University of Groningen
Software Engineering

QUESTIONNAIRE MANAGEMENT INTERFACE

Architectural Document

Mahir Hiro (s3561119)
Hleb Shmak (s3774244)
Robert Rey (s3309770)
Pal Poshyachinda (s2834839)
Krishan Jokhan (s3790746)

Contents

1 Introduction **1**

2 Client **2**
2.1 Architectural overview . . . 2
2.2 Backend . . . 3
2.3 Frontend . . . 4
2.3.1 Login Functionality . . . 4
2.3.2 Editing Functionality . . . 4
2.4 Databases . . . 5
2.5 Technology Stack . . . 5
2.6 Design Patterns and Programming Paradigms . . . 5
2.7 Frontend Architectural Decisions . . . 5
2.7.1 Toolbar . . . 5
2.7.2 QuestionsPage (current name, new name TBD) . . . 5
2.7.3 Edit Dialog . . . 6
2.7.3.1 Editor state management . . . 6
2.7.3.2 Question property definition . . . 6
2.7.3.3 Question property organisation . . . 6
2.7.3.4 Changing a question type . . . 7
2.7.4 State management: React Context API & useReducer hook . . . 8
2.7.5 UI Framework : Material-UI . . . 8
2.7.6 Drag-and-drop functionality : react-beautiful-dnd . . . 8

3 Security **9**

4 Graphical User Interface **10**
4.1 Layout of the main page . . . 10
4.1.1 AppBar . . . 10
4.1.2 Toolbar . . . 10
4.1.3 JSON section . . . 11
4.1.4 Question Area . . . 11
4.1.5 Edit Dialog . . . 11

5 Team Organization **13**
5.1 Scrum Events and Task Distribution . . . 13
5.1.1 Sprint 0: Meeting the owner, getting familiar with the framework . . . 13
5.1.2 Sprint 1: Authentication . . . 13
5.1.3 Sprint 2: Questionnaire Editor . . . 14
5.2 Trello . . . 14
5.2.1 General Usage . . . 14
5.2.2 Additions . . . 15
5.3 Github . . . 15

6 Build Process and Deployment **16**
6.1 Continuous Integration 16

7 RESTful API **17**

8 Changelog **18**

1. Introduction

Our goal for our project is to build a front-end for our questionnaire engine, that enables users to define their questionnaires using drag-and-drop interface. We do not have to deal with the back-end because the company we are working with has already implemented the back-end system. The main users of this application are professors and researchers, who need questionnaires for their research. Currently, to make a questionnaire the users of the system have to edit `JSON` files which is not user friendly at all. Here is an example of how a question is configured:

```
[{"id":"v1","type":"radio","show_otherwise":false,"title":"Voorbeeld van een..."}]
```

Clearly, it can be seen that the more questions are added to a questionnaire, the harder it will be to maintain the questionnaire. This old system of creating questionnaires needed a change in order to make it more user friendly for the questionnaire creators. It should be possible to make questionnaires by filling in text fields and options (like *Google Forms*) instead of editing bare `JSON` code. Thus we were told to create this exact system mentioned above.
Below are a few goals mentioned by the client with features he would greatly appreciate to be completed by the end of this project.

1. It would be amazing to have a drag and drop user interface.
2. It should allow users to authenticate.
3. It should allow for communication with the backend to store questionnaires.

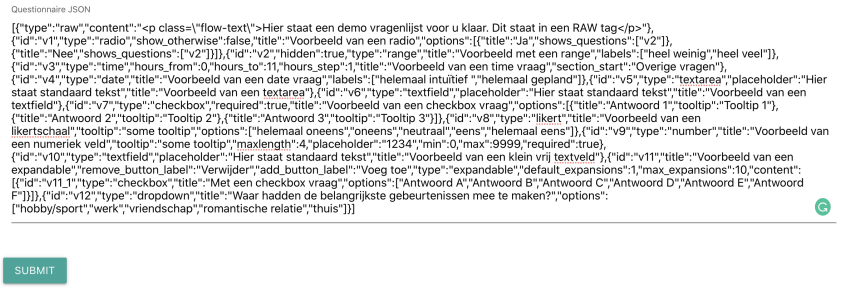
2. Client

2.1 Architectural overview

The `u-can-act questionnaire engine` consists of a `backend` which has already been written by the client and his team. The backend is the main questionnaire engine: it defines the syntax used for the questionnaire, manages recurring surveys and makes it possible to run a questionnaire and save user answers. It also stores users that make and manage questionnaires (e.g. the professors who are doing research).

As noted, the layout for a questionnaire is already defined, and a debug page is available to show how a `JSON` is converted to an organised questionnaire.
An example of the following is demonstrated below:

The `JSON` code as mentioned above is placed in this box which fully renders out the questions

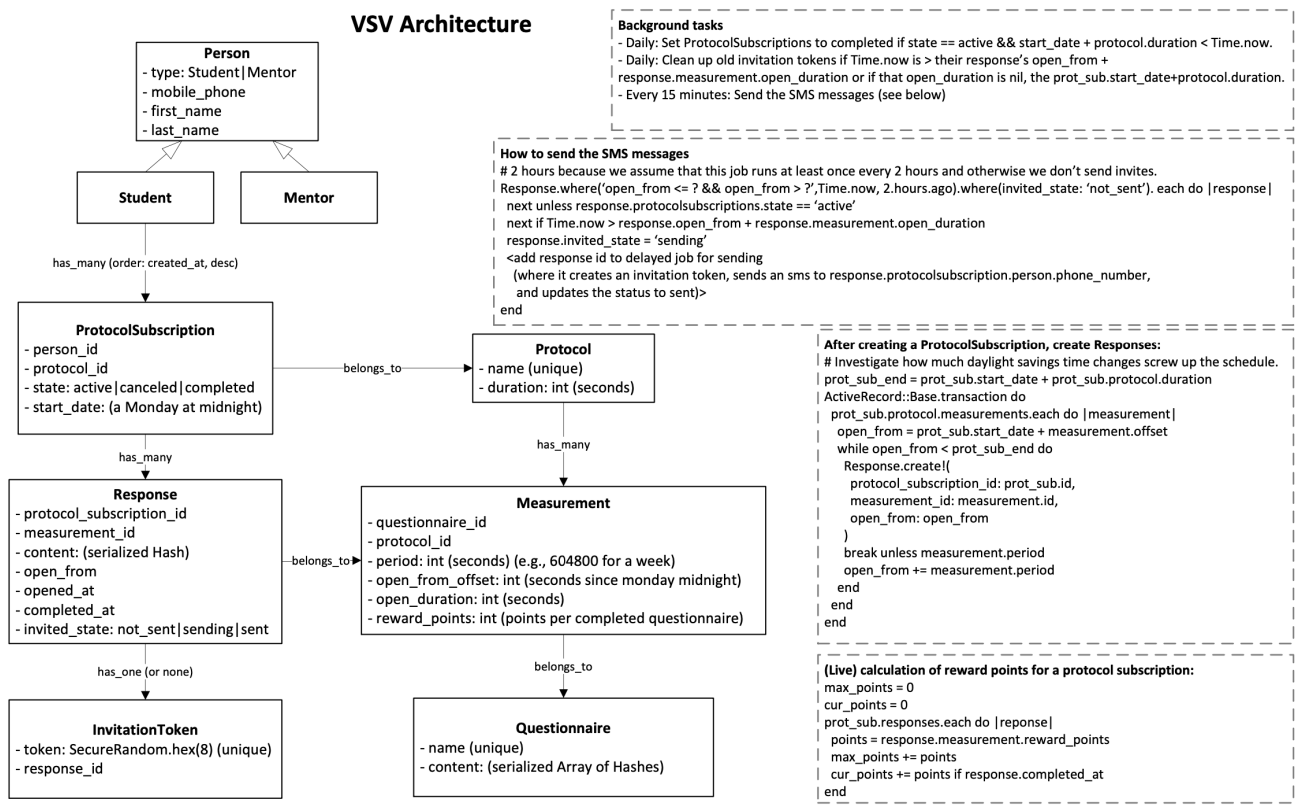


After the `JSON` is rendered in this example we can see how the questionnaire was intended to look for the users to be filled in

2.2 Backend

Getting back to the `backend` section. More details about the implementation about the code which is in `Ruby on Rails` can be found on the `GitHub` repository: [`https://github.com/compsy/u-can-act`].

With the diagram shown below we can see an overview of what was implemented before our team started working on the project.



2.3 Frontend

Frontend is the main focus of our application, as the goal of this project is to create an application, that will enable user-friendly interaction between the user and the application. There are two main activities of the application: login and editing.

2.3.1 Login Functionality

The product owner of the project [Frank Blaauw] mentioned to work on login functionality in the second block of this course. Therefore, this document will be updated accordingly with the changes in the project.

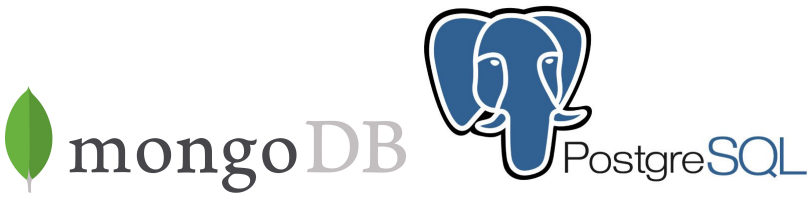
2.3.2 Editing Functionality

Editing functionality of the application will enable the user to create, delete and edit questionnaires. There are several components of the editing page: `Toolbar`,`QuestionsPage`, `QuestionPreview` and an `Edit Dialog` The editing functionality can be described as follows:

- Adding new question to the questionnaire is done by dragging the question from `Toolbar` to the `QuestionsPage`, which renders a `QuestionPreview` of the desired question type.
- Reordering the questions in the questionnaire is done by dragging the question up or down in the `QuestionsPage`, depending on where it should be in the list of questions.
- Deleting the question is done using dialogs appearing after clicking on the edit button of the question in the `QuestionPreview`.
- Editing the question (adding/deleting new options, inserting questions, changing question type) is done using the `Edit Dialog` appearing after clicking on the edit button of the question.

2.4 Databases

The backend has two databases, which consists of one each from `MongoDB` and `PostgreSQL` databases. The `MongoDB` is used for storing questionnaires, whereas the `PostgreSQL` database is used for storing personal details such as bank details and other private information. Frank told us this design choice was because of `GDPR` reasons in case one of the databases was leaked the link would not be found between the two.



2.5 Technology Stack

The technology stack for this web application consists mainly of React which is a `Javascript` frontend framework made by Facebook. We also used some additionally libraries which made the process of programming easier. The libraries used were `react-beautiful-dnd` and `Material-UI`. The backend was written in `Ruby on Rails` and the databases are in `MongoDB` and `PostgresSQL`.

2.6 Design Patterns and Programming Paradigms

We used the latest version of `React JS`, therefore hooks [`https://reactjs.org/docs/hooks-intro.html`] were used throughout the project. Hooks enable a functional programming style of coding. Moreover, it was decided to use more functional paradigm (of course, we used the object-oriented paradigm as well). It was done due to several reasons, that are described in "The functional side of React" [`https://medium.com/@andrea.chiarelli/the-functional-side-of-react-229bdb26d9a6`]. In short, it allows for cleaner code as less code is needed to achieve the features of React (in comparison to the usage of classes).

The most used pattern is composite pattern. The intention of a composite is to "compose" objects into tree structures to represent part-whole hierarchies. Together with higher-order components and higher-order functions, it makes the code more readable and neat.

2.7 Frontend Architectural Decisions

2.7.1 Toolbar

The `Toolbar` is the right-sided list of the different types of the questions with the corresponding icons, that can be dragged to the `QuestionsPage` area. Moreover, the draggable property is added to the question labels. The `Toolbar` has several question types: radio, checkbox, range, likert, textarea, number, draw, and dropdown. More information about the questions can be found via the link `https://github.com/compsy/u-can-act#questionnaire-syntax`. Questions are mapped to the questions section. We used `react-beautiful-dnd` library [`https://github.com/atlassian/react-beautiful-dnd`] to implement drag and drop functionalities of the questions. The visual design is supposed to be minimalistic.

2.7.2 QuestionsPage (current name, new name TBD)

`QuestionsPage` is a "bag" of the questions, that represent the questionnaire. `QuestionsPage` implements drag and drop functionalities using the **Draggable** and **Droppable** functionality of the `react-beautiful-dnd` library. Moreover, `QuestionsPage` supports reordering of the questions via drag-and-drop. The dragging of the questions is done using handles.

2.7.3 Edit Dialog

To edit the question's content the `Edit Dialog` are used. It possess different properties depending on the question type. The `Edit Dialog` are easy to use, with each property (of the question type) always visible, therefore it guaranties user-friendly experience of the user. All edit dialogs of all question types are managed via the single `EditDialog` file, that renders the right properties and values and questions via maps and question data.

2.7.3.1 Editor state management

The edit dialog creates a copy of the selected question as `newQuestion`, and uses the `NewQuestionContext` to make sure every property has the same data available. The usage of a context was decided to solve the problem of dependencies of properties: some properties rely on others. An example of this is the `show_otherwise` property that, other than showing a 'otherwise: ...' option in types like `checkbox` and `radio`, also influences whether properties `otherwise_label` and `otherwise_tooltip` should be shown.

2.7.3.2 Question property definition

Some question types share properties with others. That is why hardcoding properties in a dedicated edit dialog per type seemed to cause messy and repeating code. To solve this, a method was thought of to make it possible for question properties to be connected to question types as components. Managing layout and updating the right values for the backend is managed *within* the property component. This makes it possible to re-use code between various question types, and disconnects the task of verifying and updating properties from the edit dialog to the property component.
To ensure that even in the property components, as little code as needed was copied, the properties were categorised in various overarching properties:

- `TextProperty` : For simple, text-based properties.
- `BooleanProperty` : For boolean properties, which are rendered as a switch that can be turned 'off' or 'on'.
- `RegexpProperty` : Like `TextProperty` this consists of a field where the user types the desired value. Unlike `TextProperty` a validation function is called before input is accepted. An extra property that should be given when using this property is a regular expression, that will be used to validate the input
- `NumericProperty` : Input that only accepts numbers
- `TextArrayProperty` : Used to render and edit arrays of strings.

These actual properties use an instance of the overarching property they belong to. They give, apart from the `newQuestion` and `newQuestionDispatch` reference that is needed to update the question, the property name as decided on the backend, and a user-friendly name. Other props that affect the overarching property (think of layout-related props) can be given too.

One exceptional property that does *not* use above parent properties is `PrioritizedTextOptionsProperty` as this is a special property that is unique to any other. It is used to visualise the `options` property for `likert` and `dropdown`.

2.7.3.3 Question property organisation

To decide what properties are available for each question type, a map is used. Each element of the map (which is identified by the question type) contains an array of component references (not renders) of all question types available. This was decided for several reasons:

- The code for the map also acts as a clear overview of what properties are allowed per question types
- Adding a new property to a question type is as easy as typing the property name in the array definition

- Rendering all properties can by done by simply looping over the array after it has been located via the question type.

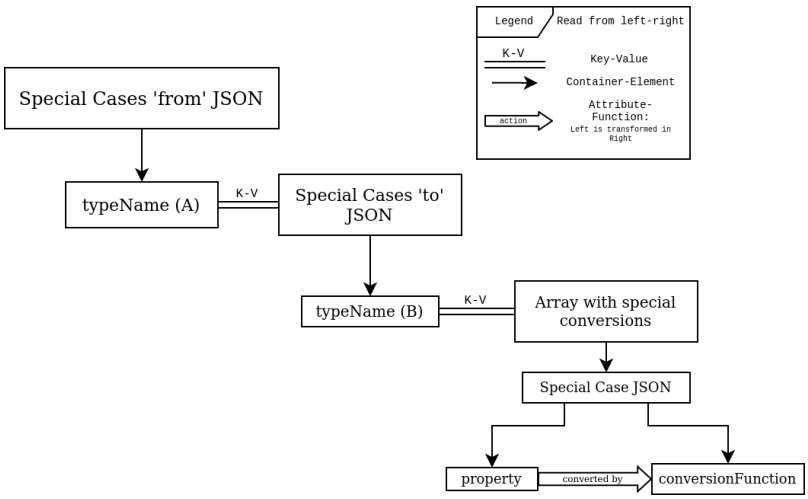
There are some downsides to this implementation. One major, and noticeable disadvantage is the lack of visual organisation of the properties. Currently, the application does now know how a property should be rendered, other than it should be rendered after a previous property with references to the context and dispatcher. A solution for this is planned in future sprints. It could be possible to create *layouts* for each question type, with how their properties should be rendered. A future revision on this document will update on what has been done to solve this problem.

2.7.3.4 Changing a question type

One of the requests of the client was to make it possible for some question types to change into others. Some shifts are as easy as changing the value of `type` in the question's JSON file. However, some contain data that are not supported (completely) by the other type. An example is the way options are stored in e.g. `combobox` and `likert`. A system was created to solve this problem. If there is a special case from type `A` to type `B`, it is stored as follows.

- A special map containing all types that have special conversion cases contains type `A`
- The value of `A` in the map described above contains a JSON, where each key is the name of the type `A` would be converted to. In our example, this is `B`.
- The value of `B` in the map described above is an array containing the actual conversion cases.
- Each conversion case has two properties: the name of the property that should be converted and the function that should be used to convert the property.

In a diagram, the map would look as follows:



'A' and 'B' refer to the types used in the text example above.

The code is made in such a way that a new case added here will automatically be supported in any other code that uses the system. When a type conversion is called, the function dedicated to the action checked whether relationship currentType-nextType (A-B). Since JSON is used, there is no search loop needed to check this relation, which is a great efficiency advantage. When the relationship exists, for each property that should be converted, its conversion function is called and the result is stored as new value for the property. This allows for clean and easy implementation of special cases and even support for new question types that could be created in the future: The only place where special cases should be registered is in the JSON, the rest of the code remains unchanged.

2.7.4 State management: React Context API & useReducer hook

To share state between components without the need to pass props, the Context API provided by React was used. Three main contexts exist in our implementation, `QuestionnaireContext` (which stores the array of questions), `NewQuestionContext` (which stores the current question being edited), and `SettingsContext` (which store settings that may need to be provided to multiple components). To edit the state of each of these contexts, the `useReducer` hook was used (a reducer was made for each of the contexts). The combination of the Context API and the `useReducer` hook allows us to easily add functionality to our application, while keeping everything organized. This approach to state management was influenced by the Redux library, and bears many similarities in code.

This approach was chosen over Redux mainly because of time constraint. In our team, none of us had prior experience with React, so learning the React framework was already going to be time consuming. Adding another big library/framework to the learning list most definitely would have slowed our progress.

2.7.5 UI Framework : Material-UI

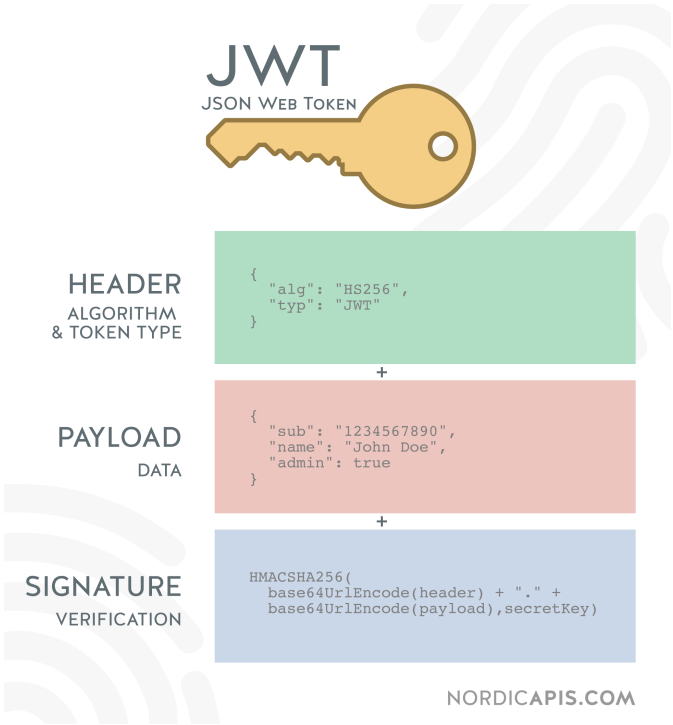
The decision to use Material-UI as the UI framework was a no-brainer. As a React framework, it makes it easy to implement user-friendly interfaces in React. Another big factor that made our team choose Material-UI was the amount of documentation (a clear victory over the others). Used by at least 189,000 projects on Github, it is easily the most used UI framework for React. Other notable frameworks considered include: MaterializeCSS (that is used by the backend to stylize questonnaires) and React Bootstrap.

2.7.6 Drag-and-drop functionality : react-beautiful-dnd

In our first version of the application, Sortable-JS was used to provide Drag-and-Drop functionality. Although very smooth and reliable for reordering questions, its lack of versatility made us switch over to react-beautiful-dnd for our second version. This library makes it very easy to not only reorder existing questions, but also clone question templates (dragging from the toolbar to the QuestionsPage).

3. Security

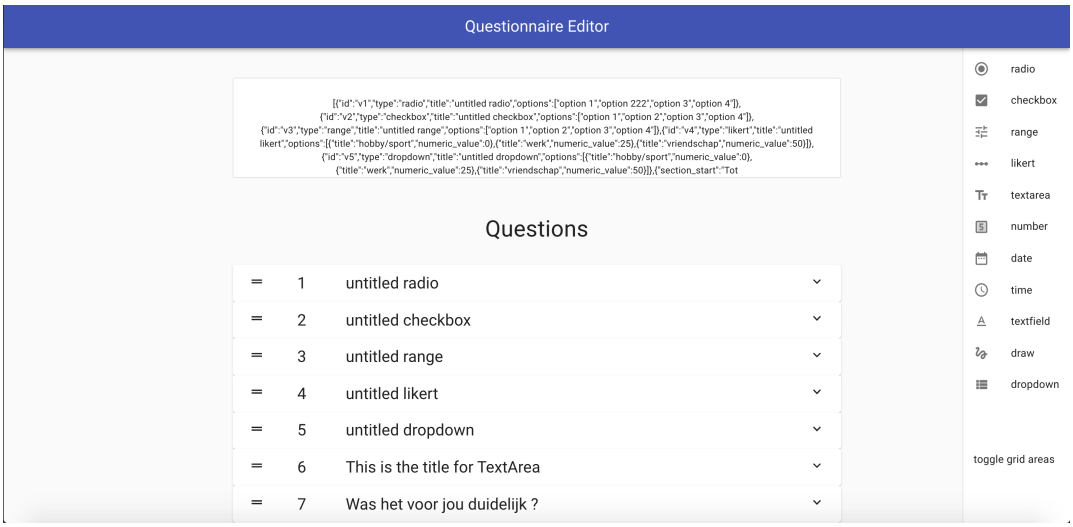
Currently the product owner [Frank Blaauw] did not require any security for this project, as he decided it was the best to focus on the authentication part of the project in the second block of `Software Engineering WBCS17001`. Frank suggested to use JSON Web Token for the authorization. In the non-functional requirements document we also mentioned a little bit about the security in our project. Furthermore, this section will be more detailed over the next few months when we cover this security section in depth. Here is a nicely detail layout of the JWT.



4. Graphical User Interface

4.1 Layout of the main page

When launching the Web application, we are immediately sent to the main section of the application as the authentication part is still left to be done in the upcoming weeks. This main section consists of a few key components which will be talked about in this section. Let us first show how the page looks before we dive into a more detailed explanation of the components.



4.1.1 AppBar

At the top of the page, we can see an AppBar which serves as a header for the components. For the time being, we have decided to go with the title "Questionnaire Editor" which most accurately reflects the main purpose of this project. Secondly, we believe that the color we have chosen, which leans towards a dark blue is the most aesthetically pleasing and fits in nicely with the other components. This is as far as it goes for the AppBar.

4.1.2 Toolbar

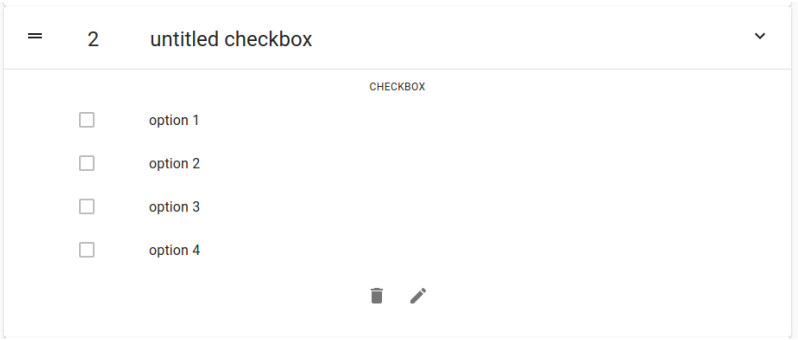
Next we have our toolbar which can be seen on the right of the page. Our toolbar contains a list of text and their respective icon for each question type. This Toolbar is also equipped with a drag and drop functionality which enables the user to select which question type he or she want to use and drop it in the question area. Moreover, we have given our draggable component a blinking color while it is being dragged, which enables the user to realise that a component is indeed being dragged. This was definitely one of the harder functionalities to implement. In order to implement all these functionalities, we have had to import libraries such as **DragDropContext**, **Draggable** and **Droppable** all from "react-beautiful-dnd". Finally we have a "toggle grid areas" component on the toolbar, which portrays all the different sections if you will, of the QuestionPage component. This feature is mainly for us (the programmers) to more easily identify components within the code in the case that we need to modify functionalities or edit the layout.

4.1.3 JSON section

This section can be found right below the AppBar in the middle of the page. It contains all the different formats (types) of questions we should implement. This section doesn't have any particular use for the user of this application, but it's been put there to help us (the programmers) in visualising how each type of question should look like. We essentially run this JSON text into a parser, which was provided to us by the client, and from there we have a much better idea of how our client wants each type of question type to look like. From there, we have built each type accordingly by trying to match as close as possible the layout of our client. The reference to the parser can be found in the *Architectural overview* subsection within the *Client* section of this document.

4.1.4 Question Area

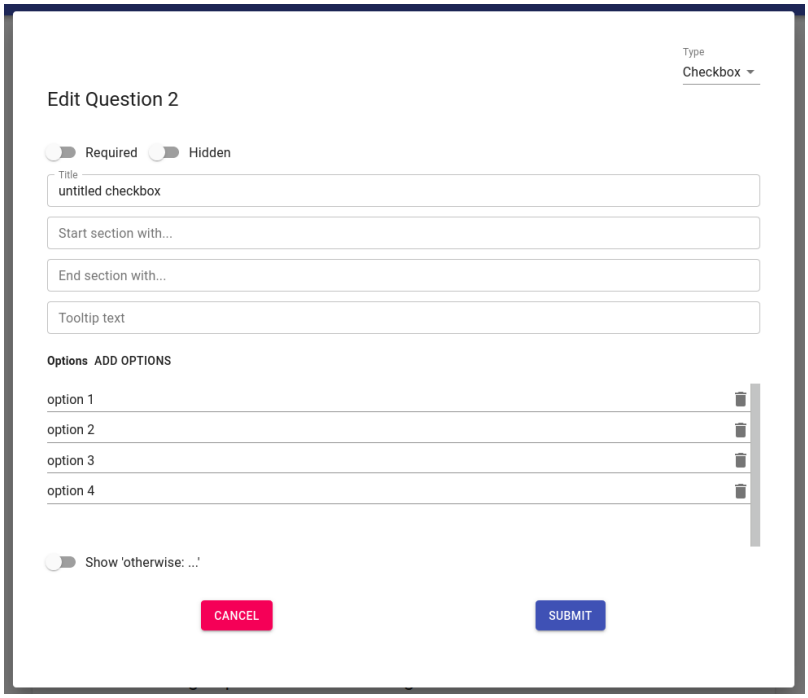
This is the main part of our application. In here we have a collection of all the question types that we have added. This section already has questions inside when the application is initially launched. This was purely made for testing our product, where we could see how each question type would look like. Each component of this section is essentially a question type and their corresponding attributes. Although each question type has its own properties, each type also share common fields and features, such as a handle, an identification number which depends on the position of the question and a title. Moreover, clicking on a question brings down all its corresponding information and attributes which are linked to its type, as well as an edit and delete button. Each time we add a question to the question area, it expands in size to accommodate for more available questions.



Clicking a question shows more detailed information about the question.

4.1.5 Edit Dialog

As mentioned previously, each question comes along with an edit button, which can be seen once it's information is expanded by being clicked on. The edit dialog contains all available properties for a question type, that can be edited. Currently, there is no distinction between required properties and optional properties, also changing a property of a question type isn't always well received. Of course these are all goals we are planning to fix in the following block. Please see an example below of what we see when we click on the edit button for the checkbox type question.



As said previously, the only fields which will be affected by this edit is the *Title* field and the *Options* field as our edit page is only a prototype and still has to adjusted for each specific type. That however does not mean the changes are not recorded correctly. All properties with their values are stored in the question's JSON, and the debug parser provided by the client can be used to show the questions properly instead for now.

5. Team Organization

This section summarizes how the team was structured and assigned to various tasks. We discuss how we deployed the (course mandatory) scrum framework and decisions regarding communication and project organisation.

5.1 Scrum Events and Task Distribution

Each of our sprints had a duration of two weeks. The sprints were each started in a meeting that followed after a meeting with our TA or client, so we would know what kind of progress was expected and what the client had in mind. Occasional meetings were done to get a view of how everyone is progressing, apart from even more regular online conversations to update each other. All meetings were at first done physically on Zernike Campus, but were forced to be organised via an online alternative after the campus closed down due to the Corona virus countermeasures.
This following subsections will summarise task distributions which used elements of scrum. Using scrum greatly enhanced efficiency in this. Thanks to the regular meetings, everyone could stay productive through extra assistance and new tasks. In total, we have had three sprints which will be summarized below.

5.1.1 Sprint 0: Meeting the owner, getting familiar with the framework

This was seen as a sprint from the 'course's point of view', where the student group met with the client and started with the requirements document. In this period, everyone started exploring the requested ReactJS framework and getting an overview on how the backend is structured.

5.1.2 Sprint 1: Authentication

The scrummaster for this sprint was Pal Poshyachinda.

The final meeting for sprint 0 directly marked the start of the first actual sprint. After discussing how everyone feels with the existing technology and frameworks, we came back on the user stories made while meeting with the client. Every available user story was given a difficulty, using `Scrumpoker Online`, an online method of giving user stories a score (where each score is a fibonacci number), until everyone compromises on one score. It was decided to focus on authentication this sprint, as it seemed the first step for a user to use the final program.
The sprint contained two parts: creating an account and the ability to log into an account. Since these tasks seemed small enough to be initially done individually, each group member was given a separate git branch to work on. The plan was to try out the framework, as for all of us, this was our first project in ReactJS, and come with a prototype.
A meeting was then made to show everyone's progress. As the way ReactJS works was completely different to all of us, Robert, Mahir and Hleb were not able to finish a prototype. Krishan focused on creating an account and made a prototype containing some of the possible elements for this user story. Ou's prototype was most complete and it contained a style similar to the existing questionnaire layout and it was decided to continue on this in the future.

The sprint ended with a meeting with the client via Google Meet. As a group, we were physically together. The prototype was not yet connected with the backend but was sufficient for the sprint. The meeting concluded to focus mainly on editing a questionnaire for the next sprint.

5.1.3 Sprint 2: Questionnaire Editor

The scrummaster for this sprint was Robert Rey.

After the demo of sprint 1, the meeting continued with deciding how the next sprint should be. The main focus would lay with editing a questionnaire, and suggestions by the client and the group were given to decide how it would look like. It was decided to have a sidebar containing all question types (in the rest of the document, this is called the `toolbar`), where a question type would be dragged to the questionnaire preview. More detailed and graphical information on this can be found in *Section 2.3: Frontend*, *Section 2.7: Frontend Architectural Decisions* and *Section 3: Graphical User Interface*. The meeting concluded with a task distribution for this sprint.
The initial distribution is summarised below.

- Toolbox with different question types: Mahir, Robert and Hleb
- Dragging a question type to the editor: Mahir Robert and Hleb
- Rendering a preview of a question type: Pal and Krishan
- Edit dialog for a question type: Pal
- Refactoring existing code: Pal and Krishan

Reaching the end of the sprint, some tasks were finished earlier than expected. In an update meeting, some tasks were assigned more people and new tasks were made:

- Dragging a question type to the editor: Bug fixes by Pal
- Rendering a preview of a question type: Ou, Krishan, Mahir Robert and Hleb (due to the amount of question types available)
- Edit dialog for a question type: Pal and Krishan (including all available question types and bug fixing)
- Refactoring existing code: Pal and Krishan

We kept each other up to date via Slack and WhatsApp. Within the groups, members worked together on bugs, architectural ideas and finishing the task.

The sprint ended with the presentation for the course, and before that, several smaller meetings were done to check if there was anything that needed more help. Thanks to redistributing ourselves over the tasks, with new knowledge of difficulty *for* each task, the sprint could be finished a few days before the deadline. The final days consisted of bug fixing and creating the presentation slides (done by Mahir and checked by everyone).

5.2 Trello

Trello is used to get a clear overview of what needs to be done. For the majority of the project, one Trello board was used, created by our TA (only 'Authentication' had its own board, but this idea was discarded later because it was better to have one central place for all tasks). In the beginning, for the majority of the team, it was hard to get used to Trello and updating the board. Even though this is still not optimal, the group eventually began to use Trello more appropriately.

5.2.1 General Usage

After the intial meeting of sprint 2, Krishan made sure that the task distribution would be available as a summary on Slack and each task, assigned, on Trello. However, even though the tasks were available, Trello was not used to organise and update sprint progress. Instead of Slack, initially, WhatsApp was used. Because lack of usage of Trello, it was not clear for everyone who was working on what part, within the subgroups. This lead to the problem that Pal and Krishan were both refactoring the same code, unknowingly from each other. This meeting marked the decision that everyone should take Trello more seriously and up to date, in order for

these mistakes to not happen again. Together with that, it was decided to update more regular on smaller things like commits on GitHub (first via WhatsApp but eventually done on Slack after a comment from the TA), so everyone could see the flow of the sprint in a more clear and detailed way.

However, usage of Trello was not done as supposed to by most members, even after deciding to do so. It was still hard for most members to update Trello, so even here the newer tasks and bugs in sprint 2 were added on the Trello board by Krishan. Having one responsible for updating Trello after a meeting can be a good thing, but Trello remained barely updated after. In general, the group became more professional regarding updating each other and organising tasks compared to the beginning and we hope that eventually, updating Trello would fully become a part of our scrum routine.

The Trello board was initially organised by our TA, with lists based on the scrum framework: *To Do*, *Doing* and *Done* with each a respective label to add to each task.

5.2.2 Additions

Making progress into the project, various additional lists and labels were made and used:

- Questions for Frank: A list where questions to our client [Frank Blaauw] could be added to be asked later.
- Ideas: A list for concepts that are not (yet) mandatory, but could become handy in the future.
- Current Sprint: a label to view what tasks are planned for the current sprint
- Optional: A label for tasks that are not necessary for the current sprint, but have a close relation to it, something that tasks in the `Items` list do not necessarily have.
- Fixed: A label for tasks that do not necessarily 'finish' A clear example is 'refactoring'. Future code might need to be refactored, and it is nice to have a fixed task in a list for this as a reminder.

5.3 Github

Git helped us track the changes we make to files, so we have a record of what has been done, and we can revert to specific versions should we ever need to. We made several branches for each of our own progress for sprint 1 which later were merged in the development branch. At the end of each sprint our development branch was merged with our master branch.

Merge issues were not really a problem. Thanks to everyone working on different parts, mostly, getting new data from a branch would not cause any. And if there were any merge issues presents, these were mostly small and could be overwritten by the local branch. Thanks to the usage of WebStorm, issues could first be clearly reviewed before things were actually overwritten. Because there were mainly only two two people assigned to refactoring code, apart from one issue in the beginning where there was a lack of communication, merge issues stayed small as these members stayed in contact when deciding who would refactor what part of the code.

6. Build Process and Deployment

6.1 Continuous Integration

Continuous integration is the practice of automating the integration of code changes from multiple contributors into a single software project. The platform we used is `Circle CI` for our pipeline choice. We chose to do this our selves because it can detect errors quickly and locate them more easily. Continuous Integration doesn't get rid of bugs, but it does make them dramatically easier to find and remove. - `Martin Fowler`

7. RESTful API

The full documentation of our RESTful api can be found at `http://localhost:8000/` when the server is running locally (code to run a local server can be found at `https://github.com/compsy/u-can-act`). This section will be developed in further detail when we start the authentication section next block where out web application will communication with our client's backend code, and where we will connect the existing code with real data retrieved from the backend.

8. Changelog

Who	When	Which section	What
R. Rey	April 4, 2020	The document	Created the document.
H. Shmak	April 4, 2020	Introduction	Introduced the application.
R. Rey	April 4, 2020	Graphical User Interface	Attached the screenshots and described the interface.
K. Jokhan	April 4, 2020	Team Organisation	Introduced the section and worked on subsection 'Trello': Additions Worked on 'Scrum Events and Task Distribution' Worked on 'Trello': General Usage Worked on Github (merge issues)
M. Hiro & H. Shmak	April 4, 2020	Client	Introduced and elaborated on Client section.
M. Hiro	April 4, 2020	Build Process and Deployment	Introduced and elaborated on Continuous Integration section.
P. Poshyachinda	April 4, 2020	Frontend Architectural Decisions	Added section on state management Added section on UI-framework Added section on DnD functionality
K. Jokhan	April 4, 2020	Frontend Architectural Decisions	Added section 'Dialogs'
M. Hiro	April 4, 2020	Team Organisation	Introduced Github section
H. Shmak	April 5, 2020	Security	Introduced Security section.
M. Hiro	April 5, 2020	Security	Security section extended.
K. Jokhan	April 5, 2020	Frontend Architectural Decisions	'Dialogs' extended.
K. Jokhan	April 5, 2020	Whole document	Fixed mistakes and inconsistencies.