Faculty of Engineering, Computer Science and Psychology
Institute of Databases and Information Systems

Assess Your Stress: Conceptual redesign of the Track Your Tinnitus system for measuring stress at the workplace

Master's thesis at the University of Ulm

Submitted by:
Bojan Klečina
bojan.klecina@uni-ulm.de

Reviewers:
Prof. Dr. Manfred Reichert
Dr. Rüdiger Pryss

Supervisor:
Dr. Rüdiger Pryss

2016

Version 2016-07-22

Satz: PDF-LaTeX 2_ε

Abstract

The *Track Your Tinnitus* (*TYT*) platform has been developed in a joint project by the universities of Ulm and Regensburg in Germany for several years. The framework was created to assist tinnitus patients in measuring and keeping track of their symptoms over extended periods of time. For this purpose, *TYT* provides a central, WWW-based platform to manage and distribute questionnaires to users, who fill in these questionnaires using their mobile devices multiple times per day when prompted by the application. In comparison to other non-computerized methods, this approach offers a more precise and reliable measurement of psychological phenomena and symptoms like tinnitus, which are generally difficult to estimate otherwise. However, the general principles behind *TYT* can also be applied to other use cases. After a new German law was passed, which aims to improve public health through a variety of measures in all areas of life, the stakeholders of the *TYT* project decided to initialize a project to apply *TYT* to measurement of stress at the workplace. Another purpose of this project was to completely redesign and rebuild the framework from scratch due to flaws in its design and the now outdated software it was built on. This new iteration of the *TYT* concept was named *Assess Your Stress* (*AYS*). The main goals were to apply the *TYT* concept to stress tracking, while also generalizing the platform to make it more open for extensions and different fields of application in the future. The project's main contributions are a detailed concept of the platform overhaul, a stable core system, upon which future extensions can be built, and a basic web-based client developed as a single-page application in *AngularJS*. While the system's application to stress tracking is prototypical in this release, it serves as a preliminary indication that the principles behind *TYT* are useful in the context of stress tracking.

Contents

1 Introduction **1**

1.1 Tracking stress at the workplace . . . 1

1.1.1 Motivation . . . 1

1.1.2 *Track Your Tinnitus* . . . 2

1.2 Generalizing *TYT* . . . 2

1.3 Outline . . . 4

1.3.1 Concept overview . . . 4

1.3.2 Mobile stress tracking . . . 5

1.3.3 Personalized feedback . . . 5

1.3.4 Extensibility . . . 5

1.4 Content overview . . . 6

2 Related work **7**

3 Requirements **11**

3.1 Use cases . . . 11

3.1.1 Guest . . . 14

3.1.2 Administrator . . . 14

3.1.3 User . . . 17

3.2 Functional requirements . . . 19

3.2.1 General . . . 20

3.2.2 Questionnaire . . . 20

3.2.3 Question . . . 22

3.2.4 User group . . . 22

3.2.5 User . . . 23

3.2.6 Results . . . 23

3.3 Non-functional requirements . . . 24

3.4 User interface requirements . . . 25

3.5 System requirements . . . 26

4 Architecture **29**

4.1 Overview . . . 29

4.1.1 General architecture . . . 29

4.1.2 Primary usage scenarios . . . 31

4.2 Static structure 32
4.2.1 Domain model 32
4.2.2 Server 35
4.2.3 Client 42
4.3 Dynamic structure 45
4.3.1 Register user 46
4.3.2 Create questionnaire 46
4.3.3 Edit questionnaire 47
4.3.4 Fill in questionnaire 47
4.3.5 Coping advice 48
4.4 User interface design 49
4.4.1 Dialog structure 49
4.4.2 Page layout 58
4.4.3 Visual design 65

5 Prototypical implementation 67
5.1 Utilized software and technology 67
5.1.1 Laravel 5 67
5.1.2 AngularJS 78
5.1.3 JWT 80
5.1.4 JSON API 81
5.2 Implementation details 83
5.2.1 Dynamic translation system 84
5.2.2 Version control 92
5.2.3 Model inheritance and user roles 93
5.3 Client 96
5.3.1 Initially hidden slider handle 96

6 Conclusion 99
6.1 Summary 99
6.1.1 Self-assessment 100
6.2 Future development 101

Bibliography 103

1 Introduction

1.1 Tracking stress at the workplace

1.1.1 Motivation

On July 25, 2015, the Preventive Health Care Act (*Präventionsgesetz*) took effect in Germany. This bill aims to improve public health through various initiatives for increased health awareness and preventive healthcare. The proposed measures are meant to encompass all integral areas and phases of life, such as infancy, education and work. One of these proposals is the implementation and scientific evaluation of preventive healthcare measures and financial investment in such programs as a joint effort of German insurance companies. Another goal of the bill is to encourage corporations to take measures for the improvement of employee health through active, preventive care. This has led to the inception of this project by the *Universität Regensburg* and *Ulm University*.

Beginning in 2013, the two universities have developed the *Track Your Tinnitus* system (*TYT*). This framework, which is described in more detail in the next section, is used by patients of tinnitus to measure and track their symptoms. The concept of this system can, however, also be adapted for other purposes. This fact and the passing of the Preventive Health Care Act was the impetus for the two institutes involved in the *TYT* project to develop a new, adapted iteration of *TYT* for the measurement and tracking of stress at the workplace. Stress has been researched from a medical standpoint and it is well-known that excessive and prolonged stress can be a serious health concern. Intense and chronic overwork has been linked to various health issues such as chronic fatigue, depression, cardiovascular disease and, in some extreme cases, it can even cause death.

Using a system similar to *TYT*, employees should be able to keep track of when and to what degree they feel stressed at work. First and foremost, this should help employees gain a more palpable sense of their workload and when and why it might be too much or too little. On the other hand, these data should also provide employers with an overview of their employees' well-being and help balance work tasks, detect signs of overwork among employees and offer assistance in taking appropriate countermeasures.

In addition to the adapted version of *TYT*, this project has another goal. *TYT* was implemented using the PHP framework *Laravel* version 3. Because *Laravel* has since changed dramatically, the parties involved in the long-term development of *TYT* have expressed the wish to not build on the current implementation of *TYT*, but instead to rebuild the framework entirely from scratch using the current version 5 of *Laravel*. Furthermore, since *TYT* was more or less customized for its one particular use case of tracking tinnitus symptoms, the new version of the system should aim for a more generalized and extensible design for forward-compatibility with future developments and extensions of the system.

1.1.2 *Track Your Tinnitus*

Beginning with [11], the universities of Ulm and Regensburg have developed the *Track Your Tinnitus* system. Due to its nature as a phenomenon that, in most cases, only the patients themselves can perceive, tinnitus is inherently difficult to measure and track accurately. *TYT* aims to alleviate this problem be providing a mobile framework for tinnitus sufferers that they can use to log the quality and intensity of their symptoms. The general usage of the system is as follows: A patient creates a user account. Next, they must complete a number of *statistical questionnaires*, which gather relevant personal and general information pertaining to the patient's illness. The mobile app will then begin issuing notifications at random intervals throughout the day, which can be adjusted by the user. On receiving a notification, the patient must fill in a questionnaire, logging the current state of their tinnitus. The data are gathered and can then be viewed as diagrams on the mobile app or the web portal. These logs provide an at least somewhat accurate overview of the patients' symptoms and can help the patients and their doctors identify factors that affect their well-being. With [11] as its foundation, *TYT* has since evolved into an ecosystem of various client implementations and server extensions (cf. chapter 2), as its general concept is suitable for various applications and use cases.

1.2 Generalizing *TYT*

While *TYT* offers a solid foundation for a mobile stress tracking application, there are several changes to the system and the concept itself that need to be implemented for this project.

TYT employs two kinds of questionnaires: *statistical questionnaires* and *app questionnaires*. Statistical questionnaires collect relevant personal and general information pertaining to the patient's illness. After creating a user account, users *must* complete these questionnaires once, and only once, before they can start tracking their symptoms. Statistical questionnaires can be filled out via both the web portal and the smartphone app and can be edited by administrators. The so-called app questionnaires can, as the name implies, only be filled out using the mobile application. Since the new version of the system aims to be more general, constraints related to questionnaires have been relaxed or dropped entirely. The system should support various types of questionnaires and newer versions of the system should be able to easily add new questionnaire types, ideally without having to modify the inner workings of the system. All questionnaires should be accessible from all platforms that the framework serves. Furthermore, for the use case of this project, statistical questionnaires must still be filled out once, but users can revisit them and edit their answers at any time. Lastly, all questionnaires can be edited and deleted and new questionnaires of all types can be created.

In the current version of *TYT*, there can only be one app questionnaire at a time and it is not editable. Moreover, its structure is hard-coded in a database table, which contains fields for each of its eight questions. Clearly, this solution is not conducive to extensibility and should therefore be replaced with a more flexible solution. As mentioned above, administrators should be able to create any number of questionnaires of any type, including app questionnaires, and edit these questionnaires as needed.

The aforementioned generalization of the questionnaire concept also has implications for the visualization of questionnaire results. In *TYT*, all results were displayed in diagrams, one for each question in the app questionnaire. However, when the number and type of questionnaires and questions is arbitrary, this solution is not optimal. A more general solution will have to employ a different approach. It should be able to group results of various questions into combined diagrams for a more terse overview of data. Additionally, edited questions also become a problem. Users must be able to differentiate between edited versions of the same questions in the diagrams in order to get an accurate representation of their results.

TYT currently includes a rather rudimentary solution for version control for questions. For statistical reasons, it is important that answers can always be linked to questions in the exact state they were in when a particular user answered those questions. Therefore, when a user answers a question, the system saves a copy of the question's text with the answer. This way, the original question texts are preserved and unaffected by changes to the question records. However, not only does this violate the *Single Source of Truth* principle [38], but the system currently does not even leverage this feature in the questionnaire editor or result views, i.e. any past question versions, which might exist inside the database, are hidden from both users and administrators. In the new iteration of the system, a proper solution for version control that can be applied to any type of entity in the system should be implemented.

Originally, *TYT* offered users the option to delete their e-mail address used for registration from the system and sign in anonymously. This requirement has been dropped from the new version of the system as it has proved impractical in the past.

TYT supports internationalization of questionnaires and questions. In the current implementation, questionnaires and questions each reference separate database tables that contain the data of their respective parent questionnaires or question that should be localized. This solution is not ideal in that the localization tables have to be adjusted manually if the questionnaire and question tables were to change. Furthermore, this solution is specific to questionnaires and questions in the system. If any other entity were to be localized, the same solution has to be implemented again for that particular case. The new iteration of this system aims to provide a more flexible and adaptable solution for internationalization, which can be used wherever necessary within the system with minimal extra implementation.

1.3 Outline

This section outlines the general aims of the system designed and implemented in this project.

1.3.1 Concept overview

Since this system is a direct successor of *Track Your Tinnitus*, the name for this system was chosen to appropriately reflect this connection: *Assess Your Stress*, or *AYS* for short.

This system, much like *TYT*, aims to provide a web-based solution for user-driven measurement and tracking using questionnaires. Unlike *TYT*, which is aimed toward patients of tinnitus, this system is an adaptation of the same concept applied to stress at the workplace. For this purpose, users complete questionnaires containing questions that are suitable to measure perceived stress levels. In its first iteration, the system offers a web-based interface, but in future developments, users should also be able to access and fill in questionnaires using mobile devices like smartphones and tablets, possibly even smart watches. After measuring their stress levels for a while, users can access diagrams that represent their stress levels as measured through the questionnaires. Users may also choose to export an overview of their results in a file format such as CSV or PDF. These results should then help employees using this system get a clearer image of their workload and mental well-being. If need be, the results can also be the basis for intervention and adjustment of workload after review by the employer. The questionnaires, which the system offers, are not "hard-coded" and uneditable, like it is the case with some similar systems described above.

Administrators can create and edit questionnaires using different question types. This particular system includes three types of questionnaires, each for different purposes: statistical, snapshot and burnout questionnaires. All questionnaires can be completed repeatedly and at any time. Statistical questionnaires collect general information on the user and are usually only completed once. Unlike the other two questionnaires types, users can edit their answers for statistical questionnaires even after they have been submitted. The so-called snapshot questionnaires are the system's implementation of the *Experience Sampling Method* [10]. They are filled out multiple times per day and create most of the data used for the analysis of users' stress levels. Burnout questionnaires are a third type that *TYT* does not offer. This category is for specialized questionnaires that psychologists use to gauge symptoms of burnout. The system might also actively encourage users to complete burnout questionnaires if it detects elevated levels of stress over extended periods of time.

Lastly, users are divided into user groups. User groups are also managed by administrators. All users belong to the *default group*, which is simply the superset of all users that the system manages. User groups and user roles are separate concepts. While the system uses user roles to grant access to system functionality to specific types of users, user groups are used for questionnaire access control. Like user roles, users can belong to any number of groups. Administrators assign both users and questionnaires to user groups. Users can only access questionnaires that are assigned to any of the groups they belong to. All other questionnaires, which exist in the system, are invisible to those users. This is useful, for example, for research groups and enables researchers to create closed groups for study participants to evaluate special questionnaires.

1.3.2 Mobile stress tracking

Like *TYT*, this system should provide a framework for mobile, web-based, user-driven measurement and tracking of stress at the workplace. Regular usage of the framework should provide both employees and employers with meaningful insights regarding their work environment. The system is also intended for use in as many types of professional environments as possible. Naturally, these professions vary greatly in regards to the physical workplace, available IT infrastructure, environmental conditions, working hours etc. In particular, a constant and stable Internet connection cannot be required of all clients, as certain professionals have to move a lot during their workday or work in environments without mobile connectivity. These constraints must be considered in the design of this system in order to be useful to as many people as possible. Ideally, the framework should provide interfaces for various types of clients, especially mobile, and be able to accommodate more and new types in the future.

1.3.3 Personalized feedback

In addition to the simple visualization of results implemented in *TYT*, the new iteration of the framework should provide personalized feedback to users regarding their stress levels. For example, upon detecting elevated levels of stress over a certain period of time, the system should offer valuable information on how to deal with stress at the workplace. It should help users detect the exact sources of their stress and offer common relaxation and coping methods. In the future, it might also be desirable to augment the system with more advanced features such as situational reasoning and analysis based on particularly stressful events logged in the system by the user. This would also enable dynamic adaptation features like automatic adjustment of notification frequencies and intelligent choice of questions tailored to the user's current situation, perhaps when the system detects that a recent event has significantly affected the user's stress levels.

1.3.4 Extensibility

Section 1.1.2 outlines the current implementation of *TYT* and points out specific issues with the system. A number of these issues pertain to a lack of extensibility. While the main goal of this project is still the design and implementation of an overhaul of *TYT* adapted for stress tracking, the shortcomings of *TYT* in regards to extensibility should be eliminated as much as possible to provide a more stable basis for future developments. This particularly concerns internationalization, questionnaires, questions, version control and client interfaces. The framework should be general enough to be forward-compatible with new types of questionnaires and questions that developers might choose to implement for their special use cases. These questionnaires and questions should be translatable into any number of languages with minimum effort. The translation mechanism itself should also be as general as possible so that it may be used for other entities within the system as well, in present or future iterations of the software. Lastly, the API for access to questionnaires, questions, answers, statistics and user administration should also be adaptable to different kinds of software and possibly even hardware clients in the future.

1.4 Content overview

The rest of this document discusses the design and implementation of the proposed system as follows:

Chapter 2 provides a general overview of previous work and research related to the subject matter of this project and highlights how this projects fits in with those past efforts. Following that, the requirements for this project are described in detail in chapter 3. Chapter 4 then defines the systems' architecture based on different views, such as the data model, use cases, system interactions and the user interface. The prototype of this system, which was implemented for this project, is presented and discussed in chapter 5. Finally, this document closes with chapter 6, which contains a summary of the project, discusses the gained insights and offers points of extension for future development.

2 Related work

This chapter discusses some previous related work, which influenced this project.

Stress and other psychological phenomena can and have been measured in a number of different ways. Some techniques, which are commonly used, rely on physiological effects that can indicate changes in a person's affect, such as body temperature, heart rate or electrodermal activity. Sensors record these effects and the data are later compiled, analyzed and correlated with one another to derive conclusions about the variables under investigation. Other techniques rely on journals. Patients or study participants must regularly update their journals with all data relevant to the phenomena that are being measured. In the case of stress, people often have to rely on their memory and recount stressful situations in their journals after the fact. However, both these techniques come with a host of problems. While physiological manifestations of psychological effects are objective and can be measurable and significant, they also tend to be imprecise, unreliable, and, naturally, only an indirect measurement of the psychological phenomenon at hand. With journals, patients and study participants are the direct source of data pertaining to the measured variables. While this approach can offer some valuable insight into internal psychological processes, which cannot be observed otherwise, the reliability of the recorded data naturally varies with each study participant. If the phenomenon cannot be recorded as it is happening, people have to rely on their memories, which further distorts the already fuzzy data. Additionally, the quality of the journal entries can vary and some people might occasionally forget to update their journals altogether. Moreover, while physiological measurements offer some degree of objectivity, journals are highly subjective and can only represent the perceived quality of some psychological process.

Today, with the spread of smart mobile devices, most importantly smartphones, and increased connectivity between devices, various research groups have attempted to leverage these new, technological capabilities to alleviate some of the shortcomings of the techniques described above. [26] uses interconnected, wireless sensors to record physiological effects of stress. [17] is a wireless biofeedback device used to signal high stress levels to its user.

The approach used in [37] is closely related to this project. A native Android application is used to measure stress in users. The application prompts users ten times per day to complete a questionnaire containing Likert scale questions about the users' current perceived stress level. This is an implementation of the Experience Sampling Method [10], which this project also uses. However, the main focus of the study was on investigating the validity of the questionnaire itself and less on the technological aspect. Also, since the application only contains one predetermined and uneditable questionnaire, it is far less flexible than the system developed in this project and cannot be easily adapted for other uses.

Another smartphone-based system that measures stress was developed in [16], although with an entirely different approach. This system uses smartphones' microphones to *crowd-sense* users' voices. A classification framework then analyzes the audio data to detect stress in the users' speech in real time. The study shows that using this technique, the system can detect elevated stress levels with an accuracy of up to 81%. While this approach is vastly different from this project, it aptly illustrated the novel possibilities that mobile devices open up for mobile health applications.

StudentLife [36] is a framework that uses smartphones and sensor fusion to measure stress and mental well-being in university students. The framework was evaluated with 48 students over a period of 10 weeks. The system uses both smartphone sensors and questionnaires to measure a number of variables like stress, mood and sleep cycles. The study data was examined for various effects and correlations. For example, it was shown that an increase in depression and stress levels in students with increasing workload at the end of the term can be found in the recorded sensor data and self-reports. Contrary to *StudentLife*, this project does not use smartphone sensors for data collection. Nonetheless, these results demonstrate that measurement of stress (and other phenomena) aided by smart devices can provide valuable insights into people's well-being and can be used to react appropriately and intervene if necessary. Another similar approach can be found in [3], which also employs data mining techniques to detect stressful events using people's digital schedules and other data sources.

[27] presents a research project that aims to improve the above-mentioned journal technique. Smartphones are used to provide users with context clues when trying to recall past events. Specifically, the system uses smartphones' GPS sensors to record the user's location at certain points in time. When users later try to recall specific situations, the system offers the recorded location data as a mnemonic aid. The study shows that this technique does improve recall accuracy in users.

A number of projects have been built on the *TYT* platform to further investigate its potential. In [24] and [25], mobile crowd-sensing applications were developed and integrated with the *TYT* platform for use in longitudinal studies. [28] uses *TYT* as a case study to identify common patterns and issues in the development of mobile business and e-health applications. A large-scale clinical study using mobile smart devices for data collection using questionnaires was conducted in [29]. Two works, [32] and [30], have taken the mobile data collection approach further by applying business process concepts to it. This was an effort to solve the issue of the immense workload of "paper-based" questionnaires in health and psychological studies. [31] explores the possibilities of mobile sensor networks in various applications like fitness and health. In [23], *TYT* is used as a tool in a study to investigate the connection between emotional states, tinnitus loudness, and tinnitus distress.

Finally, [4] presents a pervasive and unobtrusive way to detect stress in smartphone users. Instead of filling out questionnaires, users simply perform gestures commonly used for interaction with smartphones. The system can then deduce the user's stress level from the way they are performing these gestures. The study shows that this technique can distinguish stressed from relaxed users with decent accuracy using only *swipe* and *scroll* gestures and text input.

These examples demonstrate the potential of mobile crowd-sensing for the measurement of stress and other psychological phenomena. Most importantly, smart devices enable applications that are ubiquitous, independent of the users location and current activity and can be integrated into users' daily lives in a more unobtrusive way than past techniques. While a number of sensor-based approaches exist, the use of (mobile) questionnaires for the purpose of measuring stress seems to be not as thoroughly researched. This project's aim is to contribute an approach for this scenario while focusing on flexibility and extensibility, as opposed to some of the above-mentioned systems.

3 Requirements

This chapter outlines the requirements that the planned system must or should meet. Requirements are divided into four groups: the first two sections define the usual functional and non-functional requirements, which are part of most software projects. Additionally, this chapter also includes requirements for the user interface and a number of general requirements and constraints, which are called system requirements in this instance. Where appropriate, requirements are further divided and grouped into smaller categories of related requirements.

It should be noted that this list represents an abridged version of the original requirements defined for this project. Some low-priority requirements have been left out for the sake of brevity.

All requirements have a distinct descriptor and a unique identifier associated with them, which is used consistently throughout the rest of this document to refer to requirements.

A note on nomenclature: From this chapter on, both lower-case and upper-case versions of certain terms will be used. The lower-case words will be when the term at hand is discussed in a general sense, while the upper-case words signify entities as defined in this chapter and the chapter on the system's architecture, i.e. classes or objects. In cases where it may still be unclear, *italics* may be used for further clarification.

3.1 Use cases

Use cases describe the tasks that users want to complete using the target system. They therefore represent the user's perspective of the system, defining its functionality in terms of goals and outcomes. Also, use cases often include not only what the input and expected output for a given interaction should be, but also *how* users may want to interact with the system. Apart from perhaps small-scale, routine projects, software can generally not be built on use cases alone, as they only offer a general, high-level view of the desired functionality. For this reason, the subsequent section breaks the use cases down into low-level functional requirements.

The system offers different functionality for different types of users, also known as *user roles*. Therefore, the following use cases are grouped by user role, since each role has distinctly different tasks they need to complete with the system.

For the sake of brevity, the use cases are described in informal text. In cases where the user interaction is more involved, a somewhat formalized process description is also given.

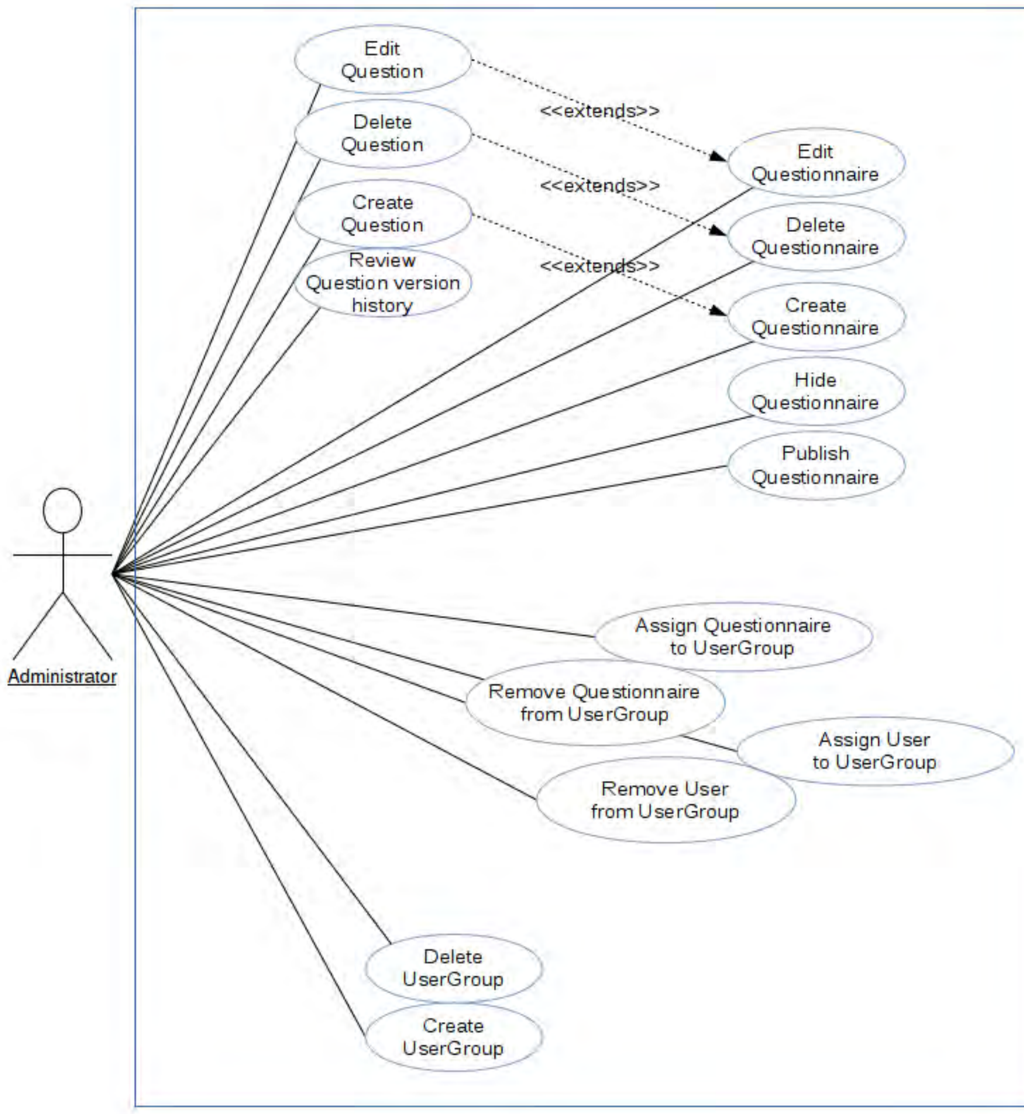


Figure 3.1: Use Cases for the Administrator role

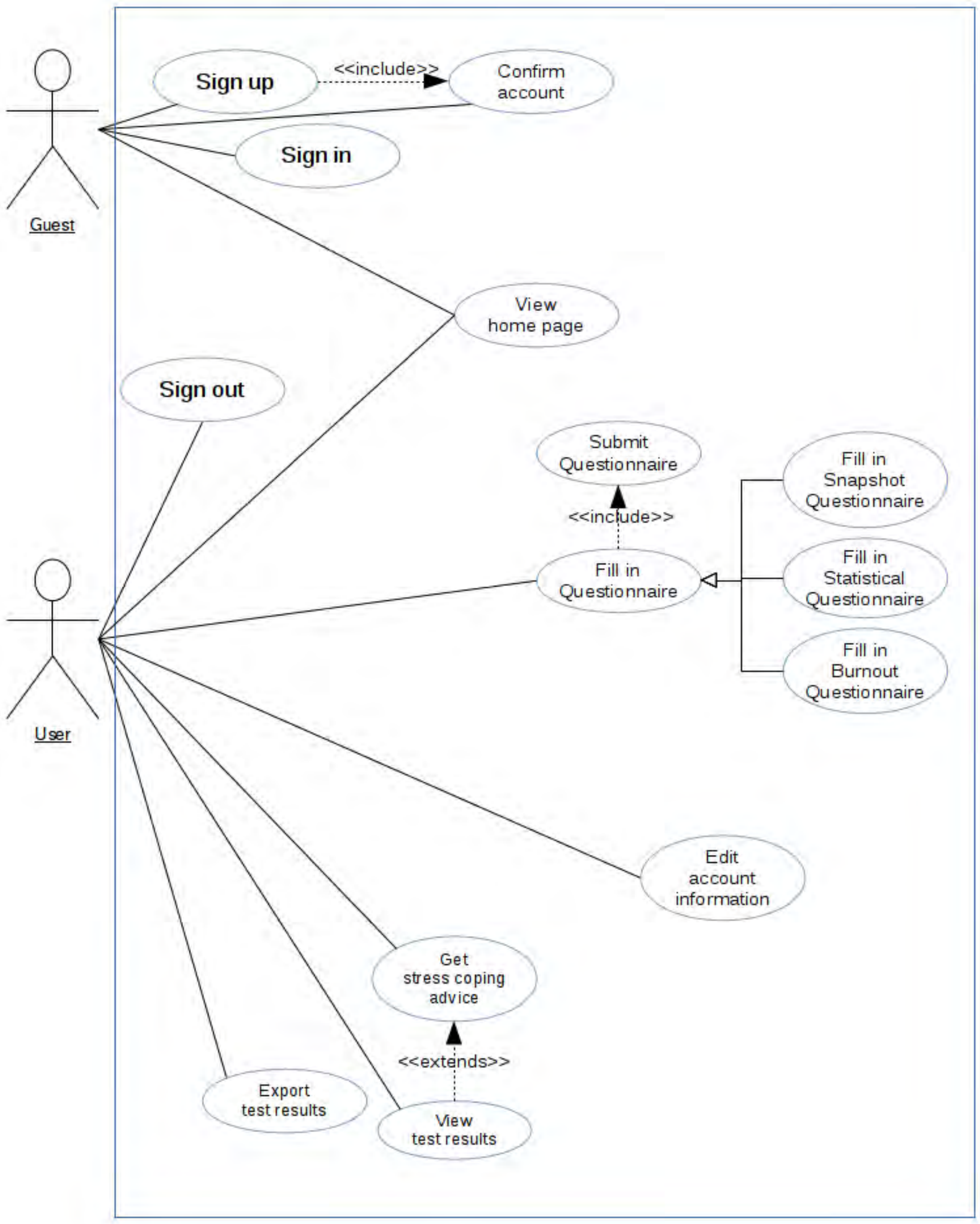


Figure 3.2: Use Cases for the Guest and User roles

The UML Use Case diagrams in figure 3.1 and 3.2 provide a graphical representation of the use cases that were identified for each user role in the planned system. These use cases are then further detailed in the subsequent sections.

3.1.1 Guest

The *guest* role is not an actual user type that the system recognizes or explicitly implements. It represents rather the *lack* of any defined role. Nonetheless, there are some functions that only non-registered users require from the system

UC001 — Sign up

Description and rationale To be able to offer any of its functions, the system must be able to distinguish users and keep records of their activities within the system. Therefore, all users must create a user account with the system before they can use it. In this iteration, the system should offer sign-up through a web-based interface.

Like most IT software that implements user authentication, this system offers a simple sign-up form, where users enter their e-mail address and password. If the entered data are valid, the system creates the user account and sends a confirmation link to the user's e-mail address. Once the user has confirmed that his e-mail address is correct by clicking the confirmation link, the system activates the newly created account and the user can begin using the system.

UC002 — Sign in

Description and rationale Once the user has created an account with the system, they must, naturally, be able to sign in to access their account and the system's functionality.

Users enter their credentials into a simple sign-in form, unless there is a active, valid session from a previous login. The system confirms the user's credentials and signs them into the system. The user can then begin using the system.

3.1.2 Administrator

One of the two actual user types supported by the system is the Administrator role. Administrators are users who manage the system "behind the scenes", i.e. they create questionnaires and questions, maintain user accounts and user groups etc. Administrators are not a superset of regular users, i.e. they do not have access to any of the functionality that users have.

UC101 — Create questionnaire

Description and rationale This system, unlike other similar systems, does not contain any "hard-coded" and immutable questionnaires, which users have to use to track their symptoms. Administrators have the ability to manage questionnaires in the framework.

Process

1. Administrator visits the *Create Questionnaire* view.
2. Administrator enters settings for the new Questionnaire (type, etc.).
3. Administrator creates Questions for Questionnaire.
4. *Optional* Administrator may assign Questionnaire to one or more User Groups. All Questionnaires are accessible to the Default Group by default.
5. *Optional* Administrator may set Questionnaire to be published immediately (see use case UC104, requirement S003).
6. Administrator submits Questionnaire.
7. *Invalid input* System informs Administrator of invalid input and Administrator can correct the input.
8. *Valid input* System creates Questionnaire.

UC102 — Edit questionnaire

Description and rationale The system enables administrators to edit all questionnaires in the system. The administrator accesses the questionnaire's edit form, which is very similar or identical to the form used to create new questionnaires. The form is pre-filled with the questionnaires current data, which the administrator then edits as needed. Once all changes are made, the administrator submits the updated questionnaire, the system validates the input and updates the questionnaire's database record. Since this system also supports version control for questionnaires, all question are submitted to version control if they are modified, meaning that all previous versions of questions remain in the system as distinct records and can be viewed by administrators at any time.

Note: The system does not currently support administrator permissions, i.e. all administrators can manipulate all questionnaires, even if they did not create them themselves.

UC103 — Delete questionnaire

Description and rationale Administrators may delete any questionnaire that exists within the system. However, the system uses *soft deletes*, i.e. deleted questionnaires are only effectively hidden from all users and remain in the database. This is important because users might lose their previous questionnaire records when questionnaires are deleted. To delete a questionnaire, administrators can either access the questionnaire's edit page and request its deletion or do so from the questionnaire overview. The system always prompts users to confirm destructive actions to avoid accidental data loss. After the administrator confirms the deletion, the system *soft-deletes* the questionnaire.

UC104 — Publish/hide questionnaire

Description and rationale Administrators may decide to create a questionnaire, but not make it accessible to users right away, possibly because it does not contain all questions yet or has to be reviewed by other staff. For this reason, all questionnaires are *hidden* by default after they are created. As noted in UC101, administrators may also have the system publish questionnaires immediately after they are created. See requirement S003 for more information.

If any of the conditions in S003 are not met, the system aborts the action and the questionnaire remains hidden. Once the questionnaire is published, it becomes visible to users in assigned user groups and they can fill it in. Administrators can also hide questionnaires at any time, making them immediately invisible to all users regardless of user group.

Like UC103, administrators can publish/hide questionnaire either from within the questionnaire's edit form or the questionnaire overview.

UC105 — View question version history

Description and rationale As described in UC102, this system submits all questions to version control when they are modified. To review any question's version history, administrators can access questionnaire that contains the question. The questionnaire's edit view contains lists of previous question version for each question, ordered in chronological order. Previous versions of questions cannot be modified.

UC106 — Create user group

Description and rationale Administrators also manage user groups. When creating a new user group, administrators only need to enter a unique name for the group. They may also choose to assign users and questionnaire from lists to assign to the group, but this can also be done later in the user group's edit view. If any users and questionnaires are selected, the system assigns them to the user group after creating it, making all of the selected questionnaires immediately available to the selected users.

UC107 — Edit user group

Description and rationale Administrators can edit all user groups in the system. They may change the group's name, which, however, still has to be unique. They may also assign and remove both users and questionnaires from the group and they may delete the group altogether.

UC108 — Delete user group

Description and rationale Like questionnaires, user groups can be deleted from the user group overview or their respective edit view. Since all users always belong to the Default Group, the users themselves are not deleted, only their association with the group. The same is true for questionnaires. However, since questionnaires must be assigned to at least one user group to be published (cf. requirement S003), questionnaires may be hidden by the system if they only belonged to the deleted group.

3.1.3 User

Apart from the Administrator role, this system also supports "regular" users. In the context of a "stress at the workplace" application, these regular users would be employees. However, to keep the concept general and open to extension in the future, the regular user role is simply called User in this system. It should therefore be noted that the User role is not equal to the superset of all users in the system including Administrators. The User may complete questionnaires created by Administrators and view their results and statistics.

UC201 — Reset password

Description and rationale Strictly speaking, this function may be assigned to guests, since a password reset request is only useful if the user does not remember their password and is therefore not currently signed it. However, an account must first exist in the system before its password can be reset. While not signed into the system, the user can simply request a new password from the sign-in form, as it is known from countless web-based authentication systems.

UC202 — Sign out

Description and rationale This is the logical counterpart to the sign-in function. Users may choose to end their session with the system at any time. They lose access to the system's protected functionality and must therefore first sign in again to gain access.

UC203 — Edit account

Description and rationale Users may edit their account information, which at this point only contains the user's e-mail address and password. The system currently does not support deletion of user accounts through the user interface.

UC204 — Fill in questionnaire

Description and rationale This is the main functionality of the system from the perspective of the User role. Users may access the system to fill in any questionnaire at any time. However, some constraints apply. These are described below. Users choose a questionnaire from a list of all available questionnaires and then proceed to fill it in. The system then records the user's input for later analysis.

Process

1. User accesses the list of all available Questionnaires.
2. User selects a Questionnaire from the list.
3. *Snapshot Questionnaire*
   a) *User has not yet completed all available Statistical Questionnaires*: System informs User that all available Statistical Questionnaires must be completed before Snapshot Questionnaires can be filled in, and returns to the Questionnaire list.
   b) *All available Statistical Questionnaires completed*
      i. User answers Questions.
      ii. System validates Answers and saves them if valid.
4. *Statistical Questionnaire*
   a) *User has completed Questionnaire before*
      i. System displays Questionnaire pre-filled with User's previous Answers.
      ii. User edits Answers and submits Questionnaire.
      iii. System validates Answers and updates them if valid.
   b) *User has not completed Questionnaire yet*
      i. System displays new Questionnaire with empty fields.
      ii. User enters Answers and submits Questionnaire.
      iii. System validates Answers and saves them if valid.
5. *Burnout Questionnaire*

a) System displays new Burnout Questionnaire with empty fields.

b) User enters Answers and submits Questionnaire.

c) System validates Answers and saves them if valid.

UC205 — Review/export results

Description and rationale After completing several questionnaires, users will want feedback from the system. They may review their answers from within their user account. In the Results section, the system generates diagrams for each questionnaire and each question that the user has completed. The type of diagram depends on the question type. For some questions, line diagrams might be more appropriate, while other question types might be more suited to pie charts or dot plots. The diagrams should be offer some degree of control for usability. For example, users should be able to zoom into sections of the diagrams and hide or show lines and labels. From the same view, users may export an overview of their raw result data in the CSV file format, possibly also as PDF.

UC206 — Request coping advice

Description and rationale When users notice that there stress levels are currently higher than usual, they may want to employ countermeasures. This system aims to assist in that by providing advice on how to cope with stress. While this first iteration of the system does not support any dynamic and intelligent analysis of user results to provide highly personalized advice, it does consider the user's current stress levels when choosing which coping strategies to offer. Users may also choose to rate the advice provided by the system after trying it out. If users rate it as useful, the system will be more likely to suggest similar strategies in the future. If it did not manage to reduce their stress with a specific strategy, the system will more likely to choose different strategies when requested.

3.2 Functional requirements

The following requirements define the target system in terms of functionality. Specifically, these requirements describe the functions that the system must offer in order to fulfill the use cases introduced in section 3.1. While the use cases characterize the system from the user's viewpoint, the functional requirements offer a more fine-grained representation of the functions that have to be implemented for users to be able to complete their tasks with the system.

As with the use cases, functional requirements are described in prose. Where necessary, more formalized definitions are also given.

Furthermore, simple functions that do not warrant entire sections for themselves may be either omitted or grouped together into one section for the sake of brevity. This is usually the case when the description in a related use case mostly matches the actual functionality, e.g. simple CRUD methods. The unique identifiers and descriptors are still listed, but detailed descriptions are left out.

3.2.1 General

F001 — Version control

Description and rationale As mentioned in the introducing chapters, one of the desired features of this system is full version control. While the predecessor *TYT* did implement version control, it was rather rudimentary and limited to specific entities in the system (cf. section 1.2). This feature is mostly necessitated by statistical integrity. In order to maintain statistical integrity when reviewing test results, the question must be preserved in the exact state it was in when it was answered by a user. This way, questions can be edited and deleted without invalidating users' statistical records. This is also the reason why the system uses *soft deletes* for records like questions and questionnaires.

This system's approach to version control is exemplified using questions, since this is the use case that was originally meant for. However, the same process can also be applied to other entities in principle.

Process

1. Question Q from Questionnaire QS is modified.
2. System creates copy Q' of Q and applies changes to Q.
3. System submits Q' to Version Control, creating Version Control Record V. V is linked to the "root" version Q and the modified version Q'.
4. If necessary, System submits any related records to Version Control as well.
5. QS is still referencing Q, now modified, and has no direct access to Q'.

3.2.2 Questionnaire

F002 — Questionnaire types

The system currently supports the following questionnaire types:

Statistical questionnaires Statistical questionnaires collect general information about the user that is relevant to the use case of the system, in this case stress at the workplace. They are different from other questionnaire types in that users can edit their answer after submitting the questionnaire.

Snapshot questionnaires Snapshot questionnaires correspond to *app questionnaires* from *TYT*. These questionnaires generally contain few questions and users fill them in several times a day to record a "snapshot" of their current mental state, in this case stress levels.

Burnout questionnaires Burnout questionnaire are a new type of questionnaire that was not supported by *TYT*. These specialized questionnaires are used by psychologists to gauge symptoms of burnout. The system may detect high stress levels over longer periods of times and subsequently encourage users to complete a burnout questionnaire, which may then offer further insights into whether the user might be experiencing burnout, which should prompt intervention by the employer.

F003 — Create questionnaire

Description and rationale When an administrator creates a new questionnaire, the system creates a new record for the questionnaire and simultaneously one record for each created question, if all input is valid. As described in use case UC101, administrators may choose to publish the new questionnaire immediately and they may assign it to user groups. If the questionnaire should be published but does not fulfill all conditions (cf. use case UC104), it will be saved but remain hidden. The administrator will be notified that it the questionnaire could not be published. If user groups are selected, the questionnaire will be assigned to them and the user groups' users will have immediate access to the questionnaire, if it has been published.

F004 — Edit questionnaire

Description and rationale When a questionnaire is edited, the system must make sure that all questions and other entities that are enabled for version control are properly submitted to version control. Furthermore, if a statistical questionnaire is edited and it has been filled out by users before, these users must be notified of the changes in the questionnaire so that they may edit their answers if necessary.

F005 — Delete questionnaire

Description and rationale When questionnaires are deleted, the system does not actually remove their record from the database, but effectively only hides them from all access by users.

As described in use case U103, this *soft delete* mechanism is used to maintain statistical integrity of users' results.

F006 — Publish questionnaire

Description and rationale A questionnaire can only be published, when it meets the conditions listed in requirement S003.

3.2.3 Question

F007 — Question types

The system currently offers seven question types, which were directly adapter from *TYT*. They differ in their representation in the user interface and the types of values they record:

1. Single choice
2. Multiple choice
3. Polar (yes/positive or no/negative)
4. Scalar (numeric scale)
5. Date
6. Single-line text
7. Multi-line text

Manage questions

As mentioned in the references sections on managing questionnaires, questions are submitted to version control when they are edited or deleted.

F008 — Create question cf. use case UC101

F009 — Edit question cf. use case UC102

F010 — Delete question cf. use case UC103

F011 — Show version history cf. use case UC102

3.2.4 User group

Manage user groups

F012 — Create user group cf. use case UC106

F013 — Delete user group cf. use case UC108

F014 — Assign user to user group cf. use cases UC106, UC107

F015 — Assign questionnaire to user group cf. use cases UC106, UC107

F016 — Remove user from user group cf. use cases UC106, UC107

F017 — Remove questionnaire from user group cf. use cases UC106, UC107

3.2.5 User

F018 — Create user account

Description and rationale When a new user enters their credentials and the system successfully validates it, the system creates a new user account. At this point, the account is not activated yet and cannot be used. The system also creates a confirmation token that is sent to the user's e-mail address. Once the user visits the link in the e-mail, the system verifies the token and activates the user's account (cf. requirement F019). The user can then begin using the system.

F019 — Confirm user account

Description and rationale As described in requirement F016, new user accounts are inactive when first created. The system creates a confirmation token, which is sent to the user. When the user visits the link containing the confirmation token, the system verifies that the token is valid. The token is a hashed value of the user's e-mail address, a *salt* value and a *Time To Live*. If the token cannot be decoded or it is too old, the system rejects it and the user account remains locked. If the token is valid, the system activates the account.

Other user functions

F020 — Update user account cf. use case UC203

F021 — Create confirmation token cf. requirement F017

F022 — Sign in user cf. use case UC002

F023 — Sign out user cf. use case UC202

3.2.6 Results

F024 — Save answers

Description and rationale Answers to questionnaires submitted by users are stored in answer records and linked to a *answer set* record. If answer records are the counterpart to questions, answer sets are the counterpart to questionnaires, i.e. they contain meta data related to the questionnaire or the answers without containing the actual answers themselves.

F025 — Render diagrams

Description and rationale Diagrams are the primary feedback mechanism for users of this system. This function collects all of a user's answers, processes them and creates diagram representations of these values to be displayed in the user's Results view.

F026 — Export results

Description and rationale Similarly to requirement F023, this function accumulates all of a user's answers and summarizes them in a format that can be exported and downloaded to the user's device. It is required to export to CSV and should also be able to export to PDF.

F027 — Retrieve coping advice

Description and rationale Users should be able to get advice from the system on how to cope with high levels of stress. As described in use case UC206, the system should contain prose descriptions of coping strategies, which it selects based on the user's current stress levels and previous user ratings. Higher rated strategies and similar strategies are more likely to be selected than lower rated strategies.

3.3 Non-functional requirements

Software not only has to conform to functional requirements, but also non-functional requirements. While these requirements are generally "softer" than functional requirements, this does not mean they are any less important, as they are often related to system properties like response time. This section outlines some non-functional requirements that were identified for this system.

N001 — Multiple language support

Description and rationale The system should support multiple languages. While modern programming frameworks commonly ship with useful localization functionality for static data, developers generally have to implement custom solutions for dynamically entered data. In this case, this is mostly relevant for questionnaires and questions, which must be easily translatable into any number of languages. The predecessor system *TYT* does contain implement a localization mechanism, but it is rather rigid and inflexible (cf. section 1.2). The new iteration of the

TYT framework should provide a localization mechanism that is simple, applicable to any entity in the system that should be dynamically translated and most importantly, it should be able to handle any number of languages in the future.

N002 — Server-client architecture

Description and rationale Like *TYT*, this system should be implemented as a client-server architecture. This is a very common pattern for modern web-based applications, as it is inherently distributed and affords interchangeable client implementations and extensibility. Traditionally, the server contains the part of the system that handles the *heavy lifting*, i.e. business logic, data persistence, session management etc. The client-side implementation should ideally only request data from the server that it then displays to the user while responding to user input. Since one of the principal goals of this system is to be flexible and extensible, this pattern remains the prime choice, as future developments will easily be able to add new client implementations etc.

N003 — Extensibility

Description and rationale As mentioned several times throughout the introduction and requirement, extensibility is an important goal in this project. This specifically pertains to the implementation of questionnaires, questions, version control, language support and clients. While this systems offers a number of different questionnaire and question types, developers should be able to implement additional types with minimal modifications to the core code of the system. This enables the system to be adapted to other use cases than just measurement of stress levels. This also means that the questionnaire and question concepts used in this system should not be specific to stress. F001 and N001 already describe the requirements for the version control and localization subsystems. Lastly, to enable easy addition of new client implementations, the server should offer a generic public API that clients then use to communicate with the server and make use of its services.

3.4 User interface requirements

This section outlines requirements for the user interface of this system. Note that all of these requirements currently only apply to a web-based interface.

UI001 — Slider

Description and rationale This requirement is directly adopted from *TYT* [11]. One of the question types supported by this system uses a slider as input for scalar values. However, since it has been shown that the initial position of the slider's handle can influence users' answers due to a phenomenon called *anchoring* [13], the slider should not display any handle before the first interaction. Once the user interacts with the slider widget, the handle is displayed and the slider's value can be manipulated.

UI002 — Usability

Description and rationale Good usability is important for any IT system. However, since this system is an application that measures psychological properties of its users, more specifically stress levels, and is meant for quick interactions throughout users' daily lives, special care must be taken in designing the user interface. Interactions should be as fast and unobtrusive as possible and, since every user of technical devices knows how frustrating badly designed interfaces can be, the application should definitely not influence users' stress levels, for better or worse.

UI003 — Color blindness

Description and rationale Since this system should be employed in various types of professional environments, color blindness must be considered in the choice of colors for the user interface. Color blindness is a common condition and can be an significant impediment for sufferers when trying to use interface that heavily rely on color. Therefore, a goal of this system should be to limit the use of color to only a few cases, where it is useful and meaningful. Important information should generally never be conveyed through color alone; color should always be accompanied by a secondary information channel such as text or symbols. Additionally, as mentioned in requirement UI002, the interface design for this system should aim not to stress users. Therefore, jarringly bright colors and extreme contrasts, which can strain the eyes, should be avoided.

3.5 System requirements

Finally, this section describes a few system requirements, which are general requirements or constraints that apply to the system as a whole and did not fit other categories.

S001 — Statistical questionnaires first

Description and rationale As noted multiple times before in other requirement definitions, users must first complete all available statistical questionnaires before they may fill in snapshot questionnaires. This requirement was adopted from *TYT* and is supposed to ensure that users do complete statistical questionnaires instead of only using snapshot questionnaires, since statistical questionnaires are important to the overall statistics.

S002 — Incomplete answer sets

Description and rationale Not all questions need to be answered when a questionnaire is submitted. Since users might not want to, be able to or have to time to answer all questions in a questionnaire at a given time, the system should respect this. With that being said, it is also required that the systems always saves complete answer sets for all questionnaires. If any questions are left out, the system should save these answers as null values.

S003 — Publishing questionnaires

Description and rationale Questionnaires must meet the following conditions in order to be publishable:

1. All of the questionnaire's mandatory data are valid (title, etc.).
2. The questionnaire contains at least one valid question.
3. The questionnaire is assigned to at least one user group.

S004 — Notification on questionnaire edit

Description and rationale When an administrator modifies a statistical questionnaire the system should notify any users of these changes, that have completed the questionnaire before. Like the first constraint in this section, this is to ensure that users' records accurately reflect the users and any changes are recorded.

S005 — Multi-language questionnaires

Description and rationale The system should only display questionnaires that are available in the user's current system language.

S006 — Optional questions

Description and rationale Questionnaires may contain optional and non-optional questions. Since questionnaires can be saved in an incomplete state, as mentioned above, this is most useful for statistical questionnaires to determine their "degree of completion". A statistical questionnaire is considered completed once all mandatory, i.e. non-optional questions have been answered.

4 Architecture

In this chapter, the architectural blueprint for this system is described from a number of different perspectives. The first section provides an overview of the planned system's general structure, including the main components and short descriptions of the key usage scenarios. The next chapter then defines the static structure, i.e. the classes and interfaces of which the framework consists, for both the server and client implementation. Section 4.3 complements the static system structure defined in 4.2 with a dynamic view of the system by describing the crucial user-system interactions and illustrating the exchange of information between the software and its users. Lastly, the general concept for the system's web-based user interface (*UI*) is presented in section 4.4.

Note that some classes and other structures may not be represented in the final implementation exactly as described here. The designs in this chapter are still part of the concept, which is independent of the platform used to implement the system. It is therefore possible that some classes are merged with others in the implementation or their functionality is provided by the web or client framework.

4.1 Overview

4.1.1 General architecture

The architecture of this framework is based on the *client-server model*, as required in N002. This software design pattern is commonly used for distributed systems, especially information systems, and it is the fundamental architectural pattern for Internet-enabled computer services. In this model, the *server* is the central *provider* of services and data, which it serves to *clients* upon their requests. This model is especially suitable for use in the Internet, as it is designed for the purpose of *decoupling*. This is crucial in very dynamic and "loose" networks, like the Internet, with enormous numbers of different computing devices communicating with each other asynchronously. More specifically, the framework designed in this project consists of a *fat server* and *thin clients*. In this variant of the client-server model, the server handles as much of the "heavy lifting" as possible, i.e. all business logic, data transformation, data persistence, user authentication and authorization etc., whereas the clients are kept as simple and ignorant of the inner workings of the server as possible. Ideally, thin clients should be strictly limited to requesting data from the server, displaying it to the user and sending user input back to the server. This

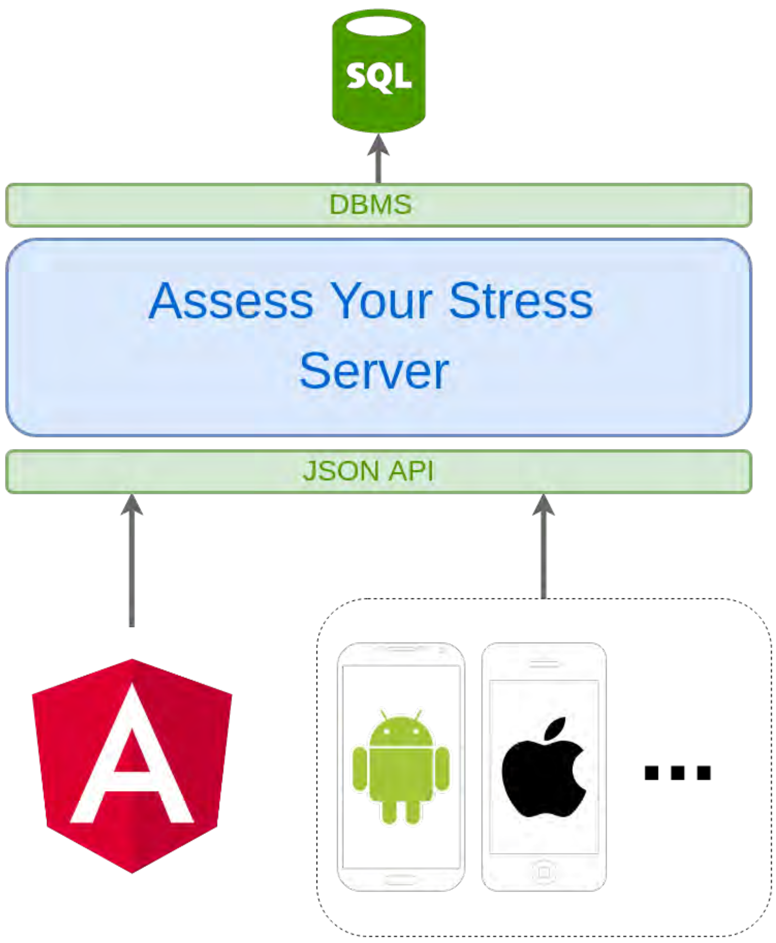


Figure 4.1: The general client-server system architecture; logos copyright of Google and Apple Inc.

essentially makes the server a "black box" from the viewpoint of the client by hiding implementation details. Furthermore, thin client implementations are more easily interchangeable, which is one of the design goals for this framework (cf. requirement N003).

Figure 4.1 illustrates the basic, high-level design of the framework. In the most general terms, the server's responsibilities are:

1. Handling data storage (questionnaires, questions, user data, statistics)
2. Handling user authentication and authorization (distinguishing regular users and administrators)
3. Providing administration interfaces for questionnaires, questions, user groups and users
4. Providing questionnaires for users
5. Collecting, analyzing and storing answers to questionnaires
6. Calculating and storing questionnaire results for each user
7. Providing questionnaire results for each user (diagrams, downloadable overview files)

All of the services and data described above are offered through a generic interface based on the *JSON API* specification. More information on this specification is provided in section 5.1.4.

As can be seen from 4.1, the web-based front end for this system is not part of the core server implementation. The web front end is designed as a separate client module, which communicates with the server using the server's generic API, like mobile clients do in the original implementation of *TYT*. The client prototype implemented in this project makes use of the *AngularJS* framework, version 2. As described above, the client implementation in this project is a *thin client*, i.e. its only responsibilities are accessing the services provided by the server, displaying data retrieved from the server to the user and sending user input back to the server. In the context of this project, this basically means that the client provides the UI for authentication, displays questionnaires, sends users' answers back to the server and displays the users' results on request.

The approach of separating the primary UI from the server has two main advantages: The original *TYT* implementation provides both a traditional website and a JSON API for mobile clients. This means that the same or very similar interfaces for data access have to be implemented twice, one "native", internal version for the website and one generic version as the API. Implementing the website as a separate application eliminates the need for internal interfaces because all clients now communicate with the server uniformly through its JSON API, as the main application is then essentially a web service. Secondly, this also has the effect of avoiding inconsistencies between the two sets of interfaces, meaning that all clients, both present and future implementations, all access the server in the same fashion and receive the same data on identical requests. This facilitates the implementation of future clients, since all clients use the same interface, as indicated in figure 4.1. Additionally, the system's maintainability is also improved because there is only a single point of entry and exit to/from the server and changes to the interface impact all clients in the same, predictable manner.

4.1.2 Primary usage scenarios

The primary usage scenario of this system can be described as follows:

A new user visits the system's website. Before they can make use of the frameworks services, they must sign up. Once they have entered valid credentials, they must confirm their e-mail address by clicking the confirmation link sent to their e-mail address upon registration. When this is done, the user account is active and the new user can start using the system. Before they can track their stress levels, however, they must first fill in all statistical questionnaires that are available. At that point, they can finally begin filling in *snapshot* questionnaires to record their daily stress levels. As soon as some data is available in the system, the user may view their results in the form of different diagrams for each questionnaire. If necessary, the user may also choose to export their results to a CSV or PDF file and download it to their device. This can be useful when discussing stress levels with employers.

Meanwhile, administrators are tasked with system maintenance. In particular, this means they can use the web interface to create, edit or delete questionnaires, add or remove them to/from user groups and add or remove users to/from user groups. While not planned for this particular iteration of the framework, they might also have the ability to view anonymous usage statistics to gain insights into how their questionnaires perform. Furthermore, the system will ultimately also provide a way for employers to retrieve statistics of their employees to get a general overview of stress levels within their company. Since privacy is a top priority in health-related applications, such statistics can only be anonymous. But since employers should also play an active role in the improvement of employee health, as mentioned in section 1.3.2, the system should at least help them gain some sense of their employee's stress levels without forcing employees to confront their employees about this topic.

4.2 Static structure

The static structure of an IT system describes the data objects that the system manipulates and stores in order to provide its functionality. This section presents the static structure of the planned framework in the form of an entity-relationship diagram (*ER diagram*), UML class diagrams and textual interface definitions. The ER diagram is part of the system's high-level *domain model*. Based on that, the low-level model class diagram and interface specifications are created through further refinement.

Since the server and the web client are to be implemented as separate applications, their static structures are each described in their own subsections. However, because the domain model is independent from how the implementation is split up between server and client, it is presented separately.

4.2.1 Domain model

Figure 4.2 shows the ER diagram for this system, i.e. an abstract, high-level model of the system in the context of the *problem domain*. As such, it does not define the system in its final, implemented form, but it is rather an approximation of users' mental model of the system. The low-level specifications in the following sections are based on this domain model.

Entities

The domain model defines the *entities* for this system and how they are interrelated. Entities represent distinct data objects that correspond to concrete entities or concepts in the problem domain, which are related to the system's functionality in one way or another. Some typical examples are users, products, comments, questionnaires, etc. These entities were extracted from the requirements outlined in chapter 3. The following entities are defined for this system:

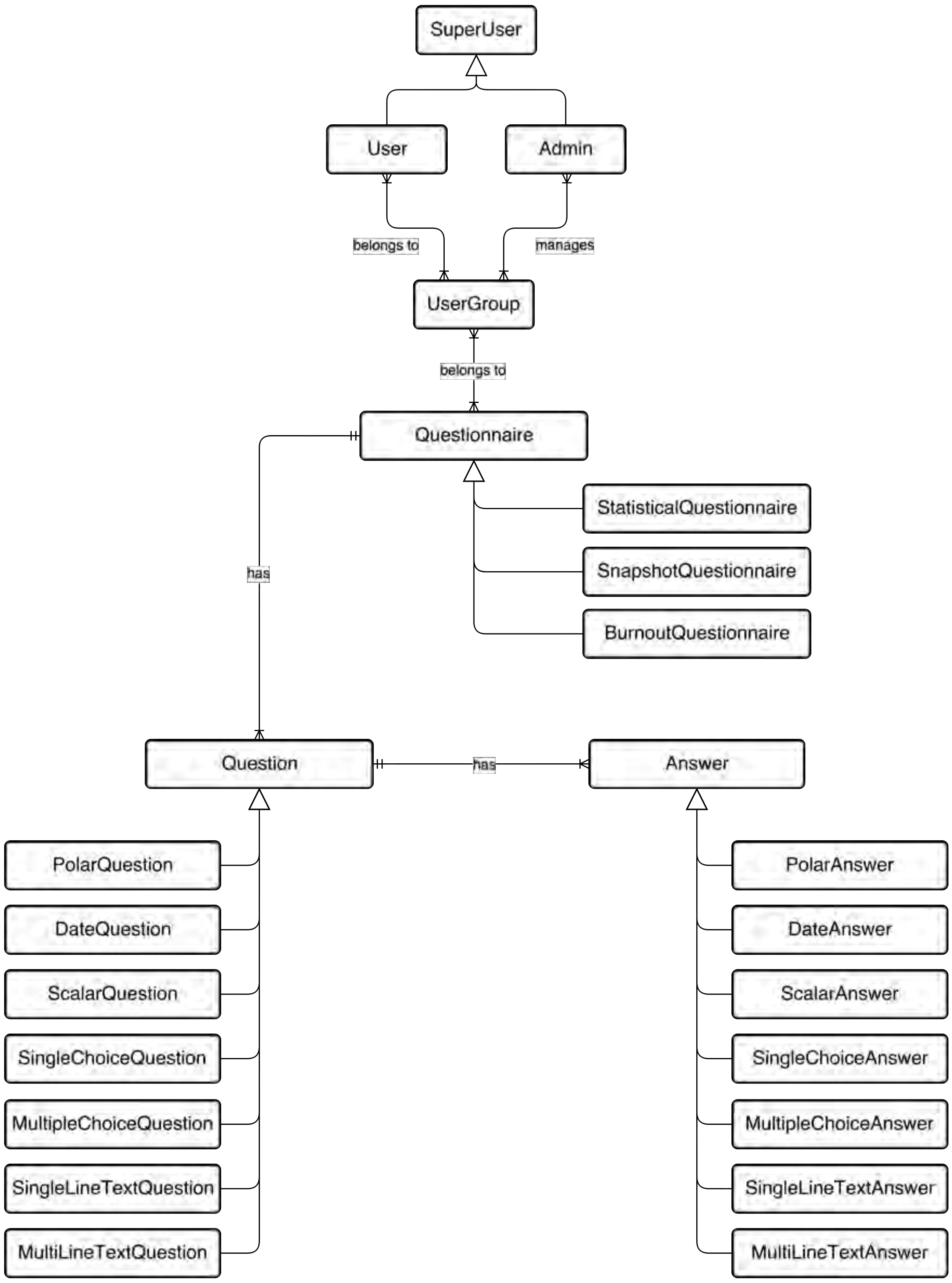


Figure 4.2: The *domain model* as an entity-relationship diagram

SuperUser The superset of all users of the system

User The *User* role, i.e. regular users (cf. section 3.1). *Users* can sign up, complete questionnaires and view their results and statistics. In the application context of this project, *Users* correspond to employees in companies.

Admin The Administrator role (cf. section 3.1). Administrators are tasked with system maintenance, i.e. management of questionnaires, questions and user groups.

UserGroup *Users* are grouped into UserGroups. UserGroups are primarily used as group-based access control for questionnaires, which are also assigned to UserGroups.

Questionnaire Generic questionnaire type (see requirement F002 for details)

StatisticalQuestionnaire Questionnaire type derived from the generic Questionnaire.

SnapshotQuestionnaire Questionnaire type derived from the generic Questionnaire.

BurnoutQuestionnaire Questionnaire type derived from the generic Questionnaire.

Question Generic question type (see requirement F007 for details)

PolarQuestion Question type derived from the generic Question. Also known as *yes–no question*; offers one generally positive (often "yes", "agree") and one generally negative (often "no", "disagree") answer, from which exactly one must be chosen.

DateQuestion Question type derived from the generic Question; accepts date and time as input from the user (e.g. *User*'s last occurrence of insomnia).

ScalarQuestion Question type derived from the generic Question; accepts a numeric value on a fixed scale as input from the user (e.g. number of days on which *User* felt fatigued in the last week)

SingleChoiceQuestion Question type derived from the generic Question; offers several answer options from which exactly one must be chosen.

MultipleChoiceQuestion Question type derived from the generic Question; offers several answer options from which one or more must be chosen.

SingleLineTextQuestion Question type derived from the generic Question; accepts a single line of text as input from the user.

MultiLineTextQuestion Question type derived from the generic Question; accepts multiple lines of text as input from the user.

Answer Generic answer type (see requirement F002 for details), meaning answers given by *Users*, not answer options contained in the Questions.

PolarAnswer Answer type derived from the generic Answer; corresponds to PolarQuestion.

DateAnswer Answer type derived from the generic Answer; corresponds to DateQuestion.

ScalarAnswer Answer type derived from the generic Answer; corresponds to ScalarQuestion.

SingleChoiceAnswer Answer type derived from the generic Answer; corresponds to SingleChoiceQuestion.

MultipleChoiceAnswer Answer type derived from the generic Answer; corresponds to MultipleChoiceQuestion.

SingleLineTextAnswer Answer type derived from the generic Answer; corresponds to SingleLineTextQuestion.

MultiLineTextAnswer Answer type derived from the generic Answer; corresponds to MultiLineTextQuestion.

Entity relationships

As can be seen from figure 4.2, the ER diagram also reflects relationships between the entities. For further clarification, the relationships presented in the diagram are to be read as follows:

- Users belong to at least one UserGroup.
- Admins manage all existing UserGroups.
- Questionnaires are assigned to at least one UserGroup.
- UserGroups contain at least one User.
- UserGroups are assigned at least one Questionnaire.
- Questionnaires consist of at least one Question.
- Questions belong to exactly one Questionnaire.
- Questions have many Answers.
- Answers belong to exactly one Question.

4.2.2 Server

Class diagrams are created based on the domain model from section 4.2.1. Each entity from the ER diagram is supplemented with data attributes, i.e. atomic units of primitive data, which are specific to each respective entity. Furthermore, the class diagram introduces classes that are specific to the implementation and not part of the domain model. This section presents the class diagrams created for the server-side implementation of the system. Since the complete diagram for the entire system is too large to be shown as a whole in full detail, figure 4.3 represents an abridged and simplified version with only the essential information. Specific classes are presented in full detail and discussed later in this section. Note: unlabeled arrows represent a "has" or "belongs to" relationship.

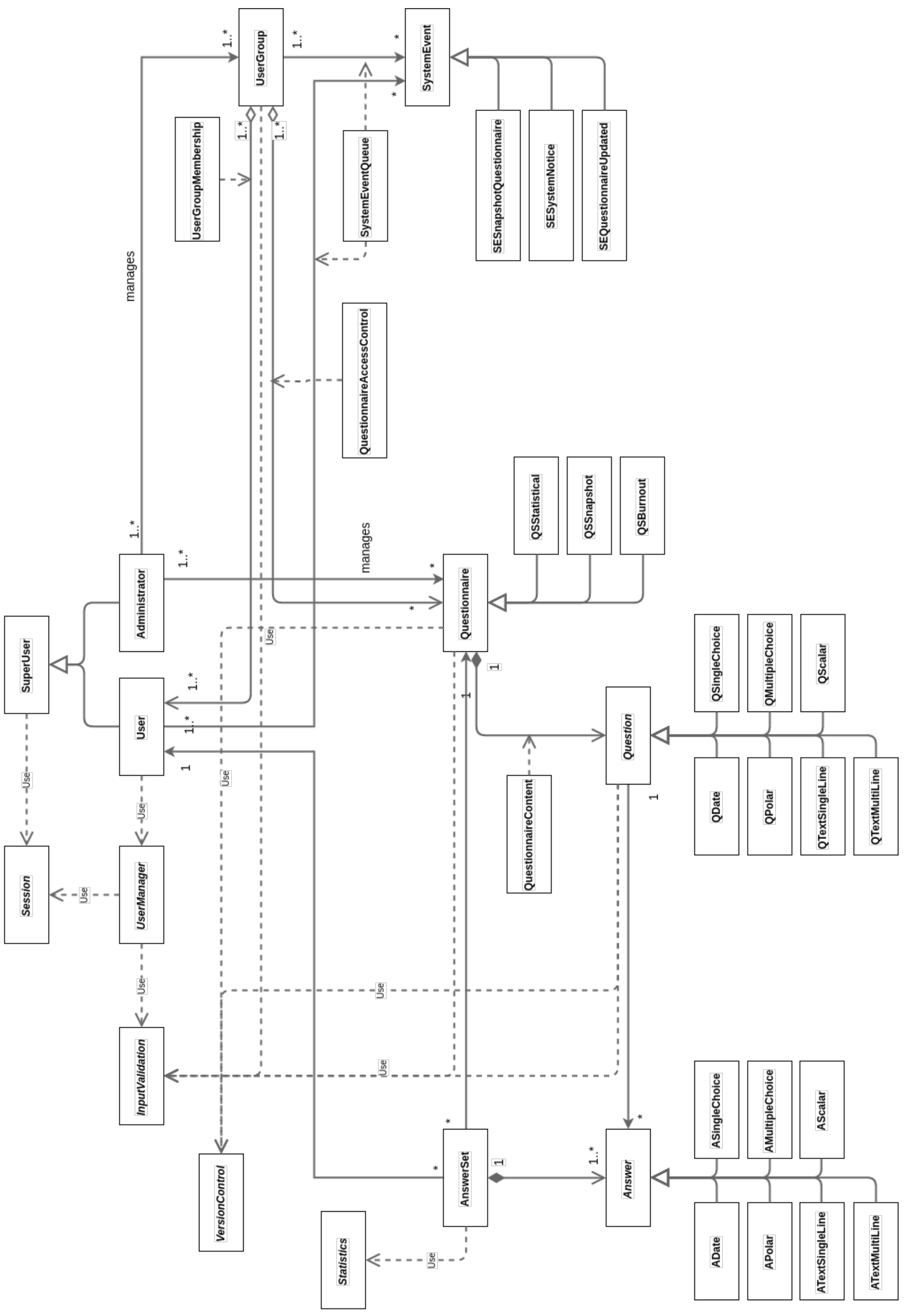


Figure 4.3: Class diagram for AYS (abridged, simplified)

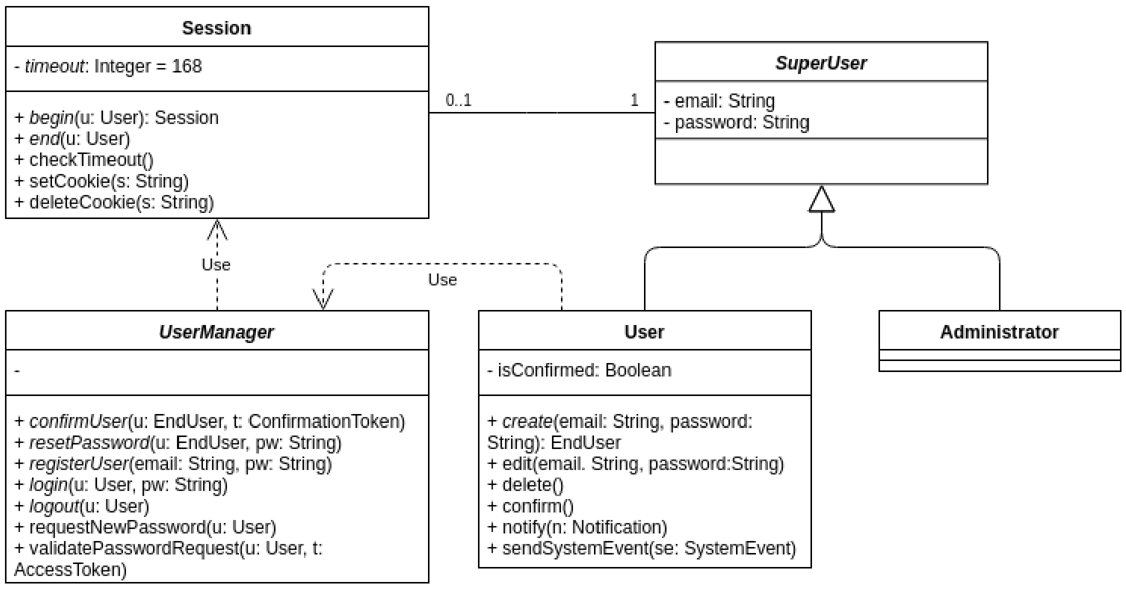

Session
- timeout: Integer = 168
+ begin(u: User): Session
+ end(u: User)
+ checkTimeout()
+ setCookie(s: String)
+ deleteCookie(s: String)
0..1
1
SuperUser
- email: String
- password: String
Use
Use
UserManager
-
+ confirmUser(u: EndUser, t: ConfirmationToken)
+ resetPassword(u: EndUser, pw: String)
+ registerUser(email: String, pw: String)
+ login(u: User, pw: String)
+ logout(u: User)
+ requestNewPassword(u: User)
+ validatePasswordRequest(u: User, t: AccessToken)
User
- isConfirmed: Boolean
+ create(email: String, password: String): EndUser
+ edit(email. String, password:String)
+ delete()
+ confirm()
+ notify(n: Notification)
+ sendSystemEvent(se: SystemEvent)
Administrator

Figure 4.4: Class diagram for the User model and related

Users

Figure 4.4 shows the specification of the user classes in full detail.

In purely object-oriented software, related user classes are a typical use case for *inheritance*. For technical reasons, however, this is not a viable solution for this system; see section 5.2.3 for more details on the role-based approach utilized in this system. Note that the user classes are represented as child classes of a superclass, even though they are not implemented as such. The reason for this is that the class diagram is part of the conceptual definition of the system, not the platform-specific definition. The same is true for a number of other, similar cases described below.

The system currently requires no personal data from the users, such as names or addresses, and only stores their log-in credentials.

The UserManager class is discussed in section 4.2.2.

UserGroups

Figure 4.5 shows UserGroup class and all related classes.

UserGroups contain only a single attribute—a unique *name* string. It also implements all functions needed to add and remove Users and Questionnaires to and from UserGroups. As can be seen from 4.5, the actual assignment of Users and Questionnaires is delegated to separate tables, namely *UserGroupMembership* and *QuestionnaireAccessControl*. These tables simply contain pairs of foreign keys of User and UserGroup or Questionnaire and UserGroup respectively for each assignment.

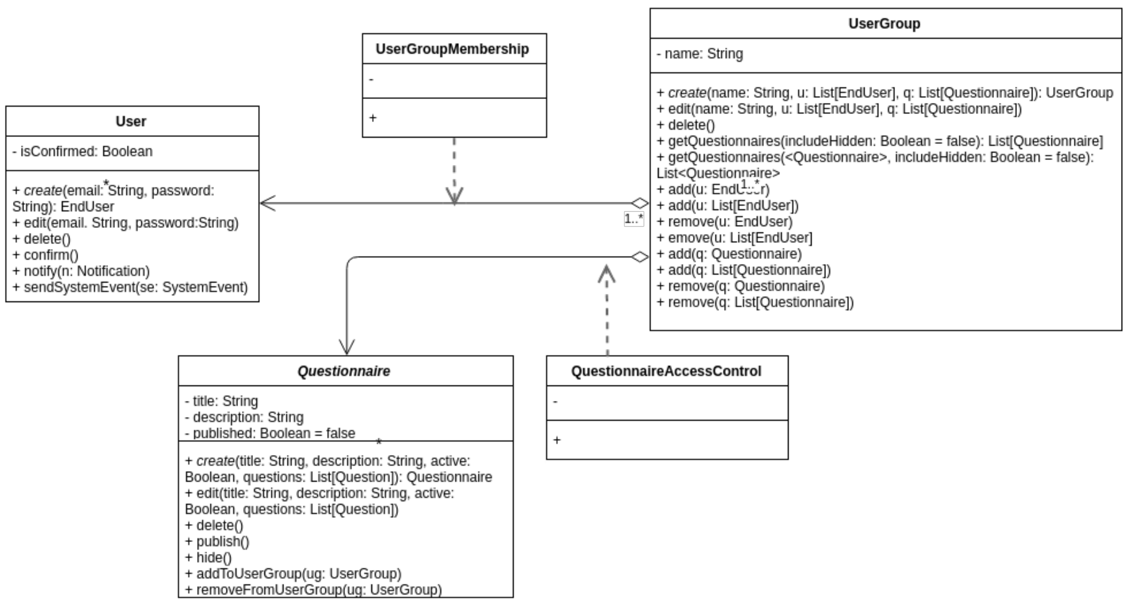

UserGroupMembership
-
+
UserGroup
- name: String
+ create(name: String, u: List[EndUser], q: List[Questionnaire]): UserGroup
+ edit(name: String, u: List[EndUser], q: List[Questionnaire])
+ delete()
+ getQuestionnaires(includeHidden: Boolean = false): List[Questionnaire]
+ getQuestionnaires(<Questionnaire>, includeHidden: Boolean = false): List<Questionnaire>
+ add(u: EndUser)
+ add(u: List[EndUser])
+ remove(u: EndUser)
+ emove(u: List[EndUser]
+ add(q: Questionnaire)
+ add(q: List[Questionnaire])
+ remove(q: Questionnaire)
+ remove(q: List[Questionnaire])
User
- isConfirmed: Boolean
+ create(email: String, password: String): EndUser
+ edit(email. String, password:String)
+ delete()
+ confirm()
+ notify(n: Notification)
+ sendSystemEvent(se: SystemEvent)
*
1..*
1..*
*
Questionnaire
- title: String
- description: String
- published: Boolean = false
+ create(title: String, description: String, active: Boolean, questions: List[Question]): Questionnaire
+ edit(title: String, description: String, active: Boolean, questions: List[Question])
+ delete()
+ publish()
+ hide()
+ addToUserGroup(ug: UserGroup)
+ removeFromUserGroup(ug: UserGroup)
QuestionnaireAccessControl
-
+

Figure 4.5: Class diagram for the UserGroup model and related

Figure 4.5 also demonstrates that there is no direct relation between Users and Questionnaires. Both are connected only through UserGroups, i.e. group membership decides which Users may access which Questionnaires. This additional level of indirection helps keep users organized and simplifies access control, as Questionnaires only have to be assigned to UserGroups, of which there are few, instead of individual users, of which there may well be hundreds or thousands.

Questionnaires and Questions

Figure 4.6 shows how Questionnaires and Questions are represented in the system and how their relationship is implemented.

Like Users, Questionnaires and Questions are both *extended* by specialized child classes. However, due to technical limitations of the *Laravel* framework, PHP's inbuilt inheritance system cannot be easily employed for this purpose. The work-around used for this framework is described in greater detail in section 5.2.3. Currently, all Questionnaire and most Question subclasses do not contain any additional attributes or functionality and are only used to distinguish Questionnaire and Question types, since they are rendered differently in the UI. The single and multiple choice Question types contain an additional *answerOptions* field, which contains the options from which Users may choose when answering the Question. The *min*, *max* and *step* attributes of the scalar Question type are used to define the range of numeric values accepted for this Question. Lastly, the polar Question type contains labels for both the *positive* and *negative* answer option.

The boolean *published* attribute and the *publish* and *hide* functions in the Questionnaire class implement the publish/hide mechanism described in requirements UC104, F006, S003.

The functions *addToUserGroup* and *removeFromUserGroup* in the Questionnaire class are convenience methods that simply "redirect" to the respective methods implemented in the User-

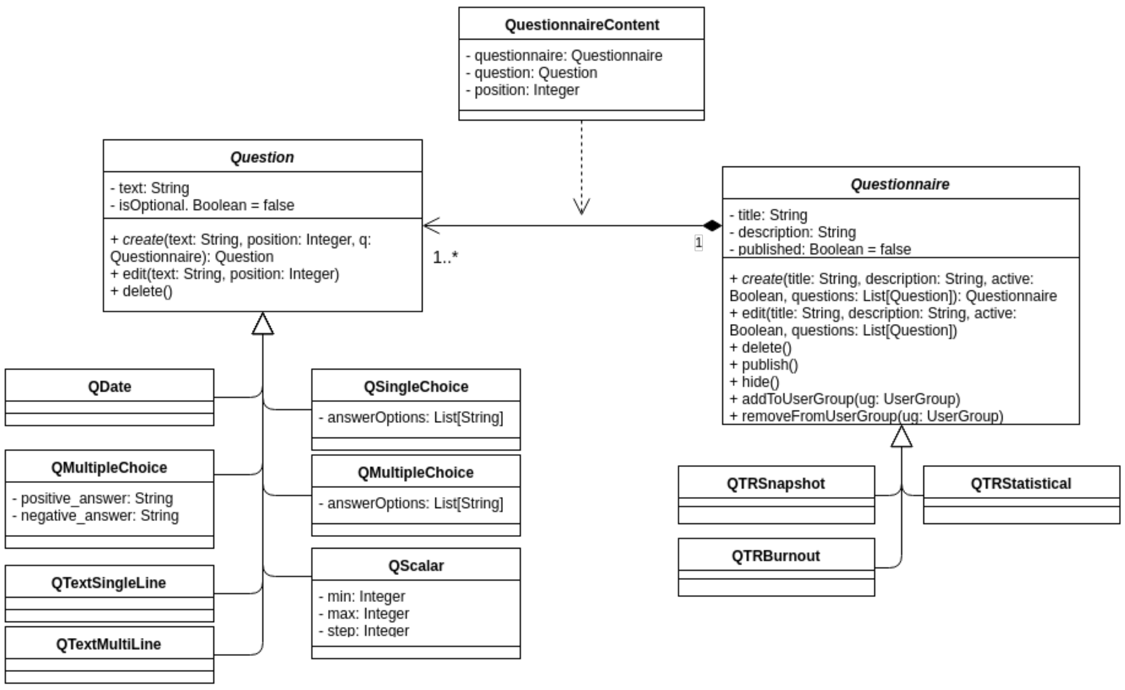

QuestionnaireContent
- questionnaire: Questionnaire
- question: Question
- position: Integer
Question
- text: String
- isOptional. Boolean = false
+ create(text: String, position: Integer, q: Questionnaire): Question
+ edit(text: String, position: Integer)
+ delete()
1..*
1
Questionnaire
- title: String
- description: String
- published: Boolean = false
+ create(title: String, description: String, active: Boolean, questions: List[Question]): Questionnaire
+ edit(title: String, description: String, active: Boolean, questions: List[Question])
+ delete()
+ publish()
+ hide()
+ addToUserGroup(ug: UserGroup)
+ removeFromUserGroup(ug: UserGroup)
QDate
QSingleChoice
- answerOptions: List[String]
QMultipleChoice
- positive_answer: String
- negative_answer: String
QMultipleChoice
- answerOptions: List[String]
QTextSingleLine
QScalar
- min: Integer
- max: Integer
- step: Integer
QTextMultiLine
QTRSnapshot
QTRStatistical
QTRBurnout

Figure 4.6: Class diagram for Questionnaire, Question and related

Group class. For the sake of keeping the code *DRY* [34], only the UserGroup implements the actual functionality needed to add or remove Questionnaires (or Users) to or from groups.

It should be noted that some of the attributes in the Questionnaire and Question classes, while noted as regular class attributes, are actually multilingual attributes, such as *title*, *description* or *text*. The implementation of dynamically translated attributes is discussed in greater detail in section 5.2.1.

As can be seen from figure 4.6, the intermediary table *QuestionnaireContents* is used to store the order of Questions in a given Questionnaire. In *TYT*, a Questions' position within the questionnaire was stored in an integer attribute on the Question class itself. While this is certainly a workable solution for the scope of this project, the position is, strictly speaking, not a property of the Question itself, like its text or answer options, and should therefore not be part of its attributes. This has the additional advantage of removing the foreign keys for Questionnaires from the Questions table and moving them to the QuestionnaireContents table.

Answers and AnswerSets

Figure 4.7 illustrates the relationship between Questions and Answers in this system.

As the class diagram shows, the Answer class can be understood to mirror the Question class. It has child classes that each correspond to one of the child classes of Question. The Answer superclass contains a single, generic *value* attribute, which is overridden by each child class with a *value* attribute of the appropriate data type. For example, the date Answer type holds a date value as its answer, the single choice Answer type stores the index of the chosen answer from

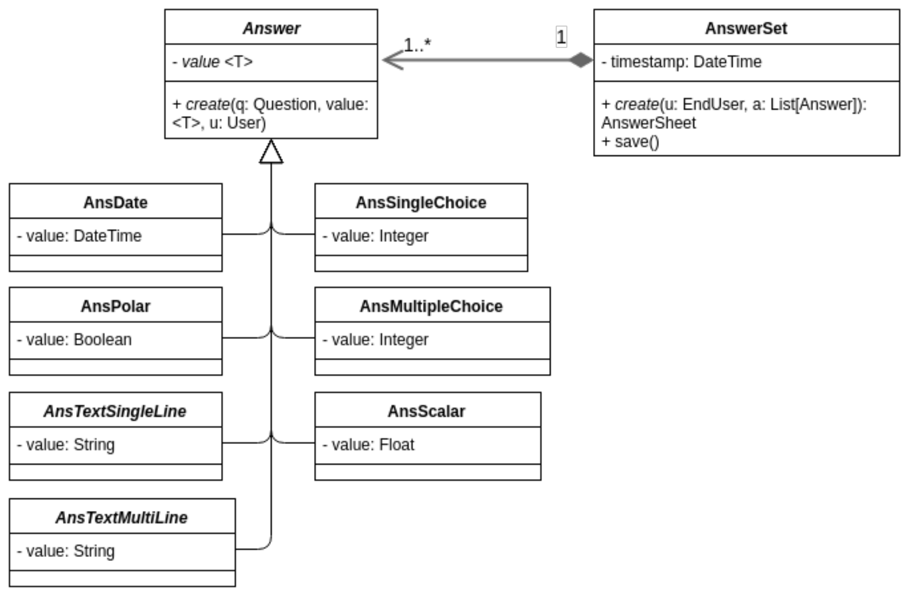


Figure 4.7: Class diagram for Answer and AnswerSet

the options array in the corresponding Question and the multiple choice type holds an array of such indices.

Like the Answer class complements the Question class, the AnswerSet class fulfills roughly the same purpose as the Questionnaire class. AnswerSet is used as an aggregate class for Answers. Every Answer, therefore, belongs to an AnswerSet and not (directly) to a Questionnaire. The primary purpose of AnswerSet is to store meta data related to the Questionnaire or all Answers given in a specific Questionnaire by a specific User, such as a timestamp of when the User started filling in the Questionnaire. This kind of meta data would not be appropriate to store within the Answers themselves. Like discussed for the case of the QuestionnaireContent class above, this additional level of indirection again simplifies working with Answers by adding. Furthermore, AnswerSets are the single point of access for Users to Questionnaires and related classes, since they have no direct relationships with Questionnaires, Questions or even Answers. This reduces coupling between classes and helps control exactly how the User class may interact with its related Answers, Questions and Questionnaires.

The approach taken in this project is quite different from the solution implemented in *TYT*. In *TYT*, answers are stored in the *standardanswers* table. Each answer record contains a foreign key of a user record. As described above, this system introduces AnswerSets, which now act as an intermediary table between Users and Answers. Furthermore, the *standardanswers* table contains columns *question1* through *question8*. These columns store the values users that input for each of the eight hard-coded questions in the only available questionnaire. Clearly, this solution is very specific to *TYT* and very hard to apply to other use cases and impossible to extend without modifying the *standardanswers* table and existing software logic. Since the new iteration of the system should be more general and extensible, the handling of answers was completely rebuilt as described above. Since each Question type now has a corresponding

Answer table that only stores the specific values needed for that particular Question type, it becomes easy to add new Question (and Answer) types to the system.

SystemEvents

Figure 4.8 shows the SystemEvent and related classes:

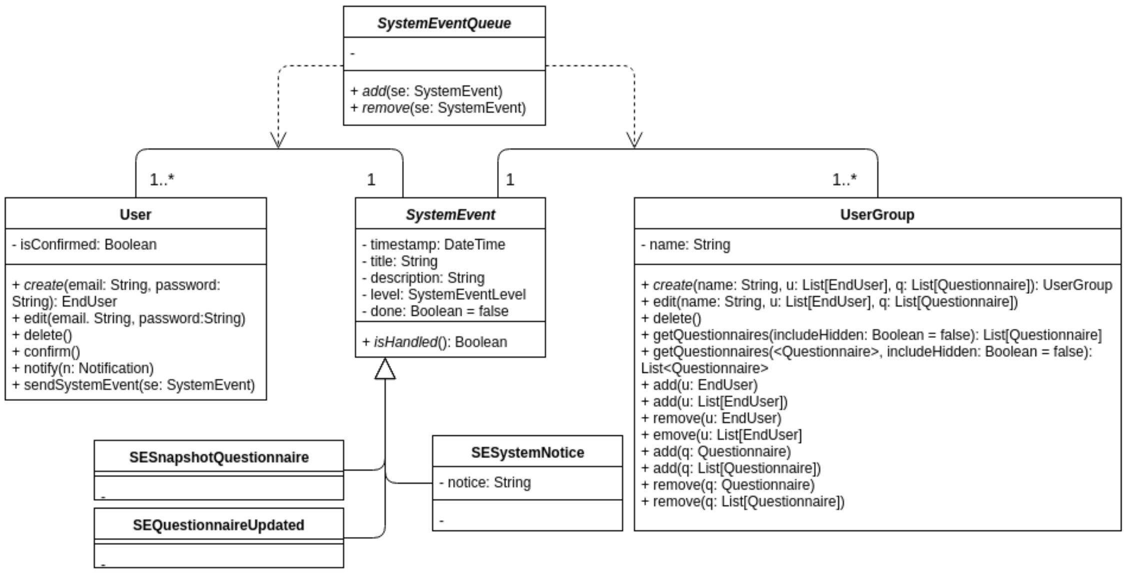

SystemEventQueue
-
+ add(se: SystemEvent)
+ remove(se: SystemEvent)
1..*
1
1
1..*
User
- isConfirmed: Boolean
+ create(email: String, password: String): EndUser
+ edit(email. String, password:String)
+ delete()
+ confirm()
+ notify(n: Notification)
+ sendSystemEvent(se: SystemEvent)
SystemEvent
- timestamp: DateTime
- title: String
- description: String
- level: SystemEventLevel
- done: Boolean = false
+ isHandled(): Boolean
UserGroup
- name: String
+ create(name: String, u: List[EndUser], q: List[Questionnaire]): UserGroup
+ edit(name: String, u: List[EndUser], q: List[Questionnaire])
+ delete()
+ getQuestionnaires(includeHidden: Boolean = false): List[Questionnaire]
+ getQuestionnaires(<Questionnaire>, includeHidden: Boolean = false): List<Questionnaire>
+ add(u: EndUser)
+ add(u: List[EndUser])
+ remove(u: EndUser)
+ emove(u: List[EndUser]
+ add(q: Questionnaire)
+ add(q: List[Questionnaire])
+ remove(q: Questionnaire)
+ remove(q: List[Questionnaire])
SESnapshotQuestionnaire
-
SEQuestionnaireUpdated
-
SESystemNotice
- notice: String
-

Figure 4.8: Class diagram for SystemEvent and related

The SystemEvent class is one of the classes that was not derived directly from the domain model defined in section 4.2, but is necessary for the implementation. SystemEvents are used to communicate information between the system or its administrators and Users. The primary reason for the inclusion of such a concept was to satisfy requirement S004, i.e. notifying users when Questionnaires that they have completed before have been edited. However, to keep the system easily extensible in the future, this concept has been generalized to be applicable to different types of events occurring in the system as well, such as generic messages from administrators. For example, this could be used to inform Users about system maintenance periods, new Questionnaires etc.

Static service and utility classes

The class diagram in figure 4.3 also features a number of *static*, i.e. non-instantiable classes that implement various utility functions. These functions are generally not suited to be included in classes representing entities. Utility classes are used like *services* throughout the system, meaning that objects may pass any data to them for processing and then retrieve some result.

UserManager The UserManager class depicted in figure 4.4 is the central facility that manages all user-related functionality like creating new user records, user authentication, verifying confirmation tokens etc.

InputValidation The InputValidation class provides validation methods for all necessary input types. All input coming into the system must first be validated by this class before it can be further manipulated or stored in the database. Validation methods accept the input that should be validated and return a boolean value indicating whether the input is valid.

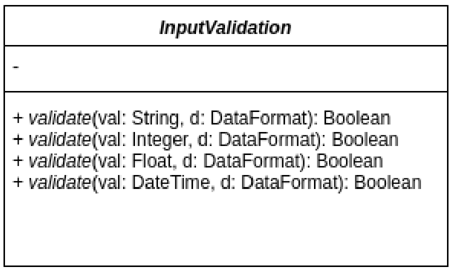

InputValidation
-
+ validate(val: String, d: DataFormat): Boolean
+ validate(val: Integer, d: DataFormat): Boolean
+ validate(val: Float, d: DataFormat): Boolean
+ validate(val: DateTime, d: DataFormat): Boolean

Figure 4.9: Class diagram for InputValidation

Statistics The Statistics class is used to calculate and visualize Questionnaire results. The *getDiagramData* method summarizes and transforms result data for a given User, either including all their results or only for a specific Questionnaire, and returns those data in a form that clients can use to render diagrams. The *export* function outputs result summaries in a given file format (CSV or PDF), which can then be downloaded by the client.

VersionControl The VersionControl class is used to manage versions of arbitrary records within the system's database. The general idea behind it is that it keeps references to records that have been submitted to version control, their related versions and the original or "root" record. The inner workings of the version control system in this project are described in more detail in section 5.2.2.

4.2.3 Client

This section discusses the design for the static structure of the prototypical web-based client. Since the data objects, representing the entities defined in section 4.2.1, passed between server and client are the basis of the system's functionality, they constitute the common ground between the server's and the client's static structure. Because the entity classes are practically identical for both the server and the client, figure 4.10 only contains abridged versions of the classes. Refer to section 4.2.1 for more details on the entities. Figure 4.10 also shows a number of static utility classes. These are shortly described below.

Since some of the "classes" depicted in 4.10 are not implemented as actual classes in the sense of object-oriented programming, the descriptions below may also refer to them more generally as "modules".

Before discussing some of the components that make up the client, a short section is dedicated to *single-page applications*, outlining the general principles behind them and why this design choice was made in this project.

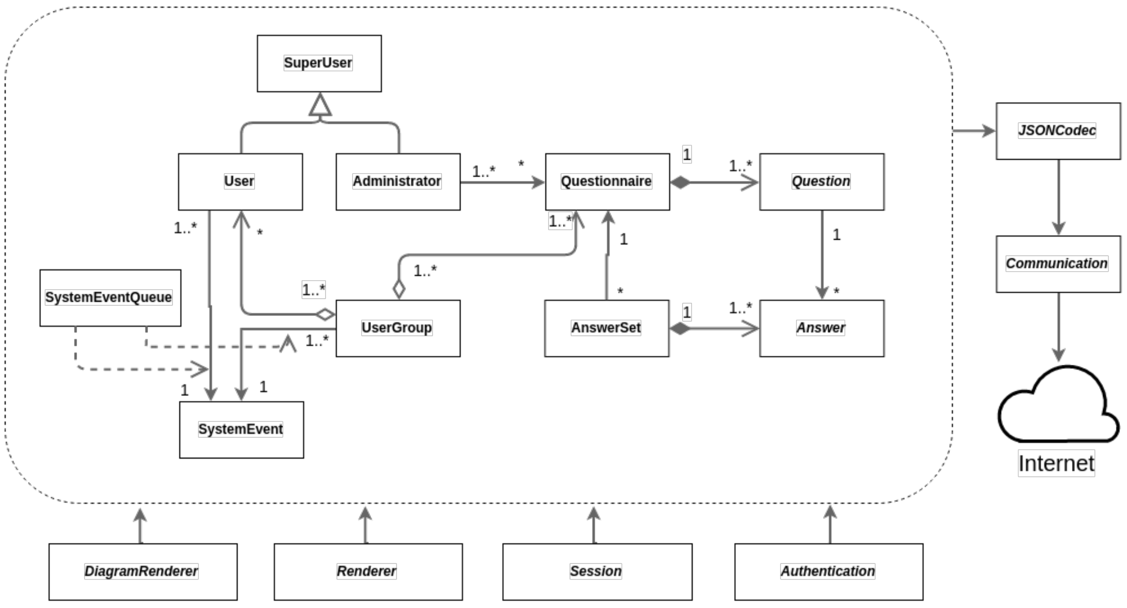


Figure 4.10: Class diagram for the client implementation

Single-page applications

Single-page applications are web applications that run on the client instead of the server, with the client usually being the user's browser. Such applications are generally built using JavaScript-based frameworks such as *AngularJS*, *Ember.js* or *React*, to name only a few. Frameworks like these were first developed around the year 2010 and have been gaining popularity ever since. Essentially, single-page applications are, as the name implies, websites that never fully refresh. In traditional web applications, when the user clicks a link, submits a form or changes the URL, the client sends a request to the server, which then generates some sort of response object, typically an HTML document, and sends it back to the client. Upon receiving the server's response, the client's browser refreshes its viewport and renders the new page. This has a clear impact on the user experience, as there is a noticeable lag between sending requests, receiving the response and finally rendering a new complete page. Singe-page applications take a different approach: Instead relying on the server to generate websites and re-rendering the entire browser viewport on each request, single-page applications utilize the server's external API to request only the data they need using AJAX requests [8]. When the requested data return, the front end application then simply modifies the appropriate parts of the DOM to display the data. The chief goal of this technique is to provide a more fluid, desktop-like experience, because only specific parts of the website change dynamically without noticeably re-rendering the whole page. This is also achieved by the smaller footprint of messages sent between the server and client. Instead of potentially very large HTML documents, the server only has to send significantly smaller data objects, most commonly in the JSON format. As discussed in 5.1.4, this format has the advantages of being small in document size (compared to comparable solutions like XML), human-readable and easy to encode and decode. These properties also help keep the server's response time as short as possible, enabling an improved user experience over the basic request-response model.

There are two main reasons it was decided to implemented the prototypical client for this system as a single-page application: First, the stakeholders for the project requested the use of a front end framework out of the desire to introduce more technically innovative solutions to *TYT*. Second, as mentioned several times before in this document, the early separation of client and server implementations prepares the system for future extensions with mobile and other clients. When the generic JSON API is already implemented and used by the single-page application as the sole communication interface with the server, the web client serves as an example and boilerplate for mobile applications.

DiagramRenderer

This module receives a User's Questionnaire results from the server or local storage and creates a JavaScript-based, interactive visualization of the data from those results. Based on meta data passed along with the actual results, the class is able to choose the appropriate type of diagram. This class also listens for user input and adjusts the diagram as necessary, e.g. when the user wants to zooming into a section of the graph or if specific parts of the graph should be displayed or hidden.

Renderer

The Renderer module is tasked with rendering the HTML pages and components within them. This is necessary because single-page applications typically do not receive generated HTML documents from the server, as explained in section 4.2.3. The Renderer receives the data objects sent from the server and modifies the page appropriately to represent the data.

Storage

The Session module deals with any data that should be stored locally on the client. Its tasks include handling the users' session (or rather an equivalent of that concept), remembering signed-in users up to a set timeout, caching Questionnaires etc.

Authentication

The Authentication module is tasked with the sign-up, sign-in and sign-out procedures. While the server is still needed to verify users' credentials, the traditional session concept is not cleanly mappable to single-page applications, which is why this module is necessary. It uses the Storage module to sign in users and remember them.

JSONCodec

This module's purpose is to transform objects from the single-page application into valid JSON objects and, conversely, to decode JSON objects received from the server and transform them into class instances that the front end application understands.

Communication

The Communication module handles the "heavy lifting" of sending messages to the server and receiving its responses via AJAX. It constructs valid, *JSON API*-compliant messages with the help of the Router and JSONCodec modules and transmits them to the server. It also validates incoming messages from the server, handles invalid responses and extracts JSON objects from the message payload for further processing through JSONCodec.

Router

The Router module holds both *internal* and *external routes* defined in the application. External routes are the server's API endpoints, to which the client sends requests and data. Users never interact directly with the server API, as the API is purely used for machine-to-machine communication. In single-page applications, the internal routes take the place of regular URIs that traditional web applications use to expose their functionality to clients. However, since single-page applications do not send regular HTTP requests to the server and do not refresh the page, changes in the URL must be handled in the front end as well. This is the Router's purpose. It intercepts changes of the URL and also listens to clicks on links, form submits etc. When a change occurs, the Router passes the appropriate component to be rendered to the Renderer according to the internal routes registered with the Router instead of passing the request to the server and reloading the page.

4.3 Dynamic structure

This section introduces the dynamic aspect of the system's design. Section 4.2 has defined the entities and classes that make up the static structure of the system. The other part of the back end's architecture is the dynamic part, i.e. how the previously defined objects actually interact. Section 3.1 describes the system's behavior in the form of use cases, which are descriptions from the users' perspective in terms of goals and outcomes. This section expands upon those use cases and illustrates the actual underlying functions that have to be implemented to provide the desired functionality. For the sake of brevity, only non-trivial processes are included in this section.

Note: As mentioned in the introduction to this chapter, the designs presented here are conceptual and might not represent the final implementation precisely. The described processes, however, remain the same in principle, even if the implementation adds some steps to the process or modifies them.

4.3.1 Register user

See also UC001, F018.

When a new User submits their credentials to sign up to the system, InputValidation first validates the input and rejects the request if the e-mail address, which must be correctly formatted and unique within the system, or the password, which has a minimum length, are invalid. Once the User has submitted valid credentials, UserManager creates the new User record in the database. It also creates a confirmation token, which is simply a 40-character-long string of random characters, and sends it to the e-mail address the User has just entered. The User then has to click the link sent in the confirmation e-mail within 24 hours. If the account is unconfirmed after 24 hours, it will be deleted. When the User clicks the confirmation link, UserManager validates the token that is contained in the query string. If the token matches the User record and it has not timed out, the account is confirmed and the User is signed it. From this point on, the User can access the system's full functionality.

4.3.2 Create questionnaire

See also UC101, F003.

As with all input received from the client, InputValidation first checks all input when an Admin creates a new Questionnaire. More specifically, it has to make sure that:

1. *title* is a non-empty string
2. *description* is valid and does not contain prohibited characters (HTML, JavaScript), if it is provided
3. An existing Question type is given
4. At least one Question is created and all contained data is valid (*text*, *answerOptions* etc., depending on the Question type)

5. If UserGroups were selected, their IDs exist and are unique
6. If the Questionnaire should be published, a boolean *published* value is passed

If the input is valid, the system creates the new Questionnaire record, along with Question records, which it then associates with the Questionnaire. If the Admin has chosen to assign UserGroups, the system also creates the appropriate relationships between the Questionnaire and the selected UserGroups.

4.3.3 Edit questionnaire

See also UC102, F004.

While the process of editing Questionnaires is generally similar to creating new ones, the editing process is slightly more involved as it deals with version control and SystemEvents.

The data passed when editing Questionnaires is identical with the data for creating new Questionnaires described above. Accordingly, the input must pass the same validation checks. If the Admin changes the UserGroup assignment, the system *synchronizes* the associated records accordingly, i.e. deselected UserGroups are unlinked from the Questionnaire and any new UserGroups are assigned. The main difference between the editing and creation process is version control and the use of SystemEvents. As discussed in requirement F001, the system automatically keeps track of any objects, that are enabled for version control, if they are modified. Questionnaire editing exemplifies this process with Questions. If the Admin edits or deletes any Questions in the Questionnaire, the VersionControl creates new entries for them and links them with their "root" versions, which are always the newest revision (see 5.2.2 for more details on the version control system). The Questionnaire has no direct access to "archived" versions and always works with the newest one. Finally, as stated in requirement S004, Users who have already completed a Questionnaire must be notified when that Questionnaire is modified. The system collects all Users that have already answered the current Questionnaire. It then creates a new SystemEvent object and adds it the each of those Users' SystemEventQueues. The next time the Users sign in, they will be notified of the changes in the Questionnaire and asked to fill in the Questionnaire.

4.3.4 Fill in questionnaire

See also UC204.

Once the client has received the requested Questionnaire, either from the server or as a cached version from its local storage, the User begins answering Questions. Once the first Answers are sent to the server, the server creates a new AnswerSet object for the Questionnaire, setting its timestamp. and the User (except for StatisticalQuestionnaires, see requirement F002). It validates and saves the received Answers and stores each as a new Answer object in association with the newly created AnswerSet. Once the client fires the "submit" command to close the Questionnaire, the server finalizes the AnswerSet.

A note on Answers: The server does not expect the Answer sent from the client to be grouped in any specific way, like grouping by "page" or all at once. The server simply expects a collection object containing valid Answer objects, which all have to store references to their related Questions. The reasons for this are twofold: The server should be agnostic to any details of the presentation, like whether Questions are displayed on pages or all together on one larger page. For this reason, assumptions about how Answers are sent should be kept minimal. Furthermore, this enables clients to employ various strategies when displaying Questionnaires. Different clients can offer different layouts based on device properties and external factors like device orientation. They may also decide how Answers are submitted, e.g. transparently in the background on a page-by-page basis without any explicit "submit" commands or all Answers at once after a final "submit". All this again results in greater flexibility and extensibility as required in requirement N003.

4.3.5 Coping advice

As described in use case UC206, Users may request stress coping advice from the system. The system chooses from a selection of pre-written pieces of advice based on the User's current stress levels and ratings that each User gives to the advice they have received from the system.

The rating system is simple: Users may access a list of previously received advice in their client. There they may rate each piece of advice on a scale from 1 to 5 based on the success (or lack thereof) the User has had coping with stress using each specific strategy. The other factor the system considers when selecting a coping strategy is the User's current stress level. The coping strategies available in the system are divided by tier. Depending on the User's stress levels, the system selects a strategy from the corresponding "stress tier". Both stress levels and ratings are combined into scores for each eligible coping strategy and the strategy with the highest score is chosen. Note, however, that not only the strategies' own ratings are considered for the score. When entering new advice into the system or modifying existing strategies, administrators may choose to define relations between strategies. Ratings of related strategies will then also affect the final scores of their related strategies to some degree.

While the current iteration does not support it, the coping strategy system may be extended with support for tagging in the future. Administrators may choose to tag advice with certain topics, like "burnout", "chronic fatigue", "depression" and so on. This way, users can give more fine-grained feedback. Instead of rating the strategy as a whole, they can rate the strategy on each individual tag. Additionally, they may explicitly signal to the system, that they would like to receive more or less advice related to a specific tag. This will then also be taken into account in the selection of strategies.

4.4 User interface design

The final component of the system's architecture is the design of its UI. In the scope of this project, the UI is conceptualized and implemented as a prototypical, web-based front end. This chapter first presents the UI's dialog structure, which describes the UI in terms of the individual *views*, generally realized as entire websites, that the user interacts with throughout their usage of the system. The dialog structure describes which users can access which views, what these views contain and how users can navigate between those views. This design is illustrated both graphically as well as textually for each major view. The other part of the UI design is the visual design of the views. This includes the page layout, color palette, fonts, utilization of images, etc. These topics are mostly discussed in text form, but some early sketches of interface designs are also included.

4.4.1 Dialog structure

The dialog structure defines the *views* of the UI and how those views behave. For the purpose of this document, views are defined as distinct portions of the browser's viewport that are used to display specific parts of the UI. In most cases, these views encompass the entire page, but they can occasionally be represented in smaller areas such dialog overlays. The diagrams below do not explicitly differentiate between full-screen views and others, as they main purpose of these diagrams is only to define the general structure of the UI. The dialog structure diagrams also include the navigation paths between views. Again, exactly *how* users navigate the application, i.e. via links, buttons, URLs etc., is left up to the implementation.

The diagrams are grouped by user role as the system presents itself with distinctly different views to each user role. Each diagram represents a larger portion of the UI, like the home page, views related to questionnaires etc.

A note on notation: In the diagrams below, views are represented as rectangles with blue borders and the view's title inside the rectangle. Occasionally, views can also represent menus from which an option must first be chosen to access an actual view—the realization of this is left to the implementation. *Nested* views are represented like regular views, but with a light blue background. This signifies that the nested view contains multiple other views, which have been grouped under the nested view to reduce clutter in the diagrams. Each nested view is represented as another complete diagram. The labeled arrows between views define the navigation paths. The label beside each arrow explains the action or condition that causes the system to navigate from one view to another, like the user clicking on a menu item, errors occurring while validating input and so on. Arrows that do not originate from outside the diagram are to be understood as the initial entry into this portion of the UI, for instance when navigating using URLs. Arrows that lead outside the diagram signify the transition to another part of the UI, like navigating back through the browser history.

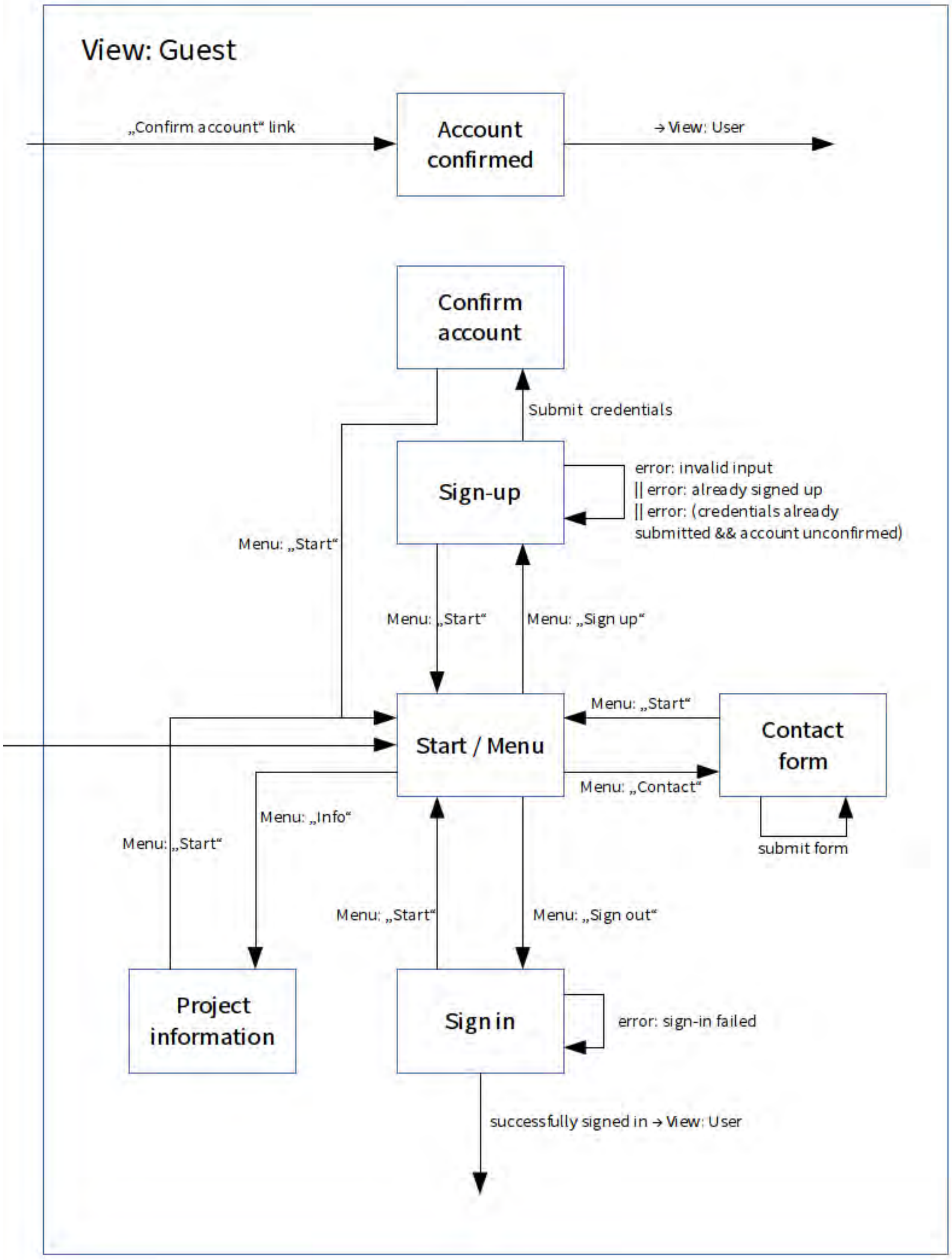


Figure 4.11: Dialog structure for the guest home view

Guest scope

As described in section 3.1, this system recognizes a number of different user types. However, the system only "knows" of signed-in users. Guests are merely a pseudo-type, but are nevertheless an important part of the system, since all new users start out as guests. Figure 4.12 shows the views that are available to guest users of the system. When guests first arrive on the home page of the system, they are greeted with general information about the system, the project and the team behind the project. The guest scope also offers a contact from that users may use to submit messages, questions and comments to the project team. Most importantly, however, guests can sign up and sign in. The user enters their credentials into the simple sign-up form in the *Sign-up* view. If there are any errors, the client highlights the invalid fields and displays appropriate error messages. Once the user enters valid credentials, they are reminded to confirm their account via the confirmation link that was just sent to their e-mail address. The *Account confirmed* view is what is displayed when the user clicks on the confirmation link in the e-mail and the confirmation token is valid. The guest is then automatically signed in and redirected to the home page in the User scope. Finally, if the guest is already registered but not signed in, the guest scope offers a simple *Sign-in* view, where users can enter their credentials and are also redirected to the User scope's home page, if the entered credentials are valid.

Administrator scope

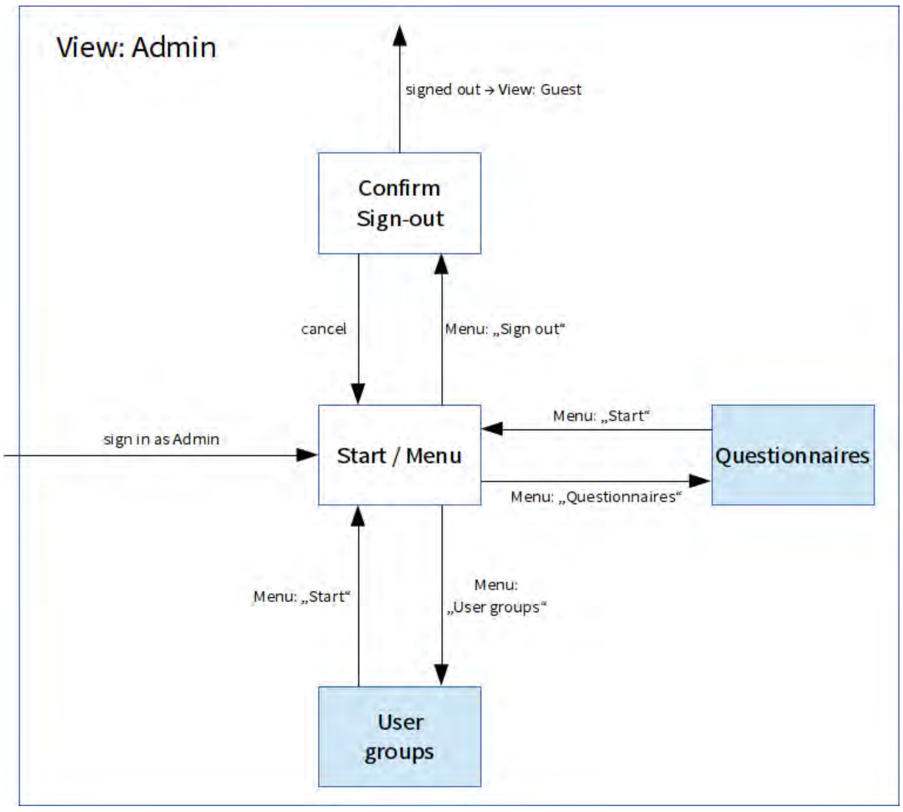


Figure 4.12: Dialog structure for the Admin home view

Home Figure 4.12 defines the views available to Administrators when they first access the system. This is also referred to as the home page or dashboard. The system redirects the user to the home page once they have successfully signed in as an Administrator. The home page itself does not contain much content and is merely the starting point for navigation. The home page would be a suitable place for a dashboard, but for the current iteration of the system, a dashboard is not useful. From the menu, which is permanently visible, the Administrator 1) sign out, 2) access the questionnaires section, or 3) access the user groups section. If the Administrator chooses to sign out, they are prompted to confirm their action and if they do so, their session is ended and they are redirected back to the home page as a guest user.

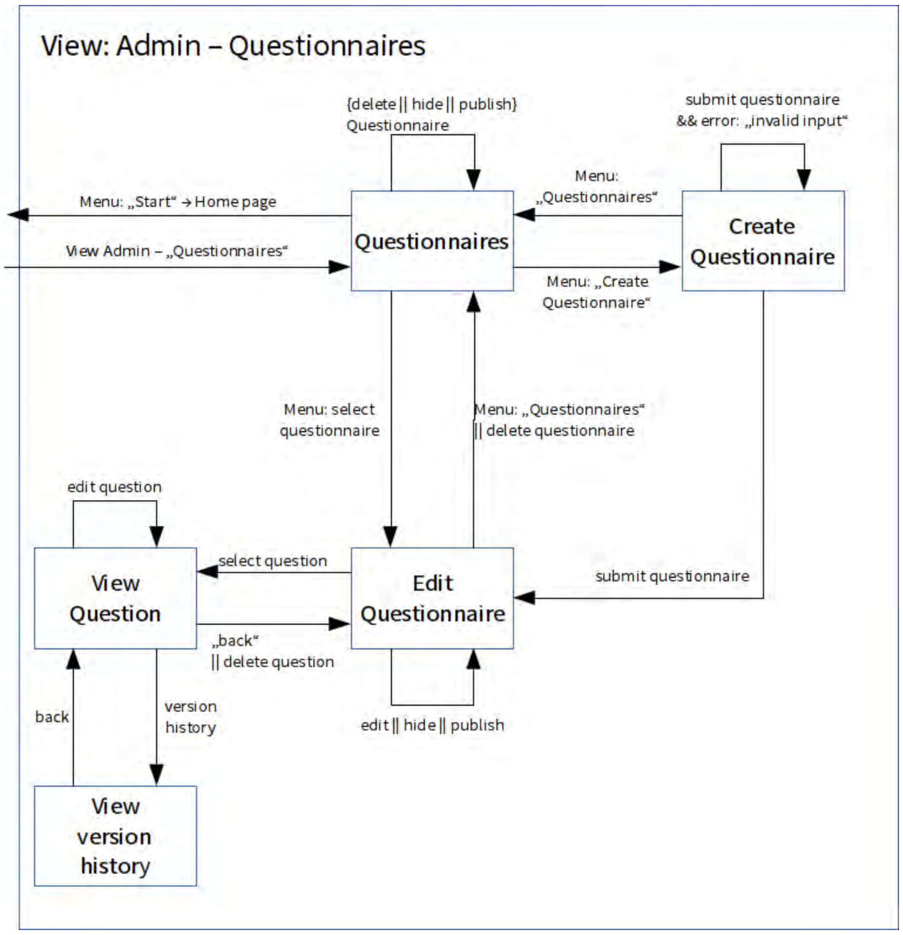

View: Admin – Questionnaires
{delete || hide || publish} Questionnaire
submit questionnaire && error: „invalid input"
Menu: „Start" → Home page
View Admin – „Questionnaires"
Questionnaires
Menu: „Questionnaires"
Create Questionnaire
Menu: „Create Questionnaire"
Menu: select questionnaire
Menu: „Questionnaires" || delete questionnaire
edit question
View Question
select question
Edit Questionnaire
submit questionnaire
„back" || delete question
back
version history
edit || hide || publish
View version history

Figure 4.13: Dialog structure for the Admin questionnaires view

Questionnaire From the questionnaires section depicted in 4.13, Administrators can view a list of all Questionnaires in the system, create and edit Questionnaires. The default view for this section is the Questionnaire list. When the Administrator navigates to the *create Questionnaire* view, the client displays the Questionnaire editor. The editor is a dynamic form that Administrators use to input all data for the new Questionnaire including its Questions. When the Administrator submits the form, the system handles it as described in 4.3.2. If there are any errors while creating the Questionnaire or its Questions, the view remains in the editor and all errors are highlighted. Once the Administrator submits a valid Questionnaire, the system saves it and the client redirects to the *edit Questionnaire* view. The *edit Questionnaire* and *create Questionnaire* views share the same editor with only a few differences: First, when the Administrator selects a Questionnaire from the list to edit, the editor's fields are pre-filled with the Questionnaire's current data. Second, Administrators additionally have the option the *hide*, *publish* and *delete* the Questionnaire from within the *edit Questionnaire* view. When the Questionnaire is hidden or published, the client does not immediately return to the Questionnaire list and, as opposed to the *create Questionnaire* view, the same is true when editing the Questionnaire, in case more changes should be made. If the Questionnaire is deleted, the client does return to the Questionnaire list as the Questionnaire cannot be displayed anymore when it is deleted. Lastly, the *edit Questionnaire* view also contains a list of previous versions for each Question in the Questionnaire. These *view Version History* and *view Question* views are likely candidates for embedded views. The overview of previous Question revisions is purely for the information of Administrators; "archived" records are immutable, as described in requirement F001. Finally, Questionnaires can also be deleted, hidden and published directly from the Questionnaire list view.

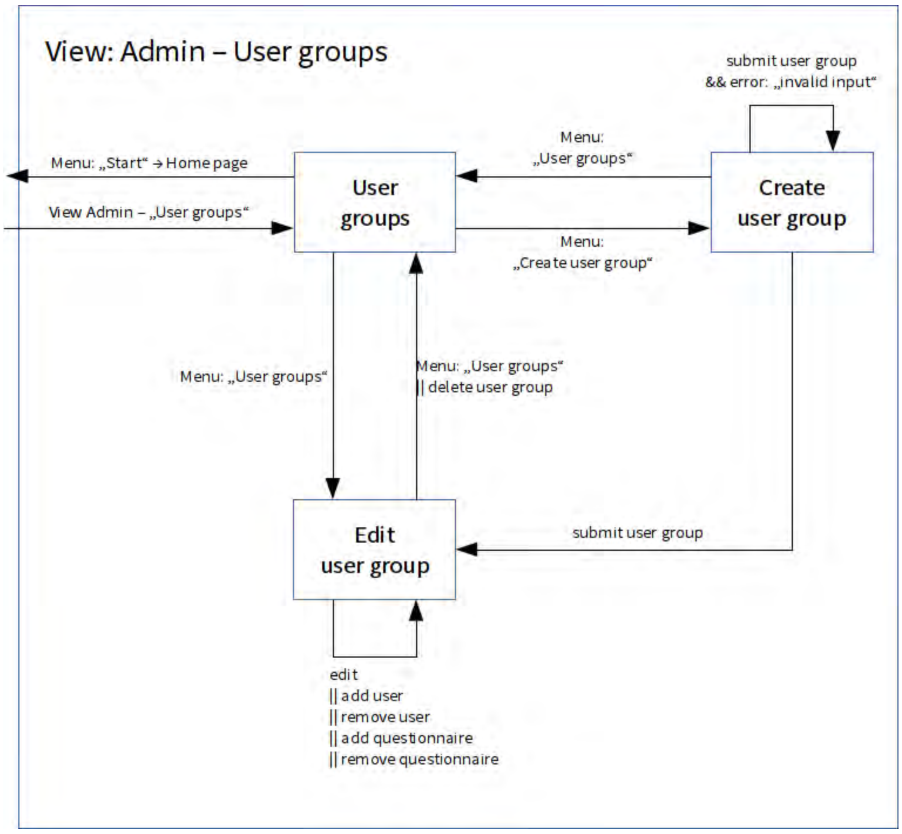

View: Admin – User groups
Menu: „Start“ → Home page
View Admin – „User groups“
User groups
Menu: „User groups“
Menu: „Create user group“
Create user group
submit user group && error: „invalid input“
Menu: „User groups“
Menu: „User groups“ || delete user group
Edit user group
submit user group
edit || add user || remove user || add questionnaire || remove questionnaire

Figure 4.14: Dialog structure for the Admin user groups view

User groups The last UI section exclusive to Administrators is the user groups section, shown in 4.14. The structure and navigation is very similar to that of the questionnaires section. Administrators may either choose the *create user group* or *edit user group* view from the menu. The UserGroup editor provides fields to input a unique name for the new UserGroup and optionally Users and Questionnaires that should be assigned. Like the Questionnaire editor, the User-Group editor remains in its view and highlights errors if there are any. Upon successful creation of the UserGroup, the client redirects to the *edit user group* view, which also operates in much the same way as the *edit Questionnaire* view. When the UserGroup is edited, i.e. its name or assigned Users or Questionnaires are modified, the view remains in the editor. The view returns to the UserGroup overview on explicit prompt from the Administrator or when the Administrator deletes the current UserGroup. UserGroups can also be deleted directly from the overview list.

User scope

Home Figure 4.15 illustrates the dialog structure for signed-in Users. Users arrive in this section of the UI when they are successfully signed in or when they navigate to the application's root URL and a former session is still valid. The *Project information* and *Contact form* views are identical with the views explained in 4.11. Users may sign out using the same *Sign-out* view as Administrators, described in 4.12. Finally, Users can navigate to the *Questionnaires* and *Results* sections, and *Account settings* view using the menu. In the *Account settings* view, Users may change their e-mail address and password, which is currently the only personal information the system kept in user accounts.

Questionnaires When the User navigates to the questionnaires section via the menu, they are first redirected to the list of Questionnaires, that are currently available to them, which can be seen from 4.16. Users can only see and access Questionnaires, that are assigned to the UserGroups of which those Users are members. When a User selects a Questionnaire from the list, they are redirected to the *Fill in Questionnaire* view. The Questionnaire is displayed as a list of Questions that it contains. Each Question is rendered with different UI widgets according to its Question type. For example, single and multiple choice Questions display a list of answer options, date Questions offer an HTML5 date input field, etc. The User completes the Questionnaire as described in 4.3.4 and when the Questionnaire is submitted, the client returns to the Questionnaire overview. In section 4.2.2, it is mentioned that Users edit their answers when they fill out StatisticalQuestionnaires repeatedly instead of creating an entirely new AnswerSet every time. For this reason, the Questionnaire view is displayed with its answer fields pre-filled with the User's last answers. Lastly, SnapshotQuestionnaires add an additional validation step to the navigation when they are accessed. Requirement S001 states that Users may only begin filling in SnapshotQuestionnaires once they have completed all available StatisticalQuestionnaires at least once. If the User does not meet this condition when they try to begin filling in a SnapshotQuestionnaire, the client returns them to the Questionnaire list and displays an error message stating that they must first complete all StatisticalQuestionnaires.

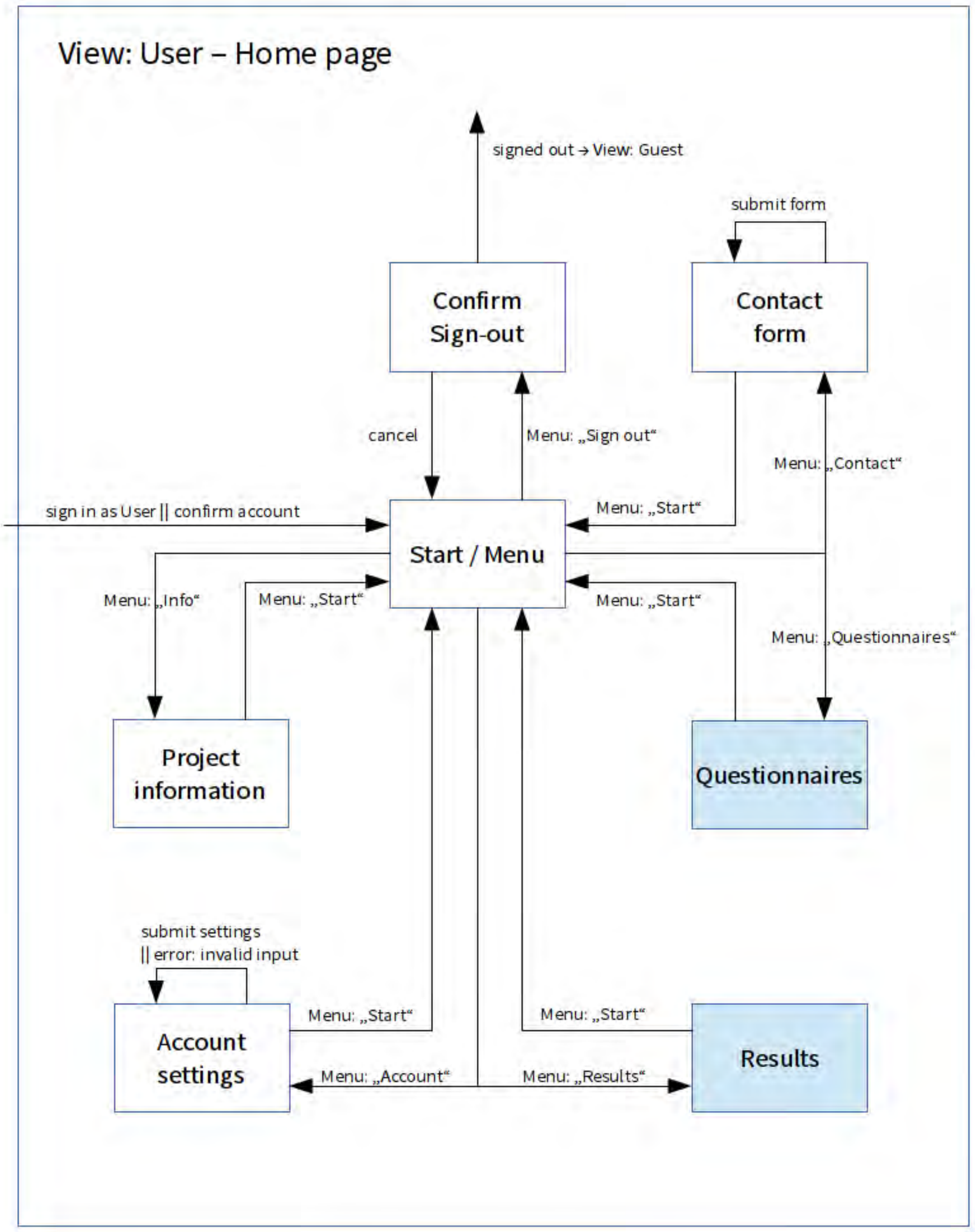


Figure 4.15: Dialog structure for the User home view

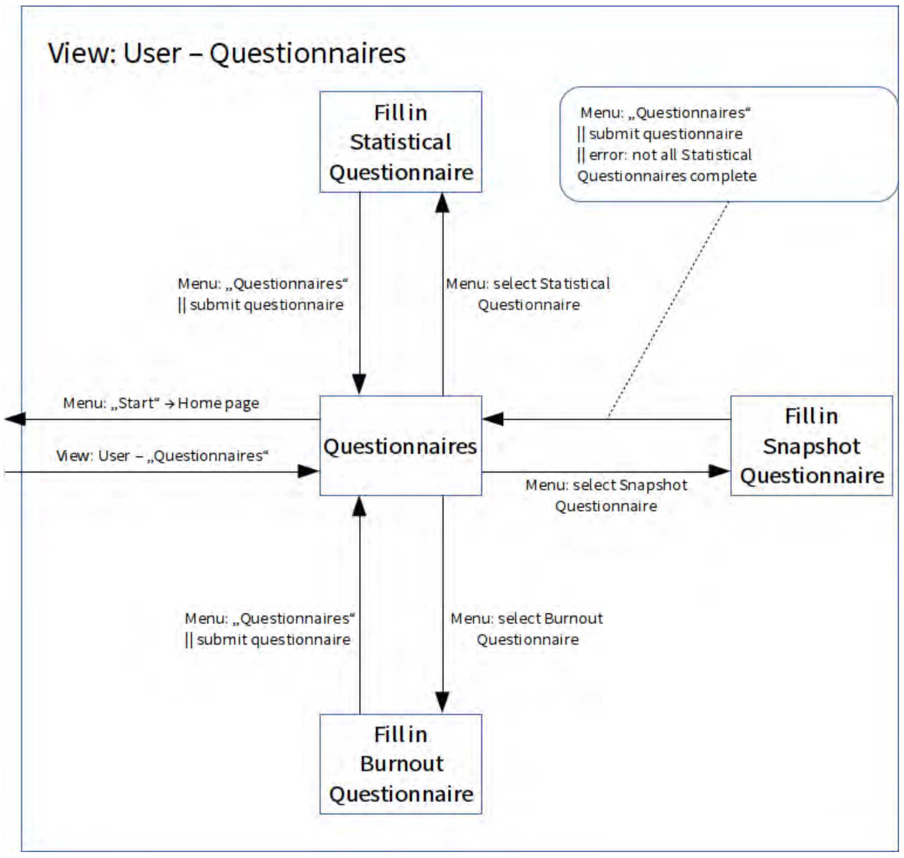

View: User – Questionnaires
Fill in Statistical Questionnaire
Menu: „Questionnaires" || submit questionnaire || error: not all Statistical Questionnaires complete
Menu: „Questionnaires" || submit questionnaire
Menu: select Statistical Questionnaire
Menu: „Start" → Home page
View: User – „Questionnaires"
Questionnaires
Fill in Snapshot Questionnaire
Menu: select Snapshot Questionnaire
Menu: „Questionnaires" || submit questionnaire
Menu: select Burnout Questionnaire
Fill in Burnout Questionnaire

Figure 4.16: Dialog structure for the User questionnaires view

Results Once Users begin tracking their stress levels with the system, they will want to view their test results. This can be done in the results section, which is structured as shown in 4.17. Users can choose to view result visualizations, export their raw result data or receive stress coping advice from the *Results* menu. The *Analysis/Coping advice* view is where Users can inspect previously requested coping strategies and request new ones. This view is also where Users will rate coping strategies. When Users choose to *Export results*, they must select a file format—at this point either CSV or PDF. The system will then generate a result overview and commence the download to the User's device when ready. Most of the time, however, Users will want to view the diagrams generated from their questionnaire results. From the *Results overview* view, the User may choose to view their results either as *Raw Data Tables* or *Diagrams*. The *Raw Data Tables* view presents the User's raw result data in a form similar to the CSV export file, i.e. as a navigable HTML table. The *Diagrams* view displays a number of different diagrams. These diagrams are generated for each Questionnaire the User has completed. The type of graph depends on the Questions in those Questionnaires; the client's DiagramRenderer module (see section 4.2.3) selects and generates the diagrams accordingly. The diagrams are navigable and interactive. Users can pan through the graph's x-axis, which corresponds to time, and select sections of the graph to zoom into. Furthermore, where the diagram type permits it, Users can also hide and display specific parts of the graph, such as lines of a line chart or segments of a pie chart. This is useful to remove clutter from the graphs and help with clarity.

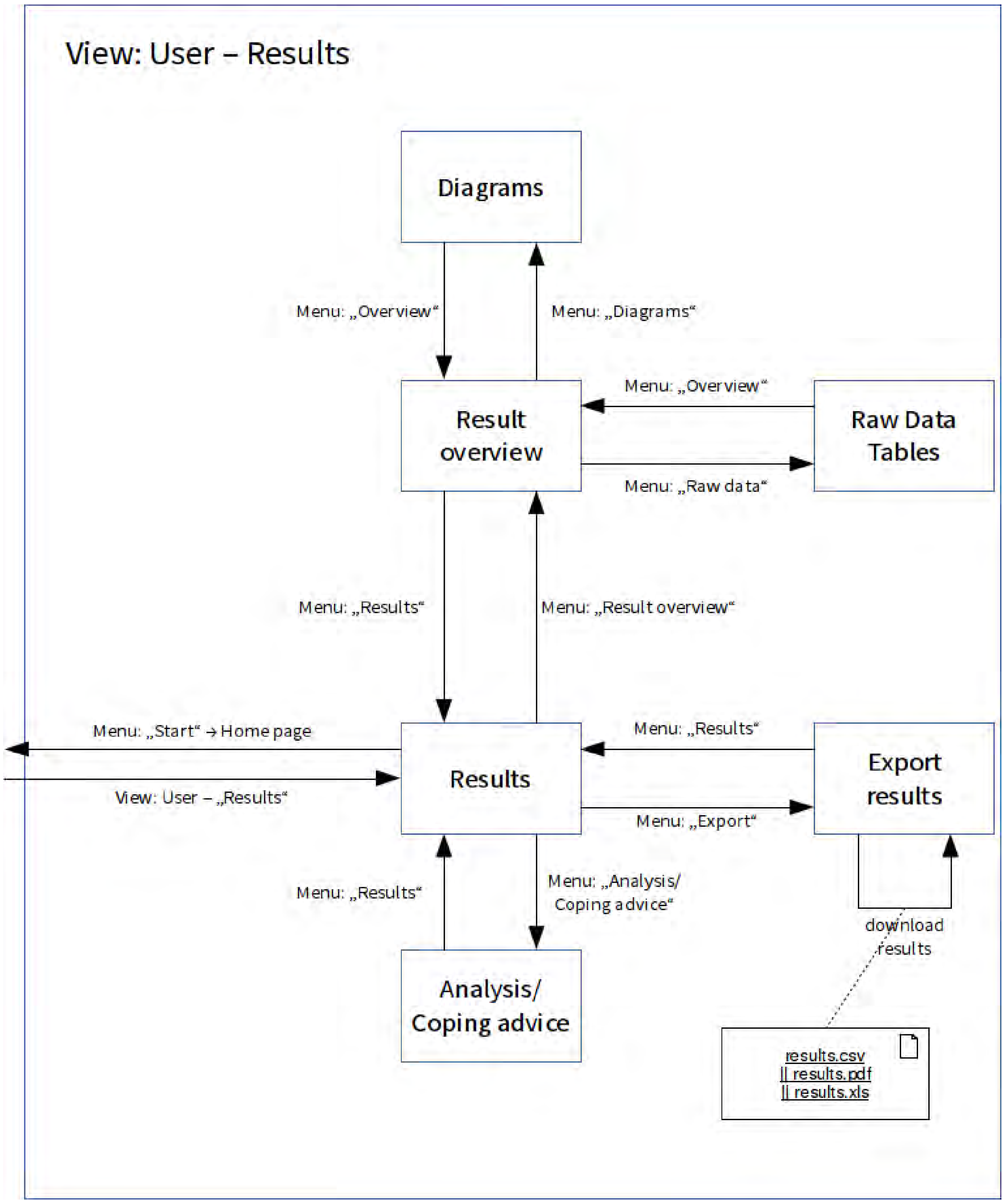


Figure 4.17: Dialog structure for the User results view

4.4.2 Page layout

This section presents a number of sketches and concepts for layouts of the UI views discussed in section 4.4.1. A few concepts will be discussed in terms of how they evolved over time, what their respective advantages and disadvantages are and what designs were finally chosen for the implementation.

Note that while wide-screen aspect rations have become the most common form factor for desktop and laptop computers in recent years [35], the sketches below are in a square-like form factor. This is because most websites still do not utilize the entire screen width on wide-screen devices and are designed with a slimmer footprint in order to display correctly even on older screens with a 4:3 aspect ratio. Furthermore, websites' width is often limited to keep text lines shorter. Longer text lines are harder to read because the visual task of "jumping" from the end of one line to the beginning of the next becomes more difficult and error-prone. Some research suggests a length of 55 characters per line for text on electronic devices [5].

Guest home page

Figure 4.18 depicts the rough design chosen for the home page as seen by guest users. It contains nothing more than a large logo for the system, a minimal sign-in form below it and a small footer at the bottom of the page. Normally, the home page would also contain short, introductory texts to present the project's purpose and functionality to visitors, but since this iteration of the system is built as a prototype, the most important part of the home page is the sign-in form. The footer contains copyright information and links to the contact form. Since guests cannot use of the system's actual functionality, the guest home page is the only view without a menu.

Navigation and page layout

While menus along the top of the screen, as depicted in figure 4.19, are arguably the most common type of navigation on the web, side menus have become more common recently. This trend is most likely due to the popularity of smart phones and tablets, which often employ this type of menu due to the commonly used *portrait* orientation [12] and limited screen real-estate. With side menus, the empty space on the sides of the main page content can be utilized. However, as can be seen from section 4.4.1, the navigation paths in this project are always fairly simply. There are only about three menu items in the navigation at any time. This means that a lot of the side menu would remain empty.

For the reasons discussed above, the more traditional top menu shown in 4.19 was chosen as the navigation for this version of the client interface to keep it simple and clutter-free. The menu contains the system's logo on the far left, which also acts as a link to the home page, next to that are navigation links and the sign-out button is aligned with the right edge of the menu. The

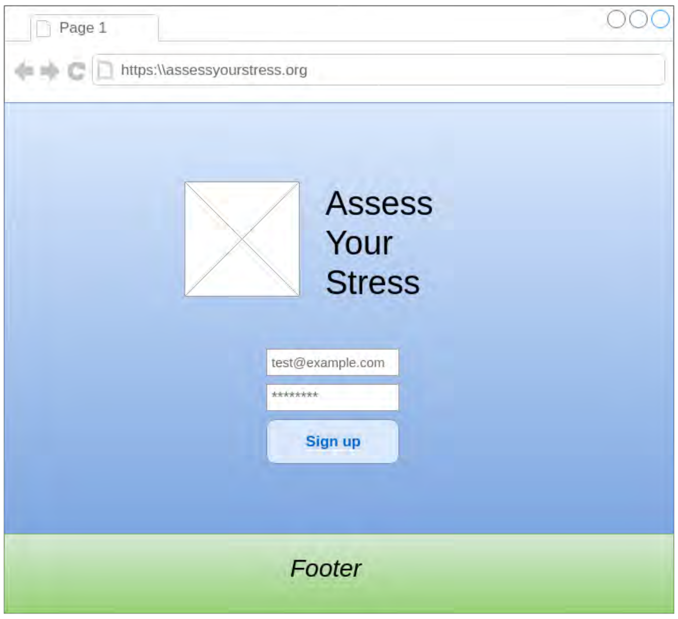


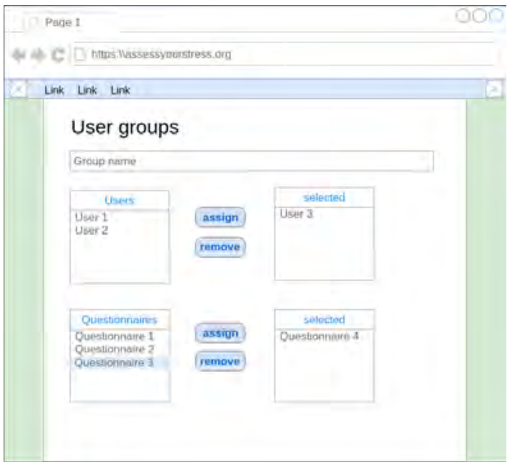
Figure 4.18: Sketch for the guest home page view

main content is hosted in a centered pane with fixed width and flexible height, but at least the full screen height. If the client device's screen is wider that the content pane, a background image spans the remaining space. The top menu along with the minimal content pane is a suitable choice for requirement UI002, i.e. that the UI should be as simple and minimalistic as possible.

UserGroup editor

The UserGroup editor is a fairly simple form, shown in figure 4.19. It only contains a text field for the UserGroup's name and two pairs of selection boxes for Users and Questionnaires. The boxes on the left contain lists of all Users and Questionnaires. The boxes on the right display the User and Questionnaires that are currently assigned to the UserGroup.

Since the number of Users and Questionnaire can be quite large, the selection boxes can also be searched using the text field above them. The potentially large number of records is also the reason why the selection box is not a standard HTML5 multiple-select box, since it would be hard to keep track of which Users are assigned to the group by the selection in the box alone. Additionally, since multiple-select boxes require users to hold the Control key to select multiple items, users might accidentally deselect everything if they accidentally let go of the Control key. This is the reason the right-hand side boxes were introduced. Administrators select the items that they would like to assign to the UserGroup and then click the *assign* button to send the selected items to the list of assigned items to the right of the list. Items can be removed from the UserGroup by selecting them in the list on the right and clicking the *remove* button. Once the desired changes are made, they can be submitted to the server by clicking the *submit* button at the bottom of the form.



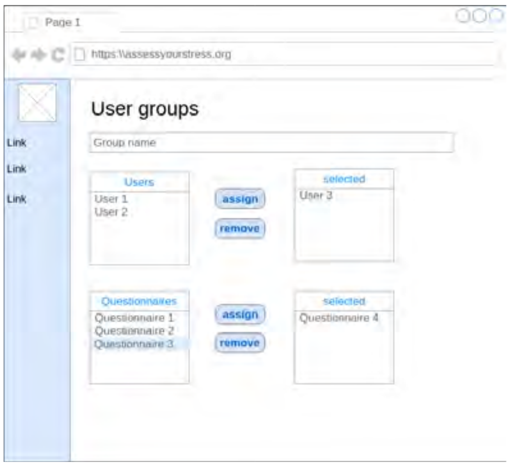


Figure 4.19: Sketch for the user group editor with top menu

Figure 4.20: Sketch for the user group editor with side menu

Questionnaire editor

Figures 4.21, 4.22, and 4.23 show three variations of the Questionnaire editor.

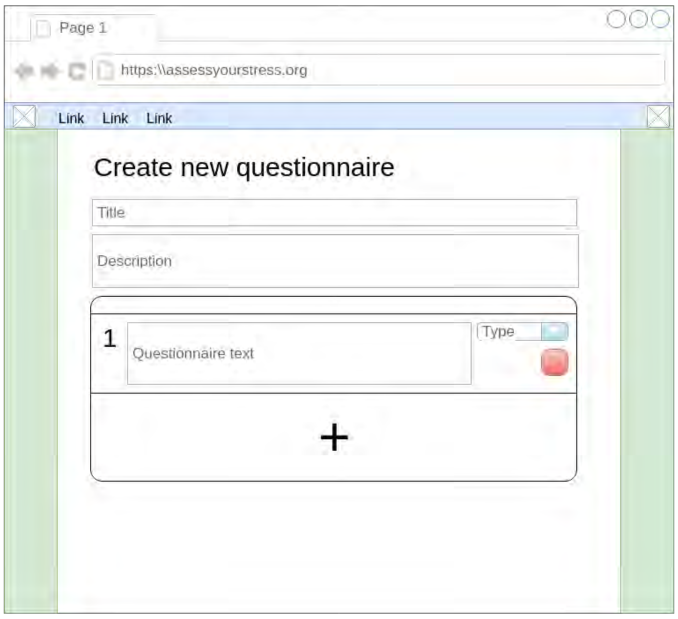


Figure 4.21: Sketch for the questionnaire editor version 1

Figure 4.21 represents the first iteration of the Questionnaire editor's design. It is the most simple of the three versions. Text fields for the Questionnaire's title and description are situated below a heading. Below that, there is a bordered section that contains the Questions. Since the number of Questions is not fixed, this list must be able to grow dynamically. The first Question is already created, but its type and content must be filled out. The type is chosen through a drop-down menu on the right. The large text box is where the Question's text must be entered. Other controls are displayed on demand, depending on the Questionnaire's type. Each Questionnaire box also contains a *delete* button that removes the Question from the list. New Questions can be added with the large *add* button at the end of the Question list. The Questionnaire can finally be submitted with the large *submit* button at the bottom of the page once the Administrator has filled all fields. However, this first version of the Questionnaire editor does not account for some other Questionnaire properties yet, like whether the Questionnaire should immediately be published and what UserGroups it should be assigned to.

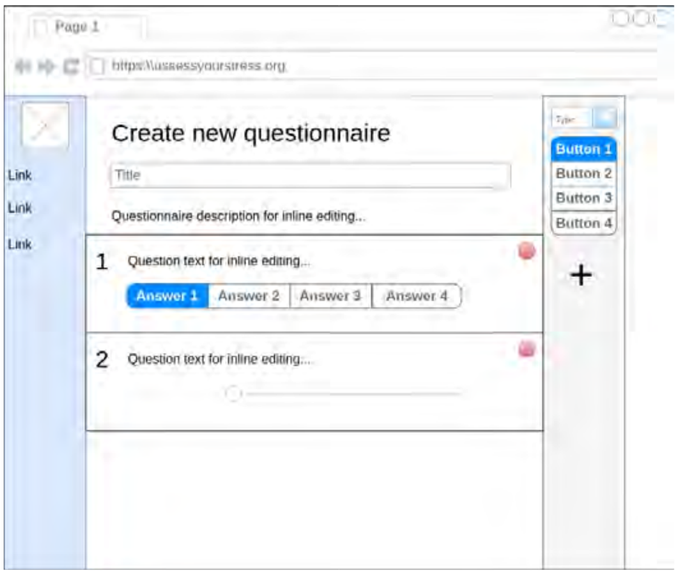

Page 1
https:\\assessyourstress.org
Create new questionnaire
Title
Questionnaire description for inline editing...
Link
Link
Link
1
Question text for inline editing...
Answer 1
Answer 2
Answer 3
Answer 4
2
Question text for inline editing...
Button 1
Button 2
Button 3
Button 4
+

Figure 4.22: Sketch for the questionnaire editor version 2

Figure 4.22 is an alternative version of the first design. This version shows how the Questionnaire editor might be realized with the side menu instead of the top menu. One problem of the first version is that the bordered section significantly reduces the space available for Questions and "squeezes" them together horizontally. This is not an ideal solution for Question types like single and multiple choice, which also contain nested, dynamically created fields for answer options. Furthermore, future Question types might be more complex and might require more space in the editor. This problem is ameliorated in version 2 because the Questions are designed to span the entire width of the content pane, which itself is also wider than the centered pane. Version 2 also includes a solution for the Questionnaire settings, that were missing from the first revision. Because these settings are general and concern the Questionnaire itself and not a specific part like the title, description or Questions, a second side bar was added on the right side of the screen. The top part of the right side bar contains all Questionnaire settings like whether it should be published immediately. The *add* button that creates new Questions was also moved to the side bar. With this layout, all the actual content of the Questionnaire, like Users will see it, is contained in the central pane while all settings and functionality is off to the side. While this partitioning helps separate content from all other properties of the Questionnaire and also gives Administrators a better understanding of what Users will actually see, the main problem is that the side bar is relatively slim. This means that it cannot contain any complex widgets that take up a lot of screen space. Future developments might extend Questionnaire and add complex functionality, which cannot comfortably be fit into the side bar. Therefore, version 2 was rejected in favor of version 3. Additionally, this type of layout works best if it encompasses the entire viewport. If the client's viewport is larger than the layout, however, the side bars would be surrounded by a background image, which eliminates the purposes of bars aligned with the edges and can look visually unpleasant. One common solution in such cases is to stretch at least one segment, in this case the content pane, horizontally to fit the viewport size. This is not a viable solution for this UI because it would only serve to create a lot of empty space and would needlessly increase the mouse pointer's travel distance.

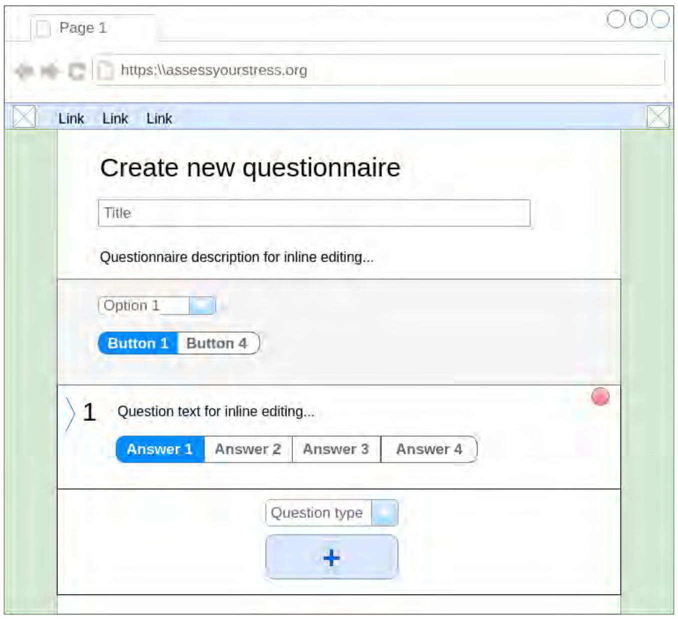


Figure 4.23: Sketch for the questionnaire editor version 3

The third and final revision of the Questionnaire editor can be seen in 4.23. This iteration returns to the top menu layout. Version 3 is essentially a combination of version 1 and 2. It adopts the linear, section-by-section layout of version 1 and combines it with the Question boxes with full content pane width from version 3. The title and description are common properties of all Questionnaire types. Therefore, they are placed at the very top, just below the page heading. The remaining content of the Questionnaire depends on its type. The Questionnaire settings, including the selection box for the Questionnaire type, is accordingly placed below the title and description segment. The third revision also adds a number a features: The text fields and labels are designed to be fairly large with plenty of negative space between elements to reduce visual clutter and create and open and light aesthetic. This has the disadvantage that each section takes up a relatively large amount of space, which can make it difficult for users to review the Questionnaire at glance. For this reason, all sections can now be collapsed, exposing only the section's heading. This also helps with another feature: Questions can be sorted using *drag and drop*. With large Questions in their expanded form, this would be difficult because the Questions would have to be dragged across a much bigger area while simultaneously scrolling the page. Another feature is inline editing. This means that the title, description and question are displayed as they would be to Users when they fill out the Questionnaire. Only when the Administrator clicks on one of those fields, they become editable. This gives Administrators a much better idea of the Questionnaire they are creating. One last minor feature is that the Questionnaire type is now chosen before the Question is added, since the Question type is one property that is likely not to be changed as much once the Question is added to the list. This reduces the amount of necessary controls in each Question box. If the Administrator does want to change the Question type, they have to remove the Question and create a new one.

Questionnaire

Figures 4.24 and 4.25 depict two alternative versions of how clients might present Questionnaires to Users.

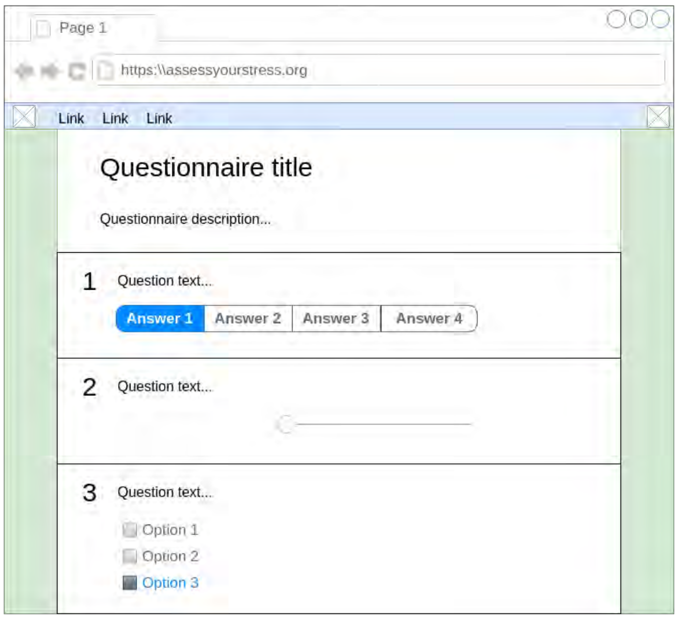


Figure 4.24: Sketch for the questionnaire view version 1

The version shown in 4.24 is the one that was eventually implemented for the prototypical client. On the top, the title and optional description introduce the User to the Questionnaire and offer advice and instructions. Below that is a simple list of all Questions created for this Questionnaire in separate boxes spanning the entire content pane's width. The Question's index is displayed on the left. The Question text is displayed near the top of the box and to the right of the index number. Below that is the dynamic part of the Question, where the type-specific content is inserted. For single and multiple choice Questions, it contains a list of answer options as check boxes or radio buttons. DateQuestions contain an HTML5 date box, ScalarQuestions contain a slider widget without initial value (cf. requirement UI001), and so on. At the bottom of the Question list is the *submit* button that Users click to finalize submission of their answers and close the Questionnaire.

Version 2, shown in 4.25, is an alternative design that explores a different layout. In this variation, the Questions are not displayed as an "infinite" list, but each Question is placed on its own page.

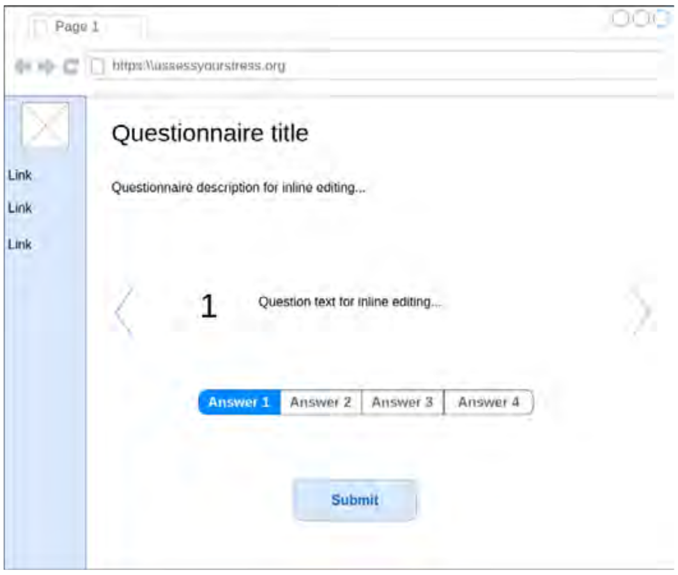

Page 1
https:\\assessyourstress.org
Link
Link
Link
Questionnaire title
Questionnaire description for inline editing...
1
Question text for inline editing...
Answer 1
Answer 2
Answer 3
Answer 4
Submit

Figure 4.25: Sketch for the questionnaire view version 2

These pages span the entire width and height of the content pane. As noted in section 4.4.2, the minimum height of the content pane is the height of the browser's viewport. This means that with this layout, Users do not have to scroll through potentially long lists of Questions. This gives each Question a lot more space, which might prove useful in the future. If new Question types are created that introduce more complex widgets for Users to input data, they might be more difficult to fit into the more compact layout in figure 4.24. On the other hand, none of the current Question types require a great amount of space, especially on large, high-resolution computer screens. This would leave a great deal of empty space, giving the UI an unfinished and unrefined appearance. Moreover, each interaction with a Question is only a few seconds long, which might make the increased amount of clicking to change pages irritating to Users. For these reasons, this is not an ideal solution for a web front end. However, the page-based approach could be suitable for future mobile clients for the system. The relatively small screen real-estate of mobile devices increases the risk of complex Question types appearing cramped and compressed. Additionally, the slim form factor forces Question lists to become longer and long lists are generally uncomfortable to scroll through on smartphones. Utilizing the entire screen might therefore be a viable option in this context. Changing pages is also faster than on desktop computers, because native mobile clients could make use of the familiar and quick *swipe* gesture to navigate through the Questionnaire.

Result diagrams

The sketch in figure 4.26 represents the *Results* view that Users access to review their Questionnaire results.

The design for this page is a natural evolution of *TYT*'s results page. Like in *TYT*, results in this project are presented as one diagram per Question. However, since *TYT* only featured one

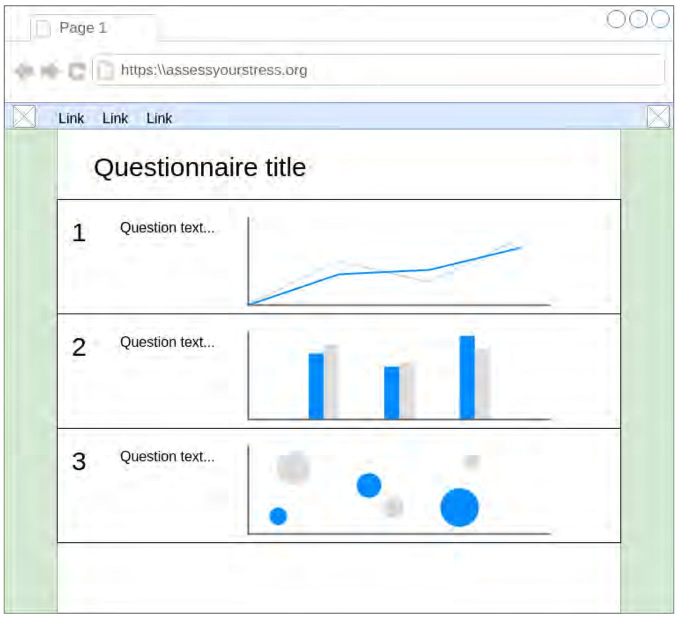


Figure 4.26: Sketch for the results view

Questionnaire with eight hard-coded Questions, the visual representation of results was slightly adjusted for the more general usage context of this iteration of the system. Users must now first choose a Questionnaire from a list of all Questionnaire that they have answered before. The system then generates diagrams for each of the Questionnaire's Questions. The diagram type is chosen based on the Question type. Answers to `ScalarQuestions`, for example, are presented as line charts as they are a suitable type of diagram to visualize value changes on a continuous, numeric scale over periods of time. Diagrams are navigable and interactive. They allow panning, zooming and showing/hiding parts of the diagrams.

Future revisions of the system will further improve result visualization by introducing combined diagrams for multiple Questions and Questionnaires. This might be realized, for instance, by enabling Administrators to create *scores* and other metrics and define logic that combines results from different Questions in that score.

4.4.3 Visual design

As mentioned above and as required by requirement UI002, the UI should be designed not to influence User's stress levels, especially not negatively. One part of this is making sure that interactions are short, immediately familiar and simple. This is ensured by the dialog structure presented in section 4.4.1 and the use of common, standard HTML elements in forms and so on. The other part is the visual design and aesthetic of the interface. This final section discusses a few of these aspects of the UI's visual design.

Spacing

As already noted in 4.4.2, the UI should be designed to appear open, friendly and light. One way to achieve such an impression is leaving ample space between visual elements and groups. Care must be taken, however, not to make the interface appear disjointed and empty with exaggerated spacing. In cases like the boxes that contain Questions in the Questionnaire view, subtle drop shadows can also be used to make interface elements appear light and spacious.

Text

The typeface used for texts also has an enormous impact on how User's perceive the interface. For this system, a clean, modern, sans serif font with slim glyphs will be chosen. Some examples for such typefaces are Helvetica, Roboto, and Source Sans. While the text color depends on the usage context, the base text color will be a dark shade of grey, since solid black text tends to appear heavy and jarring due to high contrast. For cases like error or success messages, text color should be coordinated with the background color. Letter spacing and line height will also be slightly increased to emphasize the open and light aesthetic. Italics and bold text will generally not be used, or only in rare exceptions, such as table headers.

Color

The UI's color palette plays the perhaps most important role in defining the interface's visual character. Since it should appear calm, relaxed, open and inviting, highly saturated colors and jarring contrasts must be avoided. It is also a *best practice* in usability to limit the number of used colors to a minimum. Since warm colors like red, orange and yellow are generally perceived as stimulating and arousing, they will be avoided in the general design of the UI and will be reserved to attract users' attention, like error messages or warnings. Cold colors like blue and green are usually perceived as calm and relaxed and will therefore be the main accent colors of the UI. In this aspect, the new iteration of the system matches the design decision made for the original *TYT* system. Large areas will be kept white or very light grey, possibly with a subtle, geometrical pattern in the background to give the interface more texture. Any used colors should be somewhat desaturated, as pastel colors are more unobtrusive and suitable for larger surfaces and backgrounds. Saturated colors should be reserved to convey information like signalling destructive action, success, errors and so on. The large background image should be subtle so as not to divert users' attention from their actual tasks. For example, an understated geometrical wave pattern in light shades of blue and green that fades to white could be subtle enough not to be distracting and visible enough to enrich the UI with some texture. The combination of light blue and white with a slight color gradient also evokes an association with the sky and clouds, further adding to the calm and open tone the interface should have.

5 Prototypical implementation

This chapter summarizes the project's prototypical implementation. It first introduces the software that was used to realize the system. The second part of this chapter then presents the implementation of the system. After a general discussion showing how the requirements and design described in chapters 3 and 4 were translated into working code, the rest of this chapter highlights a selection of components that exemplify the contrast with and improvement over the predecessor framework *TYT*. The focus of this discussion is particularly placed on the issues of language support, version control and class inheritance.

It was mentioned several times throughout this document that the implementation of this system only represents a prototype of the final product. Since the system is built from the ground up instead of building upon its predecessor, *TYT*, the focus of the project was to create a clean and improved concept for the new iteration of the framework. As was explained in the introductory chapter to this document, the main goals of this iteration are the creation of a flexible platform that can easily be adapted to various applications instead of building another specialized variant of *TYT*. The main areas for improvement were the system's architecture itself, which was presented in chapter 4, and the issues of language support, class inheritance and version control. Approaches for these three issues each went through several iterations, which will also be presented.

5.1 Utilized software and technology

This section is provides an overview of the platforms, frameworks, software packages and other technologies that were used to implement this system.

5.1.1 Laravel 5

Like its predecessor *TYT*, this system is built on the so-called *LAMP* stack: *L*inux (operating system), *A*pache (application server), *M*ySQL (database management system), and *P*HP (application programming language). Laravel is a framework for web applications written in PHP. It was created by Taylor Otwell and first released in 2011. The framework shares its philosophy and many of its core concepts with the web framework *Ruby on Rails* (also *RoR* or *Rails* for short), which is written in Ruby. Both of these software packages were created to make web development faster, easier and more enjoyable for developers. To this end, the toolkits are built on a few core principles:

Convention over Configuration Instead of forcing developers to manage and configure every single aspect of the web application they are building, Laravel ships with a number of powerful packages, which are configured with sane defaults that are suitable for most projects to create rapid prototypes. The Laravel core handles repetitive and complex backend tasks like CGI, URL parsing, routing, database connections, and so on. All of this enables application developers to focus on building the actual application instead of investing precious development time solving generic problems that are often very similar across projects. This does mean, however, that frameworks like Laravel or Rails make assumptions on how developers should tackle specific tasks and implement solutions. For example, the *Eloquent* ORM assumes that all *models* have an `$id` attribute as their primary key. If such restrictions do not satisfy the project's requirements, all modules can be configured as needed. If this still does not solve the problem, the built-in functionality can be replaced altogether with custom solutions. While this is all possible, it is recommended to conform to the framework's conventions as this significantly accelerates and simplifies the development process, which is particularly critical for rapid prototyping.

DRY – Don't Repeat Yourself This is a general coding principle, which stipulates that solutions should only be implemented once. Writing code that serves the same or similar purposes in different places needlessly increases the code base, diminishes the application's maintainability, makes the code more difficult to read and introduces *code smell*. Instead of solving the same problem multiple times, code should be as generic as possible within reasonable limits. This principle also applies to primitive values. Especially the use of *magic numbers* is strongly discouraged. Both Laravel and Rails encourage DRY coding.

REST The REST paradigm and its advantages were discussed in detail in 5.1.4. Both Laravel and Rails encourage the use of REST for the application's public interface. One of the reasons for this is that REST is a natural fit for the *Convention over Configuration* principle as it defines how resources should be accessed through URIs, which gives the interface routes a predictable structure. *RESTful* routes can easily be created in Laravel using the `Route::resource()` method.

MVC Another assumption that Laravel makes for the developer concerns the system's architectural pattern. Laravel applications are by default based on the *Model–View–Controller* (*MVC*) pattern. This pattern splits the system into three general tiers: The Model contains all business logic and handles data persistence. The View provides the user interface by presenting data from the Model and handling user input. The Controller is the "glue" between the Model and the View. It provides interfaces for the View that abstract from data access on the Model. Laravel ships with classes and functionality that represent each of these three tiers. For example, Models in Laravel applications typically extend the `Model` class provided by the `Eloquent` package.

Changes and new features of Laravel 5

TYT was built on version 3 of Laravel. Between the creation of *TYT* and the beginning of this project, Laravel was steadily developed, altering some core concepts and adding new features. The current release is version 5.2. The vast differences between version 3 and 5 of the framework was one of the reasons for a complete rewrite of the *TYT* system from scratch. A few of the main changes between version 5 and its predecessors are outlined below:

Laravel 5 introduces a new folder structure. The application code is placed in the `app` folder below the project's root. Laravel makes use of the PSR-4 autoloading standard, meaning that everything in the `app` folder is automatically namespaced under the `App` namespace [20]. While prior to version 5, there was a `application` directory, the `app` directory that replaces it has somewhat different contents. Before version 5, source files for models were placed in the `application/models` folder, whereas they are now simply placed in the top-level `app` folder. *Migrations* are moved out of the former `application` and into the `database/migrations` directory. Version 5 also introduces the top-level `resources` folder that holds all static resources like CSS and JavaScript files, language files and also views.

Laravel 5 also enables *method injection* for controller methods. This is Laravel's implementation of the *Dependency Injection* pattern, which provides a method for dynamic instance retrieval. First, classes that should be instantiable application-wide are registered with a central registrar. In Laravel, this is called the `ServiceContainer`. When a class A, such as a controller, depends on another class B, this dependency can be satisfied by simply passing an instance of class B into the constructor of class A. When instantiating class A, the class B object need not be passed explicitly, because it is *injected* by the ServiceContainer. The advantage of this method is that dependencies become more transparent to application developers and the instantiation is abstracted from the actual class that is passed. This enables an interface-like approach, i.e. class A only requires an object of class B and does not have to care about the specific subclass it is receiving. Laravel provides expressive methods in the `ServiceProvider` that enable fine-grained control over which instances are injected into which consumers.

TYT relied on the *OAuth* and *Sentry* packages for user authentication [11]. Beginning with version 5, Laravel now ships with a built-in authentication mechanism. This also includes a default `User` model with `id`, `email`, `name` and `password` attributes. The basic authentication solution can be enabled with a simple `php artisan make:auth` command, which automatically registers the necessary routes and provides sign-in and sign-up views. While this bare-bones approach is often enough to quickly create prototypes, it can also be extended and customized. Details on how this was done for this project are provided below.

Laravel 5 also changes how routes are mapped to controller actions. Listing 5.1, adapted from [11], shows an excerpt from the `Home_Controller` that defines an `index` method in Laravel 3.

```
class Home_Controller extends Base_Controller
{
    public function action_index ()
    {
        return View::make('home.index');
    }
}
```

Listing 5.1: *index* method defined in a Laravel 3 Controller class

Listing 5.2 demonstrates how the same method would be written in Laravel 5.

```
class HomeController extends Controller
{
    public function index ()
    {
        return view('home.index');
    }
}
```

Listing 5.2: *index* method defined in a Laravel 5 Controller class

As the example shows, Laravel 5 does not require action methods to be prefixed with `action_` anymore. The above listings also include some other, minor differences: Between version 3 and 4, Laravel has adopted the PSR-0 and PSR-1 standards [33, 21, 22], one of the effects being that all identifiers that were required to be noted in *snake_case* are now noted in *CamelCase*, like `Home_Controller` and `HomeController`. The base class that controllers extend was changed from `Base_Controller` to `Controller`. Lastly, the new version of the `index` method makes use of the `view()` helper method instead of the static `View::make()` method.

Concepts and features

While a complete introduction to the Laravel framework is far beyond the scope of this document, some of its major concepts and features, which characterize the framework from an application development perspective, are outlined here.

Laravel features the **Artisan CLI**. This interface is used for all sorts of tasks and can be extended with custom commands, which greatly increases the workflow of web application development. Artisan ships with commands that facilitate a number of common tasks for Laravel developers. These include creating Models, Controllers, and Migrations, displaying a list of routes, resetting, seeding, or migrating the database, and so on. External packages often also offer custom Artisan commands, for example to export views from the packages etc. For example, the command in listing 5.3 creates a new migration file that creates a `questions` table:

```
php artisan make:migration create_questions_table --create=questions
```

Listing 5.3: Artisan command that creates a *questions* table

Developers can construct their application's database incrementally using **migrations**. Migrations are class files that define how the database should be defined. They contain an `up()`, which is run by Laravel when the database is *migrated*, and a `down()` method, which is run when the database is *rolled back*. The `up()` method generally adds or modifies columns to tables, while the `down()` method removes columns or destroys tables. With this concept, the database schema does not have to be defined in one go, but can instead be constantly modified, even in production. The `Schema` class provides a variety of methods to create, destroy, and modify tables, columns and indexes.

Listing 5.4 shows the migration that creates the `roles` table. The concept of roles in this system is presented in section 3.1. The migration demonstrates how columns are created using methods for each data type on a `Blueprint` object. It also shows how the `down()` method destroy the `users` table when the database is rolled back. This particular migration creates a `roles` table with an auto-incrementing integer columns `id`, which is the primary key, a string column `name`, date columns for creation and update `timestamps`, and an index for the `name` column that ensures uniqueness.

```
class CreateRolesTable extends Migration
{
    public function up ()
    {
        Schema::create('roles', function (Blueprint $table)
        {
            $table->increments('id');
            $table->string('name')->unique();
            $table->timestamps();
        });
    }

    public function down ()
    {
        Schema::drop('roles');
    }
}
```

Listing 5.4: *CreateRolesTable* migration

The Laravel **Router** enables developers to map routes to arbitrary controller methods. However, if the system makes use of a RESTful API, developers can leverage *Convention over Configuration* and simply define all RESTful routes for a resource using the `Route::resource()` method. The controller then simply needs to implement the standard REST methods like `index` etc. However, routes can also be defined per HTTP verb, e.g. using `Route::get()` or `Route::post()`, for all HTTP verbs with `Route::any()`, or for multiple verbs with `Route::match()`. The Router also supports route groups, prefixes, and nesting. Groups are useful for applying *route middleware*. Middleware, in the context of Laravel, can be understood as a filtering mechanism for incoming and outgoing requests and replies. It can be used for authentication, session management, adding HTTP headers and so on.

Listing 5.5 shows an example of route definitions that demonstrates several of the above-mentioned features. It creates a group, applies the `web` middleware to it, and adds the `api` prefix. By defining a prefix, all routes defined with in the group are automatically registered with that prefix. The example defines two routes, one login route and logout route. Since the group is prefixed with `api`, the resulting routes are `[RootURL]/api/login` and `[RootURL]/api/logout`. The login route is only accessible when the request is sent using the HTTP `POST` verb and the logout route is accessible with `GET`. Both route definitions receive an option array as their second parameter, which each contains a name for the group using the `as` option and a mapping to a controller method using the `uses` option. The third definition is an example for the definition of RESTful routes. The `Route::resource()` function registers all REST routes using the given route fragment, in this case `questionnaire`, and adds all needed action methods to the given controller, in this case `QuestionnaireController`. The final parameter is an option array, in this case with the `except` option, which is used to exclude particular actions like the `destroy` action here.

```php
Route::group(['middleware' => 'web', 'prefix' => 'api'], function ()
{
    Route::post('login', ['as' => 'login', 'uses' => '
        AuthController@authenticate']);
    Route::get('logout', ['as' => 'logout', 'uses' => 'AuthController@logout'
        ]);
});

Route::resource('questionnaire', 'QuestionnaireController', ['except' => '
    destroy']);
```

Listing 5.5: Sample route definitions

Laravel features the concept of **relationships**. The framework places a variety of expressive methods at the disposal of developers that can be used to define relationships between models. For instance, a Person might *have one* Account, an Order *belongs to* a Customer and *has many* Items, and so on. Relationships are particularly useful when used with the *Eloquent* ORM, which is explained below. Laravel supports *one-to-one*, *one-to-many* and *many-to-many* relationships, which may also be *polymorphic*, i.e. models can map to different classes on a single relationship.

Listing 5.6 shows an excerpt from the `User` class oh this system. The class implements a `userGroups()` method, which returns an instance of the `Relation` class. This instance contains a collection of related models as defined by the call arguments of the relationship method that returns that instance. User and UserGroup are linked via a *many-to-many* relationship—Users can be part of any number of UserGroups and UserGroups contains any number of Users. This type of relationship requires an intermediary table that holds the foreign keys to the related records. The second call argument of the function is the name of that table, in this case the `user_group_memberships` table. The relationship function in the `UserGroup` class uses `hasMany()`, which is the counterpart of the `belongsToMany` method.

```php
class User extends Authenticatable
{
    public function userGroups ()
    {
        return $this->belongsToMany(UserGroup::class, 'user_group_memberships
            ');
    }
}

class UserGroup extends Model
{
    public function users ()
    {
        return $this->hasMany(User::class, 'user_group_memberships');
    }
}
```

Listing 5.6: Many-to-many relationship between User and UserGroup

The excerpt from an old version of the `Questionnaire` class shown in listing 5.7 demonstrates the usage of polymorphic relationships. Since many Users fill in Questionnaires, each Questionnaire has a relation with many `AnswerSet`s. However, section 4.2.1 defines a variety of different Questionnaire subclasses. Without polymorphic relationships, the `AnswerSet` class would have to define a `hasMany()` method for each Questionnaire subclass. This is detrimental to extensibility and causes lots of duplicated code, because it creates many points in the code that would have to be updated if any Questionnaire subclasses are renamed, removed are new ones are created. Instead, the `morphTo()` and `morphMany()` functions are used to define a polymorphic relationship between Questionnaires and AnswerSets. To achieve this, the `answer_sets` table has to contain an integer `questionnaire_id` column and a string `questionnaire_type` column. The type column contains the fully qualified class name of the Questionnaire and the `id` column contains its primary key value.

```
class Questionnaire extends Model
{
    public function answerSets ()
    {
        return $this->morphMany(AnswerSet::class, 'questionnaire');
    }
}

class AnswerSet extends Model
{
    public function questionnaire ()
    {
        return $this->morphTo();
    }
}
```

Listing 5.7: Polymorphic one-to-many relationship between Questionnaire and AnswerSet

Another aspect of the *Convention over Configuration* principle is the abstraction from database access. Laravel features a powerful *object-relational mapping* system (*ORM*) named **Eloquent**. Eloquent provides a fluid and expressive interface that developers can use to, retrieve, manipulate, save, and delete instances of the models they have defined in their application. Instead of having to construct SQL statements manually, developers can query the database with chainable methods that return instances of the application's models or collections thereof. This especially comes in handy in combination with relationships, introduced above. The provided Eloquent methods are far too many to list in this document, the listings below demonstrates only a few of them.

In listing 5.8, the `scopeOptional()` method defines a *scope* on the `Question` model. Scopes allow developers to extend Eloquent with their own, chainable methods that modify the current query. Here, the scope filters the query for `Question`s that are *optional*. This is achieved using the `where` Eloquent method, which receives a column name and a value as its parameters. The `where()` function is the equivalent of the `WHERE` keyword in SQL. To retrieve only the `Question`s that are optional, the `where()` method reduces the query to records whose `is_optional` value is `true`.

```
class Question extends Models
{
    public function scopeOptional ($query)
    {
        return $query->where('is_optional', true);
    }
}
```

Listing 5.8: Question scope that filters for optional Questions

Listing 5.9 shows how Questionnaires are retrieved for Questions. The design in section 4.2 states that Questionnaires and Questions are not directly related, but through an intermediary `QuestionnaireContent` model. While Laravel does offer *has-many-through* relationships to access models that are related through an intermediary model, this cannot be utilized in this case, because both `Question` and `Questionnaire` are polymorphic. The solution to handle the indirection manually as demonstrated in listing 5.9[1]. First, the `Question`'s related `QuestionnaireContent` is retrieved with `$this->questionnaireContent()->get()`. The `isEmpty()` method is used to determine whether the method has returned any records, i.e. if the the `Question` currently has any `QuestionnaireContent`. When there is a related `QuestionnaireContent`, the `questionnaire()` method can be used to get the related `Questionnaire`, because the `QuestionnaireContent` class declares a one-to-many relationship with the `Questionnaire` class. If there are any related `Questionnaire`s, the function returns the first match using the `first()` method. There are also some references to a `$withTrashed` variable strewn in. This optional flag enables developers to signal whether they would like to include *trashed* records in the query. Trashed records are records that were deleted using **soft deleting**. When soft deleting is enabled on a model, its instances are not actually removed from the database when they are deleted. Instead, the model's table contains a `deleted_at` column which saves the date and time when the record was "deleted". This makes the record a *trashed* record, which Laravel filters by default. Trashed records can, however, be retrieved using the `withTrashed()` scope.

```
class Question extends Models
{
    public function questionnaire ($withTrashed = false)
    {
        $questionnaire_content = $this->questionnaireContent($withTrashed)->
            get();
        if ($questionnaire_content->isEmpty())
            return null;

        $questionnaire = $questionnaire_content->first()->questionnaire(
            $withTrashed)->get();
        if ($questionnaire->isEmpty())
            return null;

        return $questionnaire->first();
    }
}
```

Listing 5.9: Retrieving a Question's parent Questionnaire

[1]Side note: The excerpt of the Question class in listing 5.9 is actually an older version of the class. One of the reasons it was revised is the less-than-ideal handling of inheritance. The different approaches and final solution to this issue that were explored in this project are documented in section 5.2.3

Laravel also comes with a flexible solution for **input validation**. The base `Controller` class includes a Trait that provides the `validate()` method. This method can be used in controllers to define validation rules for incoming requests in a declarative fashion. Rules are defined as an array in the form `'attribute' => 'rules'`, where `rules` is a string of validation keywords and arguments delimited by the pipe character (`|`). The framework ships with a great number of validation keywords and it also supports custom validation rules. If the request does not pass validation, Laravel automatically returns an error message to the client. Developers may also define *form requests*. These are classes that extend the `Request` class and can define validation (and authorization) rules just like described above by implementing the `rules()` method. Controllers can then simply *inject* (cf. explanation of *Dependency Injection* in section 5.1.1) the newly created Request class into a method. Laravel automatically calls the `rules()` method on the Request class and handles validation accordingly.

Listing 5.10 shows an abridged version of the input validation used for Questionnaire creation. The keys of the validation rule array are the Questionnaire's attributes, like `title`, `description`, and so on, and the values define the validation rules. The `required` rule indicates that a non-empty value must be given for the attribute. There are rules for each data type, like `string`, `boolean`, etc. Validation of nested attributes for related models is also possible, as demonstrated by the `questions` keys. The `questions` attribute is required to contain an array. This array is expected to contain arrays of Question attributes. Nested attributes in collections are validated with the asterisk character (`*`). In this case, the validation rules require that each Question have a valid Question type as returned by the `Question::getSubclasses()` method. The rule for `questions.*.min` exemplifies more complex validation logic: It requires each collection of Question attribute to contain a numeric `min` attribute, but only if that Question's type is defined as `question_scalar`.

```
class QuestionnaireController extends Controller
{
    public function store (Request $request)
    {
        //...

        $this->validate($request,
        [
            'title' => 'required|string',
            'description' => 'string',
            //...
            'questions' => 'required|array',
            'questions.*.type' => 'required|in:'.implode(',', array_keys(
                Question::getSubclasses())),
            'questions.*.min' => 'required_if:questions.*.type,
                question_scalar|numeric',
            //...
        ]);

        //...
}
```

Listing 5.10: Validation of new Questionnaires

This project makes use of a role system for users (cf. sections 3.1, 5.2.3). These roles determine which actions may be executed by which types of users. Laravel supports this functionality with its **authorization** solution. Like validation, the `Controller` class comes with built-in authorization methods. The basic idea is that developers define *abilities* that can then be checked against to allow or disallow an action. This can be achieved in a variety of ways, including using the `Gate` *facade*, built-in authorization methods, and *Policies*.

The most basic variant of authorization, which is the basis on which the alternative methods build, is demonstrated in listing 5.11. First, the ability `manage-questionnaire` is defined in the `AuthServiceProvider`. This ability is granted to users who should be able to manage questionnaire, i.e. create, edit, and delete them. It is then used in a controller to verify that the user attempting to save a Questionnaire is allowed to do so. In this context, only Administrators are allowed to manipulate Questionnaire. Therefore the ability method returns the user's `is_admin` flag. This happens in the `store()` method of `QuestionnaireController` (the same method as in listing 5.10). First, the `user()` method is used to ensure that the user is registered and signed in. The `cannot()` method, provided by the `Authorizable` Trait, is then called on the User with the `manage-questionnaire` ability as its parameter. This method returns `true` if the User is *not* allowed to `manage-questionnaire`s, i.e. if they are an Administrator. If the user is in fact not an Administrator, the method `redirect()`s to the application's root URL and includes a localized error massage.

```
class AuthServiceProvider extends ServiceProvider
{
    public function boot (GateContract $gate)
    {
        $gate->define('manage-questionnaire', function ($user)
        {
            return $user->is_admin;
        });
    }
}
class QuestionnaireController extends Controller
{
    public function store (Request $request)
    {
        if (!$request->user() || $request->user()->cannot('manage-
            questionnaire'))
            return redirect()->route('root')->withErrors(trans('auth.
                unauthorized'));
        //...
    }
}
```

Listing 5.11: Authorization example: saving Questionnaires

The features outlined above are only a small sample of the tools included in Laravel. The framework also comes with the *Blade* templating engine, modules for caching, queueing, asset management, billing, cryptography, and many other common tasks in web development. For a complete introduction to Laravel, refer to the official documentation [19].

5.1.2 AngularJS

The front end is implemented using the *AngularJS* framework. This is a departure from *TYT*, which provided its website using Laravel's built-in tools and served it in a traditional *request-response* fashion. This project uses version 2 of Angular, which is a complete rewrite of its previous code base.

AngularJS (or simply Angular) is a front end framework based on JavaScript and developed by Google. Starting out as a research project, Angular has now been widely adopted for the development of *single-page applications*. The advantages and disadvantages of single-page applications have already been discussed in section 4.2.3. In summary, single-page application break with the traditional request-response pattern to offer a more fluid and *desktop-like* user experience for web services. This is accomplished by communicating with the server's public API using AJAX and manipulating only the relevant parts of the DOM with JavaScript instead of reloading on every request. Some important features of Angular 2 are outlined below.

Concepts and features

By default, Angular 2 applications are developed in **TypeScript**. TypeScript, developed by Microsoft, is a superset of JavaScript that adds type checking to this traditionally "untyped" language. TypeScript is built on ECMAScript 2015, the most recent version of ECMAScript, which is the standard that JavaScript is based on. ECMAScript 2015 adds a variety of new features to traditional JavaScript like arrow functions, function parameter default values, string interpolation, modules, etc. Angular relies heavily on a number of those features.

Components are the central building block of Angular applications. Each Component implements the application logic that controls a defined portion of the website by handling the relevant data and functions that manipulate those data. In terms of the *MVC* pattern, Components correspond to the Model. Accordingly, the View is represented by **Templates**. Templates are reusable HTML fragments that provide the user interface for a given Component. Angular supplements

HTML with its own *template syntax*, which facilitates the development of dynamic user interface fragments. The template syntax provides keywords for many common use cases, like `ngFor` for repetition of mark-up fragments or `ngIf` and others for conditional inclusion of mark-up.

The primary way in which Components manipulate Templates and vice versa is **data binding**. The template syntax includes concise notation that binds text and input elements in the UI to data in the Component. Developers that do not utilize frameworks like Angular to build a dynamic web application have to manually implement functionality that tracks changes in the UI and synchronizes internal values accordingly, or updates the UI to reflect internal changes. This approach leads to redundant, repetitive code, is error-prone, and difficult to maintain. In Angular, developers can make use of data binding to delegate all of this work to the framework. Four types of data binding are supported:

Interpolation is a form of *one-way data binding* that inserts a value from a Component and automatically keeps it updated. Interpolation uses the `{{}}` notation. For example, mark-up like `<span>{{questions.count}}</span>` would render the current number of Questions managed by the Component and would automatically reflect any changes to that number.

Property binding is used to pass data into child Components as input. For example, a `QuestionnaireList` Component might consist of several `QuestionnaireDetail` Components, one for each Questionnaire currently in the list. Within the mark-up for the `QuestionnaireList`, the `QuestionnaireDetails` could be defined as shown in listing 5.12. The `ngFor` keyword is used to repeat the `<div>` element for each element in the Component's `questionnaires` property. The `let questionnaire of questionnaires` micro-syntax assigns each value in the `questionnaires` collection to the `questionnaire` variable on each loop. Data binding comes into play when the `questionnaire-detail` Component is inserted. The `QuestionnaireDetail` child Component (not depicted) defines an input property named `questionnaire`, which is bound to the `questionnaire` variable, set in the `ngFor` loop, using the `[]` notation. The fact that the child Component's input value is bound to a property of its parent Component means that the `ngFor` loop does not only dynamically create the list once, but it continuously tracks the `questionnaires` property and updates the list accordingly. For example, when this Component is used in the Administrator's Questionnaire list and an Administrator deletes a Questionnaire, the list is automatically re-rendered to remove the deleted Questionnaire, because the `questionnaires` property has changed.

Listing 5.12: AngularJS property binding example

```
<div *ngFor="let questionnaire of questionnaires">
  <questionnaire-detail [questionnaire]=questionnaire></questionnaire-detail>
</div>
```

Event binding works similarly to property binding, but instead of data flowing from a parent Component to a child Component, information flows from child Components or other DOM elements to parent Components in the form of *events*. The `()` notation is used to assign handler functions, implemented by the Component, to events fired by a child Component. The example in listing 5.13 shows a simply example of event binding. A `<button>` element is defined and using the `(click)` notation, the parent Component's `submit()` function is assigned as the handler method for the `click` event.

Listing 5.13: AngularJS event binding example

```
<button (click)="submit()"></button>
```

The three techniques explained above are all examples of *one-way binding*, where information flows in one direction. Angular also supports **two-way binding**, where, as the name suggests, information can flow in both directions. The notation for two-way binding is a combination of property binding and event binding, i.e. `[()]`. In the example in listing 5.14, an `input` field is bound to the `title` property of a `questionnaire` object stored in the Component. The `input` field is automatically filled with the `questionnaire`'s `title` value and any values to the `input` field are automatically passed on to the linked object.

Listing 5.14: AngularJS two-way binding example

```
<input [(ngModel)]="questionnaire.title">
```

Like Laravel, Angular also makes use of the **Dependency Injection** pattern. This is mostly used to inject **Services** into Components. Services are regular ECMAScript classes that provide some sort of service to any Component that injects it. Most commonly, Services are used to load model data from the server, handle logging tasks, etc. The Angular framework itself uses Services frequently and developers are encouraged to do so as well to extract any functionality from Components that is not directly related to their main purpose. Components that require a Service must register the Service in the Component's `@Component` *Decorator* by adding the Service's class name to the array under the `providers` array key. The Component is then aware of the Service and it can be injected by listing an instance of the Service class as an parameter to the Component's constructor method. When the Component is instantiated, its *Injector* creates an instance of the Service and passes it to the Component.

5.1.3 JWT

JSON Web Tokens (or *JWT*) is a standard (RFC 7519) for claim negotiation using encrypted JSON tokens [1]. JWTs (pronounced *jots*) are most commonly used for token-based authentication in the web. Claims between two parties are encoded in JSON objects with a specific structure. These objects are then encrypted and signed to ensure secure transmission and prevent tampering. JWTs consist of three parts: Header, Payload, and Signature. The Header contains meta information about the JWT, like type and used encryption algorithm. The Payload is the main part of the JWT as it contains the claims. The standard defines a number of three-character-long keywords for *reserved claims*, e.g. `iss` (issuer), `exp` (expiration date), etc. Developers can also define their own claims, which is useful for user IDs, user roles, and any other information that the application may require. Finally, the Signature is combination of the encoded Header, Payload, and a secret value, that is then hashed with the specified algorithm, typically HMAC SHA256. All three parts of the message are Base64-encoded and concatenated with a dot character (`.`) as a delimiter. The resulting token string is finally transmitted over HTTP in the `Authorization` header in the form `Authorization: Bearer [TOKEN]`.

JWTs have a number of advantages over comparable techniques like *Simple Web Tokens* (*SWT*). First and foremost, thanks to the compact size of JSON objects, JWTs are relatively short—so much so that they can even be transmitted in the URL as query strings. The small size makes them especially useful for mobile applications where the minimization of data traffic can be crucial. Unlike SWTs, JWTs can be asymmetrically signed with any cryptographic hashing function that both parties agree upon. Finally, as mentioned in section 5.1.4, encoding and decoding JSON objects is relatively straightforward and libraries for that purpose are available for most common programming languages. In contrast, SWTs encode verbose XML files, resulting in significantly longer tokens and requiring computationally heavy XML parsing.

The following example JWT is adapted from [2]. Listing 5.15 shows the Header and Payload parts of the JSON object in plain text. The resulting JWT can be seen in listing 5.16.

Listing 5.15: Example JWT Header and Payload in plain text

```json
{
  "alg": "HS256",
  "typ": "JWT"
}
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

Listing 5.16: Resulting JWT with secret string "secret" and hashed with HMAC SHA256

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.
TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ
```

5.1.4 JSON API

As discussed in section 4.1.1, the server for this system is implemented as a web service. It provides its functionality through a REST-like interface [7] and follows the JSON API specification [15]. The reasons for this design decision are discusses below. This section only outlines the ideas behind REST and JSON API.

REST and JSON API

Over the past decade, REST has become a popular, lightweight alternative to traditional, verbose communication protocols for web services such as SOAP over HTTP [9]. Unlike SOAP, which is independent from the transport protocol, *RESTful* APIs rely heavily on HTTP itself. REST is used to represent distinct *resources* using URIs and existing HTTP *verbs*. The REST specification defines certain conventions on how URIs should be formed to represent the resources that the server provides and what HTTP verbs should be used to access these URIs. While not all use cases in typical, modern web services can be mapped cleanly to this specification, REST is generally an attractive and simple way to implement interfaces for web services. In addition to REST, the API for this system is also designed to comply with the *JSON API* specification [15]. This is another improvement over the original *TYT* framework, which provides a custom JSON-based API. Custom APIs are flexible, but are more difficult to maintain, extend and modify. New developers, which might wish to extend the interface, first have learn and understand the way the interface is built and intended by its original developer, which increases development time and complexity. A possible solution to this problem is building the API to a generic API specification like JSON API. Since the specification, which in itself also relies on REST-like patterns, defines precisely how the API routes and messages must be constructed, new developers only need to be familiar with the core principles of JSON API in order to quickly gain a general understanding of an application's API. Furthermore, extending the interface simply becomes a matter of registering the new routes whose format is already predetermined by the specification. There are also numerous off-the-shelf plug-ins for most popular frameworks for both the back end and the front end, which further abstract from the communication layer. This essentially enables application developers to simply transmit and receive objects as they have defined them for their software without having to deal with encoding or decoding. Lastly, using JSON instead of XML to encode messages sent or received through the API has the clear advantage of a far smaller footprint. As mentioned above, XML defines a verbose syntax while JSON messages are simply *stringified* JavaScript objects, which are both shorter and more human-readable than XML documents. Moreover, the decoding overhead for JSON files is next to nothing in many frameworks, while XML documents have to go through a complex parsing process to extract any useful data from them. Both the smaller document size and lower encoding overhead make JSON especially useful for mobile devices because computational resources can be fairly limited and should be used sparingly.

JSON API format

The JSON API specification defines message formats for a great variety of different use cases, contexts, and message types, which far exceeds the scope of this document. Refer to [14] for the complete specification. Listing 5.17 therefore only demonstrates a small and abridged example of a possible server response payload that conforms to the JSON API specification. The *payload* of the JSON API response is referenced by the `data` key. This key contains either one or a collection of so-called *resource objects*. The `type` and `id` properties are required for all resource objects and reference a unique resource. If the `attributes` key is included, it must be an object and can contain any valid JSON value. As the name suggests, the `attributes` object is used to represent any relevant attributes of the resource that is being described by the resource object. JSON API also defines how *relationships* are mapped[2]. The `relationships` key references a *relationship object*. It contains all information necessary to retrieve the related models from the server, like `links` and the `type`s and `id`s.

Listing 5.17: JSON API-compliant server response (abridged)

```
{ "data": [{
  "type": "questionnaires",
  "id": "1",
  "attributes": {
    "title": "Default snapshot questionnaire",
    "description": "Please answer the questions below to measure your current
        stress level."
  },
  "relationships": {
    "questions": {
      "links": {
        "self": "[RootURL]/questionnaires/1/relationships/questions",
        "related": "[RootURL]/questionnaires/1/questions"
      },
      "data": [{
        "type": "questions",
        "id": "1"
      },
      {
        "type": "questions",
        "id": "2"
      }]
    }
  },
  "links": {
    "self": "http://example.com/questionnaires/1"
  }
}]}
```

5.2 Implementation details

The previous chapters have identified and described various issues from the problem domain and design space. In this section, a selection of solutions to these issues are discussed. The chosen issues are what sets this iteration of the *TYT* concept most apart from the original implementation. Most of these solutions have gone through multiple iterations. However, for the

[2]See section 5.1.1 for an introduction to relationships in Laravel.

sake of brevity, only the dynamic translation system's development history is discussed in detail as an exemplary case. The remaining processes are shortly outlined and their final results are presented.

5.2.1 Dynamic translation system

Frameworks like Laravel usually ship with *internationalization* functionality (often *i18n* for short), which works well for static data like text on websites. However, these solutions generally do not work for dynamic data, i.e. database records that are created and edited at runtime. While *TYT* does implement an internationalization mechanism, it has a number of drawbacks: First, the functionality is specifically built for questionnaires and questions. If the translation system were to be expanded to other classes, it would have to be built again from scratch. Moreover, the translations are stored in separate tables that mirror parts of the tables whose attributes they translate. If there are any changes to the questionnaire and question tables, these changes would have to be propagated to the translation tables and to any code that relies on them as well. All in all, this approach lacks extensibility and decreases maintainability. One goal of this project was therefore to find a more flexible solution to this problem. This section discusses the approaches that were explored to this end.

Version 1

The first version of the dynamic translation system had two major aims:

Maximum flexibility The system should be easily applicable to any alphanumeric attribute type. Any class that requires translation functionality should be able to opt into it with minimum configuration and modification. The system should work seamlessly with inheritance (see section 5.2.3 for more information on inheritance).

Transparent workflow integration Developers should be able to use dynamically translated attributes like regular attributes, e.g. if the `title` attribute of the `Questionnaire` model is translated, calls like `$questionnaire->title` should work.

The following general design was created to satisfy the above-mentioned requirements:

All translated values are stored in two tables, `translated_strings` and `translated_-texts`. These tables belong to models of the same name, which are derived from the `TranslatedAttribute` model. `TranslatedAttribute` defines the relationship methods and a `content` attribute. The `content` attribute is used to store the translated values, either with a string or text data type respectively. Since different database management systems may handle data types like strings and texts differently, this enables the system to work with both types and offer uniform, type-agnostic access to translated values using the `content` attribute. Model attributes that should be translated are *not* part of their original tables. Instead, the model in question includes the

`Translatable` Trait. `Translatable` contains all functionality concerning the dynamic translation system, including polymorphic relationships to the translation tables. All that developers then need to do to make an attribute translatable is add it to either the `$translated_strings` or `$translated_texts` array on the model, depending on the attribute's data type. When the translation tables are created and the relationships between them and the model are established, the translated attribute can then be used almost as usual. Translated attributes are modified as follows:

- PHP-style attribute access returns the value for the current system locale, for example `$question->text; // => "Do you currently feel stressed?"` if system locale == 'en'
- Analogous to standard attribute access, setting attributes creates a translation for the current default locale, or updates an existing translation.
- Setting a translated attribute to `null` deletes the related translation from the translation table.
- To retrieve or set multiple translations for one attribute at once, the pluralized version of the attribute may be used.
  - Reading the pluralized attribute returns an array of all available translations, keyed with the ISO 639-1 language code. Example: `$question->texts;` might return `["de" => "Fühlen Sie sich im Moment gestresst?", "en" => "Do you currently feel stressed?"]`.
  - Likewise, the pluralized attribute can be set to an array keyed with ISO 639-1 language codes to create multiple translations at once.
  - Set the pluralized attribute to `null` to delete all existing translations for this attribute.
- `Translatable` also provides dynamic getter and setter functions to modify values for specific locales other than the current system default. These functions are available for any attribute registered as translatable and do not have to be defined by developers.
  - `$question->getText('en');` returns the English translation for the Question's `text` attribute.
  - `$question->setText('Do you currently feel stressed?', 'en');` creates the English translation for the Question's `text` attribute and sets it to the passed value, or updates the existing translation with the passed value.
  - As with dynamic attribute access, passing `null` to a dynamic setter function deletes the translation for the attribute in the given locale.
  - Translations may also deleted explicitly with the dynamic delete function: `$question->deleteText`

The described functionality was fully implemented and was found to be working as specified. Transparent attribute manipulation lets developers handle translated attributes naturally like any other PHP class attribute and the translation tables remove any specialized columns for translated attributes from the models. However, the design and implementation of this feature has several critical flaws that led to further iterations:

The `Model` class supports *whitelisting* and *blacklisting* of attributes as a countermeasure to **mass assignment** vulnerabilities [6]. The term mass assignment means setting multiple attributes on an object at once. Without any server-side protection, malicious users may *inject* any attribute, for example by modifying a user registration HTML form and adding a hidden `admin` flag set to `true`, which would effectively grant them administrative control of the system. To counter this, Laravel requires developers to either add mass-assignable attributes to the `$fillable` array on their model or exclude specific attributes from mass assignment by adding them to the `$guarded` array.

The process of "filling" a `Model` instance with attributes is "baked" into the implementation and does not provide any programming hooks before the object is actually saved to the database. Therefore, `Translatable` has to intercept and overwrite a number of methods from the Eloquent `Model` class for its functionality to be as flexible as desired. This is where most of the problems of this approach arise. When a `Model` subclass instance is created, the `fill()` method implemented in the `Model` class checks whether the passed attributes are "fillable" (either if they are listed in the `$fillable` array or not present in the `$guarded` array) and sets them on the object if they are. Attributes that are not fillable are simply ignored. This is a problem when trying to set translated attributes on an object, because they do not actually exist in the model's table. Instead, they are delegated to the translation tables. Since the model would simply reject all translated attributes as non-fillable, `Translatable` overwrites the `getFillable()` method, shown in listing 5.18, to include translated attributes and their pluralized versions in the list of fillable attributes. However, because the translations are stored in separate tables and linked to their parent models using relationships, the translation objects can only be created when the related model instance already exists. Otherwise, the translation table has no foreign key to refer to. For this reason, translated attributes have to be cached, when a new record is being created. Once the model instance is saved to the database, the predefined `saved` and `saving` events are used to retrieve the cached translations, create the translation objects and establish the relationship with their parent object. This is clearly a very roundabout, inconvenient and error-prone approach.

Listing 5.18: getFillable() from Eloquent Model overwritten in Translatable v1

```
public function getFillable ()
{
  $original_fillable = parent::getFillable();
  $fillable_with_plural = array_union($original_fillable, array_map(function (
      $attribute)
  {
    return $this->isTranslatableAttributeDefined($attribute) ? str_plural(
        $attribute) : $attribute;
  }, $original_fillable));

  return $fillable_with_plural;
}
```

The `Translatable` Trait also has to overwrite the *magic functions* `__get()`, `__set()`, and `__call()`. As an example, the overwritten `__get()` method is shown in listing 5.19. This is done to intercept attribute access and function calls and to reroute them to the translation methods described above if the attribute matches a translated attribute. Laravel does offer the concept of *mutators* for access to attributes that are not defined on the model, but mutators cannot be defined dynamically. Therefore, developers would be forced to manually implement two mutators for every translated attribute, which violates the requirement for easy extensibility. While overwriting `Model` functions does work for this purpose, it is not a robust solution. Manipulating framework code and interfering with the intended flow of data and control is likely to introduce many points of failure in the code, which are hard to locate and debug. This is especially true when the framework code is updated. Any changes to the overwritten code are then likely to break the application. This is especially true if other Traits extend `Model` with their own functionality as well, leading to conflicts between the third-party add-ons.

Listing 5.19: Overwritten magic function __get() implemented in Translatable v1

```php
public function __get ($key)
{
  // Test for attribute accessor
  if ($this->isTranslatableAttributeDefined($key))
    return $this->handleAccessor($key);

  // Test for pluralized attribute accessor
  if ($this->isTranslatableAttributeDefined(str_singular($key)))
    return $this->handlePluralizedAccessor(str_singular($key));

  return parent::__get($key);
}
```

The values in the translation tables are matched with the right models using polymorphic relationships. This removes attributes from their model's table, where they should be. Furthermore, polymorphic relationships rely on tables to save both a foreign key and the referenced table's fully qualified class name. However, since translations reference specific attributes, not just objects, the attribute name must also be stored, adding another level of indirection and requiring additional functionality to retrieve and set relationships. Moreover, translations created in the past would break if the referenced attribute names are changed.

Lastly, this version of the translation system does not support inheritance. When supplementing fillable attributes with translated attributes, only the calling class's defined translated attributes are considered. This means that inheriting classes would have to list their own translated attributes as well as all of their parents' translated attributes explicitly, which is unnatural and cumbersome.

Version 2

After version 1 of the translation was scrapped for the reasons discussed above, version 2 was developed from it. This new iteration follows the same basic principles and goals, but the implementation was modified in a number of ways.

As mentioned in the discussion of version 1, both the former solutions for inheritance (discussed in section 5.20) and dynamic translations overwrite `Model` class code to achieve their functionality. In an effort to resolve conflicts between these two modules, it was attempted to unify their functionality in one common class called `InheritableModel` instead of relying on Traits. This class was meant to take the place of `Model` when creating new application models, i.e. all models in the application inherit from `InheritableModel`, which in turn extends `Model`. While this does not change the fact that `Model`'s functionality is modified, a solution like this would be a sufficient trade-off for the scope of this project. Portions of the `Translatable` Trait were carried over into `InheritableModel`, but some problems were solved differently:

The `$translated_strings` and `$translated_texts` arrays are now declared as static. This enables all classes to only define their own translated attributes. When the class is instantiated, the class's constructor aggregates the translated attributes of the class and all its parents and adds them to the fillable attributes. This makes working with inherited, translated attributes much more natural and encourages separation of concerns between classes. The modified constructor and the `aggregateParentAttributes()` method can be seen in listing 5.20.

Listing 5.20: Constructor for InheritableModel and aggregateParentAttributes() method

```php
public function __construct (array $attributes = [])
{
  // Aggregate fillable, guarded and translated attributes from parent classes.
  $this->fillable($this->pluralizeAttributes(array_union($this->getFillable(),
      $this->aggregateFillableAttributes())));
  $this->guard($this->pluralizeAttributes(array_union($this->getGuarded(), $this->
      aggregateGuardedAttributes())));
  static::$translated_strings = $this->aggregateTranslatedStrings();
  static::$translated_texts = $this->aggregateTranslatedTexts();

  // Add all translated attributes to $appends.
  $this->append(
    array_union(
      $this->getArrayableAppends(),
      $this->pluralizeAttributes($this->getTranslatedStringsArray()),
      $this->pluralizeAttributes($this->getTranslatedTextsArray())
    )
  );

  parent::__construct($attributes);
}

protected function aggregateParentAttributes ($array_name)
```

```
{
  $array = [];

  // Traverse the chain of inheritance up to the abstract InheritableModel class
  // and aggregate all inherited attributes.
  for ($class = get_class($this); $class !== InheritableModel::class; $class =
      get_parent_class($class))
  {
    $array = array_union($array, $class::$$array_name ?: []);
  }

  return $array;
}
```

To avoid overwriting the magic functions `__get()`, `__set()`, and `__call()`, they were replaced with two simple functions: `translation()` for singular attribute access and `translations()` for plural attribute access. While this solution relinquishes the convenience of transparent attribute access, it is much more robust and less prone to conflict with the framework or third-party add-ons. Listing 5.21 demonstrates how all the three above-mentioned functions were unified in one simple method. When `null` is passed, the translation is deleted, when no value is passed explicitly (defaults to an empty string), the translation is returned, and when a string value is passed, the translation is created or updated. If the function does not receive a locale parameter, it uses the current system default.

Listing 5.21: translation() method of InheritableModel for translated attribute access

```
public function translation ($attribute, $value = '', $locale = null)
{
  $language = $locale ? Language::getLanguage($locale) : Language::defaultLanguage
      ();
  if (!$language)
  {
    \Log::warning("Invalid locale $locale");
    return false;
  }

  if ($value === null)
    return $this->deleteTranslatedAttribute($attribute, $language);
  else if ($value === '')
    return $this->getTranslatedAttribute($attribute, $language);
  else
    return $this->setTranslatedAttribute($attribute, $language, $value);
}
```

In the first version of the translation system, translated attributes were cached on the newly created instance itself. While this does work, it relies on the object remaining in memory from instantiation up until it is saved to the database. Version 2 instead employs Laravel caching capabilities. The attributes are stored under a unique key in the cache. For this project, the local disk cache is sufficient, but separate key-value stores like Redis could just as well be used. Listing 5.22 depicts some of the functions used for caching. The `getCacheKey()` function is used to construct a unique cache key from the attribute type and the unique hash value of the current object. This cache key is used to store and retrieve translated attributes to and from the cache. The method `cacheTranslation()` stores the attributes with `Cache::put()`, passing an array of attributes and the cache key. `pullTranslations()` retrieves the cached attributes and simultaneously removes them from the cache.

Listing 5.22: Attribute caching functions in InheritableModel

```php
protected function cacheTranslation ($translation, $attribute_type)
{
  $translations = $this->pullTranslations($attribute_type);
  $translations[] = $translation;
  \Cache::put($this->getCacheKey($attribute_type), $translations, 5);
}

public function pullTranslations ($attribute_type)
{
  return \Cache::pull($this->getCacheKey($attribute_type), []);
}

protected function getCacheKey ($attribute_type)
{
  return 'translated_'.camel_case(str_plural($attribute_type)).'_'.spl_object_hash(
      $this);
}
```

The most marked improvements of version 2 over version 1 are the static translation attribute arrays and the elimination of transparent attribute access by overwriting magic methods. These measures improve extensibility, readability, maintainability, and most importantly robustness. However, this version still has some issues:

While the magic methods overwritten in `Translatable` were replaced with regular methods, drastically simplifying the handling of translated attributes, `InheritableModel` now overwrites `__construct()`. This is an approach that is arguably even more intrusive into the `Model` code and therefore not ideal. Furthermore, version 2 only slightly improves the creation of translations by utilizing the built-in caching mechanisms, but the approach itself remains the same as in version 1 and is still somewhat awkward. All in all, the translation system is still to complex and requires too much additional work to function. It was therefore further developed in version 3, which is the final version.

Version 3 — final version

Building on the insights and lessons learned from the first two attempts at building a dynamic translation system, a third and final version was developed. This iteration utilizes some of the basic ideas of the other two approaches, but in a much more minimalistic fashion.

One of the most noticeable changes is that the translation tables have been completely eliminated. The translated attributes are now moved back into their respective classes in keeping with the principles of object-oriented programming (OOP). Each translatable attribute is represented with a column of the `text` data type, which stores all translations as a serialized array. This array is formatted the same as the array returned through pluralized attribute access in the first two versions, i.e. keyed with ISO 639-1 language codes. While version 1 and 2 of the translation system also relied on a custom `Language` class to provide languages, it now simply relies on plain ISO 639-1 languages codes like the Laravel framework itself. Storing the translations on the model instance itself solves another minor issue: Updating translated attributes now works as expected. When translations are stored as records of a separate model, the parent class does not have to be saved. This can especially be confusing with transparent attribute access like implemented in version 1. When setting any other attribute, one would have to first set the attribute and then call `$save()` on the object. However, since the system creates and edits translation records in the background when updating translated attributes, `$save()` does not have to be called because the object's proper attributes are unchanged.

Laravel supports automatic *casting* of model attributes. This is useful for serialized array like the translation array. To retrieve it as a PHP array instead of a string, like it is represented in the database, the casting can be defined in the `$casts` array on the model, e.g. `protected $casts = [ 'text' => 'array' ];`.

The final translation system also does not rely on any arrays to register translated attributes anymore. Instead, when a developer accesses an attribute as a translated attribute, `Translatable` will simply attempt to access it from the database. If the attribute does not contain a serialized array that contains the requested language code (or if it cannot be saved as such), the attribute is not translatable.

The core translation functionality has been moved back into the `Translatable` Trait. It now provides a `translation()` method similar to the one introduced in version 2. Creating, editing, and deleting translations is now significantly simpler than in previous revisions, as demonstrated in listing 5.23. The `translation()` method is nearly unchanged from version 2. However, the translated attributes are now retrieved, set, and deleted through standard array access.

Listing 5.23: translation() method in Translatable v3

```php
public function translation ($attribute, $value = '', $locale = null)
{
  $locale = $locale ?: Lang::locale();

  if ($value === null)
```

```
    return $this->deleteTranslatedAttribute($attribute, $language);
  else if ($value === '')
    return $this->getTranslatedAttribute($attribute, $language);
  else
    return $this->setTranslatedAttribute($attribute, $language, $value);
}

public function getTranslatedAttribute ($attribute, $language)
{
  return $this->$attribute[$language];
}

public function setTranslatedAttribute ($attribute, $language, $value)
{
  $this->$attribute[$language] = $value;
}

public function deleteTranslatedAttribute ($attribute, $language)
{
  unset($this->$attribute[$language]);
}
```

In conclusion, the development process has led to a fairly simple, robust, and flexible translation system. The first attempt was by far the most complex, but it also offered the most transparent user experience due to dynamic attribute access and function calls. However, this transparency comes at the cost of robustness and requires some intrusive framework code manipulation. The second revision then already eliminated some of the method overwrites that caused issues in the first version. It has also traded maximum transparency for improved robustness, modularization, and code clarity by reducing the translation mechanisms to the simple `translation()` method and its helper functions. Finally, when version 2 was retired as well, the result of this process was a lean and trimmed-down, and portable solution for dynamic attribute translation. While it may be less flexible than version 1, the much improved robustness and reduced complexity is a worthwhile pay-off.

5.2.2 Version control

The version control system was developed in several iteration cycles like the dynamic translation system. In the following, the development process is outlined shortly:

In the **first iteration**, version control was implemented in a Trait called `Archivable`. The Trait provides a number of convenience methods to retrieve versions of records (previous, succeeding, first, last, etc.) and it defined the `archived` and `withoutArchived` scopes. The version control mechanism was based on self-referential one-to-one relationships. Each model that should be eligible for version control needed to have a `previous_version` integer column, which was used to store the primary key of the previous version of that particular record. This linked-list approach was functional, but since one of the criticisms of *TYT*'s implementation of version control in section 1.2 was that it required tables to add columns specifically for version control, the solution was redesigned.

The **second iteration** moved version cross-references to the newly created `version_control` table. Each record in this table referenced the "archived" record and its successor in the `next_-version` column through polymorphic relationships. In this approach, when a record should be edited, it is instead cloned and the clone is edited, while both versions are connected through the `version_control` table. This approach was simple and widely applicable, but its most significant drawback was that all references to the edited record had to be adjusted manually to reference the new, cloned record.

Therefore, a **third iteration** was created, in which it is always the original record that is edited. Its state is preserved by cloning the record before any modifications and linking it to the now edited version through the `version_control` table. This eliminated the need for manual rearrangement of references to the edited record. However, this approach had one minor flaw. When cloning a record, the version control record of the current predecessor version would have to be modified as well to reference the new clone as its successor.

The **fourth and final iteration** solved this problem by changing the relationships between model records and their version control records. Instead of linking version control records with one another, all version control records now reference the `root_version`, i.e. the original model record to which modifications are applied. The order of versions is established by simply adding a `version_number` integer column to the `version_control` table[3]. One drawback of all these approaches including the final one is that the versions do not have direct access to their former relationships. However, they can simply be accessed by requesting them from their root version. Furthermore, any relationships that are relevant to a record's state have to be archived independently.

5.2.3 Model inheritance and user roles

While inheritance is the foundational concept of object-oriented languages like PHP 5, it is not directly applicable to models in Laravel. Laravel provides functionality that handles the interaction and mapping between model classes and their database tables, but it does not support inherited models. Although models can extend each other to share their functionality with child classes, Laravel does not adapt the database model to reflect the model hierarchy. In other words, a model may have child classes, but they do not automatically inherit their parents' attributes, because they need not be explicitly defined on the model, and there is no automatic creation of child database tables. Some common solutions in such cases include *Single Table Inheritance* (*STI*). In this model, all models that extend a parent class share one common database table.

[3]The order of versions could also be implicitly derived from the creation timestamps of the version records, but dedicated version numbers are more explicit, human-readable and they prevent odd behavior. Additionally, retrieving the `n`-th version of a record becomes as simple as finding the record with version number `n` among its linked version control records.

Accordingly, this table has to contain all attributes that exist in any of the inheriting classes. That means that even if one particular child class only adds one attribute of its own, its records will contain `null` values for attributes of all its parent, sibling, and child classes. This approach is sufficient for cases where the differences between related classes are small. When related classes contain vastly different sets of attribute, however, this quickly results in large, messy tables full of `null` values. This is especially true for this project: The various Questionnaire and Question share most of their attributes, but some of their own and future extensions might very well be even more specialized. Furthermore, when considered strictly, STI violates the OOP principle of data encapsulation, i.e. an object may only have access to its own and its parents' attributes.

Due to the drawbacks of STI, it was decided early on to implement a custom class inheritance solution that works with Laravel models. Like version control and the dynamic translation system, the development of the inheritance system went through multiple iterations. This process had the same basic guiding principles as the translation system, i.e. maximum flexibility and transparent integration into the framework.

The **first draft** was designed to work as follows: In order to stay as close to OOP principles as possible, each models has its own database table. Each table only consists of columns for the classes' specific attributes. All child classes extend their parent classes with the standard PHP `extends` keywords to inherit functionality. On the database side, the class hierarchy is reflected in polymorphic one-to-one relationships between parent and child classes. Instances of inheriting classes are then mapped to multiple records of models in a hierarchy. Each subclass contains its own specific subset of attributes and is connected to other subclasses for further specialization through polymorphic `subclass` reference columns. The necessary functionality is provided by the `ExtensibleModelSuperclass` and `ExtensibleModelSubclass` Traits, that have to be included by superclasses and subclasses respectively.

This first version of the model-based inheritance mechanism shared a lot of the same issues that the first version of the translation system also had: `ExtensibleModelSuperclass` relies on overwriting the framework-provided methods `__construct()`, `__call()`, `__get()`, and `__fill()`. The getter and method invocation functions were overwritten to provide transparent access to parent attributes as if they were defined on the child class. As discussed in section 5.2.1, this approach is not particularly stable and prone to breaking when the framework code is updated. Furthermore, all attributes must be explicitly listed in a *constant* array `$own_attributes`. This is necessary to discern proper attributes from inherited attributes when mass-assigning, because developers may pass all attributes without explicitly distinguishing own and inherited attributes to keep the system transparent and close to OOP principles. Like early translation implementations, however, the related objects cannot all be created at the same time as they are split up across tables. To be able to leverage Laravel's syntax to create related objects, one of those objects must first exist in the database so that its primary key is available. Therefore, when creating an inheriting object, all child attributes must be cached and their records must be created later during the `created` and `saved` events. All of this makes this approach fairly complex, difficult to handle, and error-prone.

The **second version** was implemented as the `InheritableModel` class that was introduced in section 5.2.1. `InheritableModel` replaces the constant `$own_attributes` array with the static `$my_fillable` and `$my_guarded` arrays. This was necessary to differentiate fillable from guarded attributes, which was not possible before. Another major change that the subclass tables hold all attributes, both proper and inherited. The inheritance system provides functions to created these tables; developers are not required to gather the parents' attributes manually. `InheritableModel` also introduces the static `$subclasses` array, which was used to define a map of public class names and fully qualified class names. This array was created for several reasons. First, it creates a central place to hold public class names that can be used to differentiate Question types etc. in the client. It was considered to use the *snake_case* versions of class names as public names, but this would pose a serious security threat, because malicious users might abuse this feature to create instances of arbitrary classes in the system. Therefore, the system only recognizes classes that are listed in `$subclasses`. Also, since there is no easy way in PHP to retrieve all classes that extend a specific class, this list helps with creating the database tables for child classes in one go. Version 2 of the inheritance mechanism helps with some of the issues of version 1, like overwriting framework functions. However, it still has a number of flaws. The handling of attributes still relies on distributed tables, caching on creation, and event hooks. All in all, the approach has simply proven not as useful as expected in practical use. While it does generally fulfill its purpose, new issues with the inheritance system constantly arose with further development and it was very prone to breaking. Therefore, a third and last version was devised.

After experimenting with distributed, related tables and transparent convenience function, the concept was mostly abandoned for the reasons discussed above. The **third and final version** of the inheritance system was designed to work more smoothly within the Laravel framework and to interfere less with its intended workflows. Version 3 borrows a number of ideas from the previous versions. Similarly to version 1, the tables for parent and child classes are again split up. However, instead of transparently distributing attributes and linking the tables, the record in the parent class acts is the actual record for the object, unlike version 1, where the records in each table were treated as dependent parts of the same object. The subclass records are linked through explicit, polymorphic `extension` columns on the parent table. When obtaining an object, it is retrieved from the parent table and any specialized information must be explicitly accessed through the `extension` relationship. This is not as convenient as version 1, but significantly more robust and requires no additional, complex functionality. Furthermore, when creating new models, child attributes do not have to be cached anymore. The child models are simply created from the entire attribute set. This is possible because Eloquent simply ignores all attributes that are not declared fillable. Lastly, the central subclass register is also eliminated. In this version, there is no need to know all existing subclasses of a model, because the parent instance is the "authoritative" part of the object and subtypes can simply be distinguished by the polymorphic `$extension_type` column. Also, a convention is introduced for string identifiers of types, e.g. for the client: All subclasses must be named after the parent class, concatenated with the subtype name in *PascalCase*. For example, single choice Questions inherit from the `Question` model and are therefore named `QuestionSingleChoice`. By convention, the `QuestionSingleChoice` subtype is referenced as `single_choice` in the code where necessary. Class names can then be dynamically created from the parent class base name and the PascalCase version of the *snake_case* string identifier. This enables flexible handling of subclass names, while preventing object injection, because the parent's class name is always used as a prefix.

The final version of the model inheritance mechanism is currently used for the `Question` and `Questionnaire` classes and their respective subclasses. As was discussed in section 4.2.1, there are also multiple types of users in this system inheriting from the same, conceptual base class. The reason that the model inheritance system is not used for the User model is that it is far simpler that `Question` and `Questionnaire` and less likely to be extended in the future. Moreover, the particular use case of differentiation of users lends itself to a different, commonly used pattern, which is **user roles**. This concept was therefore introduced to the system as a by-product of the development process described above. Section 4.2.1 describes the different roles users can take on when using the framework. Roles are defined in a `roles` table. Each User has one or more Roles. Presently, the system only supports the *User* and *Admin* roles, which correspond to the roles defined in sections 4.2.1 and 4.2.1. The assignment of users to roles is managed in a `user_roles` table. This approach has the advantage that permissions for system functionality can easily be granted or revoked based on users' Role. New Roles can be created by simply adding the Role to the Roles table and assigned it to users as desired. In effect, this results in a rudimentary access control system for system functionality, similarly to UserGroups for questionnaire access. One disadvantage of a role-based approach is that all users are stored in the same Users table. For the current iteration of this system, this is not a problem since regular Users and Admins share the same attributes. However, if future implementations should add further user models that require additional attributes, this approach is not sufficient to satisfy that requirement.

5.3 Client

5.3.1 Initially hidden slider handle

In the following, the implementation of the slider without initial value is presented. This custom UI widget was required in requirement UI001. It has been shown that the position of the handle in a slider widget can have an influence on users' behavior when selecting a value [13]. Therefore, the handle is not displayed at first. Once the user clicks on the slider widget, the handle is shown and can be moved as usual. *AYS* implements this in the web-based user interface using an

HTML5 `input` element of the `range` type. While there is no way to show or hide the handle programmatically, CSS can be used for that purpose. Listing 5.24 shows how this is implemented in *AYS*. The slider handle is hidden by default by setting its `opacity` property to `0` using the pseudo-classes `-webkit-slider-thumb` and `-moz-range-thumb`. The UI detects when the slider has been first clicked by the user and marks it by adding the `clicked` class to the `input` element. The `clicked` class is then used to set the `opacity` back to `1`, showing the slider handle. The code used for this functionality was adapted from [18].

Listing 5.24: CSS controlling the slider handle

```
input[type=range]::-webkit-slider-thumb
{
  opacity: 0;
}
input[type=range]::-moz-range-thumb
{
  opacity: 0;
}

input[type=range].clicked::-webkit-slider-thumb
{
  opacity: 1;
}
input[type=range].clicked::-moz-range-thumb
{
  opacity: 1;
}
```

6 Conclusion

This chapter concludes the documentation of the *AYS* project. In the following section, the project's development is summarized. After a recapitulation of the initial goals and requirements follows a critical reflection of the development process. The requirements and results are contrasted to assess the success of the project. The oversights and weaknesses of the system, but also the lessons learned are discussed. Finally, the chapter closes with an outline of potential future improvements or extensions of *AYS* to broaden its scope and functionality.

6.1 Summary

In this project, the *Assess Your Stress* framework was conceptualized and a prototype was implemented. The development of *AYS* was initiated with the goal of creating an updated, modernized, and overall improved version of the *Track Your Tinnitus* platform. The basic principle of both systems is the same, but they differ in their fields of application: While *TYT* was created for mobile tracking of tinnitus symptoms through questionnaires and the *Experience Sampling Method*, *AYS* provides a platform that employees can use to gauge their stress levels at their workplaces.

The overhaul of the concept behind *TYT* was the focus of this work. Some of the key goals of the project was to revise the platform in a way that makes it more generalizable and applicable to more fields of application. The issue of stress tracking was used as the exemplary application to show that the *TYT* concept can indeed be applied in various ways. Most of the allotted time for the project was therefore spent conceptualizing and experimenting with different approaches to specific system components that displayed the most pronounced drawbacks in terms of extensibility and flexibility in *TYT*. These components were identified to be the dynamic translation of questionnaires and questions—or database records in general—, version control and model inheritance. In *TYT*, translation of dynamic data was purpose-built for the use cases of questionnaires and questions. A solution for version control was implemented, but only in rudimentary form, restricted to questions, and not actually used in the system. While model inheritance was not an issue covered in the *TYT* project, it was deemed relevant for the new iteration, mostly as a robust basis for the extension of questionnaires and questions in the future. At the end of the project, the concept was realized in a basic prototype to apply it to the stress tracking use case.

6.1.1 Self-assessment

A considerable amount of work was put into eliciting and defining the requirements for this system, which were categorized from high-level to low-level and recorded in a semi-formal and fairly fine-grained fashion. This was done in an effort to make the conceptual foundation for the framework as stable and refined as reasonably possible within the scope of this project. Comparing the requirements with the prototypical implementation, however, it is clear that not all requirements could be verifiably fulfilled. While the project has succeeded in producing a viable system core, some of the planned features could not be incorporated, mostly due to time limitations. One of these features that were abandoned for the current release was the overhaul of result diagrams. Originally, diagrams were planned to be generated in a more intelligent manner than the straightforward method of *TYT*. This would be especially useful in *AYS* as it now handles arbitrary numbers of questionnaires and questions. Instead of merely presenting separate diagrams for each question in a questionnaire, it would be far more insightful for users if results from different questions and questionnaires were somehow combined into fewer diagrams. This way, users could easily compare results side by side and gain a more meaningful image of their stress levels. Further, there is no support for statistical information or feedback on the administrative side. In a finished, deployed version, this would be a crucial features because employers just as much need to get feedback from the system as their employees. The *system event* functionality could also not be fully implemented. Experiments were made as a proof-of-concept, but a full, functioning messaging system was out of the project's scope. All things considered, however, it can be rated as a success that the project did result in a foundation for the system that implements the core functionality.

The process of developing crucial system components through careful and deliberate evolution was notably successful in producing useful results. The dynamic translation, version control, and model inheritance systems were all products of iterative experimentation. It is notable that the initial approaches to all three of these issues were too complex and unwieldy. The most likely cause is that is was attempted not to compromise on any of the goals set for the development of these components. It was recognized that maximum conformance with the requirements had led to solutions that offered the full, desired functionality, but were generally unstable and prone to breaking due to the close coupling with the framework. In the subsequent iterations, the solutions were therefore increasingly reduced and trimmed down as the requirements were relaxed. The end result of this process was generally lean and effective solutions to some complex tasks. This effort was therefore certainly one of the project's more notable achievements. There is also a lesson that can be learned from it, which is that the common phrase "less is more" does hold true in software engineering, where guideline is also known as the *KISS* principle—"Keep It Small and Simple".

In retrospect, it is quite clear that the planning stage of the project was allotted too much of the project's time. As mentioned above, the intention behind this was to ensure a refined and valuable basis upon which the implementation could be built. However, it is now evident that especially the requirements analysis could have been reduced significantly—the chapter on requirements already represents an abridged version of the actual requirements definitions produced during this project. While it is certainly not amiss to diligently employ established software engineering practices to the crucial phase of requirement definition, the amount of detail dedicated to each requirement is simply not necessary or particularly useful in a project of this scope, developed by a single developer.

6.2 Future development

The *AYS* framework lays the groundwork for a variety of future developments that could enhance, extend, and further improve the system.

First and foremost, a fully-featured client should be implemented to complement the system's back end core. The current prototype is fairly bare-bones and leaves out some desirable functionality. As this system is meant for use by a wide and varied audience, usability is crucial. The client should therefore also be subjected to systematic usability testing before releasing it to the general public.

The *TYT* framework provides mobile clients, that are used for snapshot questionnaires and result review. Mobile clients would also be very useful in the context of *AYS*, since people's workplaces vary greatly. Many employers may not work with a stationary computer, or they may not have steady Internet access. Furthermore, this would enable *AYS* to utilize the *Experience Sampling Method* like *TYT* does. Therefore, mobile clients that work offline would be a highly valuable addition to *AYS*.

As mentioned above, some planned functionality had to be dropped from the development of the current release due to time constraints. These features would be useful additions to the system and should therefore be implemented in the future. Among these features are improved and streamlined visualization of questionnaire results, statistical analysis for employers/administrators, and possibly some intelligent result analysis.

Visualization of results also offers some interesting possibilities for future extensions. One approach to how different questionnaires and question could be combined in unified result diagrams could be to let administrators categorize their questionnaires. They could then create custom *scores* and define what questions factor into a particular score and how. The scores would be what the unified diagrams then display. Not only would this reduce the number of diagrams needed to represent the results, users could also gain much more meaningful insight with a high-level score—say, "stress level"—than a set of different, seemingly unrelated result graphs. This is also a natural advancement since multiple questions in questionnaires are generally used in concert to measure one particular variable, and not the isolated data of individual questions.

One of the goals of the project was *personalized* feedback for users. This could not be achieved in the current release as it soon became clear that it is outside the scope of this project. Such functionality would require some level of artificial intelligence implemented in the system to analyse users' results, possibly correlating them with other data sources like schedules. This could then be used to identify the exact sources of stress in the users' daily lives and

Bibliography

[1] AUTH0: *Introduction to JSON Web Tokens*. https://jwt.io/introduction/. Version: 2016. – [Online; accessed 8-June-2016]

[2] AUTH0: *JSON Web Tokens*. https://jwt.io. Version: 2016. – [Online; accessed 8-June-2016]

[3] BAKKER, Jorn ; HOLENDERSKI, Leszek ; KOCIELNIK, Rafal ; PECHENIZKIY, Mykola ; SIDOROVA, Natalia: Stess@Work: From Measuring Stress to Its Understanding, Prediction and Handling with Personalized Coaching. In: *Proceedings of the 2Nd ACM SIGHIT International Health Informatics Symposium*. New York, NY, USA : ACM, 2012 (IHI '12). – ISBN 978–1–4503–0781–9, 673–678

[4] CIMAN, M. ; WAC, K. ; GAGGI, O.: iSensestress: Assessing stress through human-smartphone interaction analysis. In: *Pervasive Computing Technologies for Healthcare (PervasiveHealth), 2015 9th International Conference on*, 2015, S. 84–91

[5] DYSON, MARY C. ; HASELGROVE, MARK: The influence of reading speed and line length on the effectiveness of reading from screen. In: *International Journal of Human-Computer Studies* 54 (2001), Nr. 4, 585 - 612. http://dx.doi.org/http://dx.doi.org/10.1006/ijhc.2001.0458. – DOI http://dx.doi.org/10.1006/ijhc.2001.0458. – ISSN 1071–5819

[6] ENUMERATION, Common W.: *CWE-915: Improperly Controlled Modification of Dynamically-Determined Object Attributes*. http://cwe.mitre.org/data/definitions/915.html. Version: 2013. – [Online; accessed 8-June-2016]

[7] FIELDING, Roy T.: *Architectural Styles and the Design of Network-based Software Architectures*, Diss., 2000. – AAI9980887

[8] GARRETT, Jesse J.: *Ajax: A New Approach to Web Applications*. http://adaptivepath.org/ideas/ajax-new-approach-web-applications/. Version: 2005. – [Online; accessed 8-June-2016]

[9] GUDGIN, Martin ; HADLEY, Marc ; MENDELSOHN, Noah ; MOREAU, Jean-Jacques ; NIELSEN, Henrik F. ; KARMARKAR, Anish ; LAFON, Yves: *SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)*. https://www.w3.org/TR/soap12/. Version: 2007. – [Online; accessed 8-June-2016]

[10] HEKTNER, J.M. ; SCHMIDT, J.A. ; CSIKSZENTMIHALYI, M.: *Experience Sampling Method: Measuring the Quality of Everyday Life*. SAGE Publications, 2007 https://books.google.de/books?id=05e5d_KBYY0C. – ISBN 9781412949231

[11] HERRMANN, Jochen ; REICHERT, Manfred (Hrsg.) ; SCHLEE, Winfried (Hrsg.) ; PRYSS, Rüdiger (Hrsg.): *Konzeption und technische Realisierung eines mobilen Frameworks zur Unterstützung tinnitusgeschädigter Patienten*. http://dbis.eprints.uni-ulm.de/1037/. Version: March 2014

[12] HOOBER, Steven: *How Do Users Really Hold Mobile Devices?* http://www.uxmatters.com/mt/archives/2013/02/how-do-users-really-hold-mobile-devices.php. Version: 2013. – [Online; accessed 8-June-2016]

[13] KAHNEMAN, D.: *Thinking, Fast and Slow*. Farrar, Straus and Giroux, 2011 https://books.google.se/books?id=ZuKTvERuPG8C. – ISBN 9781429969352

[14] KLABNIK, Steve ; KATZ, Yehuda ; GEBHARDT, Dan ; KELLEN, Tyler ; RESNICK, Ethan: *JSON API Document Structure*. http://jsonapi.org/format/#document-structure. Version: 2015. – [Online; accessed 8-June-2016]

[15] KLABNIK, Steve ; KATZ, Yehuda ; GEBHARDT, Dan ; KELLEN, Tyler ; RESNICK, Ethan: *JSON API Specification*. http://jsonapi.org/format/. Version: 2015. – [Online; accessed 8-June-2016]

[16] LU, Hong ; FRAUENDORFER, Denise ; RABBI, Mashfiqui ; MAST, Marianne S. ; CHITTARANJAN, Gokul T. ; CAMPBELL, Andrew T. ; GATICA-PEREZ, Daniel ; CHOUDHURY, Tanzeem: StressSense: Detecting Stress in Unconstrained Acoustic Environments Using Smartphones. In: *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*. New York, NY, USA : ACM, 2012 (UbiComp '12). – ISBN 978–1–4503–1224–0, 351–360

[17] MACLEAN, Diana ; ROSEWAY, Asta ; CZERWINSKI, Mary: MoodWings: A Wearable Biofeedback Device for Real-time Stress Intervention. In: *Proceedings of the 6th International Conference on PErvasive Technologies Related to Assistive Environments*. New York, NY, USA : ACM, 2013 (PETRA '13). – ISBN 978–1–4503–1973–7, 66:1–66:8

[18] O'BRIEN, Brenna: *How to Style Input Type Range in Chrome, Firefox, and IE*. http://brennaobrien.com/blog/2014/05/style-input-type-range-in-every-browser.html. Version: 2014. – [Online; accessed 8-June-2016]

[19] OTWELL, Taylor: *Laravel Documentation*. https://laravel.com/docs/5.2. Version: 2016. – [Online; accessed 8-June-2016]

[20] OTWELL, Taylor: *Laravel Documentation: Application Structure*. https://laravel.com/docs/5.2/structure#the-app-directory. Version: 2016. – [Online; accessed 8-June-2016]

[21] PHP-FIG: *PSR-0: Autoloading Standard*. `http://www.php-fig.org/psr/psr-0/`. Version: 2014. – [Online; accessed 8-June-2016]

[22] PHP-FIG: *PSR-1: Basic Coding Standard*. `http://www.php-fig.org/psr/psr-1/`. Version: 2016. – [Online; accessed 8-June-2016]

[23] PROBST, Thomas ; PRYSS, Rüdiger ; LANGGUTH, Berthold ; SCHLEE, Winfried: Emotional states as mediators between tinnitus loudness and tinnitus distress in daily life: Results from the “TrackYourTinnitus” application. In: *Scientific reports* 6 (2016)

[24] PRYSS, Rüdiger ; REICHERT, Manfred ; HERRMANN, Jochen ; LANGGUTH, Berthold ; SCHLEE, Winfried: Mobile Crowd Sensing in Clinical and Psychological Trials – A Case Study. In: *28th IEEE Int’l Symposium on Computer-Based Medical Systems*, IEEE Computer Society Press, June 2015, 23–24

[25] PRYSS, Rüdiger ; REICHERT, Manfred ; LANGGUTH, Berthold ; SCHLEE, Winfried: Mobile Crowd Sensing Services for Tinnitus Assessment, Therapy and Research. In: *IEEE 4th International Conference on Mobile Services (MS 2015)*, IEEE Computer Society Press, June 2015, 352–359

[26] RAHMAN, Md. M. ; BARI, Rummana ; ALI, Amin A. ; SHARMIN, Moushumi ; RAIJ, Andrew ; HOVSEPIAN, Karen ; HOSSAIN, Syed M. ; ERTIN, Emre ; KENNEDY, Ashley ; EPSTEIN, David H. ; PRESTON, Kenzie L. ; JOBES, Michelle ; BECK, J. G. ; KEDIA, Satish ; WARD, Kenneth D. ; AL’ABSI, Mustafa ; KUMAR, Santosh: Are We There Yet?: Feasibility of Continuous Stress Assessment via Wireless Physiological Sensors. In: *Proceedings of the 5th ACM Conference on Bioinformatics, Computational Biology, and Health Informatics*. New York, NY, USA : ACM, 2014 (BCB ’14). – ISBN 978–1–4503–2894–4, 479–488

[27] RAHMAN, Tauhidur ; ZHANG, Mi ; VOIDA, Stephen ; CHOUDHURY, Tanzeem: Towards Accurate Non-intrusive Recollection of Stress Levels Using Mobile Sensing and Contextual Recall. In: *Proceedings of the 8th International Conference on Pervasive Computing Technologies for Healthcare*. ICST, Brussels, Belgium, Belgium : ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2014 (PervasiveHealth ’14). – ISBN 978–1–63190–011–2, 166–169

[28] SCHICKLER, Marc ; REICHERT, Manfred ; PRYSS, Rüdiger ; SCHOBEL, Johannes ; SCHLEE, Winfried ; LANGGUTH, Berthold: *Entwicklung mobiler Apps: Konzepte, Anwendungsbausteine und Werkzeuge im Business und E-Health*. Springer Vieweg, 2015 (eXamen.press). `http://dbis.eprints.uni-ulm.de/1320/`

[29] SCHOBEL, Johannes ; PRYSS, Rüdiger ; REICHERT, Manfred: 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, June 2015, 13–18

[30] SCHOBEL, Johannes ; SCHICKLER, Marc ; PRYSS, Rüdiger ; MAIER, Fabian ; REICHERT, Manfred: Towards Process-Driven Mobile Data Collection Applications: Requirements,

Challenges, Lessons Learned. In: *10th Int'l Conference on Web Information Systems and Technologies (WEBIST 2014), Special Session on Business Apps*, 2014, 371–382

[31] SCHOBEL, Johannes ; SCHICKLER, Marc ; PRYSS, Rüdiger ; NIENHAUS, Hans ; REICHERT, Manfred: Using Vital Sensors in Mobile Healthcare Business Applications: Challenges, Examples, Lessons Learned. In: *9th Int'l Conference on Web Information Systems and Technologies (WEBIST 2013), Special Session on Business Apps*, 2013, 509–518

[32] SCHOBEL, Johannes ; SCHICKLER, Marc ; PRYSS, Rüdiger ; REICHERT, Manfred: Process-Driven Data Collection with Smart Mobile Devices. Version: 2015. `http://dbis.eprints.uni-ulm.de/1136/`. In: *10th International Conference on Web Information Systems and Technologies (Revised Selected Papers)*. Springer, 2015 (LNBIP 226), 347–362

[33] SURGUY, Maks: *Differences between Laravel 3 and Laravel 4*. `http://maxoffsky.com/code-blog/differences-between-laravel-3-and-laravel-4/`. Version: 2013. – [Online; accessed 8-June-2016]

[34] VENNERS, Bill: *Orthogonality and the DRY Principle*. `http://www.artima.com/intv/dry.html`. Version: 2003. – [Online; accessed 8-June-2016]

[35] W3SCHOOLS: *Browser Display Statistics*. `http://www.w3schools.com/browsers/browsers_display.asp`. Version: 2016. – [Online; accessed 8-June-2016]

[36] WANG, Rui ; CHEN, Fanglin ; CHEN, Zhenyu ; LI, Tianxing ; HARARI, Gabriella ; TIGNOR, Stefanie ; ZHOU, Xia ; BEN-ZEEV, Dror ; CAMPBELL, Andrew T.: StudentLife: Assessing Mental Health, Academic Performance and Behavioral Trends of College Students Using Smartphones. In: *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing*. New York, NY, USA : ACM, 2014 (UbiComp '14). – ISBN 978–1–4503–2968–2, 3–14

[37] WEPPNER, Jens ; LUKOWICZ, Paul ; SERINO, Silvia ; CIPRESSO, Pietro ; GAGGIOLI, Andrea ; RIVA, Giuseppe: Smartphone Based Experience Sampling of Stress-related Events. In: *Proceedings of the 7th International Conference on Pervasive Computing Technologies for Healthcare*. ICST, Brussels, Belgium, Belgium : ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2013 (PervasiveHealth '13). – ISBN 978–1–936968–80–0, 464–467

[38] WIKIPEDIA: *Single source of truth — Wikipedia, The Free Encyclopedia*. `https://en.wikipedia.org/w/index.php?title=Single_source_of_truth&oldid=712438403`. Version: 2016. – [Online; accessed 8-June-2016]

Name: Bojan Klečina Student ID: 713280

Declaration of Academic Integrity

I hereby confirm that this work is original and that if any passage(s) or diagram(s) have been copied from academic books, papers, the Web or other sources, these are clearly identified by the use of quotation marks and the references are fully cited. I certify that, other than where indicated, the work attached is solely my own work.

Ulm, ...

Bojan Klečina