University of Groningen
Software Engineering

QUESTIONNAIRE MANAGEMENT INTERFACE

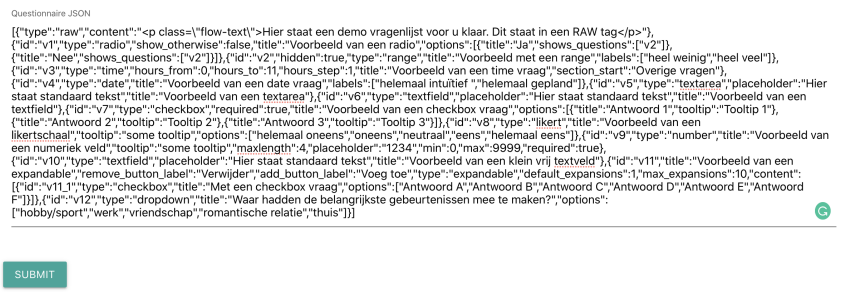
Architectural Document

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

Contents

Introduction

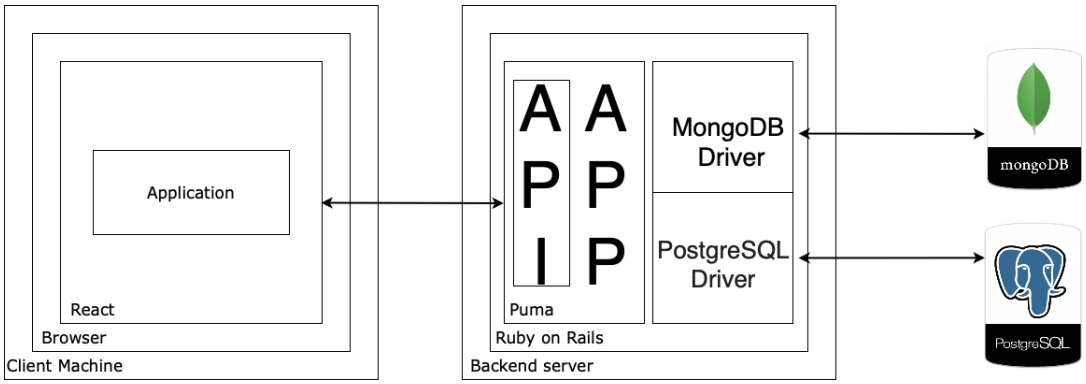
A questionnaire is a research instrument consisting of a series of questions for the purpose of gathering information from respondents. The goal for this was project to build a front-end interface for the questionnaire engine, that enables users to define their questionnaires. Currently, to be able to build such a questionnaire the users of the application have to edit a JSON format text to create a questionnaire. Which then is rendered through a back-end system which is already implemented. A short example of how JSON text for a questionnaire may look depicted below:



This system was not feasible for in the long term since editing JSON text, whether a user of the system is or is not familiar with the basic programming knowledge is not user friendly. Henceforth, the goal of this project is to create a simple yet elegant interface to ease this process when creating questionnaires. Which in turn creates this JSON by through the new interface.
Below is a picture to see how the JSON gets rendered in the system after the JSON is created.

Client

Architectural overview

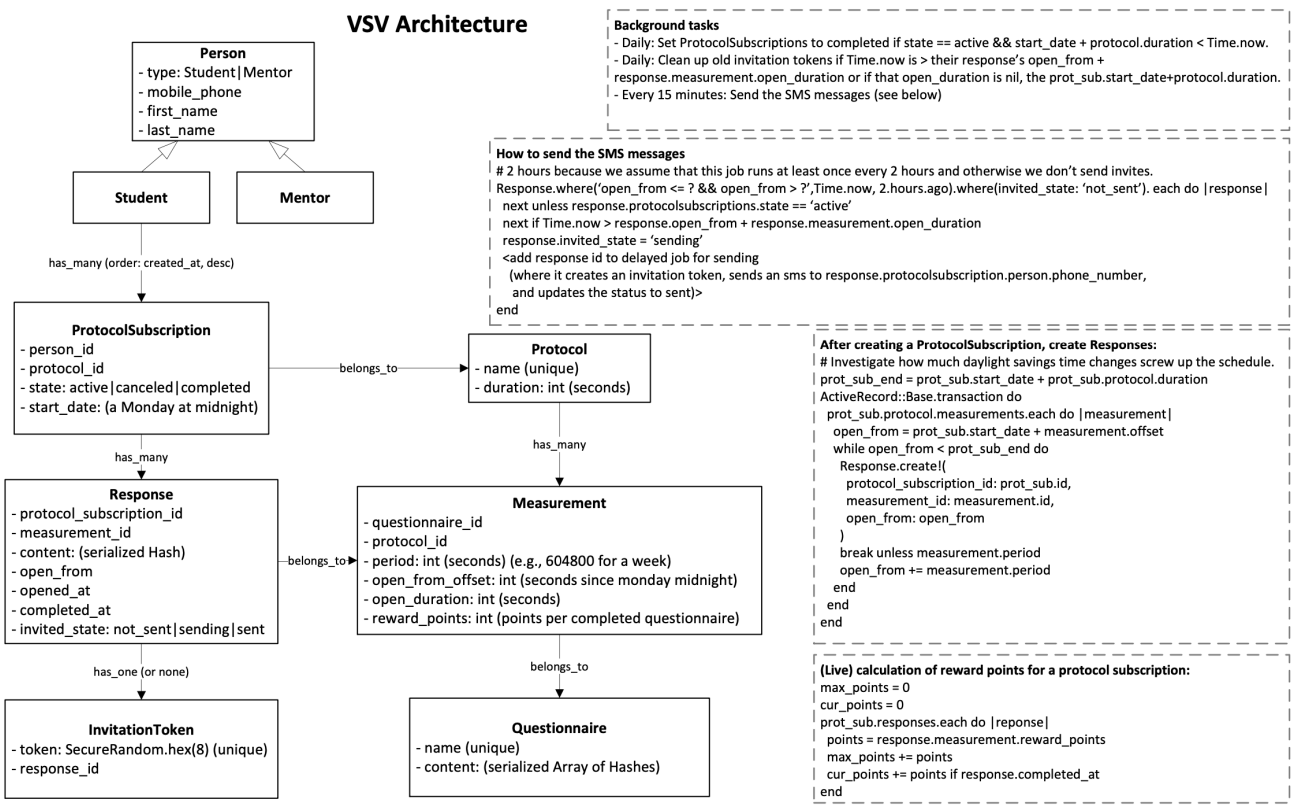


While the project does not suggest to develop the back-end, the overview of the back-end is still provided. This is done due to consistency matters, namely, it is better to grasp the structure of the project and front-end decisions made throughout the project. The document mainly elaborates on the left part of the scheme. The right part of the scheme can be found in correspondent sections: **0.1 Backend** and **0.3 Databases**.

0.1 Backend

For this project as stated in the introduction section the goal of this project was just to build the interface for this project, that being said, this was not an issue since the client already has a backend that 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). More details about the implementation about the code which is in `Ruby on Rails` can be found on the `GitHub` repository of u-can-act.

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



0.2 Frontend

As the frontend is the main focus of the application, the goal was to enable user-friendly interaction between the user and the application. There are two main activities of the application:

1. The ability to authorize oneself.
2. The ability to create a questionnaire that generates a JSON format.

0.2.1 Authorization Functionality

The authorization functionality will enable the user to save questionnaires to the backend, as well as loading existing questionnaires *from* the backend into the editor. These functionalities need admin privileges which are linked per user in the backend. The sole component that is needed to get authorization information is the `Auth0Provider` component.
Authorization functionalities can be listed as follows:

- Logging in and creating a new account.
- Using authorized requests like creating a new questionnaire and retrieving all questionnaires available on the backend.

0.2.2 Editing Functionality

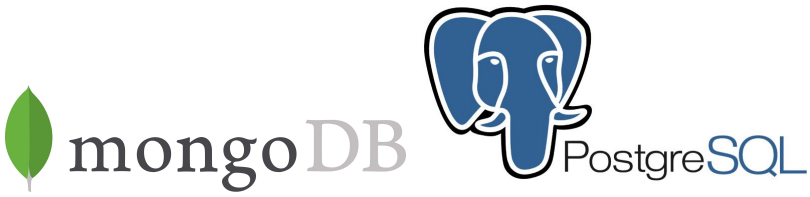
Editing functionality of the application will enable the user to create, delete, and edit questionnaires. There are several components of the editing page: a question types toolbar on the right side of the page, the list of questions located at the center of the page, a question preview for each question available, and an edit dialog for editing a question.

The more detailed explanation of the editing functionality is as follows:

- Adding a new question to the questionnaire is done by dragging the question from the toolbar on the right side to the center of the page (list of questions), which renders a preview of the desired question type.
- Reordering the questions in the questionnaire is done by dragging the question up or down in the list of questions, 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 question preview.
- 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.

0.3 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. The frontend uses local storage and API calls to save and retrieve data.



0.4 Technology Stack

The technology stack for this web application consists mainly of React which is a `Javascript` front-end framework made by Facebook. We also used some additional libraries which made the process of programming easier. The libraries used were `react-beautiful-dnd` for drag and drop functionalities and `Material-UI` for the layout and style. The backend was written in `Ruby on Rails` and the databases are in `MongoDB` and `PostgresSQL`, which a mock could be accessed via `Docker` for development. For authentication, the Auth0 service is used. To make API calls to the backend, `Unirest` is used as it allows for minimal syntax errors: property names of the call are not manually typed, but referenced through functions.

0.5 Design Patterns and Programming Paradigms

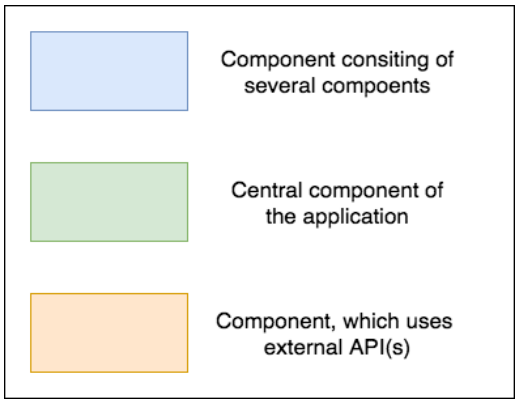
We used the latest version of `React JS`, therefore hooks were used throughout the project. Hooks enable a functional programming style of coding. Moreover, it was decided to use a more functional paradigm (of course, we used the object-oriented paradigm as well). It was done due to several reasons, that are described in "The functional side of React". 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 a composite pattern. A composite intends 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.

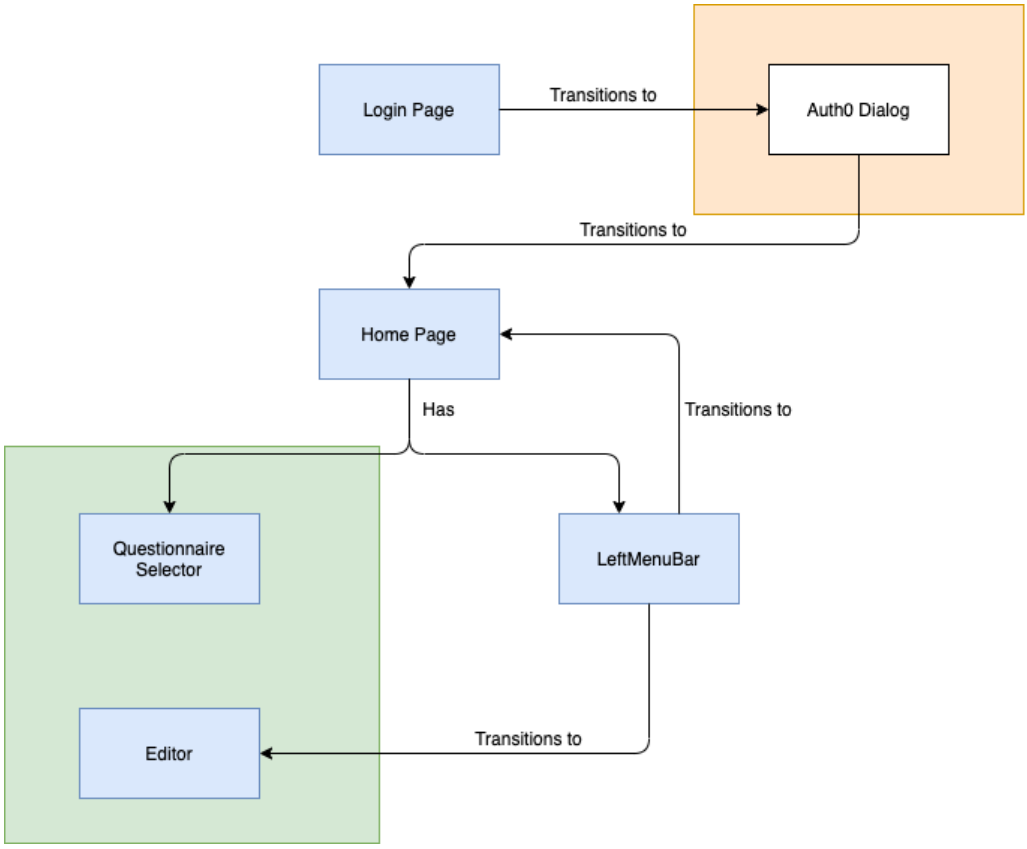
0.6 Frontend Architectural Decisions

0.6.1 General Structure

Throughout the document the following legend is used:



The general class diagram is as follows:

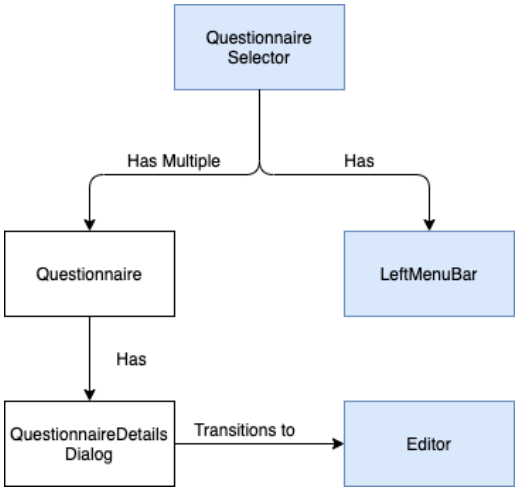


The brief descriptions of the components is given below:

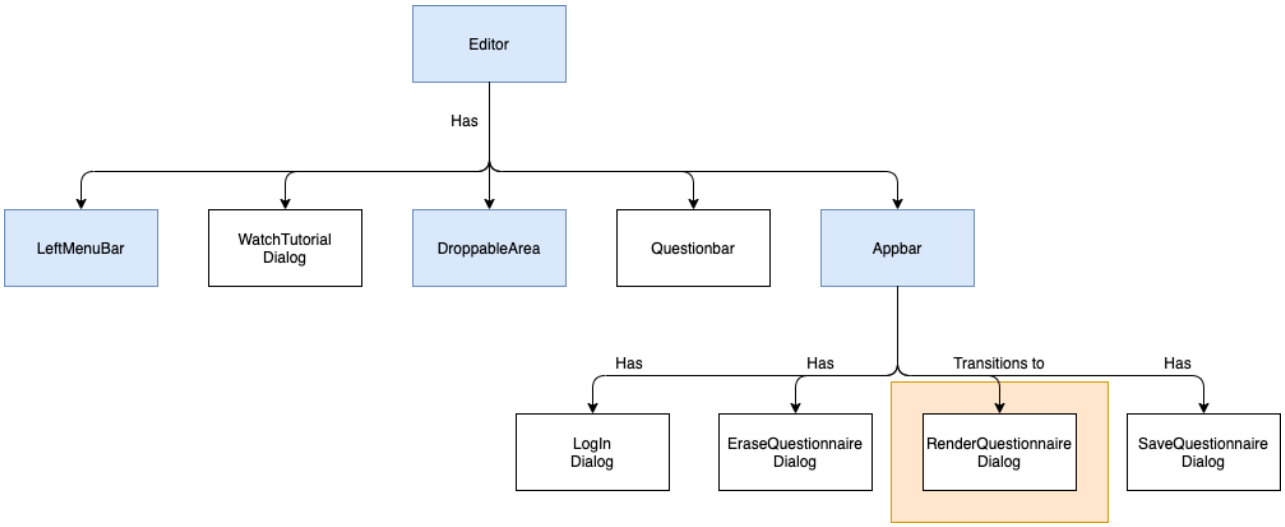
- The authentication of the user is handled via *Auth0 Dialog*, which is an implementation of the API of Auth0.
- The *Home Page* component provides the user with the functionalities of selecting already created questionnaires, modifying them, or creating a new questionnaire.
- *Questionnaire Selector* provides the user to select appropriate questionnaire in order to perform the editing of it.

- *Editor* is the core component, which provides the editing functionality.
- The *Left Menu Bar* is used for transitioning between component, moreover it enables user to select the theme or log out.

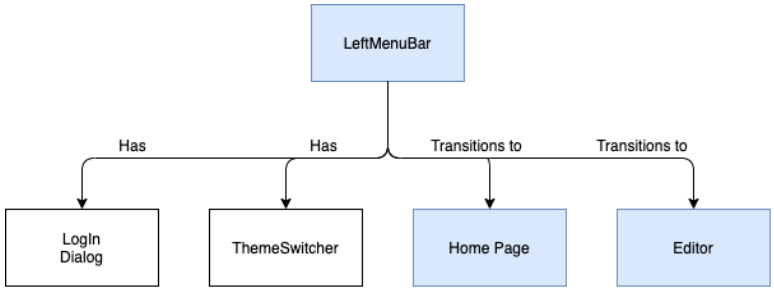
Moreover, the extended diagrams are as follows:



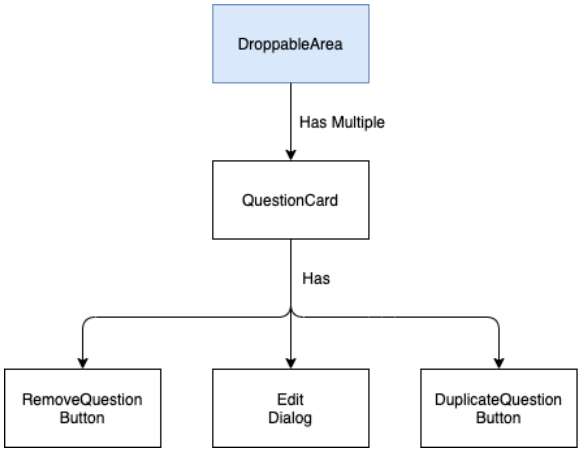
- *Questionnaire* contains a set of questions.
- *QuestionnaireDetails Dialog* enables the user to change the name of the questionnaire or duplicate it.



- *WatchTutorial* component enables user to watch the tutorial on YouTube.
- *Droppable Area* represents the ordered "bag" of questions of the questionnaire.
- *Questionbar* represents different types of the questions, that can be dragged to *Droppable Area*.
- *App bar* shows the status if the user. Moreover it provides editing functionality of the application, namely erasing, rendering and saving the questionnaire.



- *LogIn Dialog* transitions to *Auth0 Dialog* described above.
- *ThemeSwitcher* provides user with dark and mode theme.



- *QuestionCard* serves as an abstraction for the question.
- *RemoveQuestion Button* removes question from the questionnaire.
- *DuplicateQuestion Button* duplicates the question and appends the duplicate to the bottom of *Droppable Area*.
- *Edit Dialog* enables user to modify the properties of the question.

0.6.2 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 u-can-act repository. Questions are mapped to the questions section. We used react-beautiful-dnd library to implement drag and drop functionalities of the questions. The visual design is supposed to be minimalistic.

0.6.3 QuestionsPage

`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.

0.6.4 Edit Dialog

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

0.6.4.1 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.

0.6.4.2 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 categorized 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 the 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 visualize the `options` property for `likert` and `dropdown`.

0.6.4.3 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 be done by simply looping over the array after it has been located via the question type.

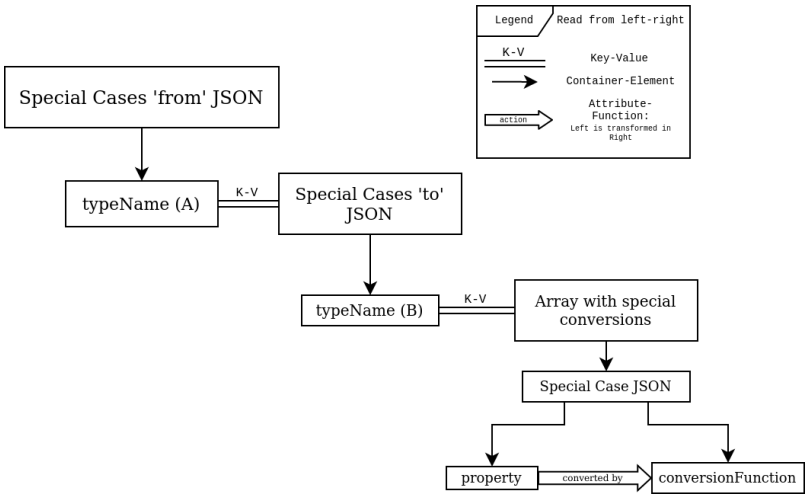
There are some downsides to this implementation. One major and noticeable disadvantage is the lack of visual organization of the properties. Currently, the application does now know how a property should be rendered, other than it should be rendered after a previous property with references to the context 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.

0.6.4.4 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 the 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 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 the 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.

0.6.5 Previous 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 the 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 the 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 constraints. In the 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 the progress.

0.6.6 Current state management: Redux, react-redux, and redux-toolkit

After further inspection, the team decided that we needed a better way to manage the state. The previous system had many issues including rendering and duplicate copies. As every component connected to a context

is re-rendered after the update, this caused extreme performance issues throughout the app. To fix this, a combination of Redux, react-redux, and redux-toolkit libraries was chosen after much review of available choices. Redux is used to share state between components.
react-redux allows us to easily access Redux state and dispatch actions using React.
`redux-toolkit` is used to simplify the reducers, actions, and action creators into a single file for each "slice" of state (e.g. state.*slice*). For the current size of the system, we found that this was an efficient way to organize the "business logic" of the application. In the case that the application gets bigger, this can be converted to using "normal " Redux to connect the supply state to a component. (using react-redux connect, mapStateToProps, mapDispatchToProps, separate action and action creators, etc.)

0.6.7 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 to React. Another big factor that made the 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 questionnaires) and React Bootstrap.

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

In the 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 the 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 questions page).

0.6.9 Showing/hiding questions functionality

In the 2nd block, the team was tasked by the client to make it possible to show or hide questionnaire questions. This was an especially tricky task because of how the system is already implemented. Many scenarios need to be accounted for when implementing this system, because of the required format of the JSON questionnaire. Because our frontend system allows for the questions to be reordered, we needed the question IDS to be calculated dynamically at render. More details about this feature's implementation can be found in the README on the project's Github page.

0.6.10 General sidebar

Implementation of the left sidebar is mostly handled in component `TemporaryDrawer`. This can be seen as a style component, like e.g. `Grid` in `Material-UI`. `TemporaryDrawer` was defined to easily create sidebars with a consistent style. The only thing a developer needs to define is a list of elements to add, and how these elements are stylized and how the sidebar is open is all handled in `TemporaryDrawer`. The layout list contains set prefixes that are used e.g. a button or dividers. However, a developer might want to add an 'unsupported' element to the sidebar later on. In order to catch these, a prefix named `custom` is accepted, that contains any `React` component as value.
The `GeneralSidebar` component mainly contains the layout and its elements, which are given to a `TemporaryDrawer` instance.

0.6.11 Home page

The home page of the application. Its main task is to list all available questionnaires for the user. The landing page's layout is inspired by Google Drive, but not made to be a complete clone of it. The layout was chosen for the great simplicity and clarity it gives: the user sees all questionnaires. As requested, questionnaires are fetched by the backend if the user is logged in and has administrator rights. Clicking a questionnaire header or the 'i' icon on a questionnaire card gives details about the selected questionnaire, and allows the user to go to the edit page with the selected questionnaire loaded into storage.
Fetching data from an outside API might take some time, and therefore status messages can be shown if such

an action is being deployed. These messages are placed on the top, in the place where questionnaires would be shown. Through this, it's easy for the user to know what is happening when there is not any questionnaire shown to them. Status messages make use of the API status implementation, which will be discussed in the next section.
The `HomePage` component is divided into two main parts. `QuestionnaireList` shows all available questionnaires and `QuestionnaireDetails` shows the details of a selected questionnaire. Each of these components handles their data fetching on their own, instead of having the main state in `LandingPage`. This was decided because an API call, to properly monitor and update, takes a lot of code, and this made it a bit unclear what code was linked to what component. To connect a selected questionnaire from `QuestionnaireList` to its details in `QuestionnaireDetails`, a state containing the current selected questionnaire key, that is available in `QuestionnaireList` as part of its fetched data, is given to `QuestionnaireDetails`. The 'setter' for this state is given to `QuestionnaireList`, to make sure the state gets updated with the data, from *where* this data came from.

0.6.12 API calls

0.6.12.1 General structure

In short, generally, a component that makes use of data fetched by the API is structured as follows:

- Tokens from Auth0 accessed through the `useAuth0` hook.
- An API Data state containing an API status and (possible) fetched data
- Asynchronous code handling the actual data fetch (connection with the backend).
- A render function for finally showing the fetched data, or a status/loading message, according to the API status stored in the data state.

This lightly follows the MVC pattern, where the render function is the View, the data state is the Model and the async code is the Controller. The following subsections will discuss the second and third steps of the call, as the first and last step does not have any particular design choices that are part of this section. The tokens are retrieved from Auth0 and its provided hook and rendering is up to the usage of the fetched data, which is beyond this section.

0.6.12.2 Data storage

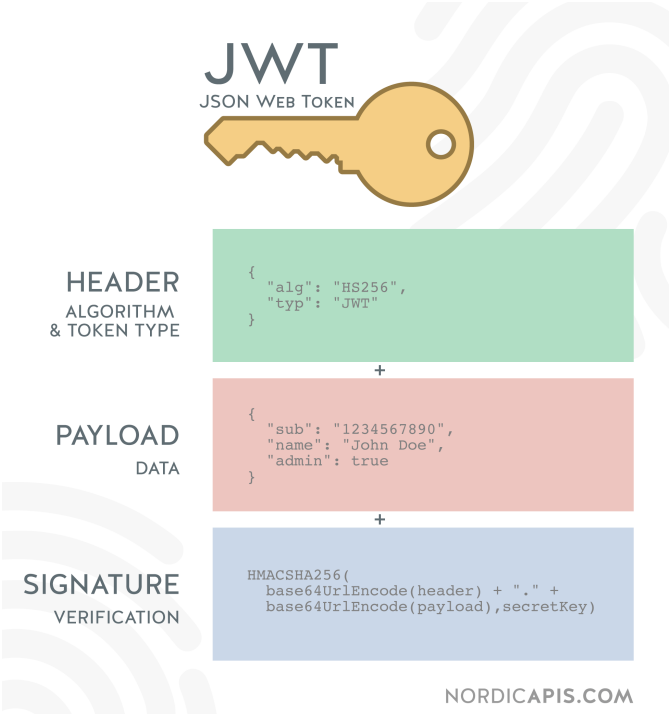
Every data that is fetched from the main backend API is encapsulated in a JSON. This JSON is part of the state, and apart from (if available) fetched data, contains a status code defined in the `API_STATUS` JSON. Storing a status code together with the fetched data was chosen to make it possible for the application to know when to *expect* some data and when not. The application can then act according to the status code, making it possible to show e.g. a loading icon when the status code is `LOADING`. Each of these API call states has initially the `INIT` code. This was chosen so that the application does not make redundant API calls when there is technically nothing to retrieve or change. To store available status codes in a (fixed) JSON was chosen to prevent manually typing a status code as a string (that is prone to spelling error bugs). This implementation also makes it possible for each status code (key in the JSON) to have a different message (value in the JSON per key), depending on what the client might prefer in the future.

0.6.12.3 Connection with the backend

The actual API call code follows a usual structure. This was automatically a standard due to having each code for API calls to be heavily based on the code generated by the Postman API manager. This also makes it possible to easily understand the code and allows for code and naming consistencies. Each step in the API call is registered in the data storage stage mentioned in the previous section. Every variable needed to make contact with the API, base URL, Auth0 configurations, etc., are stored in an `auth_config` file, so it can be easily changed with the client's configurations.

Security

Security is an essential part of the application. The main technology is `JSON Web Token`, which is used for API authentication. Moreover, the application uses the `Auth0` technology for authentication functionalities.



0.7 Auth0

0.7.1 Introduction

Auth0 is a flexible, drop-in technology to add authentication and authorization services. The application uses

it for authentication functionality.

0.7.2 Usage

Auth0 provides SDKs for various programming languages and a broad set of tutorials on how to use them according to what goal might be achieved. This made it relatively easy to create a connection between Auth0 and our app: a simple tutorial could be followed to achieve this.

0.7.2.1 Logging in

Logging in has a very simple implementation thanks to the SDK used. Like token generation, logging in is completely handled on Auth0's side, and can be accessed by calling a single login function. Auth0 *gives* the possibility to construct an own login page, that uses a special Auth0 API that goes more detailed into the logging in process. This was however not chosen as, for this particular project, having the tokens ready was the main priority. Creating a (style-consistent) login page might be something for the future.

0.7.2.2 Authorized API calls

Authorized API calls are needed for most requests to the backend. Through the backend code, it was possible to see what actual link to call as an HTTP request, and the client provided us with the information that the identity token should be given as `Authorization` header. For safety, it was decided to not store this token anywhere, and simply call the Auth0 SDK function for token retrieval from the Auth0 API. This also makes keeping the token up to date (and inaccessible after a logout) much easier, as this is simply all handled on Auth0's side.
For accessing admin-restricted data, a special access_level needs to be added ('admin' for admin-restricted calls). This is an implementation of the backend and can be added through Auth0's Rules feature. In short, this feature makes it possible to run JavaScript code on the JWT token to be given upon verification, which is exactly what is needed to add the admin privilege to a user.

0.8 Render Questionnaire

The application has the rendering functionality. The render functionality calls an external API, namely the u-can-act questionnaire render. For this call the `JSON` content of the questionnaire is encrypted base 64 and sent to the render. The API was created by the product owner [Frank Blaauw].

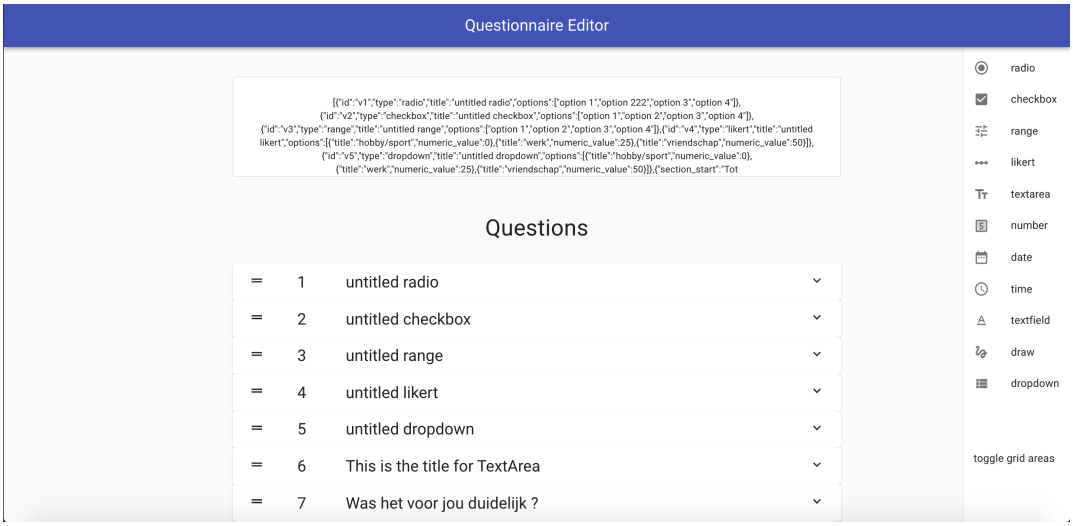
Graphical User Interface

Block A

Over the last couple of months, a lot has changed in terms of User Interface and layout of our app, which is why we have split this chapter into two separate sections, i.e one for each block. Please find all of our different components for the GUI which was done during the first block below.

0.9 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.



0.9.1 App bar

At the top of the page, we can see the app bar 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 app bar.

0.9.2 Toolbar

Next, we have the toolbar which can be seen on the right of the page. The 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 wants to use and drop it in the question area. Moreover, we have given the draggable component a blinking color while it is being dragged, which enables the user to realize that a component is indeed being dragged. This was one of the harder functionalities to implement.

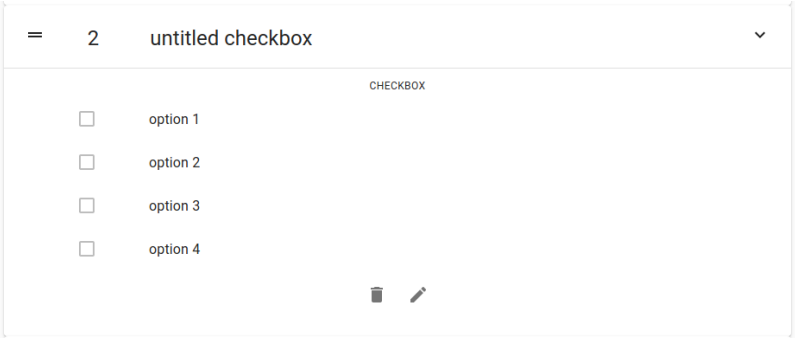
In order to implement all these functionalities, we have had to import libraries such as **DragDropContext**, **Draggable** and **Droppable** all from "react-beautiful-dnd". Finally, we have a "toggle grid areas" component on the toolbar, which portrays all the different sections if you will, of the 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.

0.9.3 JSON section

This section can be found right below the app bar 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 visualizing 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 the 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 the client. The reference to the parser can be found in the *Architectural overview* subsection within the *Client* section of this document.

0.9.4 Question Area

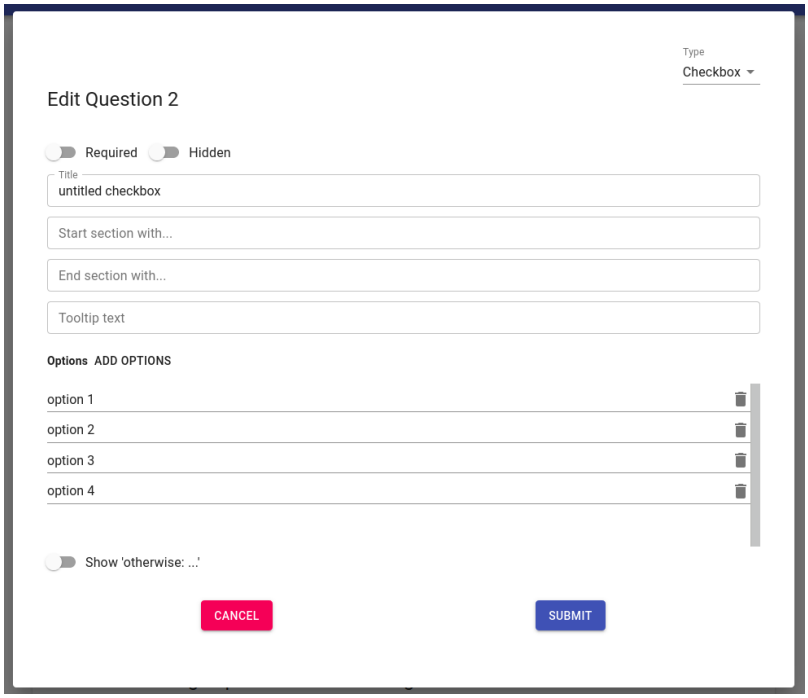
This is the main part of the application. Here we have a collection of all the question types that we have added. This section already has questions inside when the application is initially launched. This was purely made for testing the product, where we could see how each question type would look like. Each component of this section is essentially a question type and its corresponding attributes. Although each question type has its properties, each type also shares 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 edit and delete button. Each time we add a question to the question area, it expands in size to accommodate more available questions.



Clicking a question shows more detailed information about the question.

0.9.5 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 the 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.



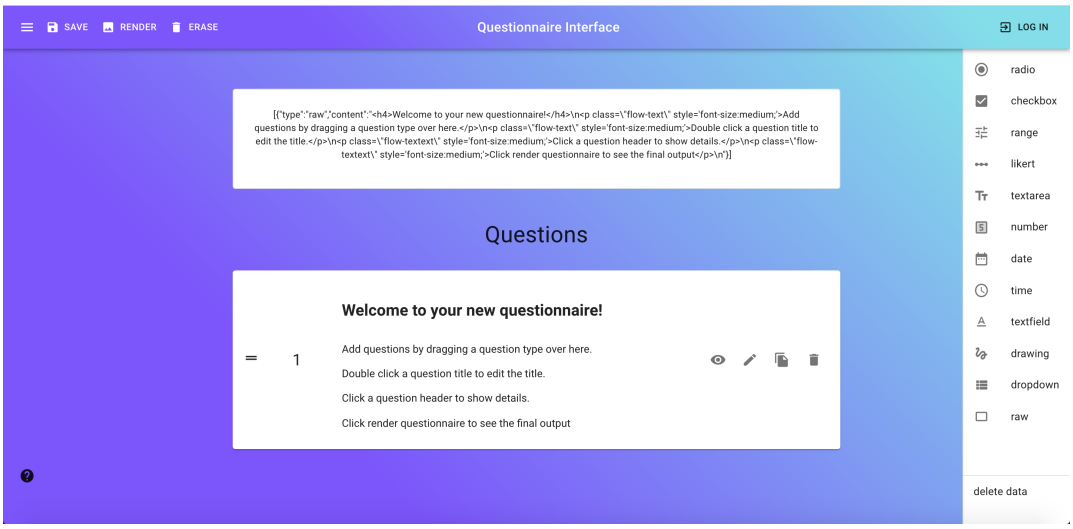
As said previously, the only fields which will be affected by this edit are the *Title* field and the *Options* field as the edit page is only a prototype and still has to adjust for each specific type. That however does not mean the changes are not recorded correctly. All properties with their values 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.

Block B

During this block, a lot has been changed in terms of the GUI, we have integrated our authorization functionality, a landing page was added, a more aesthetically pleasing color was used for the background, a Dark mode toggle was added, a side menu on the left of the page was integrated, and more. Please find all of our new features and components in more detail below.

0.10 Layout of the Main page

When we launch our app, this is now the page we are prompted to. We can see that quite a few things have changed from the product we delivered in the first block. The first noticeable difference we see is the background color of both the app bar and the main page, which has been changed to a purple-ish color, which we believe makes our app more lively. The main components of this page such as the app bar, the Toolbar, the Questions sections, and the JSON section are still in use with some added functionalities and better look. We have again broken these components into different sections below for a clearer summary of each.



0.10.1 App bar

The app bar now contains a toggle to bring out a menu (which will be explained in details below), a save button which enables the user to save their questions created, a render button which enables the user to view their questions through our client's JSON parser, an erase button which deletes all the data currently on the page and creates enables the user to start from scratch. On the right-hand side of the app bar, there is a log in button which now enables the user to log in and make use of functionalities such as save and load an existing questionnaire.

0.10.2 Toolbar

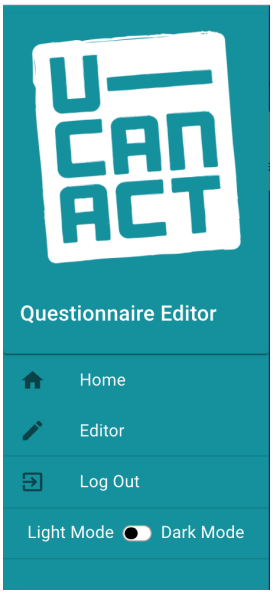
Our toolbar which can be seen on the right-hand side now contains an added question type, namely "raw".

0.10.3 Question Area

Our question area now contains a raw type question by default when the app is launched, this is to help out a new user on how to make use of this interface by giving a few instructions. This question can and should be deleted by the user once he/she is comfortable in using the app.

0.10.4 Left Side Menu

When the user clicks on the toggle button of the app bar, the menu seen below appears on the left-hand side of the page. From this menu, a user can navigate to the home page, which shows all of the saved questionnaires (the user has to be logged in to make use of this functionality), or to the editing page, which is the main page of the app already discussed previously, and finally, the user can log out (or login) through this menu. Additionally, the dark mode theme can be activated through this menu.

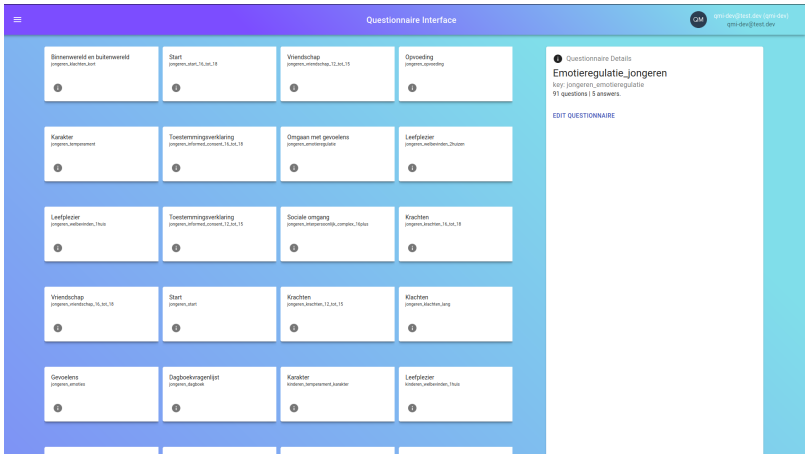


0.10.5 Video Tutorial

We have added a video tutorial section on the bottom left of the main page, which can be opened by clicking on the question mark toggle. The video gives the user a run through the application on how to use it and make the best out of it.

0.11 Layout of the Home Page

The layout of the home page (also referred to as a landing page) has similarities to the editor page. It shares the same app bar and sidebar components and uses the same style. The layout of the landing page is overall simple, and as addressed in **section 6.9**, has two components.



0.11.1 Questionnaire list

On the left side, we can see the list of all questionnaires, displayed as a grid of tiles. Each questionnaire tile shows its title in the header and the questionnaire key in the tile body. By either clicking the title header or clicking the single 'i' button in the tile the questionnaire gets loaded into the right side of the page, which is the questionnaire details section.

0.11.2 Questionnaire details

The details section shows a bit more information about the selected questionnaire. We can see the questionnaire name as the title and the questionnaire key in the body. Additionally, we can see the number of questions and answers that the selected questionnaire has, as well as a button that loads the questionnaire into the editor. This button automatically redirects to the editor page as well.

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 organization.

0.12 Scrum Events and Task Distribution

Each of the sprints had a duration of two weeks. The sprints were each started in a meeting that followed after a meeting with the 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 organized via an online alternative after the campus closed down due to the Coronavirus countermeasures.
These following subsections will summarise task distributions that used elements of the 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.

0.13 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 the TA (only 'Authentication' had its 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.

0.13.1 General Usage

After the initial 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 organize and update sprint progress. Instead of Slack, initially, WhatsApp was used. Because of the 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, for these mistakes to not happen again. Together with that, it was decided to update more regularly 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, the 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 organizing tasks compared to the beginning and we hope that eventually, updating Trello would fully become a part of the scrum routine.

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

0.13.2 Additions

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

- Questions for Frank: A list where questions to the 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.

0.14 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 the own progress for sprint 1 which later were merged in the development branch. At the end of each sprint, the development branch was merged with the master branch.

Merge issues were not 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 present, these were mostly small and could be overwritten by the local branch. Thanks to the usage of WebStorm, issues could first be reviewed before things were overwritten. Because there were mainly only two people assigned to refactoring code, apart from one issue at 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.

After reading several blogs, as well as, articles about the best practices, the team found out that using the Git Flow approach would have been the better practice in terms of development. However, the current ticket system of assigning tasks to users based on our planning poker results worked flawlessly for the scope of this project .

Build Process and Deployment

0.15 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 `CircleCI` for the pipeline choice. We chose to do this 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`. That being said CircleCI was not a requirement however the team believed it would be a great addition for the current and future scope of this project. In addition to CircleCI, we integrated the software with `netlify`. Which is described in the following section in more depth.

0.16 Netlify

Netlify is a web hosting infrastructure. It works by connecting to your GitHub repository to pull your source code then it typically runs a build process to pre-render all of your pages in static HTML. Using Netlify the team was quickly able to deploy a static website with a custom url, to provide to the client's customers. As soon as we push changes to the repository, it also triggers a deploy on Netlify. This means, even if the previous steps fail, the site gets still deployed. However, adding CircleCI now means if we break the tests, CircleCI will never do a post request to Netlify and the site won't get deployed unless we fix the issue.

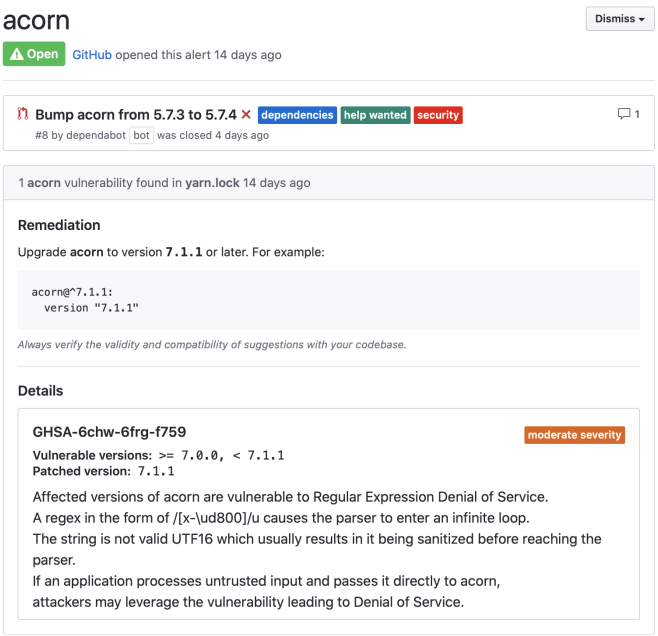
Extra Software Engineering Tools Used

0.17 Dependabot

Dependabot is an automation service that automatically create pull requests to keep your projects dependencies up to date. It has been a great asset to our team because the automation it makes sure our security is maximized in terms of dependencies.

0.18 Snyk

Snyk is a web service that continuously finds and fixes vulnerabilities in open source libraries and containers. Snyk is more secure covering for **Dependabot**. If it found any security alerts or issue, it would notify the user via github. This webservice was added to prevent security flaws while developeing the project.



0.19 Bundlephobia

BundlePhobia is a web service to find out the how much size will a npm package cost your project. It can measure the size of CSS/Sass libs too and report whether the module supports tree shaking. This was a great tool when decided over two similar libraries, for example when choosing Material UI vs Bootstrap. It made it easier to see the potential impact a library can have on the build.

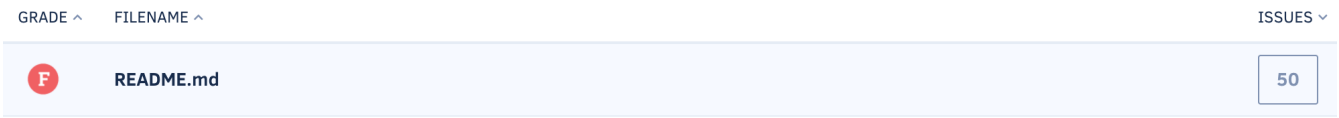
BUNDLEPHOBIA

find the cost of adding a npm package to your bundle

0.20 Codacy

Codacy is a web service for automated code reviews and code analytics. It checks the quality of the commits and notifies about all possible code smells, namely code duplicates. The platform gives the grade for each of the file and what needs to be improved. For example `Readmi` contains 50 issues and has grade **F**.

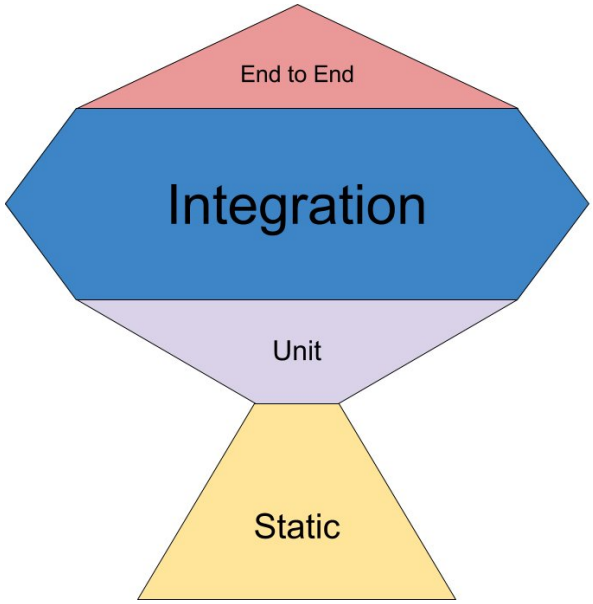
CODACY



Testing

Testing was essential in this project which is why we went through Static testing, Unit testing, Integration testing, and last but not least, End to End testing.

1. Static testing with SonarLint
2. Unit testing with Enzyme
3. Integration testing with Cypress
4. E2E testing with Cypress



The team followed a guideline like the testing trophy shown above by `Kent C Dodds` a well-known software engineer with countless experience in testing JavaScript code. It is also important to note that, the size of each of these may differ slightly based on the team values. The proportions are not meant to be taken as hard fast rules. Just general "guidelines". Since End to End test is generally the most expensive to fix in terms of time as well as monetary wise, a big focus was spent on that section of testing all the user's stories, specifically the main requirements from the client which can be visualized below in the trace-ability matrix. The team aimed for a testing coverage of 70% to 80%. Below are the testing methods in more detail of how it was performed.

Static testing

Using SonarLint, a middle ground for SonarQube as well as ESLint, two very popular tools, SonarLint was able to give us a great static testing report about each class by highlighting, major as well as, minor bugs through the code. A small list of bug which helped us detect was as follows:

1. Unused imports
2. Multiple declarations of variables with the same name
3. Unused pieces of code (dead code)

This methodology was a very cheap, reliable, and quick way to detect bugs.

Unit Testing

Unit testing is a software testing method by which individual units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures, are tested to determine whether they are fit for use. Enzyme testing utility is used in the project. Enzyme provides methods for rendering a component (or multiple components), finding elements, and interacting with elements. Moreover, most of the tests are snapshots tests, which is used for checking whether the UI does not change unexpectedly.

Integration Testing

The integration in the code consisted of mainly testing the interaction between multiple components. Using Cypress the team was able to detect a crucial bug in the making which occurred when a user-selected an option that did not reflect on another component. Testing components working together truly helped build confidence in the communications between functions in the project.

End to End testing

End to End testing, famously known as being the most time consuming usually tends to be tested when actual users use the platform, which is why the code was deployed and given out to users who would interact this project to demo and try and find faults in the system (small focus group). In addition to this interesting method, Cypress was great for mimicking user moves when it came to interacting with the UI. This testing method helped the team test most of the user's stories which as previously mentioned can be seen in the traceability matrix.

All in all, the testing conducted should provide some confidence to the client about the quality and consideration that the team took in building this project.

Below is the code coverage report with a 73% coverage rate. This was a reasonable rate since this was not including our unit test coverage.

All files

72.89% Statements `863/1184` **59.41%** Branches `363/611` **68.67%** Functions `285/415` **73.21%** Lines `809/1105`

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

File		Statements		Branches		Functions		Lines	
src/features/API		100%	2/2	100%	0/0	100%	0/0	100%	2/2
src/NavigationBarButtons		94.44%	17/18	100%	0/0	87.5%	7/8	94.44%	17/18
src/features/question		90.91%	10/11	100%	0/0	88.89%	8/9	90%	9/10
src/components/properties		90.24%	185/205	75.56%	170/225	86.05%	74/86	91.41%	181/198
src/utils		84.78%	39/46	32.35%	11/34	55.56%	5/9	85.71%	36/42
src/components		77.94%	219/281	61.05%	58/95	78.48%	62/79	77.86%	204/262
src/components/properties/TextArrayTemplate		71.66%	134/187	76.32%	58/76	61.43%	43/70	72.84%	118/162
src/components/buttons		71.43%	25/35	58.33%	7/12	100%	7/7	75.76%	25/33
src/components/previews		67.59%	73/108	51.72%	15/29	52.94%	27/51	66.02%	68/103
src/features/questionnaire		66.67%	2/3	100%	0/0	0%	0/1	66.67%	2/3
src		63.89%	69/108	38.33%	23/60	60.47%	26/43	61.76%	63/102
src/features/questions		53.33%	32/60	32.14%	9/28	64.71%	11/17	56.36%	31/55
src/components/LandingPage		49.5%	50/101	27.27%	12/44	44.44%	12/27	48.96%	47/96

APIs

In terms of testing the APIs used during this project the client willingly volunteered to test the APIs which had to be created in the back end to be tested by `Researchable` itself. Using Swagger, the client and his team tested the APIs which our current API calls which gives the team extra confidence since the code coverage report would be even higher after having tested the APIs.

Requirements Traceability Matrix(RTM)

Requirement	Module	Files Affected	Test	Passed
Create new questionnaire with any question type	FrontEnd	Question.js BottomSection.js DatePickerPreview.js DrawingPreview.js DropdownPreview.js LikertPreview.js NumberPreview.js RadioCheckboxPreview.js RangePreview.js RawPreview.js TextArea.js TextFieldPreview.js TimePickerPreview.js	CheckboxPreview.test.js DropdownPreview.test.js LikertPreview.test.js NumberPreview.test.js RadioPreview.test.js RangePreview.test.js TextArea.test.js TextField.test.js TimePreview.test.js	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓
		Edit a question/questionnaire		
Cancel the edit dialog after opening it for a specfic question	FrontEnd	Question.js EditQuestionButton.js	cancelEditDialog.spec.js	✓
Set the radius and density for a draw question	FrontEnd	Question.js EditQuestionButton.js	changeDrawingOptions.js	✓
Change the text of a dropdown question certain option	FrontEnd	Question.js EditQuestionButton.js	changeOptionOfItem.spec.js	✓
Delete an option of a specific question	FrontEnd	Question.js EditQuestionButton.js	deleleAnOption.spec.js	✓
Delete an option of a Range type question	FrontEnd	Question.js EditQuestionButton.js	deleteAnOption.spec.js	✓
Delete an option of a Dropdown type question	FrontEnd	Question.js EditQuestionButton.js	deleteAnOption.spec.js	✓
Double tap title to edit without making a change	FrontEnd	Question.js EditQuestionButton.js	doubleQuestionClickToEditTitleWithoutChange.spec.js	✓
Double tap to edit title	FrontEnd	Question.js QuestionsPage.js	doubleQuestionClickToEditTitle.spec.js	✓
Set a label for a dropdown question	FrontEnd	Question.js QuestionsPage.js	editLabelOption.js	✓
Set a specific min, max and max length for a number question	FrontEnd	Question.js QuestionsPage.js	editNumberOptions.js	✓
Change the placeholder text of a date question	FrontEnd	Question.js QuestionsPage.js	editPlaceholderOption.js	✓
Set a specific min, max and step for a range question	FrontEnd	Question.js QuestionsPage.js	editRangeOption.js	✓
Set a specific hoursFrom, hoursTo and hoursStep for a time question	FrontEnd	Question.js QuestionsPage.js	editTimeOptions.js	✓
Edit the title of a specific question	FrontEnd	Question.js QuestionsPage.js	editTitle.spec.js	✓
Edit the title of a drawing question	FrontEnd	Question.js QuestionsPage.js	editTitle.spec.js	✓
Edit the title of a raw question	FrontEnd	Question.js QuestionsPage.js	editTitle.spec.js	✓
Enable a required question property for a specific question	FrontEnd	Question.js QuestionsPage.js	enableRequiredProperty.spec.js	✓
Enable a section end for a specific question	FrontEnd	Question.js QuestionsPage.js	enableSectionEnd.spec.js	✓
Enable the tooltip for a specific question	FrontEnd	Question.js QuestionsPage.js	enableToolTipText.spec.js	✓
Change the visibility of a question	FrontEnd	Question.js QuestionsPage.js	hideQuestion.spec.js	✓
Delete existing questionnaire	FrontEnd	LandingPage.js	erase_questionnaire.spec.js	✓
Delete existing question	FrontEnd	Question.js	delete_question.spec.js	✓
Drag a question from the sidebar	FrontEnd	Question.js	dragging_questions_from_sidebar.spec.js	✓
Drag a question with the handle	FrontEnd	Question.js	dragging_questions_with_handle.spec.js	✓
Render the questionnaire	FrontEnd	Question.js	render_questionnaire.spec.js	✓
Duplicate a question	FrontEnd	Question.js DuplicateQuestionButton.js	duplicate_question.spec.js	✓
Scroll back to the top feature	FrontEnd	Question.js QuestionsPage.js	scroll-to-top.spec.js	✓
Switching between Light and Dark theme	FrontEnd	QuestionsPage.js	switch_modes.spec.js	✓
Watch a tutorial from Youtube	FrontEnd	ScrollArrow.js	watch_tutorial.spec.js	✓
Create a landing page for the home screen	FrontEnd	LandingPage.js	landing_page.spec.js	✓

0.21 Abbreviations

1. **UI**: User interface.
2. **JSON**: JavaScript Object Notation. It is a lightweight format for storing and transporting data. Often used when data is sent from a server to a web page.
3. **Deployment**: The process of setting the netlify server in order to launch the React App.
4. **API**: An acronym for Application Programming Interface, which is a software intermediary that allows two applications to talk to each other.
5. **SDK**: Software Development Kit. Code used for developing the app.
6. **Deploy**: When a software system is available for use, generally on some cloud infrastructure.
7. **JWT**: Web technology used for secure authentication.
8. **E2E**: End to End (testing method).
9. **MVC**: Model–view–controller (usually known as MVC) is a software design pattern.
10. **Swagger**: It allows you to describe the structure of your APIs so that machines can read them.

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.
M. Hiro	April 16, 2020	Whole document	Edited sections based on feedback from TA
M. Hiro	May 8, 2020	Software Engineering tools	Added section
M. Hiro	May 9, 2020	Introduction	Edited introduction section
H. Shmak	May 9, 2020	Whole document	Edited the hyperlinks
M. Hiro	May 18, 2020	Extra Software Engineering Tools	Added information about CircleCI, Dependabot, Netlify
H. Shmak	May 18, 2020	Extra Software Engineering Tools	Added information about Snyk
M. Hiro	May 19, 2020	Extra Software Engineering Tools	Added information about bundlephobia
M. Hiro	May 23, 2020	Introduction	added extra information
M. Hiro	May 24, 2020	Architectural overview	Removed overall introduction section
M. Hiro	May 24, 2020	Backend	Added information to backend section
M. Hiro	May 25, 2020	Frontend	Edited the section
M. Hiro	May 25, 2020	Architectural overview	Added overall project diagram
H. Shmak	May 26, 2020	Extra Software Engineering Tools	Added information about Codacy
H. Shmak	May 27, 2020	General Structure	Introduced General Structure section
H. Shmak	May 27, 2020	Security	Edited the section.
H. Shmak	May 28, 2020	Frontend	Elaborated on Architectural overview.
H. Shmak	May 29, 2020	Security	Introduced Auth0 & Render Questionnaire sections.
M. Hiro & H. Shmak	May 30, 2020	Architectural Decisions	Introduced General Structure section
M. Hiro & H. Shmak	May 31, 2020	Architectural Decisions	Elaborated on General Structure section
K. Jokhan	June 1, 2020	Architectural Decisions	Introduced General sidebar, Landing page and API calls section
K. Jokhan	June 1, 2020	Security	Subdivided and elaborated on Auth0 section.
K. Jokhan	June 1, 2020	Technology Stack	Added usage of Auth0, Unirest and Postman.
K. Jokhan	June 1, 2020	Frontend	Elaborated on Login functionality section.
K. Jokhan	June 1, 2020	The document	Minor spelling fixes
K. Jokhan	June 1, 2020	Graphical User Interface	Minor spelling and naming fixes
K. Jokhan	June 1, 2020	Graphical User Interface	Introduced and elaborated on landing page section
K. Jokhan	June 2, 2020	The document	Fixed spelling and naming errors
K. Jokhan	June 2, 2020	Frontend	Renamed 'Login Functionality' to 'Authorization Functionality' and elaborated on the section.
K. Jokhan	June 2, 2020	Team organisation	Removed individual sprint summaries
M. Hiro	June 4, 2020	Testing	Introduced section
M. Hiro	June 5, 2020	Testing	Added details in the testing section
M. Hiro	June 6, 2020	Testing	Added more details in the testing section
M. Hiro	June 7, 2020	Abbreviations	Introduced the section and added some points
K. Jokhan	June 7, 2020	List of APIs used	Introduced the section and added some points
H. Shmak	June 8, 2020	Testing	Introduced the section and added some points.
H. Shmak	June 8, 2020	Frontend Architectural Decisions	Updated the diagrams.
H. Shmak	June 8, 2020	Frontend Architectural Decisions	Elaborated on General structure section.
P. Poshyachinda	June 10, 2020	Frontend Architectural Decisions	Added subsection about new state management system.
P. Poshyachinda	June 10, 2020	Frontend Architectural Decisions	Added subsection hiding/showing questions functionality.
P. Poshyachinda	June 10, 2020	List of APIs used	Added APIs.
M. Hiro	June 10, 2020	Github	Elaborated on the section.
M. Hiro	June 10, 2020	Abbreviations	Added more points.
R. Rey	June 10, 2020	Graphial User Interface	Extended for Block B
H. Shmak	June 10, 2020	Frontend Architectural Decisions	Updated the diagrams and added new diagrams.
H. Shmak	June 10, 2020	Frontend Architectural Decisions	Elaborated on General Structure setion.
K.Jokhan	June 10, 2020	List of APIs Used	Removed section due to redundancy
K.Jokhan	June 10, 2020	The document	Reviewing the whole document, spelling consistencies, removed inconsistencies.
R.Rey	June 10, 2020	RTM	Added Requirements Traceability Matrix