

Design report

DEPARTMENT OF COMPUTER SCIENCE, UNIVERSITY OF TWENTE

Authors

Melcher Stikkelorum (s1560670)

Lukas Miedema (s1600885)

Remco de Man (s1579886)

Ömer Şakar (s1560158)

Supervisor

Marieke Huisman



21 April 2017

Table of contents

1. Introduction	6
2. Project definition	7
2.1. Key persons	7
2.2. Glossary	7
2.3. System type	9
2.4. Project goals	9
2.5. Scope and limitations	10
3. Project approach	11
3.1. Methodology	11
3.2. Project phases	11
3.3. Workflow	11
3.3.1. Use case management	11
3.3.2. Source code management	11
3.3.3. Branching strategy	12
3.3.4. Documentation	12
3.4. Project log	12
4. Domain analysis	14
4.1. Introduction	14
4.2. Stakeholders	14
4.2.1. The system	14
Normal operator	14
Maintenance operators	15
Operational support	15
4.2.2. The Containing System	15
Functional beneficiary	15
Purchaser	15
4.2.3. The wider environment	15
The public	15

4.3. Existing software solutions	16
4.3.1. Kahoot!	16
4.3.2. Socrative	16
4.3.3. Shakespeak	17
4.3.4. The existing Python application	17
4.4. Actors	18
4.4.1. Organizer	18
4.4.2. Participant	18
5. Requirements	19
5.1. Requirement engineering	19
5.1.1. Weekly meetings	19
5.1.2. Existing solutions	19
5.1.3. Mock-ups	19
5.2. Functional requirements	20
5.2.1. Must	20
5.2.2. Should	21
5.2.3. Could	22
5.2.4. Won't	23
5.3. Quality requirements	23
5.3.1. Functional suitability	23
5.3.2. Performance efficiency	23
5.3.3. Compatibility	24
5.3.4. Usability	24
5.3.5. Reliability	24
5.3.6. Security	24
5.3.7. Maintainability	24
5.3.8. Portability	25
6. Architectural design	26
6.1. System overview (top-level concepts)	26

6.1.1. Audiences	26
6.1.2. Quizzes	27
6.1.3. Quiz runs	27
6.2. Flow Overview	28
6.2.1. Quiz run activity diagrams	29
Playing an anonymous quiz run	29
Playing an authenticated quiz run	30
Playing a quiz run	31
6.3. Preliminary design choices	31
6.3.1. Programming language	31
6.3.2. Build management	31
6.3.3. Web framework	32
6.3.4. Application deployment strategy	32
6.3.5. Database	32
6.3.6. Object Relational Mapping framework	33
6.3.7. Prefer back end rendering over front end rendering	33
6.3.8. JavaScript and CSS dependency management	33
6.3.9. Primary Javascript Library	34
6.3.10. CSS framework	34
7. Detailed design	35
7.1. Entity relations	35
7.2. Quiz runs to separate quizzes, audiences and quiz modes	36
7.3. Websocket protocols and communication	37
7.3.1. STOMP	37
Message channels	37
Message design	37
7.3.2. Participant network sequence diagram	38
State updates from the server	39
Answering a question	40

7.3.3. Presenter sequence diagram	40
7.3.4. Statistics sequence diagram	44
7.3.5. Presentation and statistics combined	45
7.4. Saving answers to questions	46
7.5. Back end design patterns	46
7.5.1. MVC	46
7.5.2. REST	46
7.5.3. Dependency injection	47
7.5.4. Lombok	47
7.6. Security	47
7.7. Websockets and transport	48
7.8. Front end design patterns	48
7.8.1. Less	48
7.8.2. React and JSX	49
7.8.3. Parsing of date and time	49
7.8.4. Dynamic asynchronous views	50
8. Test plan & execution	51
8.1. Strategies	51
8.1.1. Unit tests	51
8.1.2. Integration tests	51
8.1.3. Concurrency and performance tests	51
8.1.4. Real world tests	52
8.1.5. Checkstyle compliance	52
8.2. Test results	52
8.2.1. Unit tests	52
8.2.2. Integration tests	52
8.2.3. Performance tests	53
Concurrency test	53
Load tester	55

8.2.4. Real world tests	56
9. Conclusion	57
9.1. Functional requirements	57
9.2. Quality requirements	57
9.2.1. Functional suitability	57
9.2.2. Performance efficiency	58
9.2.3. Compatibility	58
9.2.4. Usability	58
9.2.5. Reliability	58
9.2.6. Security	58
9.2.7. Maintainability	59
9.2.8. Portability	59
10. Future planning	60
10.1. Additional features	60
10.1.1. Data mining & analysis	60
10.1.2. Additional quiz modes & quiz mode DSL	60
10.1.3. Various types of questions	60
10.1.4. Send emails to the participant after a quiz	60
10.1.5. Using other databases management systems (DBMS)	60
10.1.6. Hosted solution	61
10.2. Maintainability	61
10.3. Usage	61
11. Evaluation & reflection	62
11.1. Roles and task division	62
11.2. Improvements	63

1. Introduction

Traditionally, students of the module Software Systems given in the first year of the bachelor Computer Science at the University of Twente have a diagnostic test in the form of a last man standing quiz. Currently, a Python application is used which utilized the Last Man Standing quiz mode. This application, however, lacks a lot of desired functionality.

In the current situation, the application only enforces the quiz mode. Questions are shown on a separate screen through an external application, such as PowerPoint slides. New questions are started through a special interface in the Python application separate from this application. The Python application automatically saves results and calculates live statistics for the Last Man Standing quiz mode. These statistics are shown through a web interface on a second screen separate from the questions. Participants answer questions through a web interface which automatically refreshes the page during the execution of the quiz.

This implementation raises some problems with the increasing amount of students participating in the module Software Systems. Most importantly, the application is not able to handle a lot of students due to the overhead that comes with the application refreshing the page during quiz execution. Also, having separate applications for enforcing the quiz mode and showing the questions is not desired as it requires someone other than the presenter to add questions during the presentation. It also raises the problem of having the need for two separate screens to show the questions and statistics at once.

In this project, we created *Quizzard*, an application that is designed and built to replace the current Python application. Our application will incorporate all functionality present in the existing application. The focus will be mainly on creating an application that is able to deal with large amounts of participants and integrates questions and statistics into one presentation. Additionally, the aim is to make the application easier to use for the people tasked with the creation and management of quizzes.

In the subsequent sections of this report, the application and the domain are analyzed. Sections 2 and 3 describe the project definition and approach. In section 4 the domain of the application is analyzed. Section 5 contains the requirements analysis. Sections 6 and 7 provide a detailed explanation of the application design, including all design choices. Section 8 contains the test plan of the application and section 9 the conclusions. Sections 10 and 11 describe future planning and evaluation of the application.

2. Project definition

In order to have a clear understanding of the project, the next sections introduce the definition of the project. A design was proposed to and approved by the client by means of a separate proposal document, which included the domain analysis, requirements, global design and the detailed design.

2.1. Key persons

This section briefly names the individuals involved in this project along with their contact information. Contact persons for the project on behalf of the University of Twente are noted below. The contact persons are further to be referred to as 'client'.

- *Marieke Huisman* *m.huisman@utwente.nl*
- *Wytse Oortwijn* *w.h.m.oortwijn@utwente.nl*

The members of the developer team "The New Bitmap Image" that have been executing the project:

- *Melcher Stikkelorum* *m.j.stikkelorum@student.utwente.nl*
- *Ömer Şakar* *o.f.o.sakar@student.utwente.nl*
- *Remco de Man* *r.j.p.deman@student.utwente.nl*
- *Lukas Miedema* *p.l.miedema@student.utwente.nl*

2.2. Glossary

These domain specific terms are used throughout the rest of the document and might need some further explanation. Terms are sorted alphabetically.

Term	Definition
Anonymous audience	A type of audience that lets participants join a quiz without authenticating in a personally identifiable way. This form of authentication is for instance realized by going to a link or inserting a token which is shared with the room.
Audience	Group of people participating in the quiz.

Authenticated audience	A type of audience that lets participants join a quiz after authenticating in a personally identifiable way. The results of a quiz can be directly tied back to a specific person and impersonation is prevented. This is realized by sending every participant an email with a unique token, binding their answers to their email address.
Back end	Back end is all application code that is 'invisible' to the user. It is comprised of all controllers, API endpoints and domain specific logic that runs on the server any user connects to and the front end communicates with.
Classic quiz	Quiz mode in which every participant gets a point for each correctly answered question.
Front end	Front end is all code related to the view of the application and code executing on the client. As such the front end entails all the HTML and Javascript of the application.
Last man standing	Quiz mode in which all participants take part as long as they provide the right answers. If an incorrect answer is given, the participant is no longer in the race for the prize. They can still answer the question though. This is usually done by having the participants 'stand up' as long as they are in the race and sit down as soon as they drop out, hence 'last man standing'.
Poll	Quiz mode in which only the distribution of the answers is shown for each question in a pie chart.
Quiz mode	The mode in which the quiz is taken. One of the following: <ul style="list-style-type: none"> • Classic quiz • Last man standing • Poll

Quizzard	The name of the application developed for this project. Also referred to by 'the application'. Quizzard is a contraction of Quiz and Lizard.
User	A physical human sitting behind a machine.

2.3. System type

The system we have built in this project is in the form of a web application. The application runs locally on the quiz host's computer. The user interface is presented through a web browser and can be reached by browsing to the IP address of the computer on which the application runs. The web server and application code are bundled together in a single package so that there is no need to set up a separate server first. Participants in the quiz can browse to the IP address of the quiz' host using their own device running any modern browser.

2.4. Project goals

First and foremost, the application should enable the teacher to organize a quiz which lets participants play and see their scores and results. The application should enforce the rules of various quiz modes on a quiz. These modes can be chosen by the organizer.

Another goal of the application was to be able to support large numbers of participants. With the ever increasing numbers of students taking the bachelor Computer Science course at the University of Twente, it is very important that the application is able to efficiently deal with the scarce network resources.

In the past, improper authentication of quiz participants has led to confusion and prevented others from participating in the quiz. The application provides a mode for proper user authentication, but also allow anonymous login functionality in cases where proper authentication is not required. The method of authentication can be set by the quiz organizer.

2.5. Scope and limitations

Even though better authentication is one of the goals of this project, bulletproof security cannot and will not be guaranteed. The system is designed for use with (diagnostic) quizzes, and not for exams with which comes a higher expectation of security. Therefore the safety measures implemented in the application may be able to be circumvented. This should, however, not pose a problem as the quiz results are not to be used as official exam results.

3. Project approach

3.1. Methodology

During development of the application, the software development model *Agile* is used. More specifically, the Scrum framework is used. This allows us to work on the software iteratively while the client is able to contribute new requirements during development. It is, however, up to the client to prioritize the added requirements.

3.2. Project phases

As part of the project, the following project phases can be distinguished. Do note that within a single sprint all these phases occur. Still, at a certain point in the project, the focus of sprints may shift more towards a certain aspect.

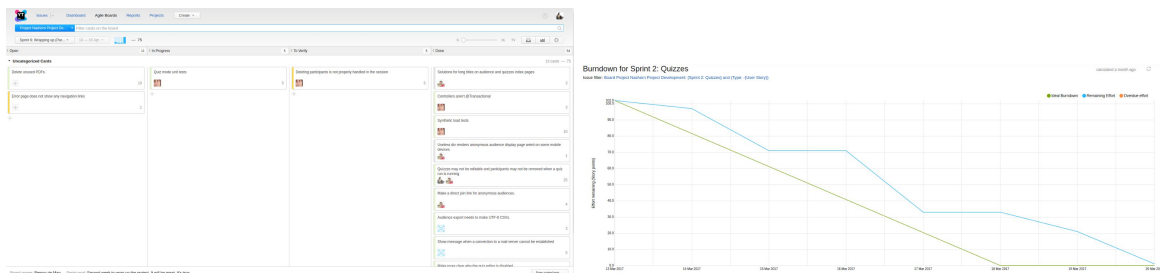
1. Requirement engineering
2. System Design
3. Development
4. Testing

3.3. Workflow

During the project, we established a workflow that will be further explained here.

3.3.1. Use case management

The use of a Scrum board is essential to the methodology we had chosen. As such we chose JetBrains' YouTrack tool. This application provides functionality to document user stories and their use cases. Each new sprint a new scrum board was created in YouTrack to track progression of features implemented in Quizzard.



3.3.2. Source code management

The application's source code is stored in a Git repository located on a GitLab instance.

We made extensive use of the continuous integration capabilities of GitLab to test our application. Using continuous integration (CI) runners running on multiple servers' pipelines were run performing unit test runs, checkstyle validation, and deployment to an online test environment. For detailed information on tests, please see section 8 on our 'Test plan & execution'.

3.3.3. Branching strategy

During development, we have applied a branching strategy in which each feature is developed in a separate branch. After completion of a feature, tests should be created to validate that the feature is in correct working order. Features can only be merged back into the master branch when all tests in that branch are passing.

When there were any bugs in the master branch that needed fixing quickly, a special fix branch was branched off of master. This fix could then be easily applied to the code in the master branch and any feature branches that were affected.

3.3.4. Documentation

At some places in the application back and front end need to interface with each other. To make sure the correct information is sent and/or received by either side we set up Swagger using which API endpoints are documented.

3.4. Project log

The table below shows the steps taken while developing the project per week. We agreed with the client to have weekly meetings. These meetings were used to review the results up to that point (sprint review, see section 3.1.) and to derive and discuss requirements for the application. One meeting with the client was skipped due to sickness.

Week	Activities
1 6/2-10/2	Requirement gathering: <ul style="list-style-type: none">• Stakeholder descriptions• Use cases• Quality requirements• Functional requirements
2 13/2-17/2	<ul style="list-style-type: none">• Review project proposal with client• Design database diagram• Define and create mock-ups for views• Draw UML diagrams• Global system architecture, including system, software, database• Set up repository and software project• Reflection proposal
20/2-24/2	<i>Vacation</i>
3 27/2-3/3	Sprint 0 <ul style="list-style-type: none">• Review project proposal• Familiarize with frameworks and tools• Finalize mock-ups

4 6/3-10/3	Sprint 1 <ul style="list-style-type: none"> • Create template from mock-ups • Implement audiences
5 13/3-17/3	Sprint 2 <ul style="list-style-type: none"> • Implement quizzes
6 20/3-24/3	Sprint 3 <ul style="list-style-type: none"> • Start implementing quiz runs
7 27/3-31/3	Sprint 4 <ul style="list-style-type: none"> • Finish implementing quiz runs
8 3/4-7/4	Sprint 5 <ul style="list-style-type: none"> • Finish Minimum Viable Product • Start writing reports
9 10/4-14/4	Sprint 6 <ul style="list-style-type: none"> • Create presentation, poster and design report draft • Additional testing of the application • Fixing problems
10 17/4-21/4	<ul style="list-style-type: none"> • Presentation • Finishing touches on the deliverables

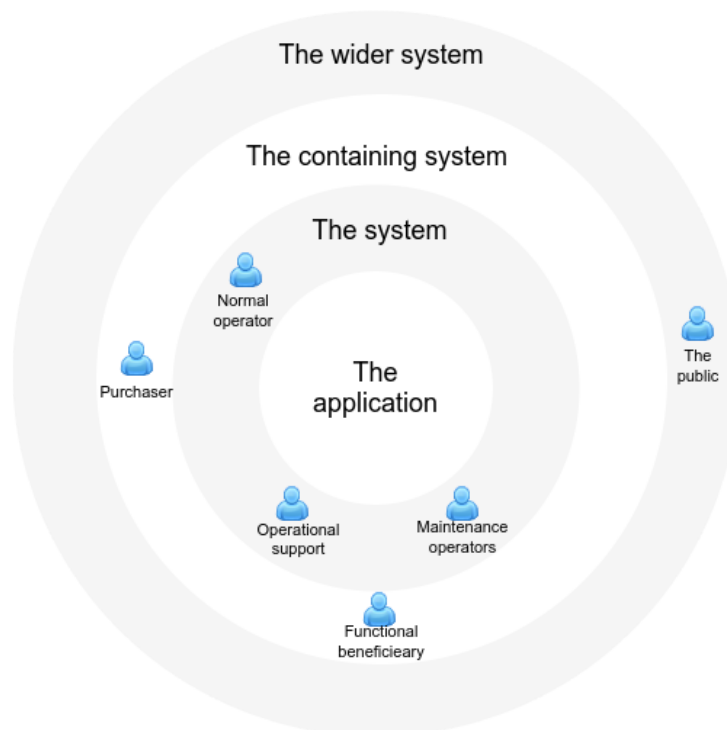
4. Domain analysis

4.1. Introduction

The application will be introduced in the educational domain in which quizzes are taken to test the knowledge of students. In particular, it will be used as a diagnostic test in the module Software Systems.

4.2. Stakeholders

In this section, the stakeholders of the application will be briefly discussed. The stakeholders are shown in the diagram below and are discussed in the following sections.



4.2.1. The system

Normal operator

All staff that will use the application to organize and facilitate quizzes. Most importantly the educational staff of the Software Systems module at the University of Twente. These could be the teachers, teaching assistants and other teaching staff that will use the system for performing diagnostic tests or any other kind of quiz.

Maintenance operators

There are no dedicated maintenance operators for the application since the application will run directly on the personal computer of the teacher. When the application is run on demand on a computer of a teacher, this teacher can be seen as maintenance operator.

Operational support

Teaching assistants and other teaching staff that will walk around during the execution of a quiz to answer the questions of the participants. They will provide the operational support for the application during the execution of a quiz. There is no predetermined operational support for creating and administering quizzes.

4.2.2. The Containing System

Functional beneficiary

The functional beneficiary of the system are the students that will participate in the quizzes, such as the diagnostic test of the module Software Systems. These will actually use the functionality of the system to participate in the quizzes.

Purchaser

Although there is no real purchaser in the sense of payment for the application, the application is commissioned by the University of Twente, which will be the 'purchaser' of this application.

4.2.3. The wider environment

The public

The public might be persons that come into contact with the system without participation in quizzes, such as people that are watching the execution of quiz. Since the system will be designed to be more generic than only for student quizzes, it might be that the public can also consist of people that participate in quizzes.

4.3. Existing software solutions

In the domain of interactive quiz applications, there are lots of applications that have functionality comparable to the current quiz application and the desired application. Some examples of these applications and their problems will be discussed in this section.

4.3.1. Kahoot!

Kahoot (www.getkahoot.com) is an online platform for creating quizzes in the form of games that can be played centralized. Questions appear on a central screen while participants have the possibility to answer from their own devices, such as a computer, mobile phone or tablet.

Kahoot allows questions to have user-submitted images or YouTube videos with up to four possible multiple-choice answers. All the questions are added individually. After a quiz has been played, it allows for results to be exported in the form of an Excel sheet. Within Kahoot all participants enter a quiz with a game pin which after they enter a display name by which they are identified. After the quiz, the results can be exported by the organizer.

Although Kahoot works great for simple quizzes, it has some shortcomings for more complex quizzes. Kahoot only supports questions in the form of text or one image. This is not really flexible for asking complex questions such as programming questions or questions based on multiple charts. Moreover, Kahoot only supports two to four answers per questions, which might not be enough for some quizzes. Also, Kahoot does not support having answer labels, forcing the user to use their colorful shapes as answer labels. Although this looks nice, this may not get the point across in certain cases.

4.3.2. Socrative

Socrative (www.socrative.com) is an another online platform, this time centralized around "rooms". With a room number, anyone can enter a room in which an organizer can hold a quiz or ask a quick question. The questions appear on the screen of the participant which can answer the question at their own pace if the organizer desires so.

Socrative allows questions to have user-submitted images with multiple types of answers: multiple-choice, true-or-false and short (open) answers. All questions are added individually. It is also possible to make a copy of a quiz created by another organizer. The organizer has the option to play the quiz at the pace of the participants or at their own pace. Results can be seen by the organizer immediately after a participant has answered a question. After a quiz, the results can be downloaded as an Excel sheet or a PDF.

Socrative has a lot more features when compared to Kahoot and therefore solves most shortcomings of Kahoot. However, Socrative does not try to integrate the presentation of the lecturer with the quiz. Socrative shows the questions directly on the screen of the participant and the lecturer gets the possibility to request the answers of the participants at every time, but it is totally separated from the presentation of the lecturer. This means that the lecturer is disconnected with the audience while executing the quiz, which might not be desirable.

4.3.3. Shakespeak

Shakespeak (www.shakespeak.com) takes integration with the organizer slides very seriously. The primary use case of Shakespeak is to integrate the questions with Microsoft PowerPoint by providing a Microsoft PowerPoint plug-in. An organizer can create their own slides in Microsoft PowerPoint and add special slides via the plug-in that are populated at runtime with the question and the results at the same time.

In terms of questions, Shakespeak only supports multiple choice questions as participants only have the multiple choice options on their own device. Shakespeak also enables students to answer by SMS, when a working internet connection is not available. The organizer has to start the quiz in advance before presenting the PowerPoint slides such that the PowerPoint plugin can connect to the Shakespeak services and fill the Shakespeak slides. Afterward, the organizer can download the results using the plug-in or the website.

Although Shakespeak has a limited feature set when compared to Socrative, it has great integration with the slides of the organizer. However, it does only support Microsoft PowerPoint on the Windows operating system. This renders it useless for LaTeX or even Google Drive customers. Besides this shortcoming, Shakespeak may only be used for educational use only, which limits the use cases.

4.3.4. The existing Python application

The existing Python application has no knowledge about the questions or the slides that are presented. It is a separate application in which the organizer should start each question manually, separate from the slides that are shown.

The existing application has the option to only allow the students that are predefined in the application. These students can answer the questions through their own device by means of a self-refreshing web page. Afterward, the organizer can download the results from the application, as long as the application is still running. The results are only kept in memory.

This application has a lot of shortcomings. This application will not be able to keep up with the ever-growing amount of participants of the course Software Systems that is expected in the coming years. The technology used by the application to keep clients in sync with the server is very resource heavy as it continually refreshes the page on the client side. Also, results in the application are not persistent over reboots and it is difficult to integrate the presentation with the quiz, as already mentioned.

4.4. Actors

As part of the discussed domain, we have found two actors. These actors will be using the quiz application. In this section, these actors and their role in the system will be briefly discussed.

4.4.1. Organizer

The organizer is the person creating, administering and organizing quizzes. The organizer will use the application to create quizzes in the system and start them for a chosen audience. The organizer wants to manage the quiz while it is running and see the statistics of the quiz afterward.

4.4.2. Participant

A participant takes part in a quiz held by the organizer. The participants take part in the quiz by using an internet connected device which is able to run a Javascript-enabled web browser. On this device, the participant can authenticate himself (when needed) and answer questions.

5. Requirements

5.1. Requirement engineering

Multiple strategies have been used to come to a final set of requirements. In this section elaborates on these strategies.

5.1.1. Weekly meetings

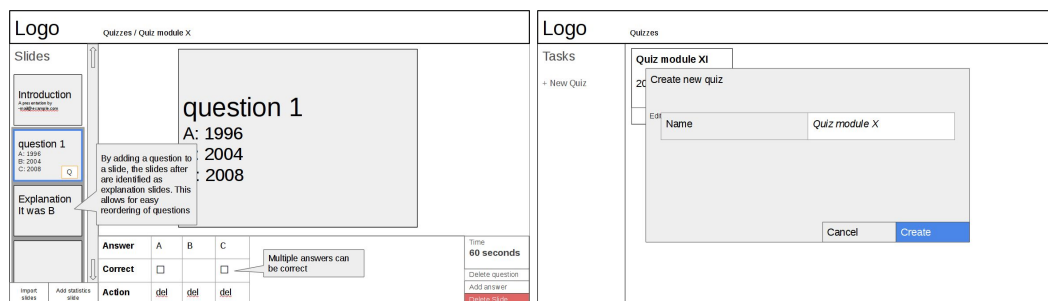
Most requirements were gathered in the first few weekly meetings with the client. During these meetings, we were able to form a clear understanding of what the envisioned application would look like. After two meetings a project proposal with an initial list of requirements was presented to the client. After some tweaking of the requirement priorities, the document was then approved. The client was able to bring forward additional requirements during the project. Whenever uncertainties arose about the implementation of requirements, different solutions were proposed to the client which then was then able to express their preferences.

5.1.2. Existing solutions

More requirements were collected by looking at existing solutions that provide some of the desired functionality. The client was very clear at expressing problems encountered when using some of the pre-existing quiz applications. A demonstration of the old Python application was requested to get a better idea of what the current workflow was like.

5.1.3. Mock-ups

Based on an initial set of requirements, a series of mock-ups was created. These mock-ups were meant to get a feeling of what user interface elements should look like and how they fitted together on the pages. The creation of mock-ups was extremely useful as it sped up the process of implementing the final user interface in HTML and informed the client of our concrete ideas for the application and its usage. The reactions of the client on the mock-ups was positive which confirmed the requirements that had been collected up to that point. The full set of mock-ups was included in the project proposal.



Some of the mockups that were created. The quiz editor (left) and the quiz runs index page (right) are shown here.

5.2. Functional requirements

This section describes the functional requirements in the form of user stories. Each user story is written in the form 'As an <actor>, I want <functionality>'. The user stories are ordered by their importance of existence in the application using the MoSCoW (Must, Should, Could, Won't) method, in which each word prioritizes the user stories.

5.2.1. Must

The user stories in this section describe the bare minimum requirements for a working application.

As an organizer, I want to

- create/read/update/delete a quiz.
The organizer should be able to define quizzes including questions and answers. The organizer is also able to change the quizzes after they are created.
- define how much time there is to answer a question
The organizer should be able to set the time (in seconds) that the participants get to answer the question. After the time limit expires, the question cannot be answered anymore.
- explain the answer to a question after the question
The organizer has the possibility to define an answer and explanation to a question, which the organizer can show after the question has been answered by the participants.
- manually start the next question
The organizer is able to determine when the next question is presented to the audience.
- choose the quiz mode
When a quiz is started, the organizer can choose the quiz mode. This mode will be used for a single run of the quiz.
- see statistics of the quiz during the quiz
The organizer can enable a screen which shows the current statistics of the quiz, like the participant scores or the distribution of answers given to the questions asked.
- integrate PDF pages/slides as questions of the quiz
The organizer can use pages from a PDF file as questions in a quiz. This enables for content-rich questions with images and annotations.
- exclude questions from the high score during the quiz
The organizer can determine to exclude a question from the high score calculation during the quiz. This might be needed in case of an invalid/incorrect question.
- restrict participation to the audience in the room
The organizer can make sure the audience of the quiz is restricted to selected participants.
- export a participation list (when available)

An organizer has the possibility to export a list of participants to a quiz when these statistics are available (i.e. not in an anonymous quiz).

- export a result list
An organizer has the possibility to export a list of results, which contains overall statistics of each question or the answers for each individual participant.
- choose a group of participants (audience) for my quiz
An organizer can choose a pre-defined audience for a quiz when the quiz is started. This makes it possible to restrict the quiz more easily to a pre-defined audience.

As a participant, I want to

- participate in a quiz
A participant should be able to launch the application and take part in a quiz if the participant is allowed to.
- see my own score during the quiz (when available)
A participant should be able to see his own score during the quiz when a score is provided by the selected quiz mode (chosen by the organizer). This score is provided on the device of the participant.
- use my own device (i.e. laptop or phone)
A participant should be able to use his own internet enabled device (with a javascript-enabled web browser) to participate in the quiz.
- see the remaining time to answer a question
A participant is able to see the remaining time that is left to answer a question on its own device.
- see multiple choice answers to choose from
A participant is able to see the possible answers to choose from on his own device. The participant can easily choose one of the answers on his own device.
- login with personal authentication
A participant should be able to login to a quiz using personal authentication when this is required for the quiz (as chosen by the organizer).
- login anonymously, without personal authentication
A participant should be able to start a quiz without any authentication when this is required for the quiz (as chosen by the organizer).

5.2.2. Should

The user stories in this section are not part of the minimum requirements for a working application, but they have all been implemented in order to create a richer experience.

As an organizer, I want to

- import a CSV file of participants to create an audience for a quiz
An organizer has the option to provide a CSV file with names and email addresses of participants, that can be imported as an audience, which can be used for restricting a quiz.

- choose if statistics should be shown on the screen
An organizer can choose if the statistics should be shown on the screen next to the questions. In this way, the organizer can choose to hide the actual statistics.
- choose if the questions should be shown on the screen
An organizer can choose to show statistics on the screen. In this way, the organizer can choose to hide the questions, such that the organizer can make a screen that only shows the statistics.
- show a PDF page as an explanation for a question
The organizer can choose to define a page or slide of a PDF file to show as an explanation to the answer instead of only a textual explanation. This can be shown on the screen after the question has been answered.
- run a new quiz with remaining questions from the previous quiz
An organizer can run a new quiz with the remaining questions of the last quiz. This makes sense when a quiz ends early, which might be the case in quiz modes like last man standing.
- pause and resume a quiz
An organizer might want to pause a quiz to resume it at a later time without loss of participants and scores.

As a participant, I want to

- compare my answers to the correct answers after a quiz
The participant has the possibility to compare his answers to the correct answers (when available) after the quiz has ended. The participant can request an overview of the questions with his answers.

5.2.3. Could

The user stories in this section describe requirements that are nice to have and will only be implemented if there is enough time.

As organizer, I want to

- reorder questions
An organizer has the possibility to reorder the questions before the quiz starts. This makes it possible to easily reuse quizzes in different orders.

As participant, I want to

- get an email with my results of the quiz (in case of personal authentication)
The participant can get an email with the results of the quiz after the quiz has ended. In this case, an anonymous participant should provide an email address. In the case of an authenticated participant, the email address is most likely already known.

5.2.4. Won't

The user stories in this section describe requirements that will not be implemented in the application and are outside the scope of this proposal.

As an organizer, I want to

- control if participants can see the questions on their own screen
An organizer can choose to show the questions of the quiz on the participant's screen.

As a participant, I want to

- see the questions on my own screen
A participant has the possibility to show the question on their own device. This way the participant can make the quiz independently on their own device.

5.3. Quality requirements

In this section, the qualitative requirements will be defined for the application. The application must meet these quality factors to fulfill the project goals.

5.3.1. Functional suitability

The requirements in this section are quality requirements that describe the functional completeness and correctness of the application.

- The application should help the organizer to easily export the results
- The application should have a correct calculation of the statistics related to a quiz
- The front end of the application must be usable in at least 90% of the mobile and desktop browsers

5.3.2. Performance efficiency

These quality requirements describe requirements for the time behavior and capacity of the application.

- The application should be able to handle at least 300 participants that take part in a single quiz in parallel
- The remaining time to answer a question should be synced with the server and never deviate more than 2 seconds

5.3.3. *Compatibility*

This section describes quality requirements that are related to the interoperability of the application with existing applications and services.

- The application should be able to use existing PDF files as questions for a quiz

5.3.4. *Usability*

These quality requirements are related to user error protection and learnability of the application. Also, application accessibility requirements are described here.

- A new user to the application should be able to understand the workings of the participant interface after one question has passed
- Color blind people should be able to use the application without use of extra software or a separate device

5.3.5. *Reliability*

The quality requirements described in this section are related to fault tolerance and recoverability of the application. Also, the availability of the application is discussed in this section.

- In case of an application crash, it should be possible to resume a quiz from the last question that has been asked to the participants
- As long as a quiz run is active, the application should be available at all times to the participating participants

5.3.6. *Security*

The quality requirements in this section are related to security of the application, such as the confidentiality, integrity, and authenticity.

- The application must prevent impersonation by users of other users when this is required by the organizer
- The application may not allow a participant to view the answers given by other participants in ways that can lead back to the participant

5.3.7. *Maintainability*

This section describes quality requirements related to the modularity, reusability, modifiability and testability of the application.

- The application allows for easy (code-wise) modification when adding new quiz modes to the application

5.3.8. Portability

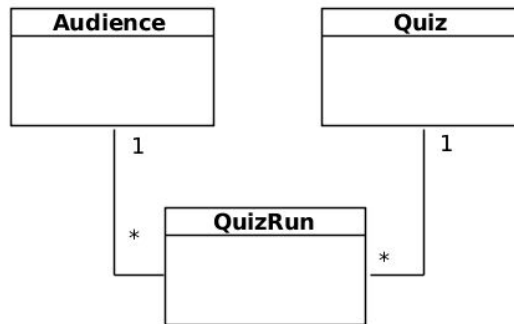
These quality requirements are related to the adoptability, installability, and replaceability of the application.

- The application should save its configuration in a separate directory
- The application must be installable on supported versions of at least the three most popular desktop operating systems: Windows, macOS and Ubuntu

6. Architectural design

6.1. System overview (top-level concepts)

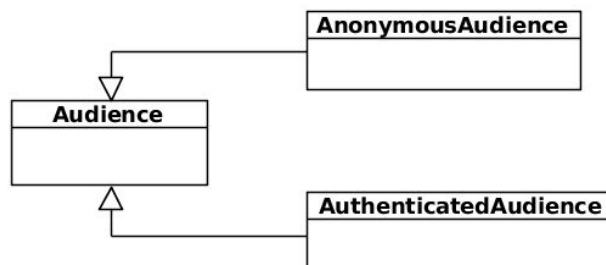
Looking at the application from a global perspective, there are three top-level concepts that can be distinguished: audiences, quizzes, and quiz runs. Each of them will be explained in this section. The design choices let to the creation of these concepts will be further discussed in section 7 on 'Detailed design'.



Abstract relations between Audience, Quiz, and QuizRun

6.1.1. Audiences

An audience is a group of people that participate in a quiz. There are two subtypes of audiences that can be distinguished: authenticated and anonymous ones.

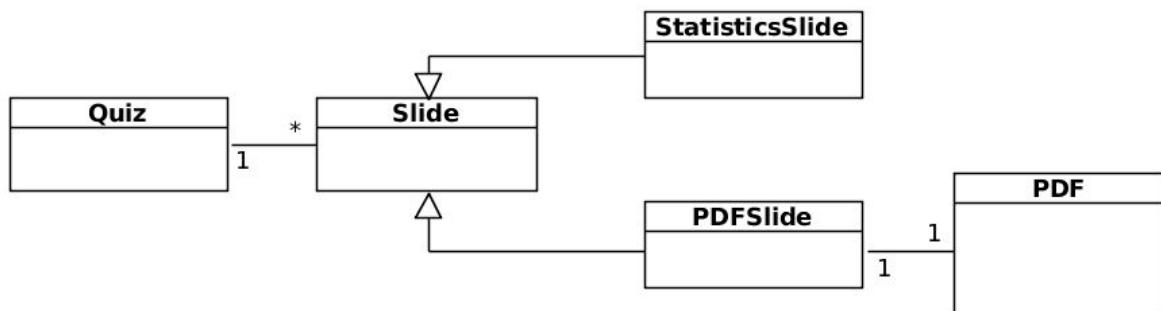


The AuthenticatedAudience requires a list of participants and will send all of them a unique token to login. This ensures some integrity and prevents impersonation. On the other hand, the AnonymousAudience allows for easy participation. A link is shared, for instance in the presentation room, via which anyone can join. Participants choose their own name, as such, it's trivial to impersonate others.

6.1.2. Quizzes

A quiz is an ordered set of slides. Some of the slides have questions with answers attached to them. A single quiz can be started multiple times in different modes via the QuizRun type, which will be discussed later.

An integral part of the application is the use of PDFs as a medium for quizzes. Essentially a quiz wraps around a PDF file. It does this through PDF slides, each of which is tied to a specific page in a PDF file on disk. This looks roughly like the following:



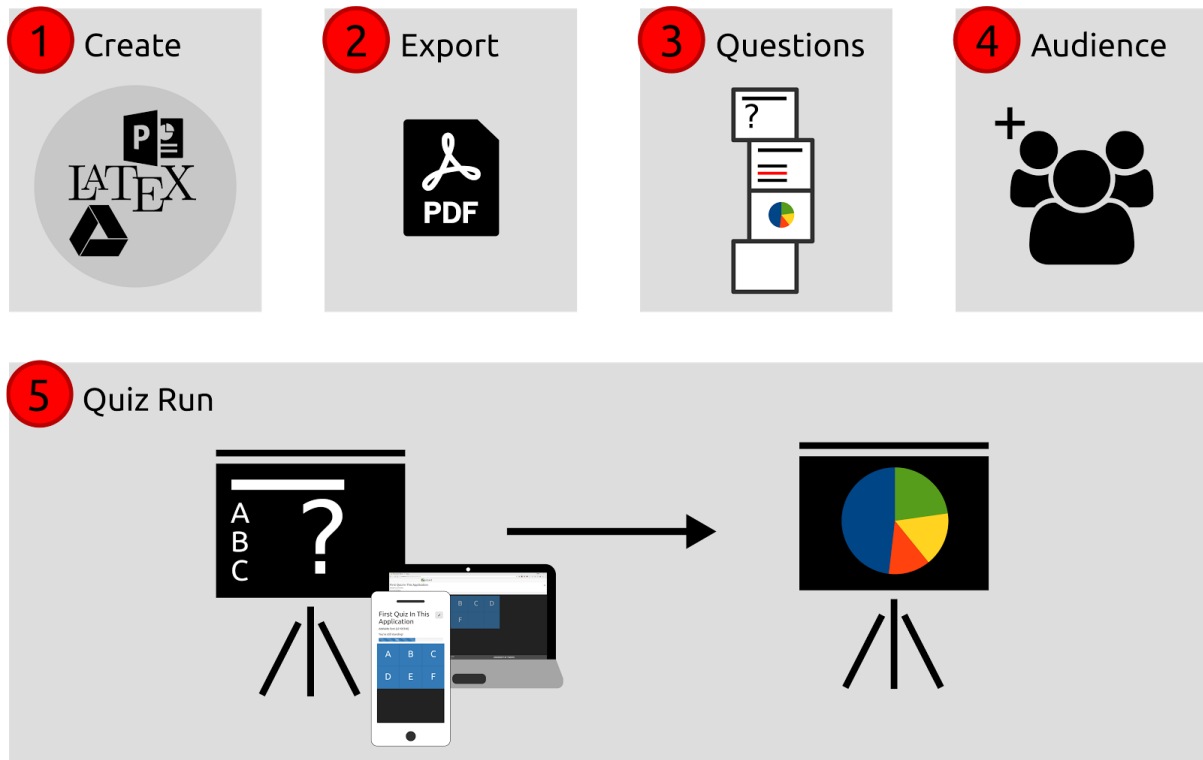
With this structure, it is also possible to have slides from different PDF files in the same quiz, choose orderings that differ from those in the

6.1.3. Quiz runs

A quiz run is an instance of a quiz tied to an audience. Additionally, a quiz run has a quiz mode attached to it. For every question that has been presented an instance of QuestionRun is created. For every participant that answers a question, a ParticipantAnswer is created. If a participant refrains from answering or is added after the quiz has started, no instance of ParticipantAnswer is present.

6.2. Flow Overview

In this section, we will have a look at how Quizzard is meant to be used. Using the diagram below the flow of the application will be discussed step by step.



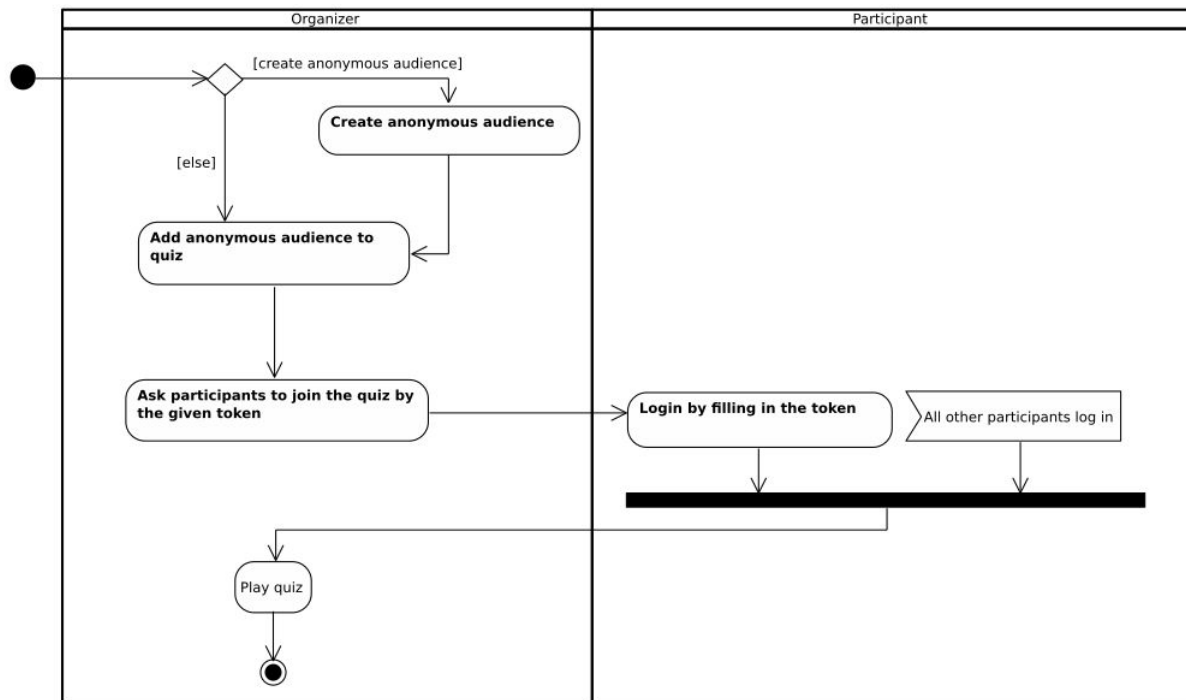
1. Create a presentation in any application capable of exporting documents as PDF.
2. Export the presentation to a PDF file.
3. Import the PDF. Add questions to slides and add statistics slides to the presentation.
4. Create an audience for the presentation.
5. Run the quiz. During the presentation participants in the audience can answer questions. Statistics slides added to the presentation will show statistics depending on the quiz mode.

6.2.1. Quiz run activity diagrams

During the design of the application, multiple activity diagrams were created to get a better understanding of the how quiz runs should be implemented.

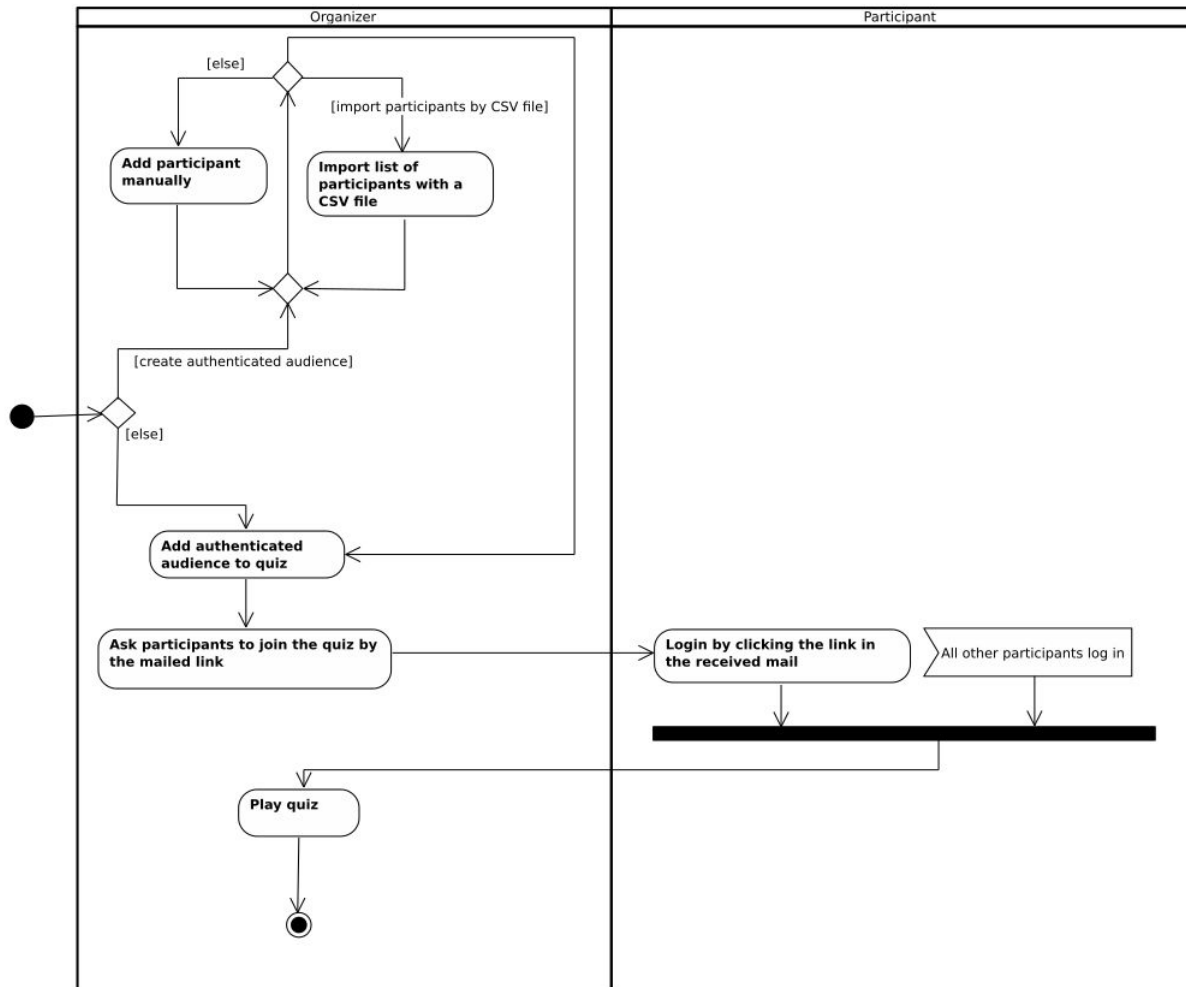
Playing an anonymous quiz run

The following diagram shows how a quiz run is played with an anonymous audience. The organizer starts by linking an anonymous audience to the quiz run if there is no audience linked yet. After the audience is linked, the organizer asks the participants to log into the quiz with the quiz token. The activity "Play quiz" is shown in detail in section 6.2.1.3.



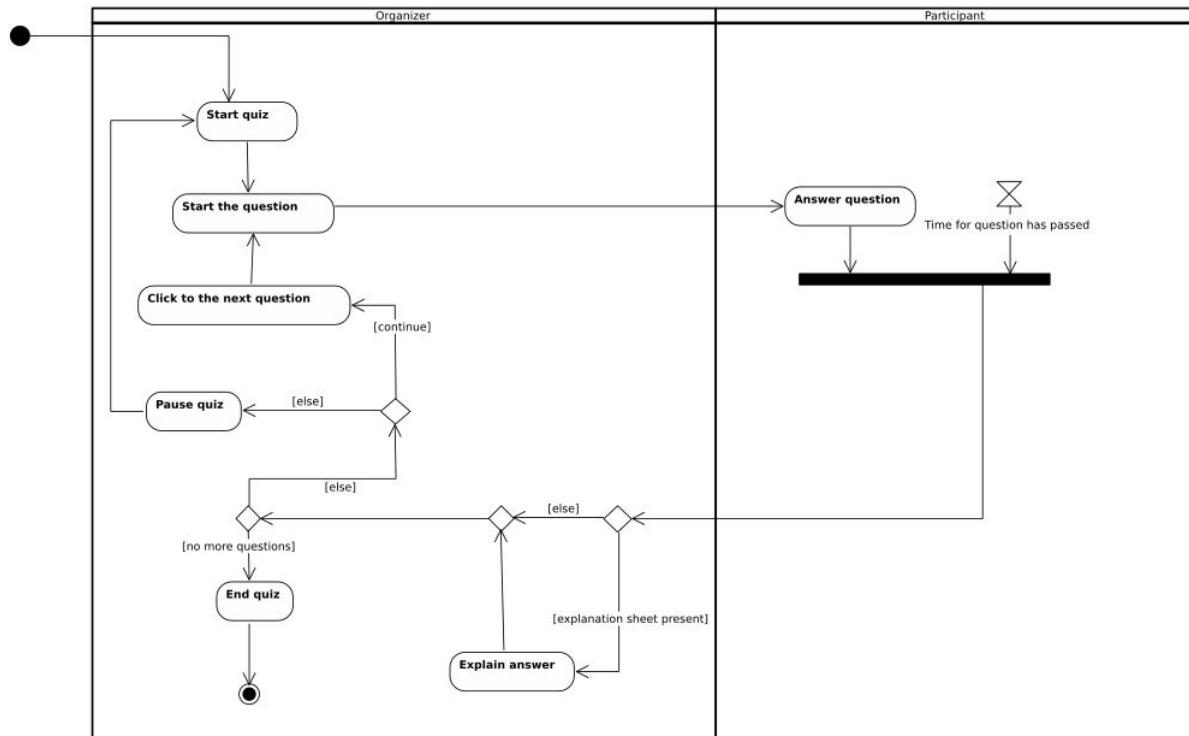
Playing an authenticated quiz run

The following diagram shows how a quiz run is played with an authenticated audience. The organizer starts by linking an authenticated audience to the quiz run if there is no audience linked yet. After the audience is linked, the organizer asks the participants to log into the quiz with the emailed link. The activity "Play quiz" is shown in detail in section 6.2.1.3.



Playing a quiz run

The following diagram shows how a quiz run goes. The organizer starts the quiz and so the first question. The participants either answer the question or the time for the question passes. If there are explanation sheets, the organizer can explain the answer. Before going to another question, the organizer has the option to "pause" the quiz, to resume it later.



6.3. Preliminary design choices

Here we will discuss decisions made regarding the implementation of the application before the actual design phase of the application had commenced.

6.3.1. Programming language

Since all project members are acquainted with Java this language was chosen to build the back end in. Furthermore, the use of Java makes the application supported on a wide range of devices that offer support to the Java runtime environment.

6.3.2. Build management

The Quizzard project is managed using Maven. Maven provides a unified way of managing projects and doesn't require IDE specific functions or tools, which is needed as we use build servers which have to be able to compile and test the code automatically without an IDE. As a result, testing and building Quizzard, therefore, can be done solely using Maven commands. Maven is centered around the pom.xml file in the root of the project.

6.3.3. *Web framework*

It was decided to keep the deployment strategy of the new application similar to that of the Python application, which implies that the application will run on the organizer's machine. This meant that whatever web framework used, it would have to be portable and not require a large number of dependencies to be installed. As such, Spring Boot was chosen as it satisfies both these requirements. Spring Boot is an extension of the Spring MVC framework which eases configuration significantly and therefore lessens the 'boot' time of the project.

Spring is able to produce 'fat jar' executables. These jars include an application server which removes the need to set up a dedicated application server like Tomcat for the application. This simplifies deployment of the final application and again reduces the need for extra configuration of such a server. Additionally, it relies only on Java 8 which is easy to install on a lot of platforms, without adding external repositories or executing potentially dangerous scripts as root.

6.3.4. *Application deployment strategy*

The existing Python application runs directly on the organizer's machine. We evaluated alternatives to this deployment model, but it was quickly established that creating a permanently hosted instance would lead to a lot more requirements in terms of permission management and user authentication (for more details on this subject see section 10.1.6). They would also limit the flexibility of the application, as the data would be stored on university hosted servers which might not be accessible to third parties. As such, the decision was made to run directly on the user's computer instead of having a single instance running on a distant server.

6.3.5. *Database*

As a result of the deployment strategy, the database needs to be just as portable as the application. An embedded SQL database management system was needed. Various SQL databases were carefully evaluated, and SQLite was chosen due to the wide availability of tooling for the database files, which helps make the application extra flexible and make troubleshooting easier. Other contenders were Apache Derby, H2DB, and HyperSQL. All these database management systems are able to run embedded and have excellent Java support, but aren't as popular as SQLite and as such may be more difficult to troubleshoot.

Further reading

- SQLite <https://sqlite.org/>
- Apache Derby <https://db.apache.org/derby/>
- H2DB <http://h2database.com/>
- HyperSQL <http://hsqldb.org/>

6.3.6. Object Relational Mapping framework

To ease the interaction between the database (SQL) and Java objects an Object Relational Framework was needed. An ORM framework helps mapping entities to data in the actual database. As such, complete entities can be queried from the database and obtained as a Java objects.

Hibernate was evaluated at first, as it's the go-to option for any development in conjunction with Spring Boot. Hibernate takes a Java-first approach, with the tables and the SQL queries all being generated under the hood. However, Hibernate's extreme abstraction from the database makes it hard to write complex queries efficiently, and as such another framework was needed. Java Object Oriented Query, or JOOQ, turned out to be a more suitable framework which allows more low-level interactions with the database, without having to deal with the complexities of the Java Database Connectivity (JDBC) API.

JOOQ uses a schema first approach which allows us to tightly control the schema's features. JOOQ automatically generates data access objects (DAOs) and model classes based on our schema and allows for type-safe SQL queries where needed. These features are used throughout the application. JOOQs code generation is used in combination with Flyway, which manages migrations of the database schema.

Further reading

- JOOQ <https://www.jooq.org/>
- Hibernate <https://hibernate.org/>
- Flyway <https://flywaydb.org/>

6.3.7. Prefer back end rendering over front end rendering

A popular way to make modern web applications is to use so-called *front end rendering*, in which the Javascript front end consumes JSON REST API's which are then displayed. We have had some experience making these kind of sites and know that for simple pages there is a lot of overhead and code duplication involved. As such, the front end is kept as light as possible, offloading rendering and pre-processing of the HTML to the back end whenever possible. In some cases where rendering by the back end is not feasible we use React (see section 7.3.8).

6.3.8. JavaScript and CSS dependency management

Modern web applications have a lot of dependencies to manage. Our application depends on multiple JavaScript libraries and CSS frameworks, which all have their own dependencies. In order to manage those dependencies we decided to use NodeJS in combination with the Node Package Manager (NPM). This choice was made based on the possibility to integrate NodeJS and NPM into Maven. Also, the Node Package Manager is one of the largest JavaScript and CSS dependency managers out there, which gives us access to all the JavaScript and CSS dependencies we want.

Further reading

- NodeJS <https://nodejs.org/>
- NPM <https://www.npmjs.com/>

6.3.9. Primary Javascript Library

When the project was first set up, it was decided to use AngularJS as the main JavaScript library for the front end. After some testing, specially with mixed back end and front end rendering, this decision was reconsidered because of the nature of AngularJS. AngularJS wants to control the whole page as a single page application. This behavior is, as explained, undesired as we do not want to make a single page application. Therefore, React was chosen as alternative to AngularJS. React is developed by Facebook to make interactive JavaScript components on an existing page. This is more aligned with the way we wanted to combine front end rendering with back end rendering, and as such it was chosen over Angular. Although React libraries to make single page applications exist, React can handle simple components on an existing page just fine.

Further reading

- React <https://facebook.github.io/react/>

6.3.10. CSS framework

In a modern web application, it is important to have a logical layout and consistent styling throughout the application. In order to style a web application, so called Cascading Style Sheets (CSS) are used. Since writing a working cross-browser CSS layout takes a lot of time, there was decided on using the Bootstrap CSS framework as starting point for the application. Bootstrap provides a powerful grid layout and styling for most HTML components, such as form input fields. Although Bootstrap provides great utilities for a consistent layout, it does not provide complex styles needed for some advanced views in our application. These views were designed from scratch on top of existing Bootstrap components.

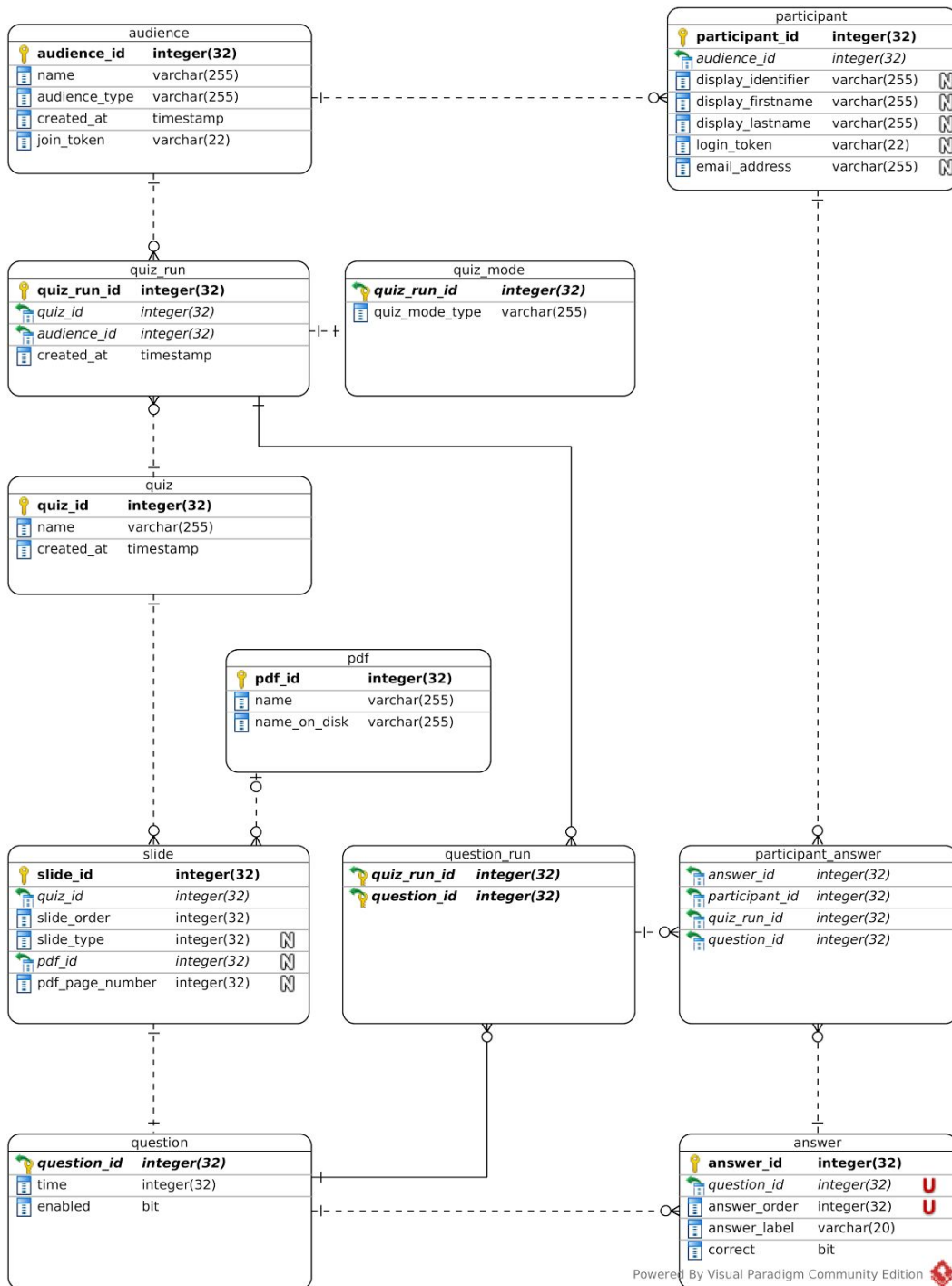
Further reading

- Bootstrap <http://getbootstrap.com/>





7. Detailed design

7.1. Entity relations

The entities in our application and their relations to each other are shown in the entity relation diagram below.



Legend

	Column is (part of) the primary key
 or 	Column is part of a foreign key
	Column is nullable

Note that the program used to generate the above diagram (Visual Paradigm) does not support TEXT columns, so TEXT columns are represented as VARCHAR(255), although those would have no 255 character limit in SQLite.

Some things to note about this diagram:

- An audience can be in multiple quiz runs, and a quiz can be in multiple quiz runs. The same audience and quiz can even have multiple quiz runs
- Every question that is presented during a quiz run will get a 'question_run' entry. This marks that a question has been presented even when no participant answered it.
- PDFs are not stored in the database, but in a folder next to the application. The pdf table, however, describes the files in this folder. PDF filenames are sanitized and duplicate names are resolved after upload, which is the name_on_disk field.
- The mode of a quiz run is represented as a 1-to-1 relationship with a quiz_mode entity. This relation is for future extensibility where a quiz mode has configurable options. These options would then be stored in the quiz_mode table.
- The participant table has a lot of fields that are nullable. This is because both anonymous and authenticated participants are stored in the same table. The presence of at least a name is enforced in the application for an anonymous participant, and the presence of an email address is enforced for an authenticated participant.

7.2. Quiz runs to separate quizzes, audiences and quiz modes

The separation of quizzes and audiences as is currently present in the application wasn't so obvious at first. Initially, quizzes were tied to an audience and had their own quiz mode set. This, however, does not allow the organizer to reuse quizzes. We found that this was such a major inconvenience that a quiz should not be tied to an audience and quiz mode directly. Quiz runs were created in reaction to this and allow quizzes to be reused.

7.3. Websocket protocols and communication

The communication is handled with STOMP over websockets. In this section, we will provide some background on how this works and why we chose to use these techniques.

7.3.1. STOMP

Websockets offer bi-directional data flow and as such let the server push a new state to the client, instead of having the client periodically check if a new state is available. Websockets themselves, however, only offer a very low-level message-based service. To still enable the development of a well-structured application, we use STOMP, or 'Simple Text Oriented Message Protocol' over websockets. STOMP provides many HTTP-like features, like headers, destinations in the form of URLs and verbs like MESSAGE and SUBSCRIBE.

Message channels

STOMP works with so-called message channels. These channels are identified with URL-like names, for instance, /quizruns/5/present is an example of such a channel. Both clients and the server can send messages to these channels with a STOMP MESSAGE frame, which is somewhat similar to an HTTP POST. Where STOMP differs from HTTP is that it does not have a strict request-response based architecture - where in HTTP every request is returned by a response from the server - even an empty response is a response - STOMP does not have this. Instead, with STOMP you can "subscribe" to a message channel with the SUBSCRIBE frame and then any message sent over that channel will be received by the client.

Message design

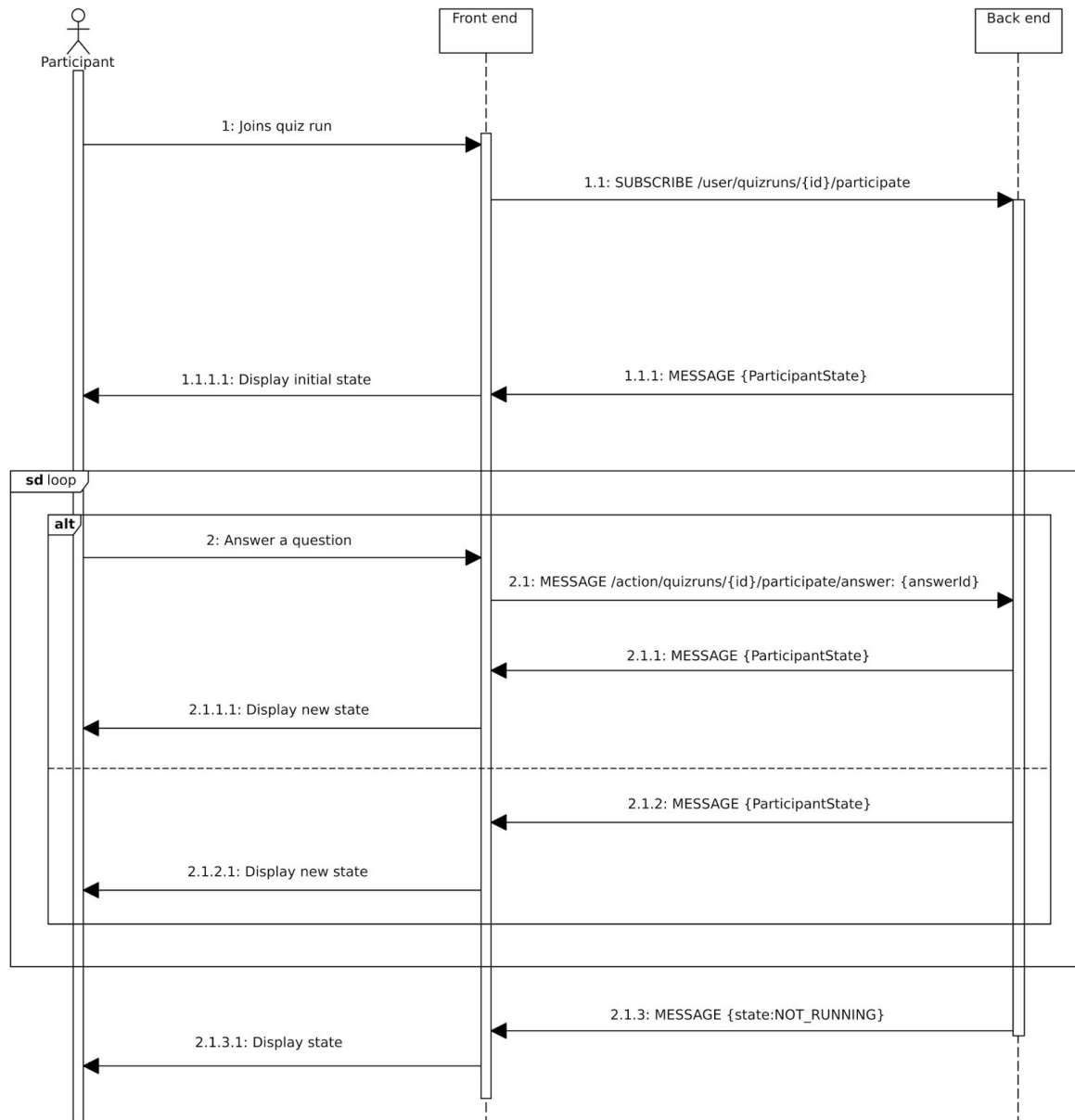
The back end always sends the complete state over the message channels to the front end - there are no partial updates or events going over the channel. This means the front end does not need to keep track of an internal state and can instead display exactly what the back end returns without keeping track of the history.

Further reading:

- STOMP <https://stomp.github.io/>

7.3.2. Participant network sequence diagram

The following diagram show the communication between the participant and the application over websockets. The participant initially joins a quiz run. When a question is started the participant answers the question. When the quiz ends the participant gets a final message that the quiz run has ended.



The participant connection consists of a single channel (/user/quizruns/{id}/participate) to which the participant subscribes (1.1 in the diagram). This is done via a STOMP SUBSCRIBE frame, as indicated in the diagram. Directly after connecting it will receive a state update describing the current state. Every time the internal state for the participant changes, a new state update is sent from the server to the client. The objects that are returned are always of the same type.

State updates from the server

Displayed below is an example of a state update when a question is available for the participant to answer (as JSON):

```
{
  state: DISPLAY_QUESTION,
  questionId: 5,
  timeTotal: 30_000,
  timeLeft: 21_540,
  quizModeState: (...),
  answers: [
    {
      answerId: 10,
      answerLabel: "A",
      selected: false,
    },
    {
      answerId: 11,
      answerLabel: "B",
      selected: true,
    },
    {
      answerId: 12,
      answerLabel: "C",
      selected: false,
    }
  ]
}
```

The `quizModeState` field depends on which quiz mode is selected (last man standing, poll mode, classic mode). For more information about how the `quizModeState` object is created, please refer to the technical manual.

The `timeTotal` and `timeLeft` values are in milliseconds. The answer array contains only information that is important for the participant: the `answerId`, which is needed for communicating which answer is chosen to the back end, the `answerLabel` and a boolean `'selected'`, indicating if this answer was selected.

The available values for `'state'` are:

- `DISPLAY_QUESTION`, as shown above
- `DISPLAY_NOTHING`, for when there is no question. All other fields, except the `quizModeState`, will have a value of null.
- `NOT_RUNNING`. All other fields are null. This is sent to close the connection, like in step 2.1.3 in the above sequence diagram.

Answering a question

To answer a question, the participant sends the answerId to the server (2.1 in the diagram) via a STOMP MESSAGE frame. The payload is the answerId integer encoded as a JSON string. If the answer changes the participant state, a state update will be sent to the client. This behavior is different than it would be with HTTP since STOMP is not a request-response protocol, where any message can wait for a response from the server. Instead, a message is fire-and-forget, after which the server can send a new message as a reply, which is received asynchronously.

7.3.3. Presenter sequence diagram

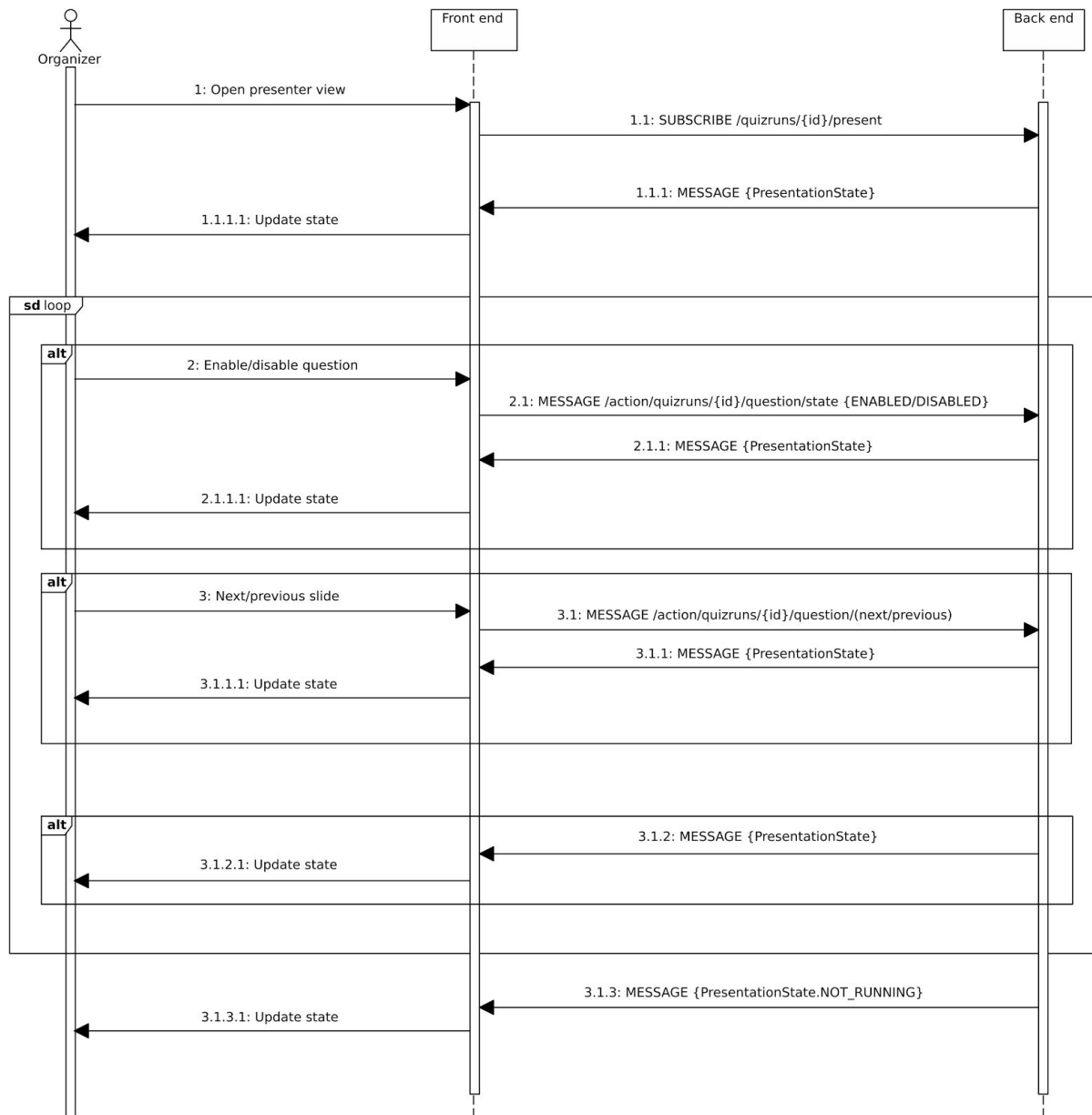
The presenter protocol consists of two STOMP channels:

1. `/quizruns/{id}/present`
2. `/quizruns/{id}/statistics`

The `{id}` placeholder is the integer identifier of the quiz run.

On the present channel (1), things like the current slide, number of slides and the question (when available) is broadcasted. On the statistics channel, only statistics information is broadcasted. This separation is made to support having a dedicated screen for displaying only statistics (the 'scoreboard'). This scoreboard view only has to subscribe to the statistics channel.

Below is a sequence diagram entailing the exchange of the presenter without the statistics channel, which will be discussed later.



In contrast to the participant, there are now three actions available:

1. Disable/enable a question (2.1) by sending either "ENABLED" or "DISABLED" MESSAGE to /action/quizruns/{id}/question/state
2. Go to the next slide (3.1) by sending an empty message to /action/quizruns/{id}/next
3. Go to the previous slide (3.1 as well) by sending an empty message to /action/quizruns/{id}/previous

As with the participant view, any change on the server causes it to send a full state object to the client. This also includes changes from other presenter front ends - for example when the organizer has two tabs open on the presenter page, clicking in any of them on the 'Next' button makes both tabs display the next slide.

The `PresentationState` object looks like this:

```
{
  state: DISPLAY_SLIDE,
  previousSlide: SlideDetails,
  slide: SlideDetails,
  nextSlide: SlideDetails,
  navigation: {
    currentSlide: 5,
    totalSlideCount: 26
  },
  canDisableQuestion: true,
  canEnableQuestion: false
}
```

The "state" field can be either

- `DISPLAY_SLIDE` (as shown above),
- `NO_MORE_SLIDES`, for when the end of the presentation has been reached, and
- `NOT_RUNNING`, when the quiz run was stopped but the presenter is still subscribed to the channel. In this case, all fields other than 'state' are null.

The `SlideDetails` object represents the slide and its information at which we will look in a moment.

The state also includes the current position in the presentation (number of slides, current slide) and if there is a question to either enable or disable. The `canDisableQuestion` and `canEnableQuestion` fields are both false in only two cases:

- There has not been a question yet in the current slide or slides occurring before the current slide
- The presentation is at the end in the '`NO_MORE_SLIDES`' state.

These fields are never both true, as that would make no sense.

Now the SlideDetails object may look like this:

```
{
  slideId: 207,
  slideOrder: 56,
  slideType: PDF_SLIDE,
  pdfPageNumber: 18,
  pdf: {
    pdfId: 10,
    name: "Some Pdf File.pdf",
    nameOnDisk: "SomePdfFile (2).pdf"
  },
  question: {
    questionId: 5,
    time: 60,
    enabled: true,
    answers: [
      {
        answerId: 28,
        questionId: 5,
        answerOrder: 25,
        answerLabel: "A",
        correct: false
      },
      {
        answerId: 28,
        questionId: 6,
        answerOrder: 26,
        answerLabel: "B",
        correct: true
      }
    ]
  }
}
```

In the presentation state, three slide detail objects are included. This lets the front end download the PDF for the next slide, in case the slide type is a PDF_SLIDE. PDFs are downloaded from a normal URL outside the websocket protocol.

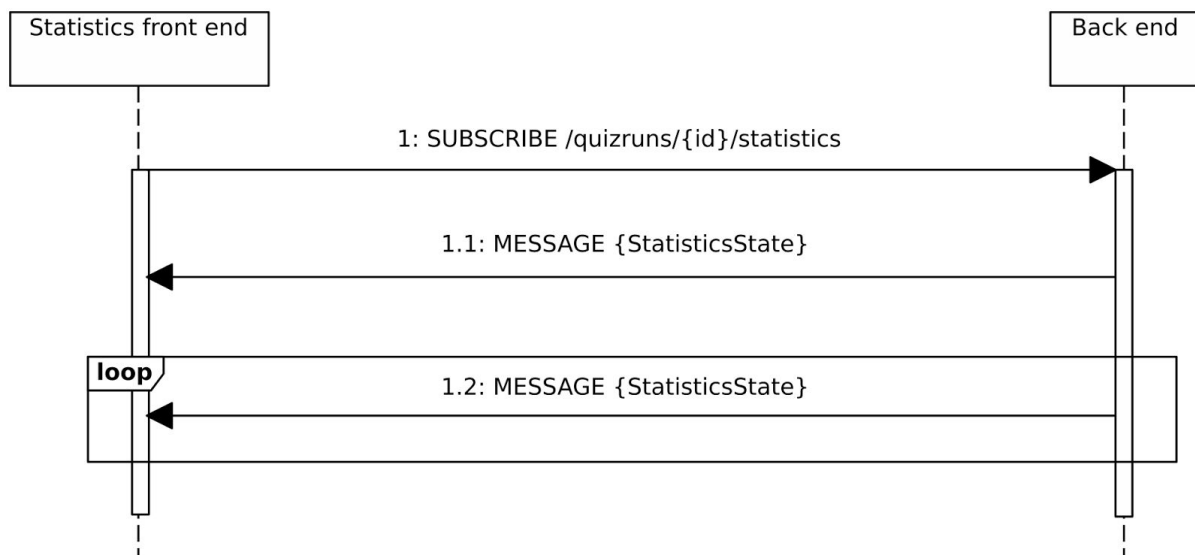
The slideType is either PDF_SLIDE or STATISTICS_SLIDE. When it is STATISTICS_SLIDE, all pdf-related fields are null (pdfPageNumber and the pdf object).

If there's a question, the slide will have a question object (as shown above). If there is no question, this field will be null. The question also contains a list of answers.

Both the answer objects and the slide itself contains an "answerOrder" and "slideOrder" field respectively. These are used to sort the objects on and enable reordering like it can be done in the quiz editor.

7.3.4. Statistics sequence diagram

As mentioned, statistics are communicated via a channel independent of that of the presenter. This is because there is a pop-out view available which displays only statistics, which just has to subscribe to the statistics channel (the scoreboard). The following diagram shows the communication between the front end statistics slide and the back end. Whenever a statistic slide is shown, a message is sent from the back end with information about the statistics slide belonging to a quiz mode.



In the above diagram, it becomes clear that the statistics front end purely listens to whatever the back end sends it, and can exert no control over it. Whenever the organizer moves to the next slide (previous diagram), this will prompt an update over the statistics channel. Like all the other protocols, only complete state updates are pushed which means the client does not need to store previous states.

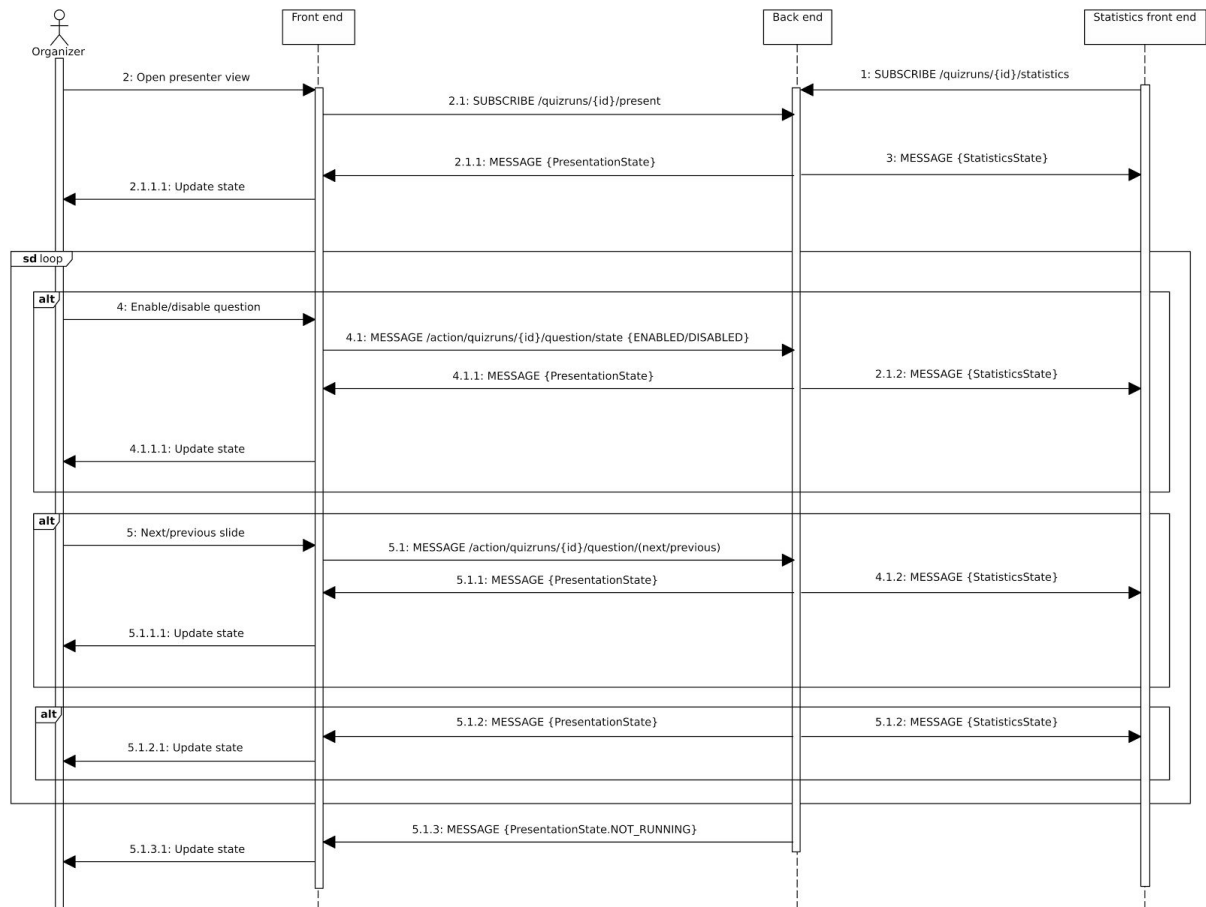
The StatisticsState object may look like this:

```
{
    quizModeType: LAST_MAN_STANDING
    state: (...)
}
```

The design is quite similar to the statistics bit of the participant state. It includes the type of the quiz mode (LAST_MAN_STANDING, CLASSIC or POLL) and it includes a state object. The state object depends entirely on the quiz mode that is active.

7.3.5. Presentation and statistics combined

The statistics sequence diagram is closely intertwined with the network sequence diagram. When put in a single diagram, it looks like this:



In this diagram, we see two front ends: one for the presentation, and another for the statistics. This is what it might look like when you have a dedicated statistics front end open. What can be distilled from this diagram is that for every `PresentationState` message the back end sends to the presenter front end, a new `StatisticsState` object will also be sent to whoever's listening on the statistics channel.

7.4. Saving answers to questions

The application has some interesting requirements when it comes to persisting participant answers. First and foremost, the performance of the application: it should allow some 300 participants to answer a question at the same time. Furthermore, the application should maintain a consistent state to be able to pause and resume a quiz at will.

To achieve these features a special mechanism was devised. Upon receiving participant answers the application stores them into memory. Only as the next slide - the slide after the question, this is - is triggered the answers are stored in the database in a single transaction. This ensures that either all answers are saved or none are saved at all (e.g. in the case of error).

7.5. Back end design patterns

To better understand the structure of the back end we will discuss what patterns were used. This section should offer insight in why these patterns were used and where they are applied.

7.5.1. MVC

The Model View Controller pattern is used throughout the application. Individual web pages are controlled by their controller class which loads an HTML view from the resources directory. The controller supplies model classes to the view when needed. Most model classes are automatically generated from the database schema by the JOOQ framework. In cases where data not directly available from the database is required by the view, specialized data transfer objects are used. The model data is included in the HTML views using the Thymeleaf template engine which pre-processes views server side.

The MVC pattern was not only chosen because it is inherent to the the web framework we use, but also as it is common practice to write web application in an MVC way. MVC helps writing structured and easy to follow the code.

7.5.2. REST

Many front end React components need to perform actions on the database which is managed by the back end. The React components connect to the back end through Restful API endpoints. REST is used as all team members were already used to designing and writing RESTful API endpoints. Additionally, REST has only a minimal overhead because of its use of HTTP methods for CRUD operations. Additionally, REST endpoints accept and serve JSON (JavaScript Object Notation) data that is easily created or consumed by the front end. Note that, since our implementation of endpoints relies on the current session, they are not truly RESTful as a true REST API requires to be completely stateless.

7.5.3. Dependency injection

Bringing together database functionality and controllers can be a difficult task. To make access to often used services, such as database access objects, easier we use dependency injection (also known as Inversion of Control). Spring provides a so-called Inversion of Control Container. Classes annotated with one of the Spring stereotypes `Repository`, `Service` or `Configuration` are applicable to be loaded in the IoC container. Whenever such a class is referenced in the application code and annotated with `Autowired` it will be loaded into the IoC container and injected where needed. The lifetime of injected objects can be controlled using the `Scope` annotation.

Dependency injection is used throughout the application for accessing the database through Data Access Objects. Additionally, commonly used functionalities are modeled as services which can be injected where needed.

7.5.4. Lombok

While writing a large application like Quizzard there is a lot of repetitive programming work involved. Classes need their getters, setters, and constructors which are all-important for the function of the application but contain no real logic. This creates a large amount of boilerplate code which has to be maintained and spent time on - time which could have been used for other, more important things. While there are solutions to overcome writing all this boilerplate code be it by declaring all fields public or by means of smart IDE suggestions, these are far from ideal as well. By making all the fields public, you lose encapsulation, and when you let the IDE generate those methods you still have the problems of maintaining them.

Lombok is a tool that helps developers not only write less boilerplate code but also hides the code from view completely. Through the use of attributes and a bytecode manipulator, writing code is sped up considerably. For example, the `Data` annotation instructs Lombok to automatically add getters and setter for all fields. These functions will be invisible in the class itself but can be used by other classes in the application like they were normal methods. We chose to use this tool to decrease the time needed writing repetitive code such as getters and setters. Lombok is mainly used in model classes that were not automatically generated.

Further reading

- Project Lombok <https://projectlombok.org/>

7.6. Security

Security for an application like this was challenging, as it does not match the typical security model of a web application. This is because in a typical application you log in once with a user that is valid for the whole application. Quizzard, however, does not have the concept of a user - there's a `Participant`, but the `Participant` is bound to an audience. The physical user sitting behind a computer can be logged in as multiple `Participants` at once in different audiences.

The spring security framework has the concept of Authentication with a Principal. The Authentication is directly bound to the user visiting the website. The Authentication has a list of roles the user has (like ORGANIZER), and has a Principal object that represents the user. It is expected that you supply your own Principal entity there that represents the user. As mentioned, Quizzard cannot easily provide such a user-representing class as there is no 1-to-1 relationship between a logged in user and a participant entity. A participant is locked to an audience, and a user can enter multiple tokens to effectively become multiple participants in different audiences. Instead, the principal used in the application is a UserAccessList object. The UserAccessList contains a map that maps audienceIds to participantIds the user has access to. The Authentication only differentiates between if someone has entered the organizer token or when they have not, and returns a different list of roles depending on that.

7.7. Websockets and transport

Quizzard faces an interesting challenge regarding pushing new states to over 500 participants in real-time. The old Python application served a web page that would reload periodically. This setup is less than ideal: it puts a lot of stress on the web server and the network, and participants may see an old state for a short while. These problems are solved by using websockets.

7.8. Front end design patterns

In order to understand the structure of the front end, the design patterns used for the front end are discussed. This section should offer insight in why these patterns were used and where they are applied.

7.8.1. Less

Managing CSS files for front end applications can become very complex for large applications. This is mostly because CSS by default lacks some important features, such as importing other CSS files and nesting some complex classes. Therefore, the front end CSS files are written in Less. Less is an extension of the CSS language with support for nested statements, variables and importing different Less files. The Less files are translated to CSS using the Less compiler, which is included in the application.

The Less files for the application are included in the `/src/main/web/less` directory of the application sources. Each view that needs additional styles has a separate Less file that contains the CSS classes for that view. All these Less files are included using import statements in the main `nashorn.less` file. This pattern gives the advantage of easily compiling this Less file to one CSS file, which is included in the application upon compilation.

The front end builds upon Bootstrap to provide consistent styling. Bootstrap has Less source files that can be used to modify Bootstrap before including it in a web application. In order to keep Bootstrap managed by the NPM package manager, the

Bootstrap Less source is included through a Less include statement in the main `nashorn.less` file.

7.8.2. *React and JSX*

As already mentioned, the front end makes use of React as main JavaScript library. React defines itself as a view only framework in which views on the page are divided into components. Components can be nested into other components to minimize duplication of code.

React components can be written in just plain JavaScript, but there is also an XML flavored JavaScript syntax called JSX. JSX makes it possible to write HTML snippets directly inside the JavaScript code. This makes it very easy to define how a React component should be rendered. Therefore, all components of the application are written in the JSX JavaScript syntax.

For the application, all React components that are written are ECMAScript 6 classes that include each other via ECMAScript imports. These imports are resolved on compilation time when the compiler converts the JSX components to one JavaScript file.

All React components are included in the `/src/main/web/jsx` directory of the application sources. The classes are divided over multiple directories. Each directory contains components that are working together in one view, except for the `components` directory, which contains reusable generic components.

Further reading

- ECMAScript 6 Features <http://es6-features.org/>

7.8.3. *Parsing of date and time*

The front end has a special React component which uses the `momentjs` JavaScript library which parses the date time on each page. This component tries to show the date and times on different pages in the local timezone of the user. In order to detect the timezone of the user, the `momentjs` timezone plug-in is used.

In every place where a date and time is shown, the back end renders the timestamp as HTML5 data attribute and the `react-datetime` CSS class is added to the HTML element. This attribute is parsed by the component to get the UTC time. This UTC time is converted to the local timezone and shown.

To let the front end parse and show the dates, the back end must add the HTML5 data attribute and the CSS class to the element which contains the date.

7.8.4. *Dynamic asynchronous views*

The front end includes some views that are updated asynchronously with data from the back end. New information can be requested by the view through the REST API or can be send to the view through a websocket subscription.

Because React is a view-only framework, it does not contain code to request and receive data from REST endpoints or websockets. This is done through different JavaScript libraries. In order to not clutter all the React components with code of these libraries, two types of React components can be distinguished: stateless and stateful components. The stateful components contain some controller code that retrieves state from the back end which is then saved in the state of the component. Stateless components do not contain this code and rely on the state of their parent component.

As a quick example let's look at the quiz editor component. There is one top-level component that is stateful and contains all controller code. This component communicates with the REST API to get all data and saves it in the state of the component. This state is passed to the stateless components that actually form the quiz editor, such as the sidebar, question editor and quiz lint tool. These stateless components update based on the data they get from the stateful parent component.

Since these views actually need to have some context to parse the data they retrieve from API endpoints some model classes are created for the most complex views. These model classes are plain ECMAScript 6 classes. They are used to save and maintain the data used by the stateful React components that function as controller.

8. Test plan & execution

Testing software is extremely important to be able to say something about the quality of the application. Testing is an important part of our development cycle. Every newly completed feature should be tested and will not be accepted before all test in the application pass. In this chapter, we will discuss in detail how the application is tested.

8.1. Strategies

Testing the application is done in multiple ways. We will look at the various test strategies, how they are implemented and what they are supposed to test.

8.1.1. Unit tests

Most of our unit tests are used to determine the proper functioning of all controllers, resource endpoints, and websockets. For the controller and resource endpoints tests there are special test services available: `ControllerTestService` and `ResourceTestService` respectively. These classes provide easy access to data access objects (DAOs) for persistent storage access and a `MockMVC` endpoint simulating connections to the back end. Using `MockMVC` we are able to test all common HTTP methods on our controllers and check return HTTP status codes as well as the model returned by a controller.

8.1.2. Integration tests

Using integration test we aim to test the front end in combination with the back end. Although most of the back end is already tested through the use of unit tests, it is important to verify that actions can be achieved successfully by using the elements of the user interface. Using the `Fluentlenium` framework pages can be modeled as Java classes using which actions on the actual web page can be invoked. These tests assure page element are actually rendered and that the user is able to perform the desired actions on a given page.

8.1.3. Concurrency and performance tests

The project has some important performance requirements and as such key parts of the code are tested for performance and correctness under concurrent loads. It is important to test under high, multi-threaded loads as certain concurrency-related problems may only show up in those cases.

8.1.4. Real world tests

In our initial planning, a real world test was planned in week 8 of the project. In week 7 Pieter-Tjerk de Boer was contacted since he was the only Computer Science teacher teaching a very large crowd of students. Unfortunately, Pieter-Tjerk's teaching activities had already seized at the proposed time of testing. A second opportunity for a real world test with some seven participants could not be realized due to an apparent case of miscommunication.

To adjust for the lack of real world tests, a synthetic load test was introduced. The load tester is a separate test tool which connects to an actual instance of the application. It simulates a configurable amount of participants and talks to the application instance using actual websocket connections. The simulated participants pick random answers to answer questions.

We have conducted occasional small real world test sessions with the four of us to determine whether new functionality has been implemented to our satisfaction. Complementary to this the application was tested multiple times during client meetings together with the client.

8.1.5. Checkstyle compliance

We value a consistent look of the application code. To achieve this a checkstyle compliance test is in place. Using Checkstyle a set of rules was defined which is imposed on all application code. Most of the rules make sure the code remains readable. To name a few rules that are tested:

- One-character variable names are disallowed
- Code lines may not be longer than 120 characters
- Correct indentation levels

8.2. Test results

8.2.1. Unit tests

A feature branch can only be merged into our master branch when all tests are passing. As part of the continuous integration effort, all tests are run whenever a commit is pushed to the online repository. These tests have proven their worth time and time again. Many breaking changes made during development have been detected and corrected thanks to these tests.

8.2.2. Integration tests

Integrations tests are also run as part of the continuous integration pipeline. The same as with unit tests hold with front end tests as well. A feature branch can only be merged into our master branch when all tests (including front end tests) are passing.

Some edge cases were discovered with these tests where the front end did not communicate with the back end properly. Also, some misspelled words in the front end were found this way.

8.2.3. Performance tests

The performance of the application was tested in two ways:

- By means of a unit test which tests the critical sections which will experience high load, but does not set up real network connections and instead mocks the network code with Mockito. It simulates a question, having all participants answer concurrently and then click to the next slide to prompt storing all the answers.
- A load tester client which needs to be run on another computer. It simulates participants all the way from logging in to the point of sending answers over the websocket connection.

Concurrency test

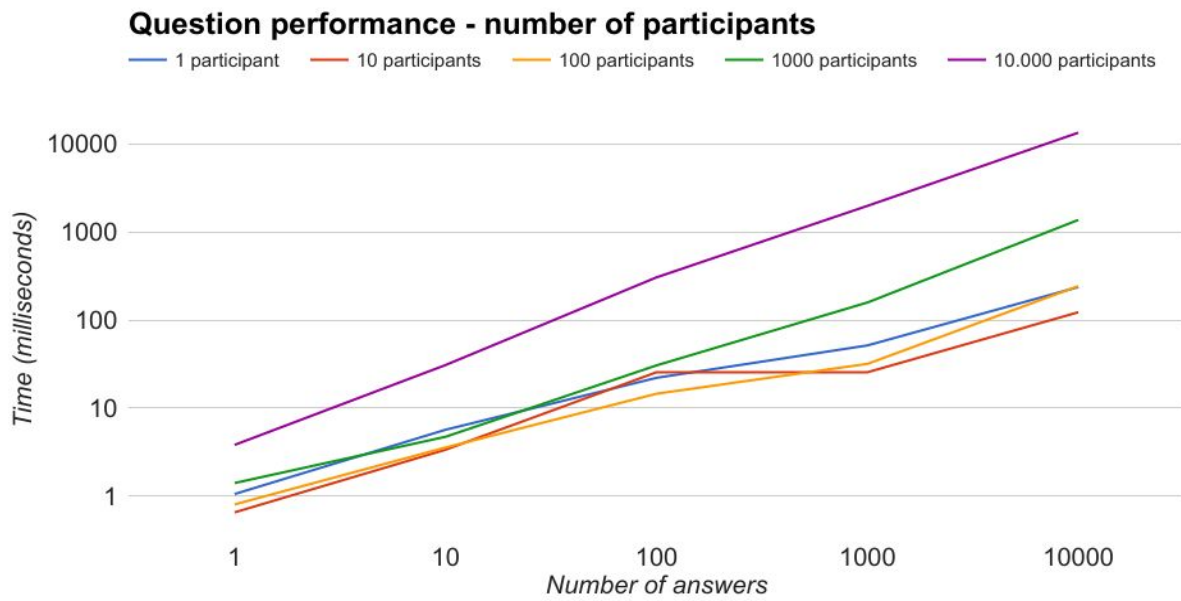
Data from the concurrency unit test can be put in a graph with a varying amount of participants and answers given by the participants. Note that the number of answers is the total number of answers given.

These results were obtained on a Lenovo W540 laptop with an i7-4700MQ (4 cores, 8 threads), 8GB DDR3 (1600Mhz), Ubuntu 16.04 with Oracle Java 1.8 (build 1.8.0_121-b13).

The test does the following things:

- Open a question slide
- Have n participants send in questions until m questions have been sent
- Go to the next slide and wait for the application to store all the answers

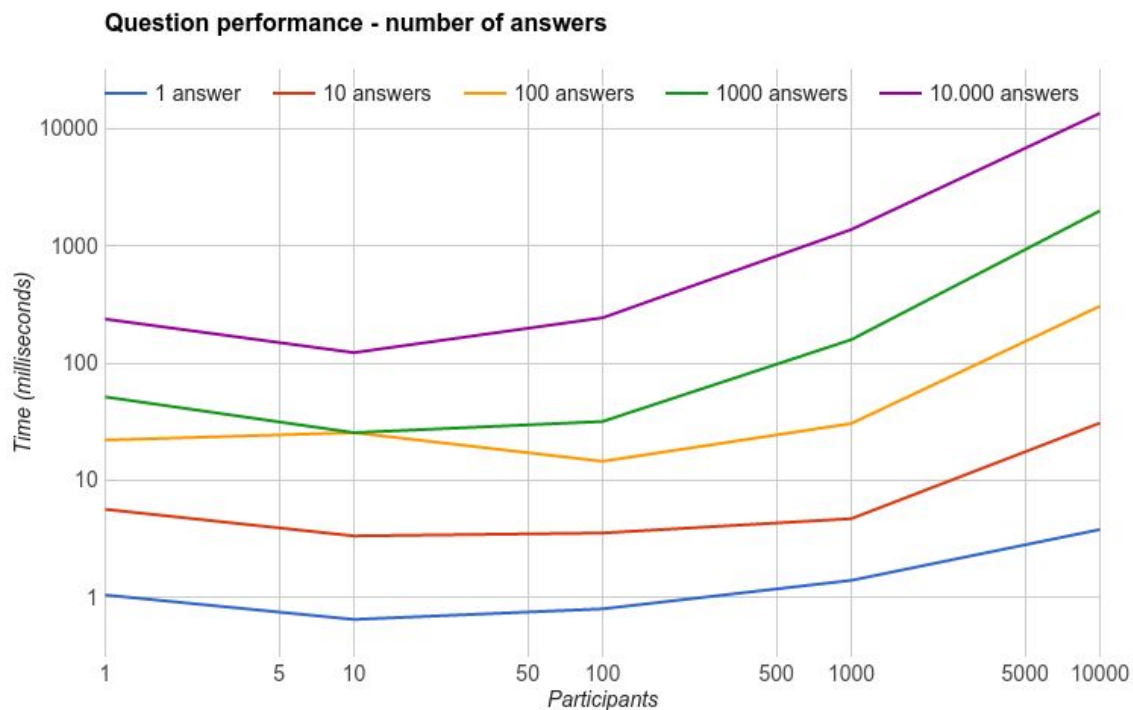
Where $n \in \{1, 10, 100, 1000, 10.000\}$ and $m \in \{1, 10, 100, 1000, 10.000\}$. Every combination of n and m have been run 20 times and the average time is taken.



Performance test showing how a given number of participants is affected by the number of answers given

What can be discerned from the above graph is that the performance is linear over the amount of answers submitted. If participants submit twice the number of answers, it will take twice as long to process. This is obvious - the amount of time per answer is fixed and as such, this is an $O(n)$ algorithm.

The data can also be represented with a given number of answers, looking how the answer time varies when changing the amount of participants.



Performance test showing how a given number of answers is affected by the number of participants

An interesting quirk of the JVM shows up here. The JVM needs time to "warm up" the code and JIT compile it. With just one participant, this either cannot happen or cannot be amortized over enough participants, causing the average time to be higher than with 10 participants.

Increasing the number of participants does *not* result in a linear increase in the amount of time taken to store the answers. This is because the quadratic nature of some of the data that needs to be searched from `ParticipantSubscriptionRegistry`. Many of the operations in this class are linear-time operations, whereas they could have been constant time with different data structures. The choice for these less-efficient data structures was because of their other advantages - the code is much easier to understand and to keep bug free in this way.

Looking at the graph - 500 participants handing in 2 answers each (changing their mind once) yields 1000 answers, which takes about 100 milliseconds. This means the application passes the performance efficiency requirement, as it proves that at least this component can support 300 participants in a single quiz.

Load tester

The load tester is a client that can be used to perform a full-stack test. It has a CLI interface that requires a host, a join token, a quiz run id and the number of participants to simulate. After connecting it has options to have all participants send a

random answer. The tool proved that the application can handle the 300 participants required by the performance efficiency requirement.

8.2.4. Real world tests

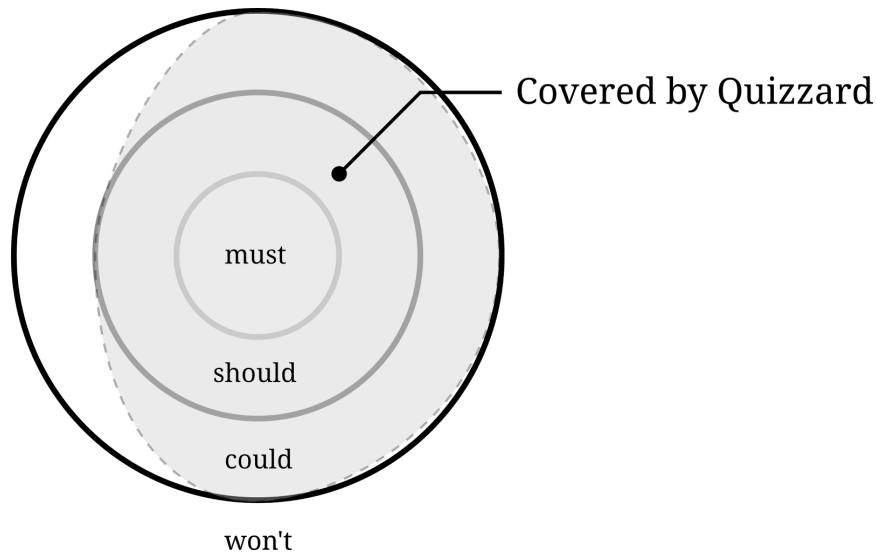
Although no real world test have been conducted we have gained valuable insights from our synthetic load test. These tests have proven that our application remains stable with up to 1000 users connected at the same time.

Our small real world test sessions have brought about many subtle changes to how the quizzes work. Multiple times the application exhibited unexpected behavior during these sessions. Often times these flaws could easily be fixed and would have gone by unnoticed otherwise.

9. Conclusion

In this section, we will look back at the requirements as defined in section 6 and see which of those have been implemented during the project.

9.1. Functional requirements



The above Venn diagram indicates how user stories covered by the Quizzard applications relate to their MoSCoW classification. All user stories from the must and should priority classes have been implemented, and some from the 'could' class. No user stories have been implemented from the won't class.

9.2. Quality requirements

9.2.1. Functional suitability

The application supports exporting results of a quiz run by the organizer. The organizer can export the results in two different ways. The results can contain a list of the answers given by the participants or the results can contain a list of ones and zeros for the correctness of questions. This requirement has been proven to work by a unit test.

For the different quiz modes, the application has a way to calculate the statistics. The correctness of those calculation has been proven with numerous system tests.

The participant front end is developed according to the mobile-first principle. This means that the participant front end is developed in order to be optimized for mobile devices but it is still viewable with a desktop or laptop.

Because the organizer front end is not meant to be viewed on a mobile device, it is not developed according to the mobile-first principle. However, the pages are still optimized for mobile devices afterward, so the pages can still be used with a mobile device.

9.2.2. Performance efficiency

The application can comfortably handle 300 participants. For more information about the performance and the performance tests of the application, see section 8.2.3.

The time to answer a question in the front end is synced with the back end. During the system tests, this was evaluated and it is expected to work on 90% of all devices.

9.2.3. Compatibility

The whole application is built around PDFs. Already existing PDFs with quizzes can be imported into the application and used for quiz runs.

9.2.4. Usability

There were several system tests where participant came into contact with our application for the time. These tests showed that participants easily understood what to do and how to answer a question.

The participants and the organizer can use their own device to play a quiz. Thus no additional software or a special device is needed for someone to use the application.

9.2.5. Reliability

In the case of an application crash, it is possible to resume a quiz from the last question that has been answered by the participants. When a question slide is shown, all participant answers are stored in application memory. When the organizer goes to the next slide all the in-memory participant's answer are stored in the database. If the application were to crash during a question the answers will not have been stored. After restarting the application the quiz run will start at that question, since it is not answered yet.

After a quiz run is started, participants can still join the quiz run by logging in with their token. This has been tested with system tests and during several occasions like meetings with the client, the demo market, and the end presentation.

9.2.6. Security

The application can identify different participants if it is desired. When a participant is added to an authenticated audience, they will receive their own token by which the application identifies them. With unit and systems tests it was proven that participants could not answer questions for other participants via the application.

9.2.7. Maintainability

The application allows for the implementation of additional quiz modes. For more information about how to implement a new quiz modes, see the technical manual section 3 "Extending quiz modes".

9.2.8. Portability

The application stores its PDFs in a separate folder and the rest of the data in an SQLite database. This folder and database file is located in the same directory as the jar file. These files could be given to another person and they would have the exact data and PDFs.

There are some properties of the application that can be configured before the application is started like the SMTP mail server. These properties are in a file called *application.properties*. For a list of configurable properties see the owner's manual. Just as with the PDFs folder and SQLite database, this file can be given to another person to use for their own instance of the application.

Since the application is built with Java 8, the application can be run on any device with a working Java JRE 8. This includes Windows, macOS and Ubuntu.

10. Future planning

In this section, we will have a look at future of the Quizzard application.

10.1. Additional features

All required features are present in Quizzard, however, this does not mean there isn't any room for improvements to the application. Below there will be a list of possible improvements which could be implemented if more time was given.

10.1.1. Data mining & analysis

As has been stated, quiz runs that use authenticated audiences store who answered what on which question. These results could be used to help students that scored low. With the help of data mining tools within the application, the organizer can easily track the progress of participants and aid them when needed.

10.1.2. Additional quiz modes & quiz mode DSL

Currently, the application support three quiz modes. Additional quiz modes can be added in order to support the needs of different organizers.

To take this idea a step further, a quiz mode language could be created and integrated into the application. Using this domain specific language (DSL) for creating quiz modes organizers would be able to build their own quiz modes within the web interface.

10.1.3. Various types of questions

At the moment the application supports only one type of question: multiple-choice. There are a number of additional question types which could be implemented. Some examples are 'estimation questions' which allows a question to have a range of answers which are correct, true-or-false questions and open questions.

10.1.4. Send emails to the participant after a quiz

After a quiz has ended a mail could be sent to the participant with their results. This could be done automatically after a quiz has ended. It could also be an option for the organizer to choose a moment at when to send the results.

10.1.5. Using other databases management systems (DBMS)

Quizzard uses SQLite to store the data locally on the organizer's machine. Other database management systems could be used that could possibly speed up the time needed to read and write to the database.

10.1.6. Hosted solution

At the start of this project, we specifically opted to run the application off of the organizer's machine. This means an organizer needs to start the application each time he wants to take a quiz. Alternatively, the application could be hosted on a central server which is available at all times. To achieve this a large part of the application will have to be rewritten keeping this requirement in mind.

During such a rewrite the authentication model should be reevaluated carefully. The current model only supports a single organizer. Every user which want to run the application runs it on their own machine, which means support for multiple organizers was not an absolute necessity. If the application would be running on a server, however, this authentication model does not hold up anymore. Thus multiple organizers with their own audiences, quizzes, and quiz runs have to be supported.

10.2. Maintainability

The application will be used in the Software Systems module. Additionally, the client can also use the application in other modules or for other purposes. There are no explicit plans to provide services to maintain and support the application after it has been delivered to the client, but if such services are desired these plans are negotiable.

10.3. Usage

The application is planned to be used in the next quarter of Computer Science during the module 'Programming paradigms' 2017 as well as in the module 'Software systems' next study year.

11. Evaluation & reflection

11.1. Roles and task division

Every team member has been SCRUM master at least once. Each team member worked on various areas in the application. Below follows a crude task division in which the most important tasks of each member are noted.

Ömer

- front end (static)
 - mock-ups to HTML
- back end
 - controllers
 - controller tests
 - other tests

Melcher

- front end (static)
 - mock-ups to HTML
- back end
 - controllers
 - CSV exporting/importing
 - controller tests

Lukas

- back end
 - controllers
 - websockets
 - quiz modes
 - security
 - initial schema

Remco

- front end (React)
 - quiz editor
 - quiz presenter
 - participant view
 - statistics views

During the project features being worked on were tracked using YouTrack. Using YouTrack every team member could see what issues were part of the sprint and needed work or were already being worked on.

11.2. Improvements

Every person in this team had already worked with the other members of the team on projects during other modules of the computer science bachelor. Additionally, we had experience working as a team on a large University financed project. As such, we already knew each other very well and were comfortable working with each other. No problems worth mentioning occurred during the project.

One improvement which could be made in a future project is the better distribution of some of the tasks. For example one of the team members took on the job of the front end which then became extremely knowledgeable on that subject. Other team members were not as informed about the front end. If that member were to become sick during the project, it would take a considerable amount of time for another member to familiarize with the front end. With addition to losing time due to the sickness of a team member, more time is wasted than needed.