style guide. `https://angular.io/guide/architecture` (2017) Accessed: 2017-09-22.

[15] Schobel, J., Schickler, M., Pryss, R., Reichert, M., Elbert, T., Ruf-Leuschner, M., Schauer, M., Brunnemann, N.: QuestionSys - A Generic and Flexible Questionnaire System Enabling Process-Driven Mobile Data Collection. `https://www.uni-ulm.de/in/iui-dbis/forschung/laufende-projekte/questionsys` (2017) Accessed: 2017-09-24.

[16] Schobel, J., Ruf-Leuschner, M., Pryss, R., Reichert, M., Schickler, M., Schauer, M., Weierstall, R., Isele, D., Nandi, C., Elbert, T.: A generic questionnaire framework supporting psychological studies with smartphone technologies. In: XIII Congress of European Society of Traumatic Stress Studies (ESTSS) Conference. (2013) 69–69

[17] Schobel, J., Pryss, R., Schickler, M., Ruf-Leuschner, M., Elbert, T., Reichert, M.: End-User Programming of Mobile Services: Empowering Domain Experts to Implement Mobile Data Collection Applications. In: Mobile Services (MS), 2016 IEEE International Conference on, IEEE (2016) 1–8

[18] Schobel, J., Pryss, R., Reichert, M.: Using Smart Mobile Devices for Collecting Structured Data in Clinical Trials: Results From a Large-Scale Case Study. In: 28th IEEE International Symposium on Computer-Based Medical Systems (CBMS 2015), IEEE Computer Society Press (2015) 13–18

[19] Community, G.S.: The Global Sharpers Survey. `http://www.shaperssurvey2017.org` (2017) Accessed: 2017-09-25.

[20] MD, S.S.: How Many Patients Should A Primary Care Physician Care For? `https://medcitynews.com/2014/02/many-patients-primary-care-physician-care` (2014) Accessed: 2017-09-24.

[21] Rappleye, E.: How long is the average wait at a physician's office? `http://www.beckershospitalreview.com/hospital-physician-relationships/how-long-is-the-average-wait-at-a-physician-s-office.html` (2015) Accessed: 2017-09-25.

[22] Google: Google forms - create and analyze surveys, for free. `https://www.google.com/forms/about/` (2018) Accessed: 2018-02-25.

[23] Google, Community: Google Docs Help Forum. `https://productforums.google.com/forum` (2017) Accessed: 2018-02-25.

[24] United States Government US Army: Systems Engineering Fundamentals. CreateSpace Independent Publishing Platform (2013)

[25] Unverzagt, A., Gips, C.: Impressum und Anbieterkennzeichnung. In: Handbuch PR-Recht. Springer Fachmedien Wiesbaden, Wiesbaden (2018) 257–270

[26] Chung, L., Nixon, B.A., Yu, E., Mylopoulos, J.: Non-Functional Requirements in Software Engineering (International Series in Software Engineering). Springer (2012)

[27] Mansilla, S.: Reactive programming with RxJS: untangle your asynchronous JavaScript code. Version: P1.0 (december 2015) (online-ausg.) edn. The Pragmatic Programmers, Place of publication not identified (2015)

[28] Reactivex: Reactive-extensions/rxjs. `https://github.com/Reactive-Extensions/RxJS` (2016) Accessed: 2018-01-17.

A

Source Code

This appendix lists some important source code.

```
login() {
    if (this._loginError) {
      this._loginError = false;
    }
    this.isLoading = true;
    this.onlineService.authenticate(this.username,
    this.password)
    .then((isAuthenticated) => {
      if (isAuthenticated) {
        this.authService.loginNewUser(this.username,
        this.password)
        .then(() => {
          this.isLoading = false;
          [...]
          this.navCtrl.setRoot(Home);
        }, () => {
          console.log("wrong credentials");
          this.isLoading = false;
        })
      }
    }).catch((error) => {
        this.isLoading = false;
        this._loginError = true;
    });
  }
```

Listing A.1: Two Phase Login

```
ionViewDidLoad() {
 this.sqlService.getPinnedQuestionnairesMetaList()
 .then((questionnaireList) => {
   this._availableQuestionnaires = questionnaireList
   .map((questionnaire)=>{
     questionnaire.collapsed = true;
     questionnaire.selectedLang =
      this.selectLanguage(questionnaire);
     return questionnaire;
   });
 });
}
```

Listing A.2: Home screen fetches pinned questionnaires

```
myPurchases() {
    return this.http
    .get(this.ROUTE_MY_PURCHASES,{}, this.getHeaders())
    .then((response:HTTPResponse)=>{
      if(response.status == 200){
        [...]
        for (let purchase of purchases) {
         let relatedProduct = products.find(
          (product) =>
           product.id === purchase.relationships.product.data.
              id
          );
          let metaObj = {
            purchaseId: purchase.id,
            purchasedAt: purchase.attributes.created_at.date,
            id: relatedProduct.id,
            contact: relatedProduct.attributes.contact,
            name: relatedProduct.attributes.name,
            description: relatedProduct.attributes.description,
            createdAt: relatedProduct.attributes.created_at.
               date
          }
          metaList.push(metaObj);
        }
        return Promise.resolve(metaList);
      }else{
        throw "An error occured";
      }
    }).catch(()=>{
      return Promise.resolve([]);
    })
}
```

Listing A.3: Fetching the List of Available Questionnaires

```
export class GraphTransformation {


  groupMap: any;

  transformGraph(nodes: any, links: any) {
    this.groupMap = null;
    return this._groupGraph(
      this._assignLinks(
        this._generateKeyValuePairs(nodes),
        links
      ),
      false
    );
  }

  _generateKeyValuePairs(nodes: any) {
    let map = {};
    let groupMap = {};
    let nodeTmp = deserialize<Node[]>(Node, nodes);
    nodeTmp.map((node) => {
      if (!isNullUndefinedOrEmpty(node.isGroup) &&
      node.isGroup == true) {
        groupMap[node.key] = node.executableComponent;
      } else {
        if(node.key == 1 || node.key == 2){
          groupMap[node.key] = node.executableComponent
        }
        map[node.key] = node;
        node.nextNode = [];
      }
    });
    for (let key in map) {
      if (map.hasOwnProperty(key)) {
        if (map[key].category == "Join") {
          map[key] = this._createDataNodeGateway(map[key]);
        }
      }
    }
    this.groupMap = groupMap;
    return map;
  }

  _assignLinks(map: any, links: any) {
    let startNode = null;
    for (let link of links) {
      if (map[link.from].category == "Start") {
        startNode = map[link.from];
      }
      map[link.from].nextNode.push(map[link.to]);
    }
    return startNode;
  }

  _groupGraph(graph: any, stopAtJoin: boolean) {
    let currentGroupId = graph.group;
    let currentNode: any = graph;
    //Functionality on End Node
    if (currentNode.category == "End") {
      [...]
      currentGroup.content
      .push(this._createContentNode(currentNode));
      return currentGroup;
    }
    let nextNode: any = graph.nextNode[0];
    //Functionality on Page
    if (currentNode.category == "Invisible") {
      [...]
      while (currentGroupId == currentNode.group) {
        currentGroup.content
        .push(this._createContentNode(currentNode));
        nextNode = currentNode.nextNode[0];
        currentNode = nextNode;
      }
      currentGroup.nextNode
      .push(this._groupGraph(currentNode, false));
      return currentGroup;
    }
    if (currentNode.category == "Join") {
      if (stopAtJoin == false) {
        currentNode.nextNode = [];
        currentNode.nextNode
        .push(this._groupGraph(nextNode, false));
      }
      return currentNode;
    }

    //Functionality on Page/Gateway
    if (currentNode.category == "Page" ||
      [...]
    }

    //Functionality on Start
    if (currentNode.category == "Start") {
      [...]
      currentGroup.content
      .push(this._createContentNode(currentNode));
      currentGroup.nextNode
      .push(this._groupGraph(currentNode.nextNode[0], false));
      return currentGroup;
    }

    //Functionality on PathLabel(skip PathLabel)
    if (currentNode.category == "PathLabel") {
      currentNode.nextNode = [];
      currentNode.nextNode
      .push(this._groupGraph(nextNode, false));
      return currentNode;
    }
    //Functionality on Split
    if (currentNode.category == "Split") {
      let splitPaths = currentNode.nextNode;
      let dataNodeGateway =
        this._createDataNodeGateway(currentNode);
      dataNodeGateway.nextNode
      .push(this._groupGraph(splitPaths[0], false));
      splitPaths.shift();
      for (let n of splitPaths) {
        dataNodeGateway.nextNode
          .push(this._groupGraph(n, true));
      }
      return dataNodeGateway;
    }
  }

  ...
}
```

Listing A.4: Graph Transformation

```
_synchronizeSelection() {
    let selectedItems = this._getSelectedItems();
    let resultIds = this._getItemIds(selectedItems.results);
    let questionnaireIds
    = this._getItemIds(selectedItems.questionnaires);
    this.sqlService
    .getDataToSynchronize(questionnaireIds, resultIds)
    .then((resultList)=>{
      for(let resultId of resultList){
        this.sqlService.getResultForUpload(resultId)
        .then((result)=>{
          this.onlineService.submitResult(result)
          .then((response)=>{
            this.sqlService.setResultSynchronized(resultId)
            .then((success)=>{
              if(success){
                this._results.find((result)=> result.id
                == resultId).synchronized = true;
              }
            }).catch((error)=>{
              console.log("SYNCHRONIZE ERROR",error)
            })
          }).catch((error)=>{
            console.log("SUBMIT ERROR", error);
          })
        });
      }
    })
  }
```

Listing A.5: Synchronizing selected questionnaires and results

```
 [
    { "id": 1, "color": "#f44336" },
    { "id": 2, "color": "#2196f3" },
    { "id": 3, "color": "#8bc34a" },
    { "id": 4, "color": "#ff9800" },
    { "id": 5, "color": "#673ab7" },
    { "id": 6, "color": "#607d8b" }
]
```

Listing A.6: Color source for marking

```
export class EngineInstances {


  constructor([...]) {
   ...
     this.resumableInstances =
      this.sqlService.subscribeToEngineInstancesForAdmin();
     ...
  }

  ionViewDidLoad(){
    this.sqlService.subscribeToEngineInstances()
    .subscribe((data)=>{
    });
  }

  _continueInstance(instance: any) {
    this.sqlService.getEngineInstancePayload(instance)
    .then(payload => {
      this.engine
      .loadInstance(this.navCtrl, payload, instance.instanceId)
         ;
      this.engine
      .start(
        {
          title: instance.title[payload._opt[0]],
          isTest:instance.isTest
        }
      );
    });
  }
}
export class SqlStorageService {

  subscribeToEngineInstances() {
    return this._engineInstances.asObservable();
  }

  subscribeToEngineInstancesForAdmin() {
    return this._engineInstancesForAdmin.asObservable();
  }

  _updateEngineInstanceObservers() {
    this._engineInstances.next(this._localInstances);
    this._engineInstancesForAdmin
    .next(this._localInstancesForAdmin);
  }
}
```

Listing A.7: Listing resumable instances

```
   _getFilteredList() {
    return this.sqlService
    .subscribeToEngineInstances()
    .take(1).repeatWhen(() =>
      Observable.timer(0, 2000))
      .map(instances => instances
        .filter(instance =>
        new Date().getTime() -
        instance.date.getTime() < this._timerInMilliseconds)
        .sort((a, b) =>
        b.date.getTime() - a.date.getTime())
      );
  }
```

Listing A.8: Filtering sessions of the last `x` minutes

List of Figures

List of Tables

Name: Philipp Jung Matrikelnummer: 860410

Erklärung

Ich erkläre, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den . . . .

Philipp Jung