University of Groningen
Software Engineering

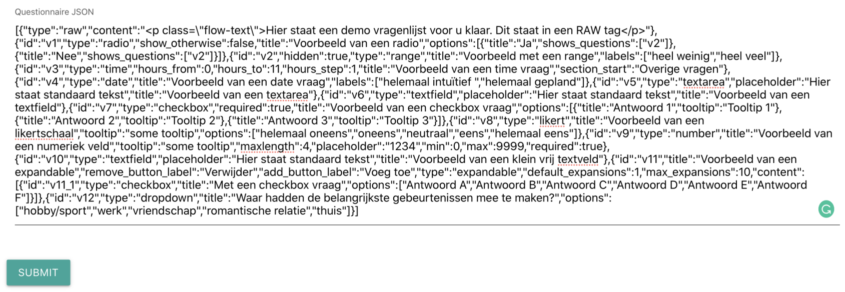
QUESTIONNAIRE MANAGEMENT INTERFACE
Architectural Document

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

Contents

Introduction

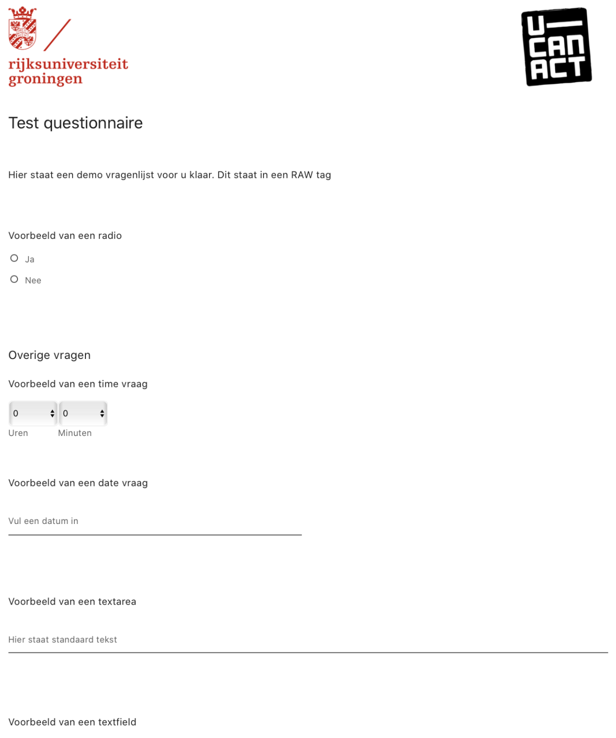
A questionnaire is a research instrument consisting of a series of questions for the purpose of gathering information from respondents. The goal for this was project to build a front-end interface for the questionnaire engine, that enables users to define their questionnaires. Currently, to be able to 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:



Questionnaire JSON

```
[{"type":"raw","content":"<p class=\"flow-text\">Hier staat een demo vragenlijst voor u klaar. Dit staat in een RAW tag</p>"},
{"id":"v1","type":"radio","show_otherwise":false,"title":"Voorbeeld van een radio","options":[{"title":"Ja","shows_questions":["v2"]},
{"title":"Nee","shows_questions":["v2"]}]},{"id":"v2","hidden":true,"type":"range","title":"Voorbeeld met een range","labels":["heel weinig","heel veel"]},
{"id":"v3","type":"time","hours_from":0,"hours_to":11,"hours_step":1,"title":"Voorbeeld van een time vraag","section_start":"Overige vragen"},
{"id":"v4","type":"date","title":"Voorbeeld van een date vraag","labels":["helemaal intuïtief ","helemaal gepland"]},{"id":"v5","type":"textarea","placeholder":"Hier
staat standaard tekst","title":"Voorbeeld van een textarea"},{"id":"v6","type":"textfield","placeholder":"Hier staat standaard tekst","title":"Voorbeeld van een
textfield"},{"id":"v7","type":"checkbox","required":true,"title":"Voorbeeld van een checkbox vraag","options":[{"title":"Antwoord 1","tooltip":"Tooltip 1"},
{"title":"Antwoord 2","tooltip":"Tooltip 2"},{"title":"Antwoord 3","tooltip":"Tooltip 3"}]},{"id":"v8","type":"likert","title":"Voorbeeld van een
likertschaal","tooltip":"some tooltip","options":["helemaal oneens","oneens","neutraal","eens","helemaal eens"]},{"id":"v9","type":"number","title":"Voorbeeld van
een numeriek veld","tooltip":"some tooltip","maxlength":4,"placeholder":"1234","min":0,"max":9999,"required":true},
{"id":"v10","type":"textfield","placeholder":"Hier staat standaard tekst","title":"Voorbeeld van een klein vrij textveld"},{"id":"v11","title":"Voorbeeld van een
expandable","remove_button_label":"Verwijder","add_button_label":"Voeg toe","type":"expandable","default_expansions":1,"max_expansions":10,"content":
[{"id":"v11_1","type":"checkbox","title":"Met een checkbox vraag","options":["Antwoord A","Antwoord B","Antwoord C","Antwoord D","Antwoord E","Antwoord
F"]}]},{"id":"v12","type":"dropdown","title":"Waar hadden de belangrijkste gebeurtenissen mee te maken?","options":
["hobby/sport","werk","vriendschap","romantische relatie","thuis"]}]
```

Clearly, 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 too create a simple yet elegant interface to ease this process when creating questionnaires. Which in turns creates this JSON by through the new interface. Below is a picture to see how the JSON get's rendered in the system after the JSON is created.



Test questionnaire

Hier staat een demo vragenlijst voor u klaar. Dit staat in een RAW tag

Voorbeeld van een radio

○ Ja

○ Nee

Overige vragen

Voorbeeld van een time vraag

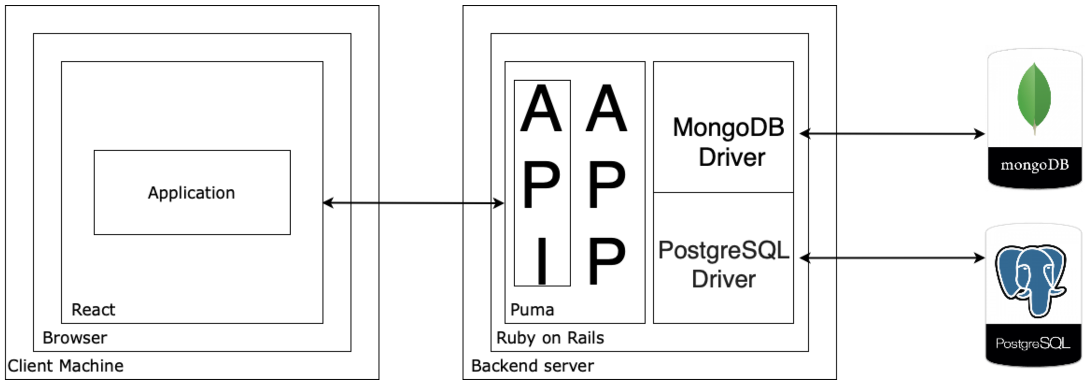
Voorbeeld van een date vraag

Voorbeeld van een textarea

Voorbeeld van een textfield

Client

Architectural overview

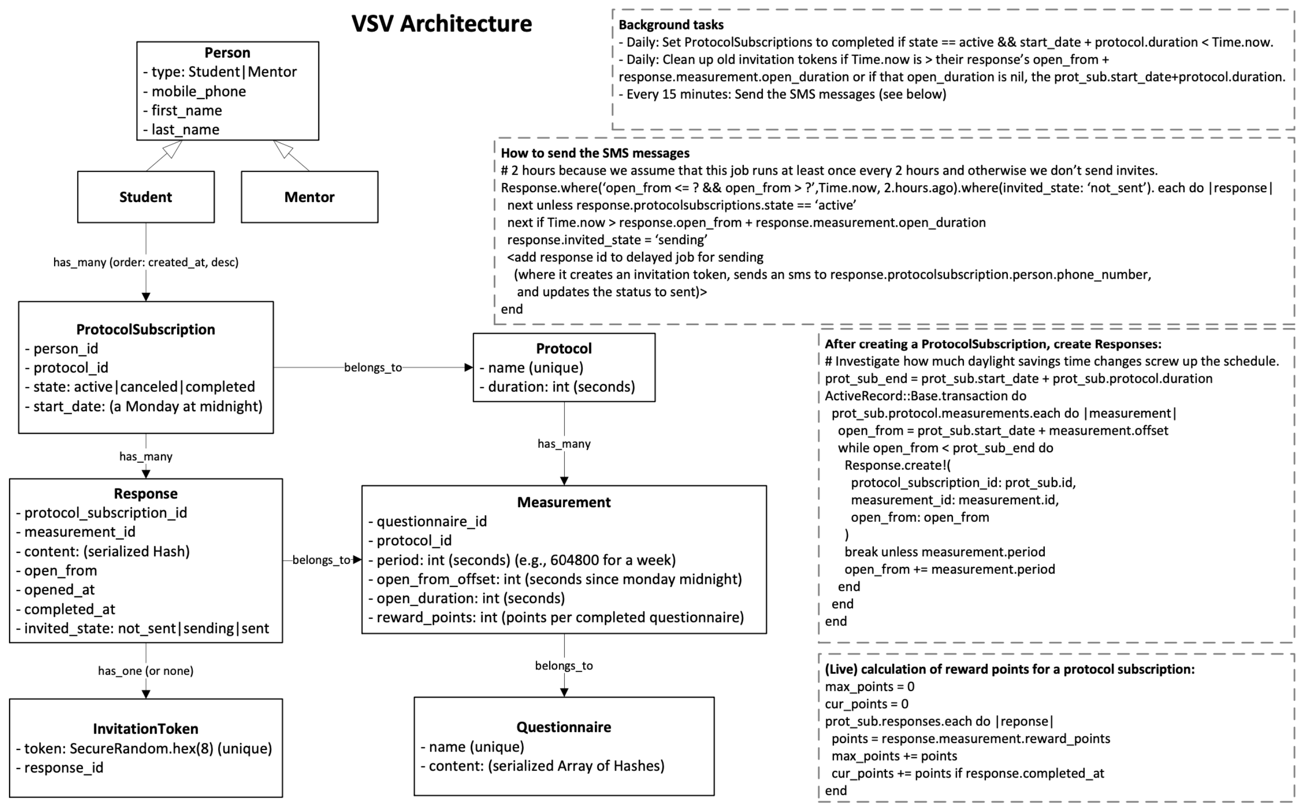


While the project does not suggest to develop the back-end, the overview of the back-end is still provided. This is done due to consistency matters, namely it is better to grasp the structure of the project and front-end decisions made throughout the project. The document mainly elaborates on the left 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 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). 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.


VSV Architecture
Person
- type: Student|Mentor
- mobile_phone
- first_name
- last_name
Student
Mentor
has_many (order: created_at, desc)
ProtocolSubscription
- person_id
- protocol_id
- state: active|canceled|completed
- start_date: (a Monday at midnight)
belongs_to
Protocol
- name (unique)
- duration: int (seconds)
has_many
has_many
Response
- protocol_subscription_id
- measurement_id
- content: (serialized Hash)
- open_from
- opened_at
- completed_at
- invited_state: not_sent|sending|sent
belongs_to
Measurement
- questionnaire_id
- protocol_id
- period: int (seconds) (e.g., 604800 for a week)
- open_from_offset: int (seconds since monday midnight)
- open_duration: int (seconds)
- reward_points: int (points per completed questionnaire)
has_one (or none)
belongs_to
InvitationToken
- token: SecureRandom.hex(8) (unique)
- response_id
Questionnaire
- name (unique)
- content: (serialized Array of Hashes)

Background tasks

- Daily: Set ProtocolSubscriptions to completed if state == active && start_date + protocol.duration < Time.now.
- Daily: Clean up old invitation tokens if Time.now is > their response's open_from + response.measurement.open_duration or if that open_duration is nil, the prot_sub.start_date+protocol.duration.
- Every 15 minutes: Send the SMS messages (see below)

How to send the SMS messages

```
# 2 hours because we assume that this job runs at least once every 2 hours and otherwise we don't send invites.
Response.where('open_from <= ? && open_from > ?',Time.now, 2.hours.ago).where(invited_state: 'not_sent'). each do |response|
  next unless response.protocolsubscriptions.state == 'active'
  next if Time.now > response.open_from + response.measurement.open_duration
  response.invited_state = 'sending'
  <add response id to delayed job for sending
    (where it creates an invitation token, sends an sms to response.protocolsubscription.person.phone_number,
     and updates the status to sent)>
end
```

After creating a ProtocolSubscription, create Responses:

```
# Investigate how much daylight savings time changes screw up the schedule.
prot_sub_end = prot_sub.start_date + prot_sub.protocol.duration
ActiveRecord::Base.transaction do
  prot_sub.protocol.measurements.each do |measurement|
    open_from = prot_sub.start_date + measurement.offset
    while open_from < prot_sub_end do
      Response.create!(
        protocol_subscription_id: prot_sub.id,
        measurement_id: measurement.id,
        open_from: open_from
      )
      break unless measurement.period
      open_from += measurement.period
    end
  end
end
```

(Live) calculation of reward points for a protocol subscription:

```
max_points = 0
cur_points = 0
prot_sub.responses.each do |reponse|
  points = response.measurement.reward_points
  max_points += points
  cur_points += points if response.completed_at
end
```

0.2 Frontend

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

1. The ability to login
2. The ability to create a questionnaire which generates a JSON format.

0.2.1 Login Functionality

The login 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 login information is the `Auth0Provider` component.
Login 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: `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.

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` 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` 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. The `Postman` API manager is used to debug API calls to the backend, and moreover, was used its code generator as a base for the actual API call code in our application.

0.5 Design Patterns and Programming Paradigms

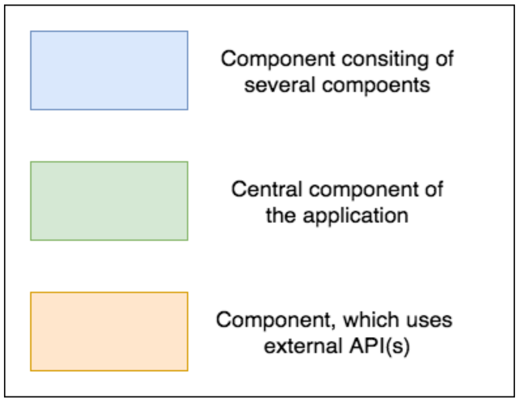
We used the latest version of `React JS`, therefore hooks were used throughout the project. Hooks enable a functional programming style of coding. Moreover, it was decided to use more functional paradigm (of course, we used the object-oriented paradigm as well). It was done due to several reasons, that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

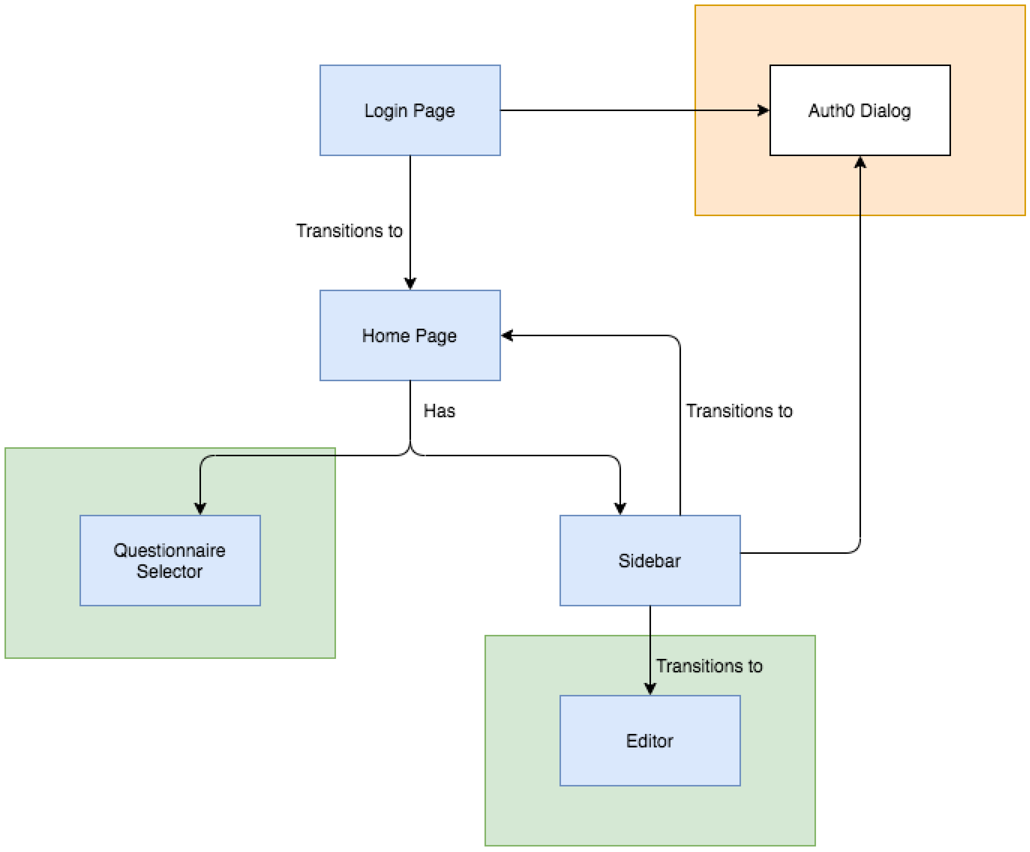
0.6 Frontend Architectural Decisions

0.6.1 General Structure

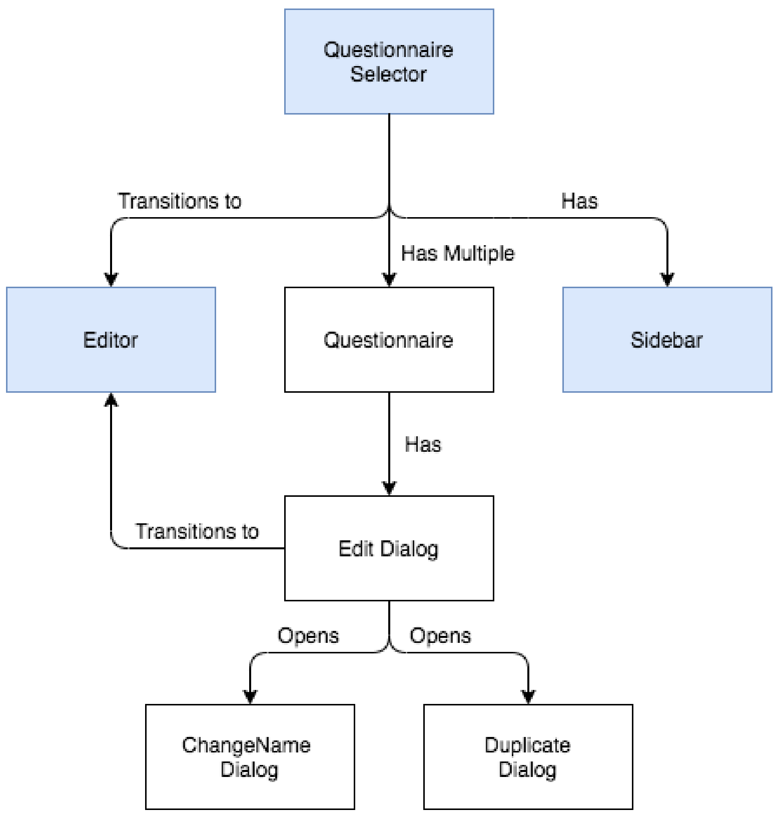
Throughout the document the following legend is used:

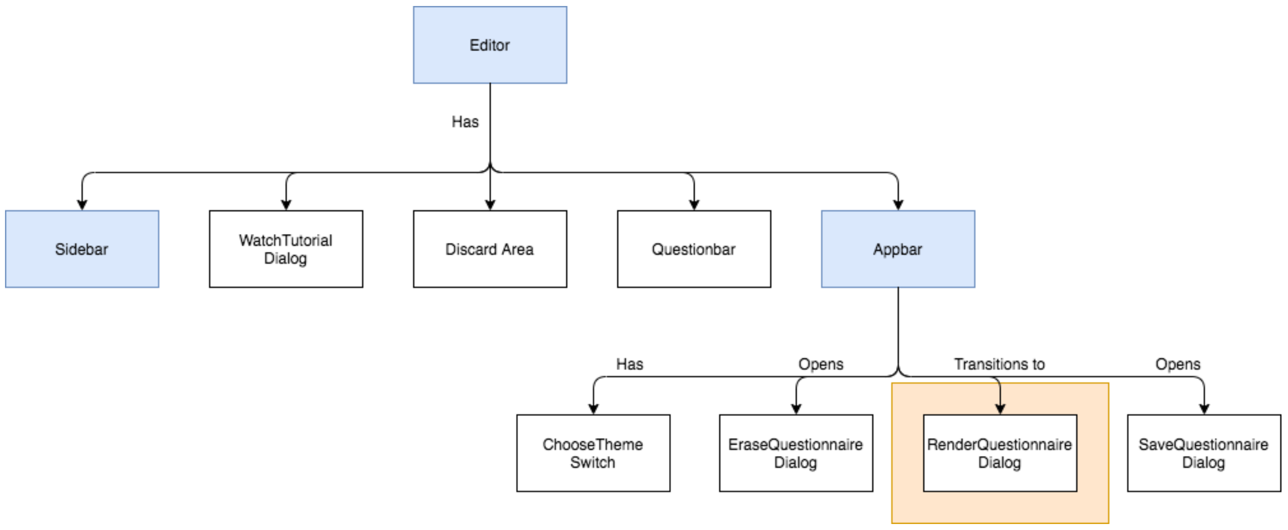

Component consiting of several compoents
Central component of the application
Component, which uses external API(s)

The general class diagram is presented below



Moreover, the extended diagrams are as follows:





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 (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.

0.6.4 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.

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 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`.

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

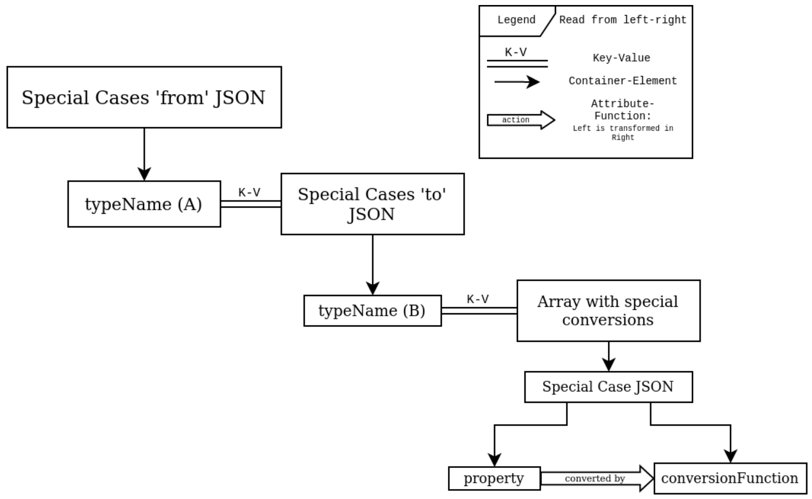
There are some downsides to this implementation. One major, and noticeable disadvantage is the lack of visual organisation of the properties. Currently, the application does now know how a property should be rendered, other than it should be rendered after a previous property with references to the context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

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 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.

0.6.5 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 constraint. 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 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 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 questonnaires) and React Bootstrap.

0.6.7 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 QuestionsPage).

0.6.8 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 actually open is all handled in `TemporaryDrawer`. The layout list contains set prefixes that are used for e.g. a button or dividers. However, a developer might want to add an 'unsupported' element to 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.

0.6.9 Landing page

The landing page acts as home page in 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 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, that will be discussed in the next section.
The `LandingPage` 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

handle their data fetching on their own, instead of having a main state in `LandingPage`. This was decided because an API call, to properly monitor and update, generates 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 actually came from.

0.6.10 API calls

0.6.10.1 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 aync code is the Controller. The following subsections will discuss the second and third steps of the call, as the first and last step do 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, that is beyond this section.

0.6.10.2 Data storage

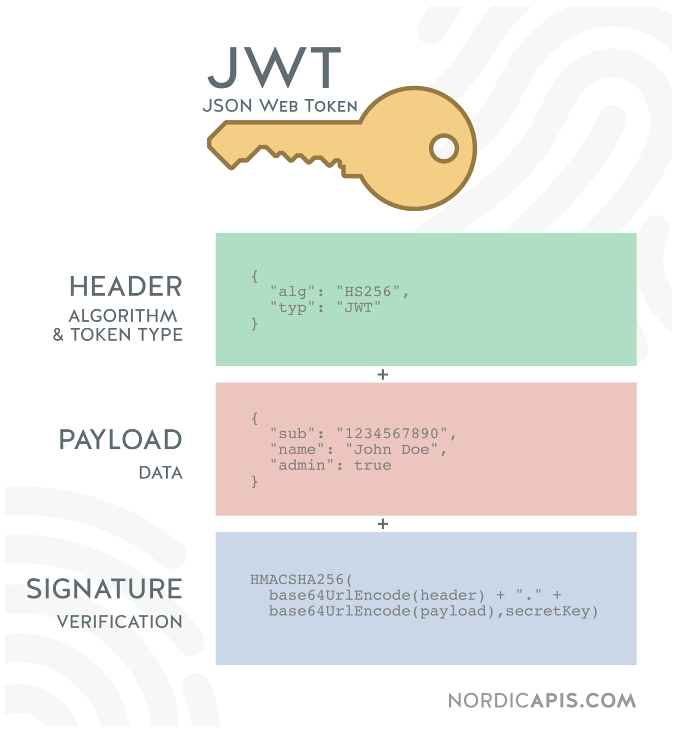
Every data that is fetched from the main backend API is encapsulated in a JSON. This JSON is part of the state, and apart from (if available) fetched data, contains a status code defined in the `API_STATUS` JSON. Storing a status code together with the fetched data was chosen to make it possible fo the application to know when to *expect* some data and when not. The application can then act accordingly 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. These status codes are 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.

0.6.10.3 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 of the code generated by Postman. 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, to have it easily changed when the client wants to insert its own data.

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 the login functionality.



```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

+

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

+

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),secretKey)
```

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 the authentication functionality. The product owner specifically mentioned not to introduce the create functionality for the application. Therefore, only already registered within Google users should be able to log in the application.

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.

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 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 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-privilige to a user.

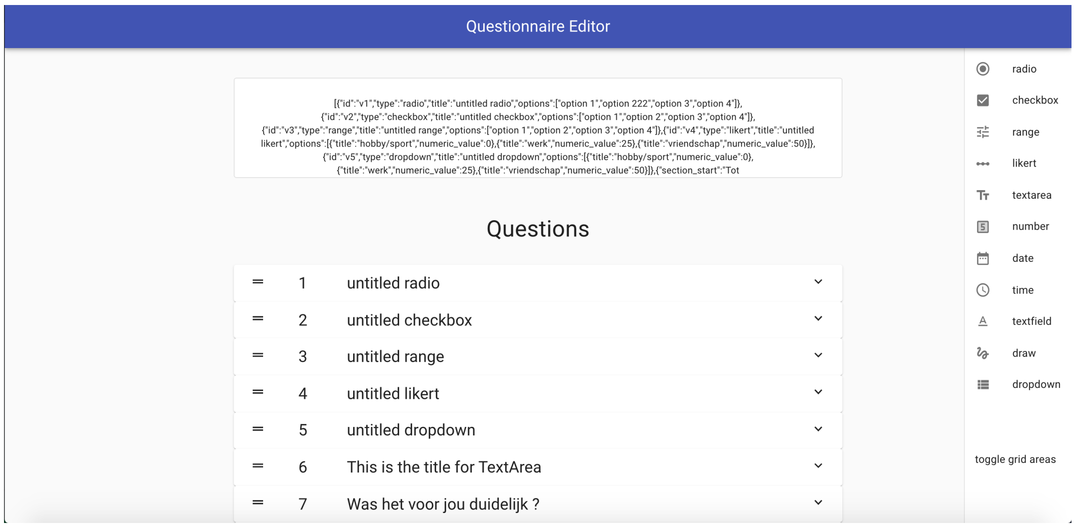
0.8 Render Questionnaire

The application has the render 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

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.



[{"id":"v1","type":"radio","title":"untitled radio","options":["option 1","option 222","option 3","option 4"]},
{"id":"v2","type":"checkbox","title":"untitled checkbox","options":["option 1","option 2","option 3","option 4"]},
{"id":"v3","type":"range","title":"untitled range","options":["option 1","option 2","option 3","option 4"]},{"id":"v4","type":"likert","title":"untitled likert","options":[{"title":"hobby/sport","numeric_value":0},{"title":"werk","numeric_value":25},{"title":"vriendschap","numeric_value":50}]},
{"id":"v5","type":"dropdown","title":"untitled dropdown","options":[{"title":"hobby/sport","numeric_value":0},
{"title":"werk","numeric_value":25},{"title":"vriendschap","numeric_value":50}]},{"section_start":"Tot

=	1	untitled radio
=	2	untitled checkbox
=	3	untitled range
=	4	untitled likert
=	5	untitled dropdown
=	6	This is the title for TextArea
=	7	Was het voor jou duidelijk ?

0.9.1 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.

0.9.2 Toolbar

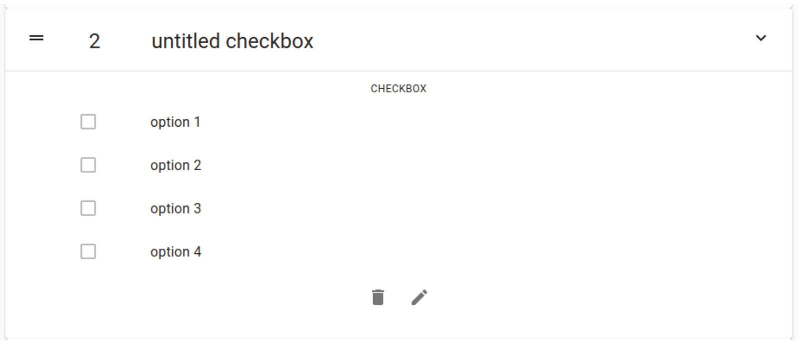
Next we have the toolbar which can be seen on the right of the page. The toolbar contains a list of text and their respective icon for each question type. This Toolbar is also equipped with a drag and drop functionality which enables the user to select which question type he or she want to use and drop 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 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.

0.9.3 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 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

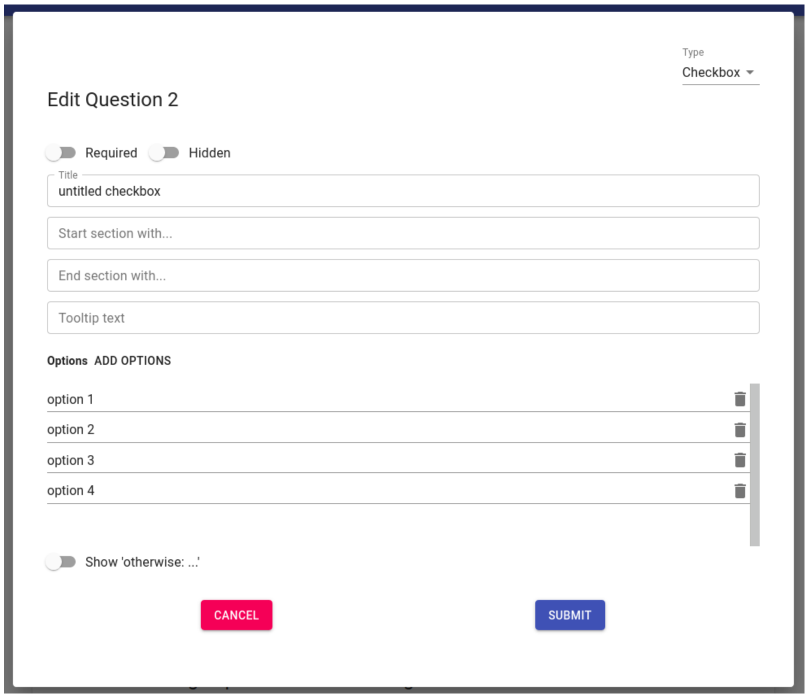
This is the main part of the application. In here we have a collection of all the question types that we have added. This section already has questions inside when the application is initially launched. This was purely made for testing the product, where we could see how each question type would look 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.

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 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.


Type
Checkbox

Edit Question 2

Required Hidden

Title
untitled checkbox

Start section with...

End section with...

Tooltip text

Options ADD OPTIONS

option 1
option 2
option 3
option 4

Show 'otherwise: ...'

As said previously, the only fields which will be affected by this edit is the *Title* field and the *Options* field as the 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.

0.10 Layout of the Landing Page

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.

0.11 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 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.

0.11.1 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.

0.11.2 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 the 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.

0.11.3 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. By redistributing 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).

0.12 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 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.

0.12.1 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 the scrum routine.

The Trello board was initially organised 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.

0.12.2 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.

0.13 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 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.

Build Process and Deployment

0.14 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 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.

0.15 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 prerender 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.

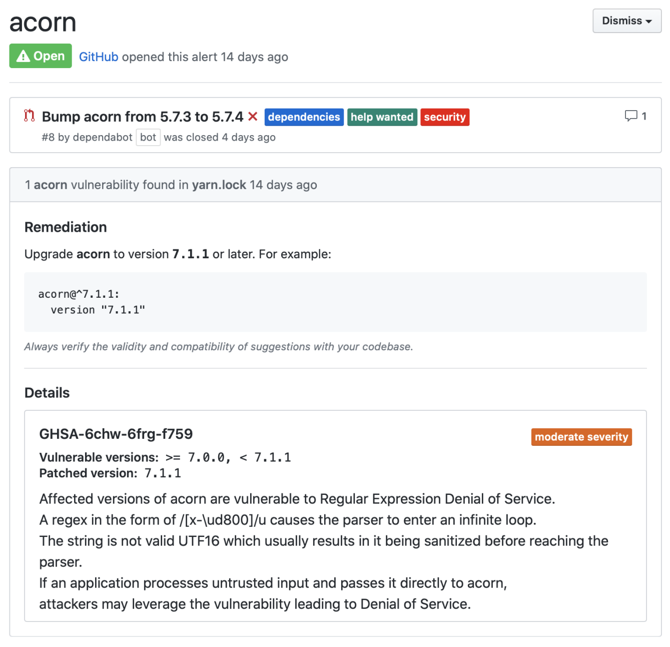
Extra Software Engineering

0.16 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.

0.17 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.



acorn

Dismiss

Open GitHub opened this alert 14 days ago

Bump acorn from 5.7.3 to 5.7.4 ✕ dependencies help wanted security 1

#8 by dependabot bot was closed 4 days ago

1 **acorn** vulnerability found in **yarn.lock** 14 days ago

Remediation

Upgrade **acorn** to version **7.1.1** or later. For example:

```
acorn@^7.1.1:
  version "7.1.1"
```

Always verify the validity and compatibility of suggestions with your codebase.

Details

GHSA-6chw-6frg-f759 moderate severity

Vulnerable versions: `>= 7.0.0, < 7.1.1`
Patched version: `7.1.1`

Affected versions of acorn are vulnerable to Regular Expression Denial of Service.
A regex in the form of /[x-\ud800]/u causes the parser to enter an infinite loop.
The string is not valid UTF16 which usually results in it being sanitized before reaching the parser.
If an application processes untrusted input and passes it directly to acorn,
attackers may leverage the vulnerability leading to Denial of Service.

0.18 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.

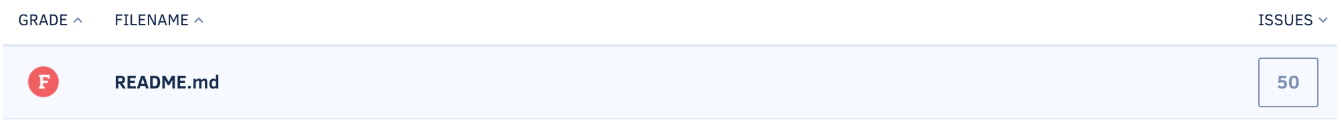
BUNDLEPHOBIA

find the cost of adding a npm package to your bundle

0.19 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



GRADE	FILENAME	ISSUES
F	README.md	50

RESTful API

The full documentation of the 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 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 the client's backend code, and where we will connect the existing code with real data retrieved from the backend.

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