Design Report

TCS Design Project

DUNK

Students

Gerk-Jan Huisma,	s1978373,	g.huisma@student.utwente.nl
Wouter Looijenga,	s2171899,	w.looijenga@student.utwente.nl
Tommy Lin,	s1840932,	t.z.lin@student.utwente.nl
Xue Li Hu,	s2088010,	x.l.hu@student.utwente.nl
Andras Katona,	s2185334,	a.m.katona@student.utwente.nl

Supervisor

Faiza Bukhsh,	f.a.bukhsh@utwente.nl

Contact Person Dunk

Karol Fels,	k.w.fels@student.utwente.nl

Abstract

This report describes the Design Project of Technical Computer Science for Group 6. During the Design Project, the group made designs for and implemented a prototype for Dunk. Dunk is a web application that serves to replace existing survey systems. The report elaborates on all phases of the project. Firstly, the requirements were specified and a global design based on these were made. Afterwards, more detailed designs were made, and these were implemented. The tests, including a plan, were done in between. An evaluation of the Design Project is given, reflecting back on all phases. Finally, a recommendation for future work is made.

Table of Contents

1. Introduction

Dunk is an application that uses a user-centred approach to increase the number of survey respondents and reinvent the process by which organizations collect data that decisions are based on. Surveys are a core part of how organizations make their decisions. However, surveys in general are long and tedious. This makes completing a survey often boring and feel like a waste of time.

Currently, the University of Twente uses EvaSys for their surveys regarding finished courses. However, these surveys are often skipped by students for several reasons. They are very long and tedious, monotonous, as the same questions are always asked. They can only be submitted once the complete survey has been done, which means that information given by students who do not answer all the questions and are not able to submit, is lost from the final data set. Dunk wishes to change that, by asking one question at a time and treating each question as a survey on its own. This approach requires a lesser time commitment and attention span from the users and allows them to give valuable feedback without having to answer all questions of a very long survey.

This report describes the realization of a platform for both survey producers and consumers in the form of a web application. This web application would be a realization of Dunk's core principles and serve as a basis for future development.

2. Requirement specification

Before starting to work on the application, the requirements of the application should be specified and prioritized, so that clear goals and tasks can be made. These requirements were divided into three categories, based on priority:

- ➢ Must have features that must be implemented during the development time
- ➢ Should have features which should be implemented if they took little development time or if there was sufficient time left
- ➢ Could have features that could be implemented if the project was finished very early, but likely would not be implemented and were not considered during planning.

2.1. Must have features

R-1.1 The system must be able to register accounts with different roles
These are the survey producer and consumer roles. As the application will give different functions based on the users' needs, the users should be able to indicate this when they register.

R-1.2 The system must store user profile data (such as login credentials)
Because users must be able to log in, so this information should be stored.

R-1.3 Producers must be able to make surveys for consumers
This is a core functionality, without surveys, there is nothing for consumers to interact with.

R-1.4 The system must store surveys in a database
Surveys must be stored, in order to later serve them to consumers.

R-1.5 Producers must be able to review answered surveys
As the goal for the producer is to get feedback, they must be able to review their survey.

R-1.6 Producers must be able to receive their data in an aggregated and visualized form
Survey makers must get their data in a clear and summarized form, such as a graph, to save time and effort. It should also be possible to download data in an aggregated form,

such that the survey producers can use tools of their choosing to further analyse and visualize the data.

R-1.7 Producers must be able to create branching surveys
Branching surveys allow producers to select different subsequent questions based on answers given by consumers. As some questions will become more relevant based on previous answers, this might give producers more relevant data at the end.

R-1.8 Survey consumers must be able to answer survey questions
This is an absolute necessity, as without answers there is no data.

R-1.9 The system must store consumer answers in a database
Answers are stored to serve as feedback for the producers.

R-1.10 Survey progress of consumers must be saved automatically
To improve user experience, consumers must be able to leave the application and continue answering surveys whenever they want. Every answer is stored in the database.

R-1.11 Producers must be able to invite consumers to a survey through an URL
This allows producers to share the survey with their target group.

R-1.12 The final product must be hosted in production with Amazon Web Services (AWS)
Dunk currently uses AWS to host their website and has decided that it would like to continue using their services. AWS offers, among other cloud solutions, several ways to host servers, databases, file storage solutions, routing and load balancing of server instances.

2.2. Should have features

R-2.1 Producers should be able to invite others to a survey through a QR code.
This allows producers an alternative method of sharing the survey with their target group. QR codes are especially useful in scenarios where survey consumers are reached via non-electronic means such as posters, flyers, letters, etc.

R-2.2 Producers should be able to mark their survey as private
A password is generated for private surveys and only consumers with an URL or QR code, and the correct password, gain access to the survey. This way, a producer can ensure that only consumers with the password can access the survey.

R-2.3 Consumers should be able to answer surveys without creating an account
This improves the privacy situation, as consumers are not forced into making an account and giving their personal data.

R-2.4 Consumers should be able to provide feedback about the survey.
With this, surveys can be further improved upon.

R-2.5 Consent pop up & Terms of Service should be added for doing surveys and registering accounts.
Users should be aware of their privacy rights and what happens with their data.

R-2.6 Accounts should be deletable, including the user data.
Any user should be able to be erased from the database, to implement the right to be forgotten from the GDPR.

2.3. Could have features

R-3.1 The system could keep track of consumers that filled in surveys and reward the user with points based on a reward system.
Consumers could be incentivized into giving as many answers as possible. This can be done by implementing a reward system, which tracks points. The consumer could see their progress, how many surveys were completed, how many questions were answered and how close they were to certain rewards.

R-3.2 The system should be able to spot bots or suspicious behaviour.
This is necessary to stop people from using bots to participate in a large number of surveys to get rewards without actually taking part in them.

R-3.3 The producer could receive automatically generated feedback based on the statistics from the answered surveys.
If enough data is available, models could be used to generate (simple) conclusions that could be given as feedback, so the producers do not have to manually look at the data and spend time on the feedback.

R-3.4 Utilizing machine learning, consumers could have more specific surveys based on previous data that they submitted.
This leads to more relevant surveys given to consumers, which means they do not have to spend time looking for relevant surveys.

R-3.5 The consumer could earn extra rewards for giving feedback.
As producers would receive more valuable information on how to create their surveys in the future, this should be rewarded if it is useful.

R-3.6 Database security and measures against leaks could be taken.
As this project involves a lot of personal data that is stored, correct security measures should be taken to prevent leaking of this data. As this will take more time and expertise, this is left as an optional feature.

3. Global Design

3.1. System description

The application consists of a database and a single server application, divided into three views, a general view, the producer page and the consumer page, which will be shown to the user depending on the role of the user.

The general view consists of the public endpoints, which are used for registering an account, logging in, and guest consumers. These pages are available for users who are not logged in with an account. The guest consumer page will only be accessible with a direct URL given by a producer, as specified in requirement **R-1.11**. This ensures that people do not need to register an account before answering surveys, but they will not be able to answer public surveys and, in the future, participate in the reward system.

Access to the other private views is restricted, to simplify the application for the user. As most consumers will not need the survey creation features, and vice versa for the producers, restricting the other view should not impede the functionality of the application for each user group. The small group that does want to do both features, can simply register a second account.

All views are divided into a front and back-end. The front end will be what the users actually see and interact with. Think of the fields to put in data, drop down menus or buttons. These are used to get the data you want to use from the user.
The back end will handle the connection with the database. This is where the data is stored and read.

The general view includes registering and signing in. The front end has fields for user data, and should also validate during registering to ensure that the user data is valid, such as an actual e-mail address or a password conforming to standard security. The back end only needs to create and read the user data.

For the producers, the front end includes a survey creator, and a page to review the results. The back end will handle creating, reading, updating and deleting surveys and questions, and reading the answers given by consumers.

The consumers, including the guest consumer page, will instead have an answering page for the front-end, where the question will be displayed and possible answers, such as buttons for multiple choice or fields for open questions. The back end should be able to read questions, and create, read, update and delete answers.

Both private views also have access to the user page, where they can read, update and delete their user data.

3.2. Key features

Based on the requirements, the project can already be divided into several main tasks, which can later be divided further during actual implementation. To implement the questions and answers, as well as the separate user views, a clear database should be designed and developed, which also allows for expansion in the future.

Users should be able to create, read, update and delete data from the database. To achieve this, a back-end should be written which allows interaction between the database. As such, a database system should be chosen, as well as a tool to interact with this in a programming language.

With these features, a responsive front end can be created. The front end should be split into different views, as the producer and consumer have different functions. To do this, frameworks and libraries which facilitate responsive front end design and state management should be decided as well.

Lastly, as this is a web application, it should eventually be hosted. A framework to make the application a server should also be chosen. With this, a host should also be decided, to make sure the application works when hosted on a server.

3.3 Preliminary choices

Before starting to work on the project, it is important to think beforehand about what tools the project needs. This could be the programming language used, but also frameworks that can be useful in the application. By thinking about this beforehand, there is a clear idea of what needs to be done.

3.3.1. Programming language

Before starting work on the application, the programming language is an important choice to make. As the application is a high-level application, high level languages are suitable. Initial choices were JavaScript, TypeScript and Java. These were chosen because all members were either familiar with these, or in the case of TypeScript which is a superset of JavaScript, are very similar to familiar languages.

3.3.2. Third party tools

Starting from the top, AWS was chosen to host the project, as it was already being used, and it allows for both database and application hosting.

To actually host the project, Node.js and Express.js were chosen. These are some of the most widely used frameworks for web applications and APIs in JavaScript, and are compatible with TypeScript. They are easy to learn and use, have a wide range of functions, and are well-documented.

For the back-end, as AWS uses Microsoft SQL Server for their Relational Database Service (RDS), a SQL dialect had to be chosen. T-SQL was chosen as this is Microsoft's own dialect of SQL. To accommodate this in the back-end, mssql[1] was chosen. This package allows for interaction with the database, and was made to specifically support Microsoft SQL servers. While none of the members had experience with this package, it had good documentation and a wide range of features, allowing people with little experience to utilise it.

For the front-end, React will be used to build the front-end. Several members were already experienced with this, and it is very versatile. Next.js will be used as well, as it allows for server side rendering on top of React.

3.3.3. Preliminary architecture

As there are separate views, the choice was also made to have different repositories for each main function. These were the authentication, consumer, middleware and producer repositories. This would keep the repositories isolated and structured. To interact with each other, Docker was chosen. Docker allows the separate repositories to be hosted as separate containers that can interact with each other through channels. Docker was chosen as some members had experience with it, is widely used and has good documentation, so other members could also learn it easily.

[1] https://www.npmjs.com/package/mssq

4. Detailed Design

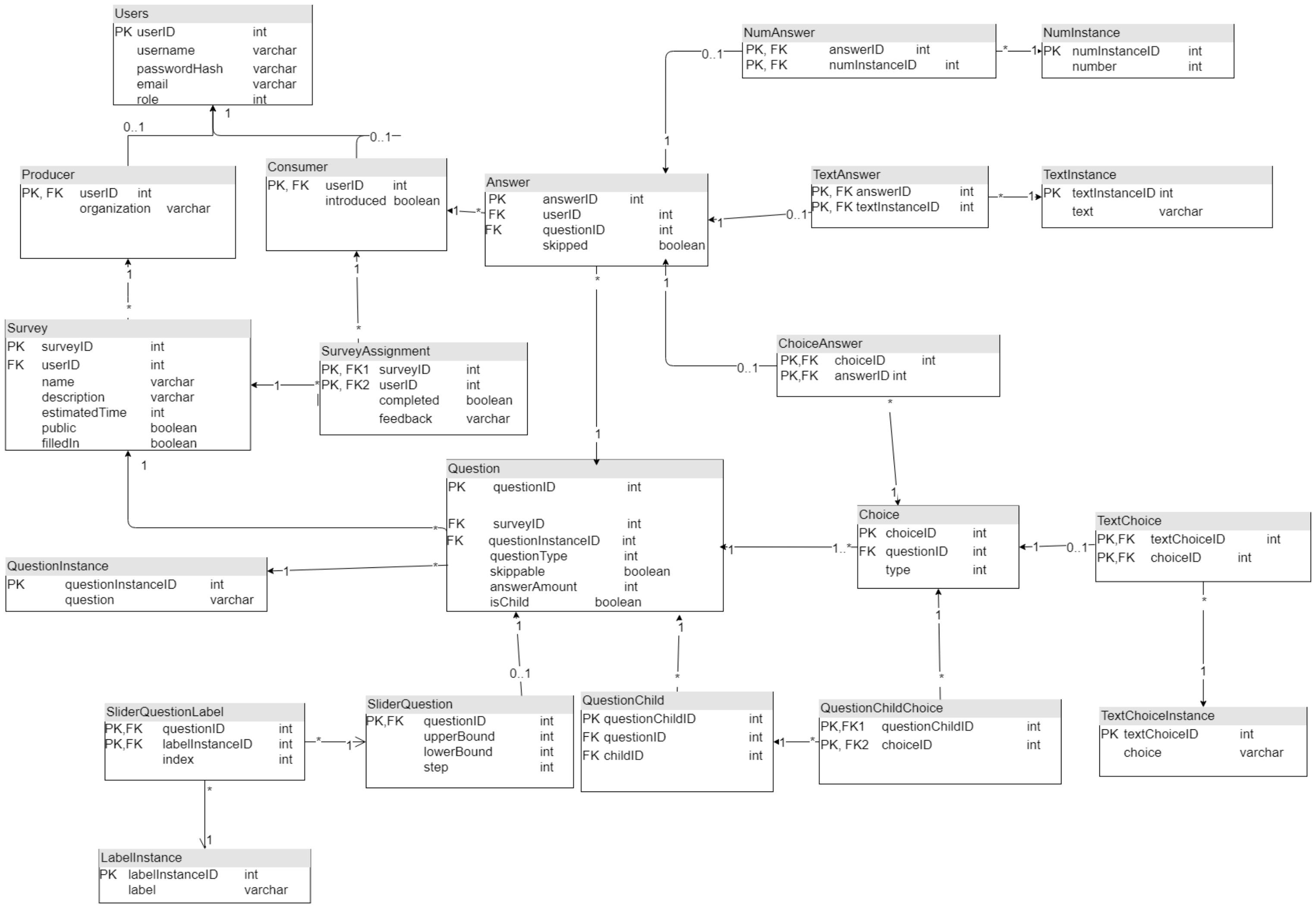


Figure 1: Diagram of the database structure, including data structure

4.1. Back end design choices

4.1.1.API

The API is designed to ensure that it is future proof. For this reason generally the URIs of the API endpoints should refer to some atomic interaction with the system. There should be separate API endpoints which deal with the different types of questions and answers for example. This way we can ensure that in the future, adding more features can be done with low chances of it having a negative effect on the rest of the system. The highest levels of collections of the API can be split into three parts: the producer, consumer and user part. The responsibilities of the producer and consumer parts provide the functionality for each user most importantly creating and answering questions/surveys, and user collection provides authentication, general information and settings. An example URI would be:

```
consumer/closed/questions/123/answer/choiceQuestion
```

The closed collections exist on both the consumer, and producer side. This encodes whether you need to be signed in as the given role to access some resource, the open collection only exists on the consumer side, as this is where we have more extensive functionality for anonymous users.
With the design report there should also be an in depth documentation of the API endpoints. This documentation describes in detail the resources provided by the API as well as, what is the correct format for the input data, what are possible outputs and errors. This documentation should be used by the people continuing the development of the Dunk project, to help them understand the capabilities, usage and design of the system. These specifications of the API do not provide any guidelines for the further extension of the system, but it does provide some specifications for the endpoints required to implement our required features even if it was not implemented in the end for some reason.

4.1.2. Database design

This section describes the design of the database. It elaborates on which design decisions were made. A diagram of the complete database can be seen in Figure 1. For the database design each table will be explained with their data and data types.

In general, the database was designed with the idea that it should be expanded in the future. This allows the client to add functionality which was not needed in the prototype made during the project.

One way to accomplish this was to give all types of questions and answers a separate table. Expansion is facilitated, as a new type of question or answer can easily be added through a new separate table, instead of trying to update a current existing table. Furthermore, the actual text of the questions and answers were made into instances in separate tables. A question would instead refer to the id of the text. This was done to centralize all of this data, which makes it store less data in general. It also allows for easily searching all the different questions and answers. This could be used in future work, for example, for a question bank which displays all current questions and answers.

Starting off with the Users table, this table stores the user credentials (username, passwordHash and email) which are necessary for the user account, and their role, either Producer or Consumer, which is represented as 0 for producer, or 1 for consumer. This table is a generalization of 2 other tables, Producer and Consumer. This is done to differentiate between the 2 user types in the database, so that different views can be given depending on the role.

The Consumer table, as a specialization of Users, also stores whether a specific consumer has been introduced. This is done so that the introduction page will only be shown to consumers that have not yet been introduced. This structure also allows future development to implement consumer specific data, such as the point & reward system, as mentioned in R-3.1.

The Producer table is a specialized table from Users, and currently only holds data about the user's organization. This is currently not used in the application, but merely acts as an example that this table can be used for producer-specific functionality. The organization could be used to automatically assign created surveys to specific consumers, depending on the organization.

The Survey table contains the name of the survey, a description of it, an estimated time of completion, whether the survey can be accessed publicly and if it has been filledIn. The survey references the Producer that created it with the userId. The name, description and estimatedTime would be used to showcase to the user, to give them an idea about the survey. Public would be used to determine whether this survey should only be assigned to people with a survey link, or whether it should be publicly accessible. filledIn would be used to show to the producer if they are still working on the survey.

The table SurveyAssignment is the association between a user to a specific survey. This is done as multiple users could be associated to one table, and one user could also be associated with multiple surveys. This table determines whether a user has fully completed the survey, because then the survey could be hidden from the user, or moved to a list of completed surveys. It also stores general feedback about the survey, so the Producer can view this.

An instance of a question is kept in a table called QuestionInstance. As mentioned before, all of the “Instance” tables keep the actual text strings, to centralize all this data.

The Question table references the surveyId of the survey with which the question is associated, and it has a questionInstanceId as foreign key. This Id links to the actual text of a question, as explained before.
The Question also contains the type of question, for example, a multiple choice or open text question, which are represented as integers in the database. It also stores the answerAmount so the front-end knows how many answers to display, skippable to determine whether a consumer should be allowed to skip this question, and the boolean isChild. This boolean would be used to determine whether this question should only be accessible as a child, and as such, should not show up for a consumer that has not yet done the expected previous question.

The SliderQuestion table is a specialization of a Question, and an example of how this database structure can be expanded to accommodate for different types of questions. It contains data specific to a SliderQuestion, such as the lowerBound, upperBound and step to show in a slider. These would be used to show in the front end, to allow the consumer to interact with the slider to choose their answer.

Each slider question would also have labels, to show what each step in the slider means. These are stored in LabelInstance. The label column simply holds the string to show at a slider step.

The labels are associated with a slider question through the association table SliderQuestionLabel. The index also keeps track of the position of the label, so you can show the correct label at the correct step for a slider question.

The QuestionChild table is necessary for creating surveys which branch out depending on the given answers to some previous questions. This table contains the id of the original question and the question followed by it. By creating this tree-like structure the design also helps in differentiating between questions which are ordered, and need a previously answered question, and the questions that can be taken in any order. Any question id which is not found in this table, is an unordered question and can be done at any time.

The Choice table is used for creating answers to multiple choice questions; it contains the id of the question the option belongs to and the type of the option. The type would serve no purpose in the prototype, as there is only one type of option which is TextOption, but it is there for future proofing the design of the database, in case later the client would like to extend the functionality of the system by adding more types of options.

The TextChoiceInstance stores the text of a choice. The association table TextChoice associates this text instance with a Choice. This association table is necessary to facilitate the possible extension of the system mentioned above. The alternative would have been to do it similar to Question and QuestionInstance, where the Choice and ChoiceInstance table reference each other directly. But in that case there is no method to add different types of Choices later.

The QuestionChildChoice is an association table between the QuestionChild and Choice tables. To make it possible to create a branch for multiple choice questions. This would be done by linking a specific questionChildId, which knows what question has a follow-up question, and a specific Choice from a multiple-choice question. With this, the data necessary to redirect a consumer to the next question based on their choice would be available.

The Answer table contains the userId of the consumer answering, the questionId which links the answer to the question, and a 'skipped' boolean. This is to let a consumer skip a question, if allowed by the producer of such a question, to increase the autonomy of the consumer and allow them to choose what to answer.

Similar to Choices and Questions, the actual instances of the answers are kept in separate tables. This is again for future expansion. This structure allows for new Answer types in the future, without having to fill the Answer table with null values for existing answers. Between each Instance, there is an association table to link the answer with a specific instance.

4.1.3. Database and API responsibilities

This section describes how the tasks to complete a goal of an endpoint, are shared between the API and database. As in what is done with the business logic of our backend and what is done with using the database procedures.

The only responsibility of the database procedures is inserting new data, updating old data, deleting data, and searching for data. Everything else should be provided by the rest of the backend. This means that the backend provides authentication, authorization, and validation of inputs. This means that by the time we execute any endpoints that require authorization, and/or have inputs, we can be certain when executing the database procedures that every input is valid and the user has the rights to use the endpoint connected to that procedure.

Also almost all API endpoints that change the database have their own separate database procedures for interacting with the database, they all use different procedures to make their changes, and all of the procedures are also made to be as atomic as possible to make it possible and easy to further extend the system. This while increasing the number of procedures needed made their development much easier, and also made it easier to work together as any endpoint could be completely developed by one person without much interference with the others.

However by creating atomic procedures, the update and create endpoints which interact with resources across multiple tables can have problems with partial failures, as in when one of the operations fails after a previous one has gone through, these partial failures could have been handled more efficiently in the database with SQL Transactions, but due to a lack of experience and time the decision was to handle these partial failures in the logic of the backend instead of the database. These decisions made the development process much easier for us, and they provide extensibility, for future work on the project.

4.2. Front end design choices

4.2.1. General

React Features

React allows developers to create large web applications that can modify data, without reloading the page. The reason for choosing React over other popular front-end frameworks such as Angular or Vue is twofold. Firstly, React is not a complete framework. This makes React more lightweight and versatile and gives React the benefit of a shallower learning curve compared to the other frameworks, which makes it easier for new learners to adopt. Secondly, several members already had extensive experience creating web applications with React.

Static Site Generation

React and Next.js, which are JavaScript libraries used for building reactive web applications, will be used for realizing the pre-existing UI designs. However, certain advanced features require the use of third-party libraries. Next.js is one of these libraries and it is used to generate static HTML pages from React components at build time or render the components at the server-side on each request. This improves initial page loading times, as React normally is rendered on the client-side, which leads to visible delays, especially on older or slower devices. Furthermore, statically generated assets are significantly smaller in terms of bundle size and can be cached by a CDN with no extra configuration to boost performance.

Typescript

Typescript is a superset of JavaScript and the programming language of choice for developing the project. Typescript adds static type checking, classes and interfaces to JavaScript, making it easier to read and debug the code as well as producing highly reliable JavaScript. Besides, the React library ecosystem already supports type definitions, therefore searching type support for third-party libraries is no longer an issue.

Redux

For managing the state of the web applications, another JavaScript library called Redux will be used to provide a single source of truth for all React components. State management tools are important because they prevent unexpected behaviour and can share data between the components without the system having to constantly provide the data states. This makes debugging easier since this allows for consistency between components. The consistent states allow you to easily view where states get changed and where related issues are. In addition, Redux has a proper compatibility with React, as there exist official React bindings, developed by the Redux team.

Bootstrap & React-Bootstrap

The first advantage of Bootstrap is the fact that this framework is equipped with a responsive grid system. This system takes care of the different views of a website or web application. Another advantage of Bootstrap is the fact that this framework has many standard functions. You can use the framework for formatting simple elements in HTML, but also for formatting tables, buttons, images and forms. All elements you create in Bootstrap are given a professional look. Besides some basic features, Bootstrap also has many advanced features. Creating complex forms and navigation structures, for example, is also very easy within this framework.

React-Bootstrap replaces the ordinary Bootstrap JavaScript version. Each component has been built from scratch as a true React component, without unneeded dependencies like jQuery. React-Bootstrap has evolved and grown alongside React, making it an excellent choice as a foundation for user interfaces.

Sass & CSS

Sass, which is a CSS preprocessor and a superset of CSS, is the preferred way of customizing Bootstrap theming and allows use of several very useful mixins and functions predefined by Bootstrap, for creating reliable and consistent user interfaces. It provides clear files, with readable code, changes in the style of a website can be implemented quickly. In addition, it saves time by omitting complex CSS code, as well as a much better maintainability for future programmers who will work on this project.
Furthermore, the use of CSS modules, which are CSS files in which all class names and animation names are scoped locally by default, works really well with the modular behaviour of React components.

SWR

The name SWR is derived from stale-while-revalidate, a HTTP cache invalidation strategy popularized by HTTP RFC 5861. SWR is a strategy to first return the data from cache (stale), then send the fetch request (revalidate), and finally come with the up-to-date data. It was decided to use the SWR React hook, created and maintained by the same team as Next.js, for retrieving data from the API using GET requests. Under the hood this React hook uses the fetch API that is built into JavaScript by default, however extends this behaviour by implementing the SWR protocol. With SWR, React components will get a stream of data updates constantly and automatically. And the UI will be faster and more reactive.

4.2.2. Consumer Client

Mobile-First Design

Because the primary users of the Dunk application are expected to be mobile users, the consumer client was designed with a mobile-first approach. Using this approach, we looked at the existing UI designs provided by Dunk and started creating the front-end for the smallest mobile screens first, working our way up to larger screens. We decided that we wanted to keep the same look and feel on the desktop screens as on the mobile screens. This is why we chose to stop scaling at a relatively low breakpoint using CSS max-width properties, as opposed to rearranging or replacing parts of the user interface to better fit the needs of desktop users.

Layout

A main part of the consumer client is the global layout component. This component actually implements the mobile-first design and ensures that every page of the consumer client, including both the authenticated and anonymous user view, look exactly the same. The component is controlled using Redux and it is possible to set a background image and color. The card component that is on top of the background and is actually used to display data, has CSS property position set to relative. This allows us, if desired, to position parts of the content of each web page absolutely with respect to this card.

Navigation

Since the main usage of the consumer client is expected to be on mobile screen sizes, we opted for a bottom navigation component that contains routes to several key parts of the consumer client. This bottom navigation component is part of the global layout and when visible, it will be always placed in the same spot in the global layout, to ensure a consistent user interface.
As Redux is used to control the state of the global layout component, it is also possible to use it to show the bottom navigation in the global layout when on an authenticated page or hide the navigation bar on the anonymous survey page. Also, Redux can be used to set which route is currently active.
The consumer client, as discussed in the general section, is built using Next.js, which has a file-system based router built on the concept of pages. The Next.js router allows doing client-side route transitions between pages, similar to a single-page application. This gives an almost native-like experience when transitioning between pages of the producer client.

Add Survey

In order to share surveys created by a producer, it was necessary to implement some form of survey publishing mechanism. As explained in the backend section, we chose to generate a JSON Web Token on the server side, which uses a server side secret to ensure that the survey id that is inside the token is not tampered with. This way we can assure that only people with a valid token can access the accompanying survey. In the producer client, an URL that includes a valid survey token is generated when the publish button is pressed. This URL can be shared to anyone and anyone who accesses it gets directed to the add survey page. On this page, business logic has been put in place to handle the validating of users, associating new or existing users with the survey belonging to the URL or skipping authentication and answering a survey anonymously. See figure 2 for a detailed overview of this logic.

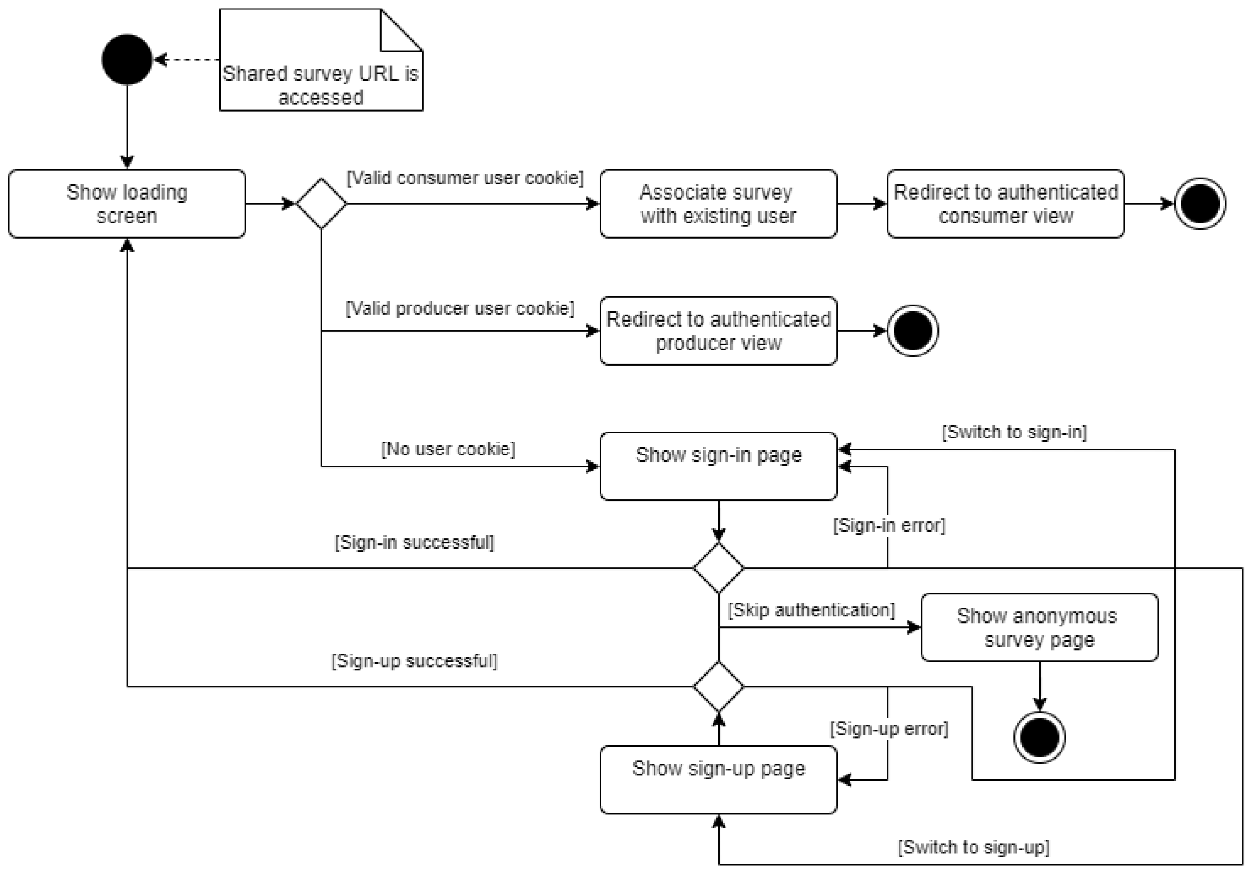

Shared survey URL is accessed
Show loading screen
[Valid consumer user cookie]
Associate survey with existing user
Redirect to authenticated consumer view
[Valid producer user cookie]
Redirect to authenticated producer view
[No user cookie]
Show sign-in page
[Switch to sign-in]
[Sign-in error]
[Sign-in successful]
[Skip authentication]
Show anonymous survey page
[Sign-up successful]
[Sign-up error]
Show sign-up page
[Switch to sign-up]

Figure 2: Activity diagram of the front-end business logic for accessing a new survey

4.2.3. Producer Client

Desktop-Only Design

The primary users of the producer client are most likely going to be desktop users. On desktop it is easier to analyse survey results because of the wide variety of desktop software for data analysis. Furthermore, because of the significantly larger amount of screen real-estate, it was also much easier to design a survey creator with questions preview and extensive questions creation options. In our opinion, we think it should be possible to design a fully responsive survey creator as well, but because of time constraints and expected usage it was not deemed a hard requirement to design a responsive version of the producer client.

Navigation

In terms of navigation, we opted for a sidebar component that contains routes to several key parts of the producer client. This sidebar component is rendered globally and is visible on every page at the same place, to ensure user interface consistency.
The sidebar is actually fully responsive and scaled down well to mobile screen sizes.

Redux is used to control the state of the sidebar component on a global level. For example, if the sidebar is expanded or not and which route is currently active.
The producer client, similar to the consumer client, also uses the Next.js router to allow client-side route transitions between pages, similar to a single-page application.

4.2.4. Home Page

Static Home Page

The static home pages are implemented according to the pre-existing designs provided by Dunk. The home page contains a sign-in button that links to the sign-in page shared by the survey producers and consumers.

Shared Sign-in Page

This page is the main way of authenticating and authorizing survey producers and the only way that existing survey consumers can access their account without associating themselves with a new survey. It was necessary to support this behaviour, because for a survey consumer it should always be possible to access their account in case they want to access, alter or delete personal data, the account or redeem potential rewards. The business logic of the shared sign-in is rather straightforward and only entails one API request to validate user credentials and obtaining potential redirects to other, restricted parts of the Dunk system. See figure 3 for a detailed overview of this logic.

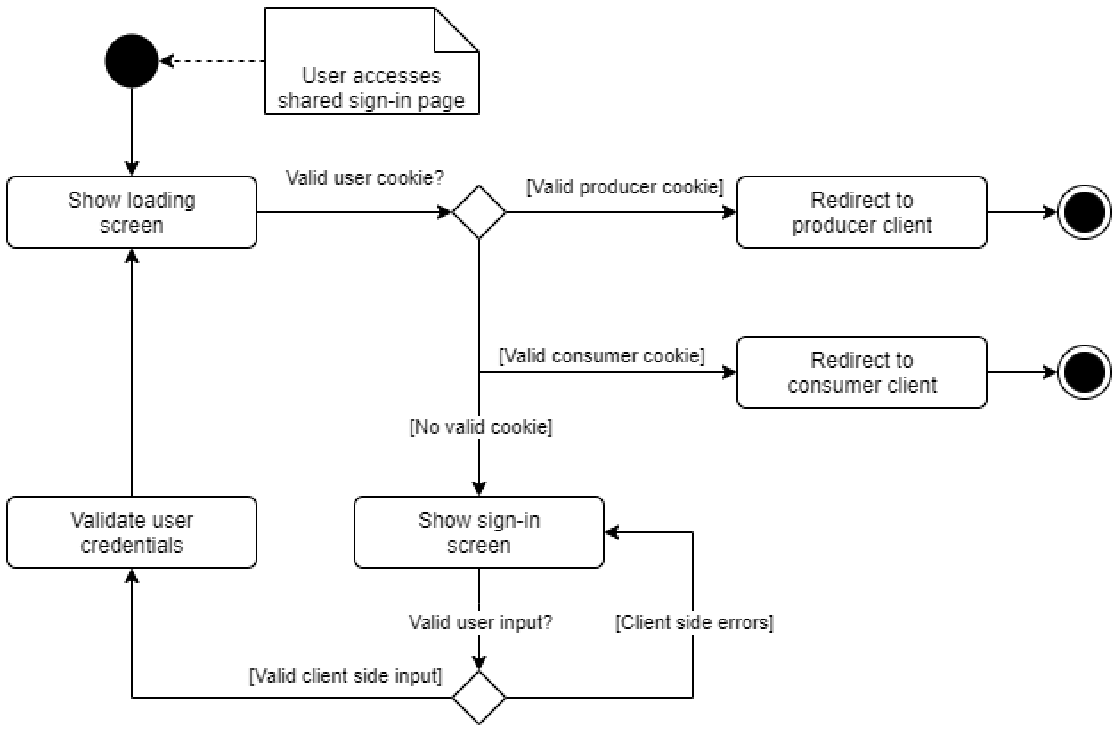


Figure 3: Activity diagram of the shared sign-in page business logic

5. Implementation

5.1. Database Implementation

The database made use of docker. Docker was chosen due to the versatility it provides for running the application on cross platforms for example on Linux and Windows. The decision was due to issues with cross platforms. Since Windows applications do not necessarily run-on Linux systems. Since we wanted to have a cross platform addition. We used docker for this issue. Docker builds from the backend the skeleton of the project. For each operating system Windows or Linux, it builds the skeleton it uses. With this the cross operating system is available.

However, after some discussion we switched to AWS (Amazon Web Services). As the Docker consumed too much memory and start-up time to be feasible within working hours. Therefore, the decision was taken to switch to AWS. Since AWS provided significant lower loading times and less computational work for the laptops. Afterwards the database server was chosen for the interaction of the database. For SQL-servers the following were compared:

- Mssql
- Node-SQL server
- Oracledb
- Sequelize

Mssql was chosen, because of better compatibility with docker. After arduous testing with a local database on the laptops. The decision was made to use the online database of Amazon. Since the computation and load time were significantly reduced by moving to the online platform which reduced the delay for each task the computer needed to finish. This resulted in a more efficient time procedure. Afterwards the data tables were made using SQL queries. For each query the structure of the database was adhered as shown in the schema. For each schema the variables were set up.

Afterwards SQL procedures were made for the most used queries. For example, creating a survey or creating an account. The procedures were made to increase the efficiency of the application. Since by using this, we prevent boilerplate code.

Afterwards we needed to choose the protocol to test the made procedures and connection of the server. There were multiple options, but two were the most prevalent:

- TSQLT
- TsJest

TSQLT tests the back end with the SQL procedures at the core. While TsJest is more used at the front-end where the calls are checked. After weighing down the pros and the cons of these two protocols, TsJest was chosen. Since TsJest can both test the SQL queries and the connection with the client and the server which results in less testing code needed while testing both the back end and the front-end. TSQLT was not used since the AWS needed to be altered to allow the plugin to be used and furthermore it only tests the query and not the connection. Which results in needing to find another protocol to test the protocol.

After weighing down both options, TsJest was used, due to its capability to test both the front-end and back-end and not having to change the structure of the server.
After implementing these mentioned programs and protocols the database was made. There were cascades added to simplify the deleting process. The delete cascades were limited to prevent one possible SQL query to delete all the data. Therefore, specific tables were linked with each other to cascade.

The result was a reduced possibility of complete data deletion of the database. Since the table in lower importance will not delete the table of higher importance or vice versa. This approach was chosen after a long discussion with our supervisor in one of our weekly meetings. As cascades allow for a finer deletion process using SQL queries compared to the alternative. Where each deletion needs to be set up in a format which could become unreadable if delete queries were created from the root. The cascade of deletion can be viewed below in figure 4. In the figure the color represents which cascade is linked to each other. For example, when the consumer chooses to delete his profile. The user needs to be removed from the table.

Therefore, a link was added to the foreign key constraint by delete cascade. This can be viewed by the color as the user table is linked with the producer and consumer with the color blue for visualization. For some of the tables a double color is added. As for the consumer, it has a double color. Namely red and blue. As a survey assignment needs to be removed if the user deletes his/her account. Some of the tables do not require multiple removement, because of data preservation mentioned in our meetings. As data is quite valuable. Therefore, when the consumer deletes his/her account the answer remains, but the userID is set to null.

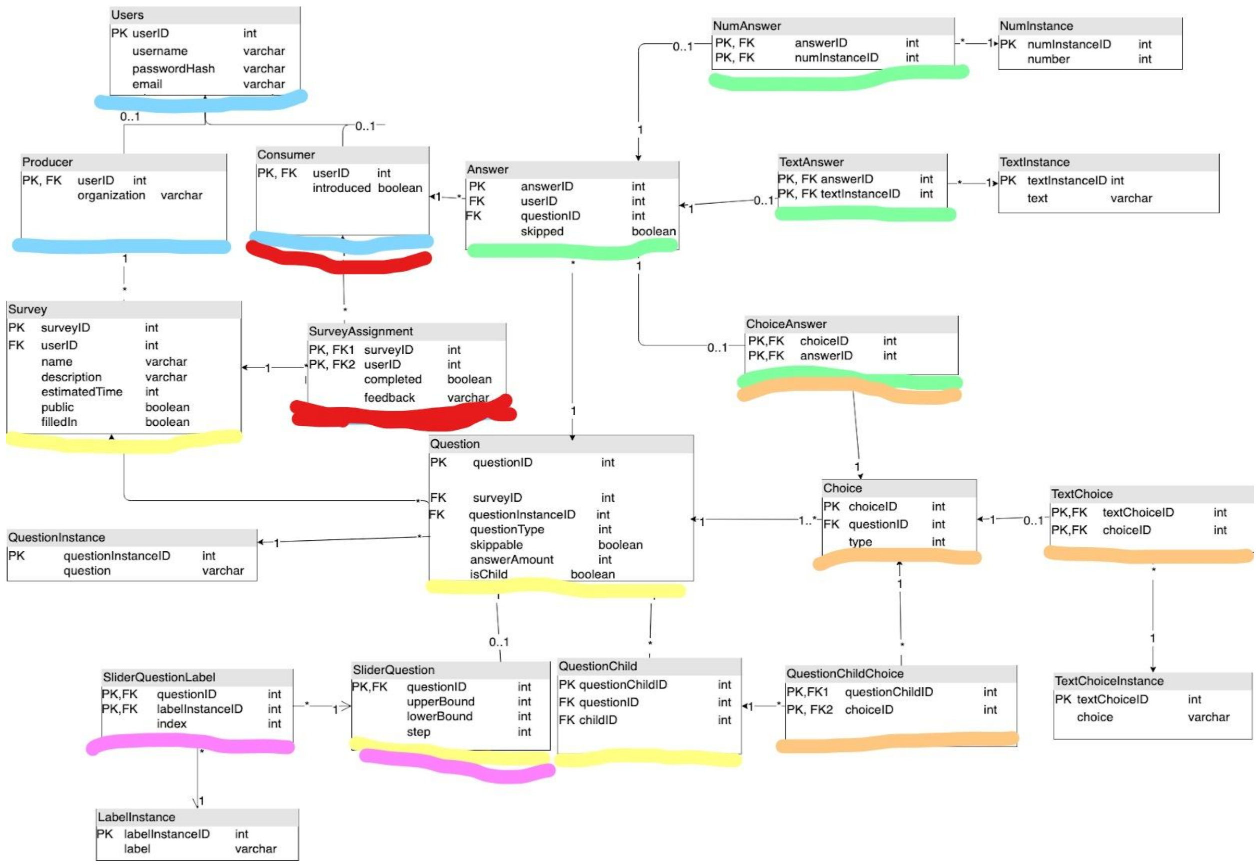


Figure 4: Cascade deletion of the database

Some new database tables were added to the original schema for the slider question. Since the client preferred a more defined slider question in the design, The decision was made to include two tables for the slider question. Slider Question Label and label instance. The tables were added to give the slider question a label and the instance of the label. As the slider question contains a certain type.

5.2. Database Procedures

Procedures were made to increase the efficiency of the platform. The decision was taken after a discussion of possible queries. Since certain actions tend to be performed on the database for example creating an account or answering a question. The SQL query remains the same in each of these iterations with only minor changes of the input values.

Therefore, the decision was made to create a template of the query function. Where the base of the query is made with variables containing the user input. The efficiency is increased by having these stored procedures ready. Which allows for higher writing speeds as the query is partly produced. The database queries were split up in three sections. The producer, consumer, and authorization SQL procedure. The query naming of the procedure is prod for producer, con for consumer and auth for authorization. The naming convention was chosen to differentiate the queries.

As for example a consumer should not be able to create a query or change other users' passwords. The authorization server contains the authorizations, adding and deleting users and surveys and their related questions or answers. The consumer server contains SQL queries for answering and deleting their account. For the producer server, procedures for creating and deleting surveys and questions were made, as well as publishing surveys. There were some discussions about the deletion of data. Since there were quite a lot of foreign key constraints with the deletion. Furthermore, data preservation of the answers was also quite important.

5.3. Back-end implementation

5.3.1. API

The API was implemented using Express.js, which is a back-end web application framework, it has been referred to as the de facto standard Node.js web application framework, and multiple group members had experience using it.
We use Express to create the hierarchical structure of the URI routes. This can be done very easily and intuitively with express leading to maintainable and readable code.
Expres

5.3.2. Authentication/Authorization middleware

Express also offers the middleware feature, which are functions that always run before some set of API endpoints, which you can define using routes, and they have access to the same data provided in the request. This is very useful when defining things such as authentication/authorization, as some endpoints need to run it while others do not or they only need some variation on it.

The auth middleware is made up of two functions one for authentication of the user cookie provided on login, and the other is for authorizing the user.
The authentication function takes the request, and uses the JsonWebToken library to verify that the user cookie contained in the request is valid, and was created by dunk. In this case the middleware adds the user object to the request object so that any subsequent middlewares have access to the user object.

Now if we assign the previous middleware to a route, every middleware that was assigned to that route after the previous middleware will have access to the user object on the request object.
The authorization middleware takes a role which is either consumer or producer, and if there is a user object, it checks whether they have the right role. The user gets redirected in case they have the wrong role, or are not signed in.

5.3.5. Database connection

The backend connects to the database server using node-mssql, a Microsoft SQL client for Node.js. Using this connection the server contains utility functions to invoke each procedure that is used by the endpoints. There is an environment variable specifying the schema when a procedure is called this way multiple databases can be created for different purposes such as testing, production or development without them interfering with each other.

5.4. Front end implementation

5.4.1. Consumer Client

General

The consumer front-end was designed for mobile usage first. The general idea is that most people will be answering on their phone. However, the design is fully responsive and also looks nice on larger screen sizes. We created a React component that can be reused and rendered across all pages that comprise the consumer front-end. This ensures a consistent UI across the entire client and should speed up developing the front end in the future. These reusable components are used on both the authenticated and the anonymous view on the consumer side.

Authenticated View

When the consumer chooses to log in, they will experience the authenticated view. Meaning that there is a navigation bar at the bottom of the screen with several tabs This allows you to easily switch between pages on the smartphone and is touchscreen-friendly. This navigation bar is one of our reusable components. On this bar you can switch between a profile information page, survey page and a reward page.

The profile page contains information about the consumer and contains the logout button. The survey page is for answering questions. Depending on the type of question you have to fill in your answer in a certain way. In addition, a skip button has been added to skip a question if it prefers not to be answered by the consumer. Lastly, the reward page is intended to add a reward system in the future of this project, but has no functionality at the moment.

Anonymous View

If the consumer has chosen to complete the survey anonymously, the consumer will then experience the anonymous view. The anonymous version shares various similarities with the view for authenticated users.However, the reusable Redux component for the navigation bar is removed for anonymous consumers. That makes sense because the consumer prefers not to reveal his data and does not need a profile page. Moreover, this ensures that the reward page is redundant, since the answers and the questions are no longer linked to a consumer. In this view, the user can only see the survey page where questions can be answered and skipped.

5.4.2. Producer Client

General

The producer side is made for PC users only. Although it is possible to do it on the mobile phone, making surveys is easier on the computer. On the PC there is a better overview and pages are all made scrollable. Many design choices have to be changed to make a proper mobile version, since the current sidebar and question creator are not suitable for a mobile display yet. In the end we opted for a sidebar without a top navigation bar. This provides a better overview and a neater appearance, otherwise the top-bar would be empty and take up unnecessary space.

Surveys Overview

The goal is on one hand to provide a familiar experience and functionality to the producers creating the surveys. While on the other hand providing a novel system of managing data, questions and providing surveys to consumers all on a single platform. To realise this, a table has been chosen to display all surveys made by the producer. In this table it is possible to create, update, download and delete the surveys by selecting and clicking on icons and buttons in the table. The producer is able to add new surveys by means of a plus-button positioned in the top-left corner. In the modal that opens, the producer is required to give the survey a name and description.

Survey Creator

By clicking the edit button in the survey overview, the producer will be redirected to the survey creator page. On this page we have opted for a topbar, toolbar and a scrollable side-page. The top bar is used for adding a question to the toolbar and for publishing the survey. Moreover, the topbar is used to monitor when the survey was last edited.

In the toolbar it is possible to add a question, choose the type of question, enter the question and create labels for questions. In the current state of the project, the producer can, by means of a dropdown, choose from four different types of questions: text, number, slider and multiple choice questions; After a type has been chosen, it is possible to enter the question itself in the field below.

An overview of the added questions can be found on the scrollable page next to the toolbar and besides the overview of the questions it is possible to delete them with a delete survey button. In this overview you can scroll up and down to edit different questions after they have been added. The toolbar shows the correct information based on which question you are viewing in the overview. For each type of question it is possible to enter the answer in the overview on the scrollable page. After the survey has been created, the producer can publish it by clicking the publish button in the top bar. This creates a URL that the producer can share with consumers. In the future of this project, the producer can also share the survey by means of a QR code.

5.4.3. Home Page

Static Home Page

In addition to the consumer and producer side, there is a general home screen and introduction pages as well. It has been decided to place the signin button on the home screen. The already existing home page is implemented in the current project and the sign-in button was added to it in the top right corner. This is intended for the producers to log in because in most cases consumers are led to dunk via a survey link. Finally, the home page provides general information about Dunk and why it is different from existing survey tools.

Shared Sign-in Page

Finally, a shared sign-in page is implemented, including buttons for sign in with Google and sign-in with Apple. The general idea is that the producer and consumer both log into this page. However, consumers can also choose to complete a survey anonymously. In that case, the consumer does not end up on the sign-in page and will be redirected to the survey directly. The producers and consumers will return to the home page again when they decide to logout.

5.5. Hosting in Production

5.5.1. NGINX

For the first version of Dunk, preferably it should be possible to run the entire system on one machine or one AWS virtual server instance. To meet this requirement, NGINX is used as a web server for all of the front-end clients. At the same time, NGINX also functions as a reverse proxy for API requests from the front-end clients to the Node.js process in which the API runs. This process can be running on another port (locally) or on another machine altogether. So in later stages it would be possible to split up the web server and API over different AWS virtual server instances and use NGINX as a load balancer as well to distribute traffic over different virtual server instances running the Node.js API.
The reason NGINX was picked over, for example, Apache web server, was because NGINX outperforms Apache when it comes to static content processing and high traffic levels. Furthermore, it can also function as a reverse proxy and has load balancing features.

5.5.2. Amazon Web Services (AWS)

There is another possibility for hosting the Dunk system, that actually does not involve NGINX, at least not as a load balancer or reverse proxy. AWS offers several services that can replace and most likely outperform a manual setup with NGINX. For the initial testing phase, before the first funding round of the Dunk start up, this would most likely turn out too costly, however later on this setup might be preferred over the NGINX setup described in the previous section.

AWS offers CloudFront, which is a global Content Delivery Network (CDN). A CDN is a geographically distributed group of servers which work together to provide fast delivery of Internet content and can be used to help cache content at the network edge, which improves website performance.

In our case, we could use CloudFront to deliver the statically generated front-end clients over HTTPS, thereby ensuring a low time to first byte and fast, responsive user experience all over the world. Furthermore, CloudFront also has options for adding a reverse proxy between the distributed contents and other server instances, such as the Node.js API, running on the AWS infrastructure.

6. Testing

This section describes the testing phase of the project. This includes the overall test plan, which details the planning and choices made before actually writing the tests, and then the test results are discussed to determine whether the tests are sufficient and what conclusions can be drawn from these.

The overall results were that the unit tests completely passed, but for the integrated tests some failed; the cause of this is known, and is mostly related to the testing structure. This will be described more in detail in its separate section.

6.1. Test plan

This section describes the overall test plan before actually writing the tests for the application.
This includes the depth of testing, tools to be used for testing, what the rough idea is to get from the tests, when a test should pass and when a test is expected to fail, a description of both the unit tests of the used procedures, and the integrated tests of API endpoints.

To accomplish this, it was expected that a separate schema in the database would be sufficient to fully isolate the test data and get no interference from the regular data.

6.1.1. Scope

The goal of the tests is to be as thorough and complete as possible, to determine what the exact functionality is compared to what is expected. This should also include failing tests to determine whether the error handling works as expected.

To do that, both the individual procedures should be unit tested, but also how the system as a whole interacts, in the integration tests of the endpoints.

For each of the tests, failing tests should also be written in case it is possible for the tested function to fail. This is described more in depth in “Scenarios”.

6.1.2. Tools

Jest[2] was chosen as the testing framework for the project. Firstly, it works with TypeScript, the chosen language to write the application in. Furthermore, it is a lightweight framework that is fast because of how it organizes tests, by first running

[2] https://jestjs.io/

previously failed tests. While it works without any additional configuration after installing, it has a very wide range of features that can be enabled and used, such as configuration files, Setup and Teardown, and specific assertion types. It is also compatible with other frameworks used in the application, such as React and NodeJS. It could also be used for the endpoint testing, due to it mocking endpoint data with test agents.
Lastly, since none of the members had specific experience with a testing framework, Jest was also useful as it has extensive documentation, wide usage and as such a lot of assistance could be found online.

Another tool used to ensure proper testing was Supertest[3], which is used to mock endpoint and server data. This is necessary for the integration testing, as a method to mock the server is necessary to send the HTTP request to.

6.1.3. Scenarios

The main scenario, that should always be tested, is the normal scenario, in which a user does not attempt to change data structures. However, it can not be assumed that this is always the case.
Checks were made to ensure that the sent data structure matches the expected data. These would also be checked. An example of this could be when a user tries to delete a survey, which that user is not the owner of. The expected response should be that nothing is deleted, as it should be checked whether the user is the owner of the specified survey.

6.1.4. Metrics

The assumption was made that all test suites should pass without any failures. Any failures that do show, imply that the functionality does not work as expected. Tests which expect the function to fail, and which fail in the expected manner, should pass in the test suite.

6.1.5. Unit testing

The procedure files would all be tested with unit testing. The unit tests always include a test with correct data, expecting a correct result. If a function requires an id to use, such as to make a survey, a test was also done with an invalid Id; to ensure consistency, the Id 0 was used for the failure tests.

[3] https://www.npmjs.com/package/supertest

For the CREATE procedures, the expectation is that the procedure will return either the Id of the created object or 1, simply as a status code. The expectation of the failing tests would be that the value 0 is returned.

For the READ procedures, the expectation is that the procedure will return either an Id if a specific Id is requested, or an array if data is requested. An incorrect test would return either 0 for an Id, or an empty array for an incorrect input.

For the UPDATE procedures, the procedure itself is expected to return a 1 on success, or 0 on failure. Besides using the return code to determine whether the test was successful, the updated data was also selected from the database, to ensure the data was changed correctly. This was done with a separate SQL query, not one of the procedures, to ensure that the result is independent of any other procedure.

For the DELETE procedures, where besides the return code being tested, a similar thing to the UPDATE procedure was done, where a separate SQL query was used to determine whether the data was deleted correctly.

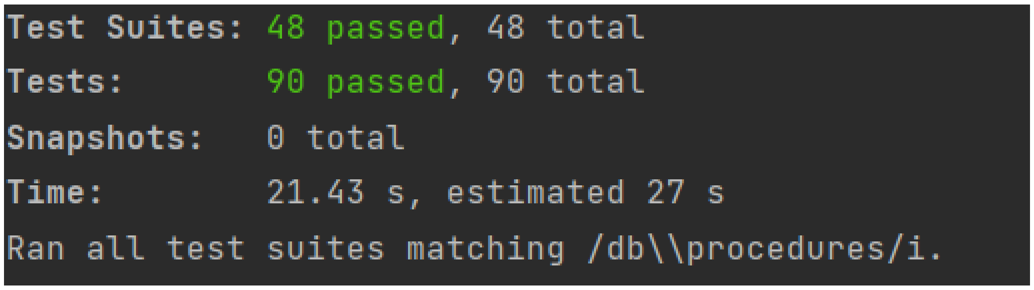
6.1.6. Integration testing

For the integration tests, as mentioned in Tools, Supertest was used to mock server endpoints. Similar tests to the unit tests were done, where every correct functionality was done, with a corresponding correct result. Failure tests were done by attempting to access the resource without the correct cookie, which stores the user Id. This was used to determine whether people could access data that did not belong to them.

6.2. Test results

The unit tests ran as expected. As mentioned above, the failure tests would also run as a pass in case the failure is expected. As such, the tests all run according to expected behaviour. All in all, there were 48 different procedures tested, with a total of 90 tests for these, as can be seen in figure 5]. This is because for some procedures, there were no edge cases that would be checked at the procedure level. For example, the DELETE procedures which do not utilise an id to be deleted, did not have a failure test. This is because the validation whether this delete is valid is done on the endpoint level, as it requires matching the user with the object to be deleted, which is separately done in the endpoint in this application.

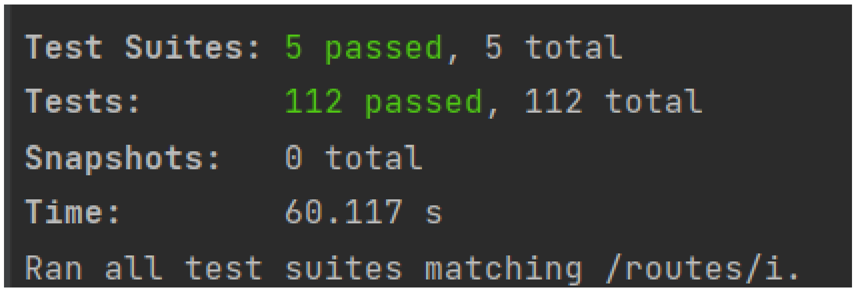
Coverage tests were also used, however, not much was done with the result of these tests, due to time constraints. Future work could expand on the tests, to ensure coverage would improve, or find out whether there is unnecessary code which is not utilised. In general, the coverage was usually higher than 80%, which we thought to be sufficient for a minimum viable product. The coverage can be found in Appendix Section 1.



```
Test Suites: 48 passed, 48 total
Tests:       90 passed, 90 total
Snapshots:   0 total
Time:        21.43 s, estimated 27 s
Ran all test suites matching /db\\procedures/i.
```

Figure 5: Result of the unit tests

For the integration tests, all tests passed as well, as can be seen in figure 6
The integration tests did have some issues during development. Each test suite would also try to start the actual server, as such multiple servers were attempted to be hosted on the same port. This gave an issue, but because the integration tests use a mock server, this did not interfere with the actual tests. The Express server had to be restructured to accommodate for the tests to work perfectly and pass.



```
Test Suites: 5 passed, 5 total
Tests:       112 passed, 112 total
Snapshots:   0 total
Time:        60.117 s
Ran all test suites matching /routes/i.
```

Figure 6: Results of the integration tests

The coverage of the integration tests are also shown in Appendix Section 1. For some files, the coverage is very low, or even 0 sometimes. This can be explained because the endpoint tests also use the procedure files. The edge cases for the procedures are not used, either because these tests were not interesting for integration tests, or because validation happening at the endpoint causes these branches to not be used in the procedure files.

7. Evaluation

7.1. Planning

Some alterations were made from the planning as certain tasks took longer than expected or needed some refinements. Certain tasks were done earlier than others and some were delayed. The following tasks were created the green ones were completed in time, the other ones in red had some delays:

- Database design
- Authentication and authorization server
- Database development and connectivity
- API development
- Consumer frontend
- Producer frontend
- Design report
- Testing

The database design was finished in time. The authentication and authentication server were delayed due to the database design needing to be finalized. The producer and consumer front-end were finished one week later than planned. As the initial design and outward appearances were debated for quite some time. The authentication and authorization server and API development took more time than expected. As the database connectivity and the connection with the use of the API development should have been finished before the authentication and authorization server. Which resulted in a delay for the server. For the API development, the connection with the front-end and back-end was delayed by a week. Due to delays of the front-end with the design and the back end. As the connection could only be finished when both parts were finished.

Overall, the planning could have been improved by starting some of the parts earlier for example the database connectivity and the API development. The design report and testing were finished within the allotted time slots. Overall, after reviewing our progress we are quite delighted. We were able to implement most of the required functionalities.

7.2. Responsibilities

The responsibilities for each part of the project implementation were subdivided in back-end and front-end.The team was split up to take care of the two mentioned divisions of the project.

The back-end group:

- Tommy Lin
- Andras Katona
- Xue Li Hu

The front-end:

- Wouter Looijenga

Front-end, back-end and development operations:

- Gerk-Jan Huisma

The subdivisions worked quite well as by splitting the project in front-end and back-end each person was able to perform their task more proficiently as the tasks were more aligned with their respective domain. There were some more subdivisions made as each part of the sub division had variable completion time. For the back-end the following task division was made:

Tommy Lin:

- Database design
- Procedures for producer, consumer & user account
- Worked mostly on create & read endpoints for producer
- Database & API testing, focusing mostly on failure testing

Andras Katona:

- Database design, setup, procedures(ones necessary for endpoints), testing
- Database and API testing
- Authentication
- API implementation: most of the PUT endpoints, getting questions of survey as consumer and as producer, both open and closed for consumer, getting the link to a survey as producer

Xue Li Hu:

- Database design
- SQL procedures namely deleting(producer and consumer) and answering questions(consumer)
- Cascade implementation for the delete
- API endpoints for the consumer(namely answering and skipping questions)

Gerk-Jan Huisma:

- Producer and consumer front-end

- API design + documentation
- Authentication middleware
- Development setup (Docker, Next.js development servers, selecting and setting up test runner)
- Hosting (NGINX, AWS services, Database)

Wouter Looijenga:
- Consumer and producer front-end
- Introduction pages
- Front-end Testing

The parts where everyone worked together were the proposal report, final design report, the presentations and the initial choice of the project. Overall the responsibilities were more or less equally divided between each member of the group. However, due to the changes of the general structure during the project weeks and experience of other team members. Some team members received more workload compared to others. This was decided after careful discussion and if the team member in question needed aid the other team members aided that person as best as possible.

7.3. Requirements

The following requirements can be viewed in the Appendix section 2. The requirements in green have been implemented while the rest is not yet implemented. Due to the time constraints mentioned before. Not every part of the functional requirements were completed. Most of the functionalities were implemented for a working application. In future works these requirements can be implemented in order to improve the Dunk product.

7.4. Final result

In the end, a minimum viable product of Dunk was made. While not all the requirements were implemented as initially expected, the most important features were implemented. As mentioned above in “Requirements”, these were not implemented due to time constraints.
The current product implements the user creation and signing in.
The producer pages are functional, and can make surveys and questions, but are missing the branching questions and feedback on questions and surveys.
The consumer system implemented all the required features, which allows them to answer questions. Some low priority features were finished for consumers, such as anonymous answering.

Authentication has been done to ensure that users are always on the correct page, and do not have access to features that should not be accessible.

7.5. Conclusion

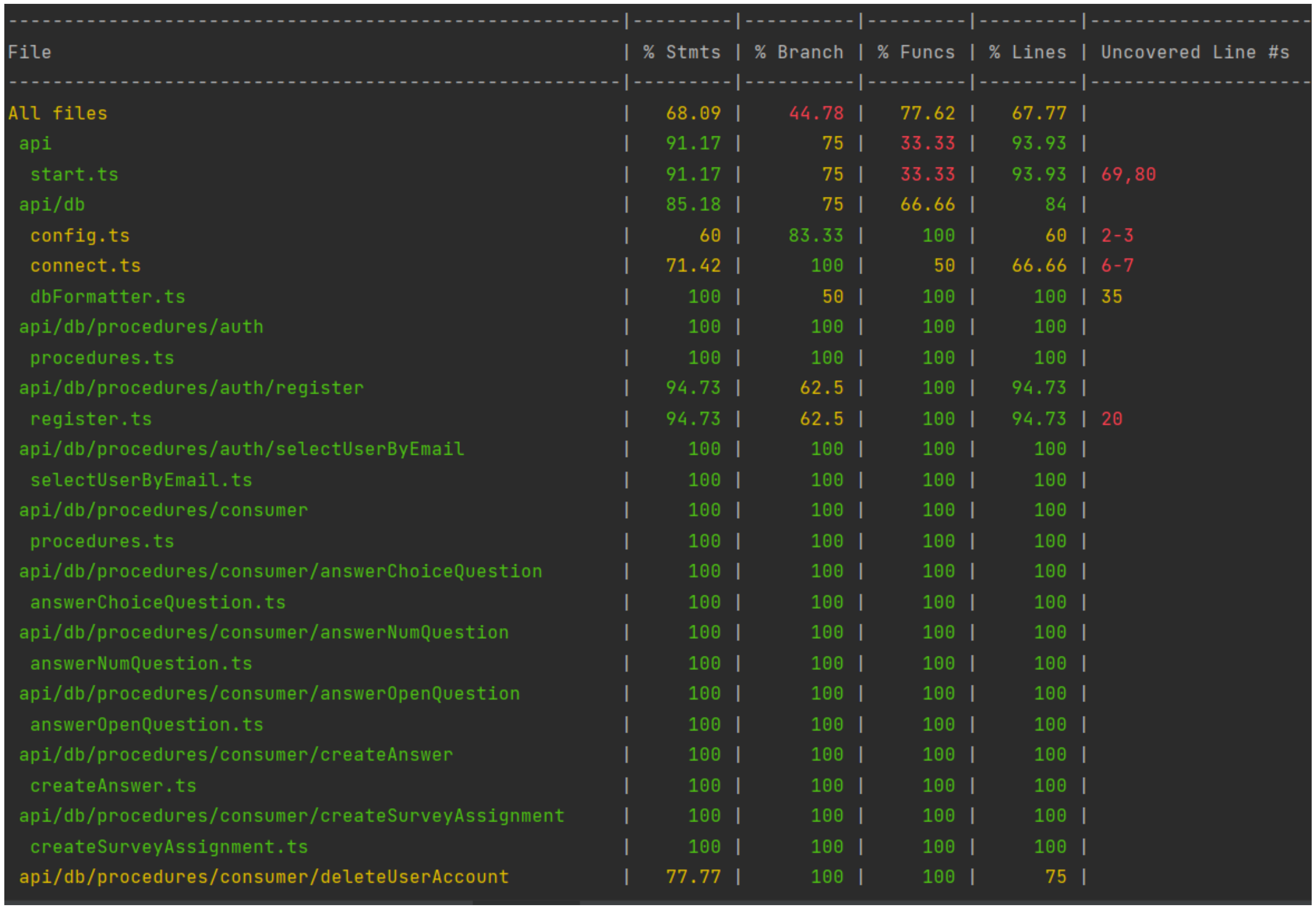
The goal of this project was to create the designs for and implement Dunk, a new survey system. We can consider this a successful project, as the design and implementation are functional, and do not contain any major issues.

However, several of the requirements were not implemented. This has also been discussed with the client, who was satisfied with the current result. Future development is necessary to realize these missing features.

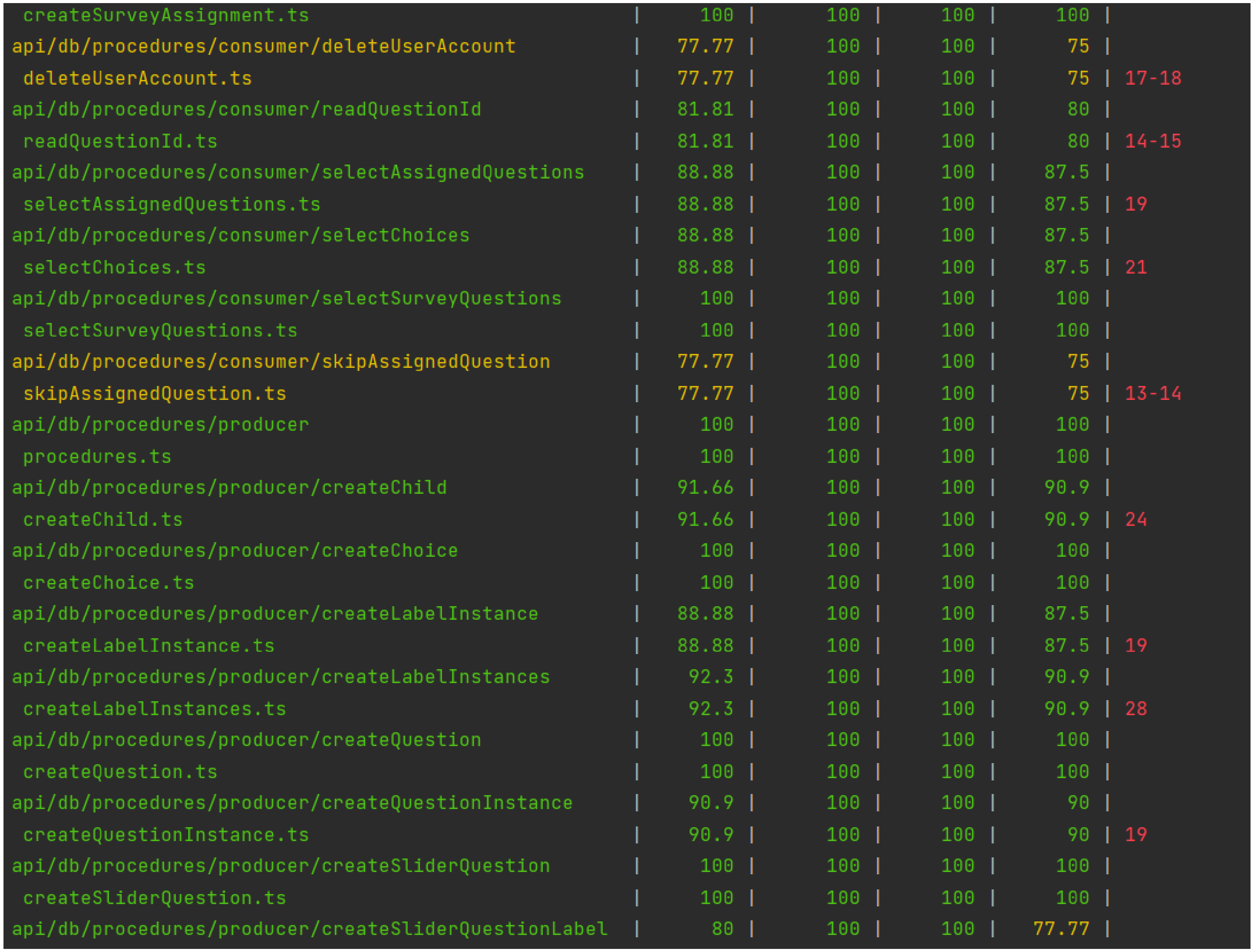
The project allowed the team to design and implement a large web application. This provided the team with new experience, and the group has learnt a lot. Not just in development, but also in requirement analysis, planning, documentation and communication.

Appendix

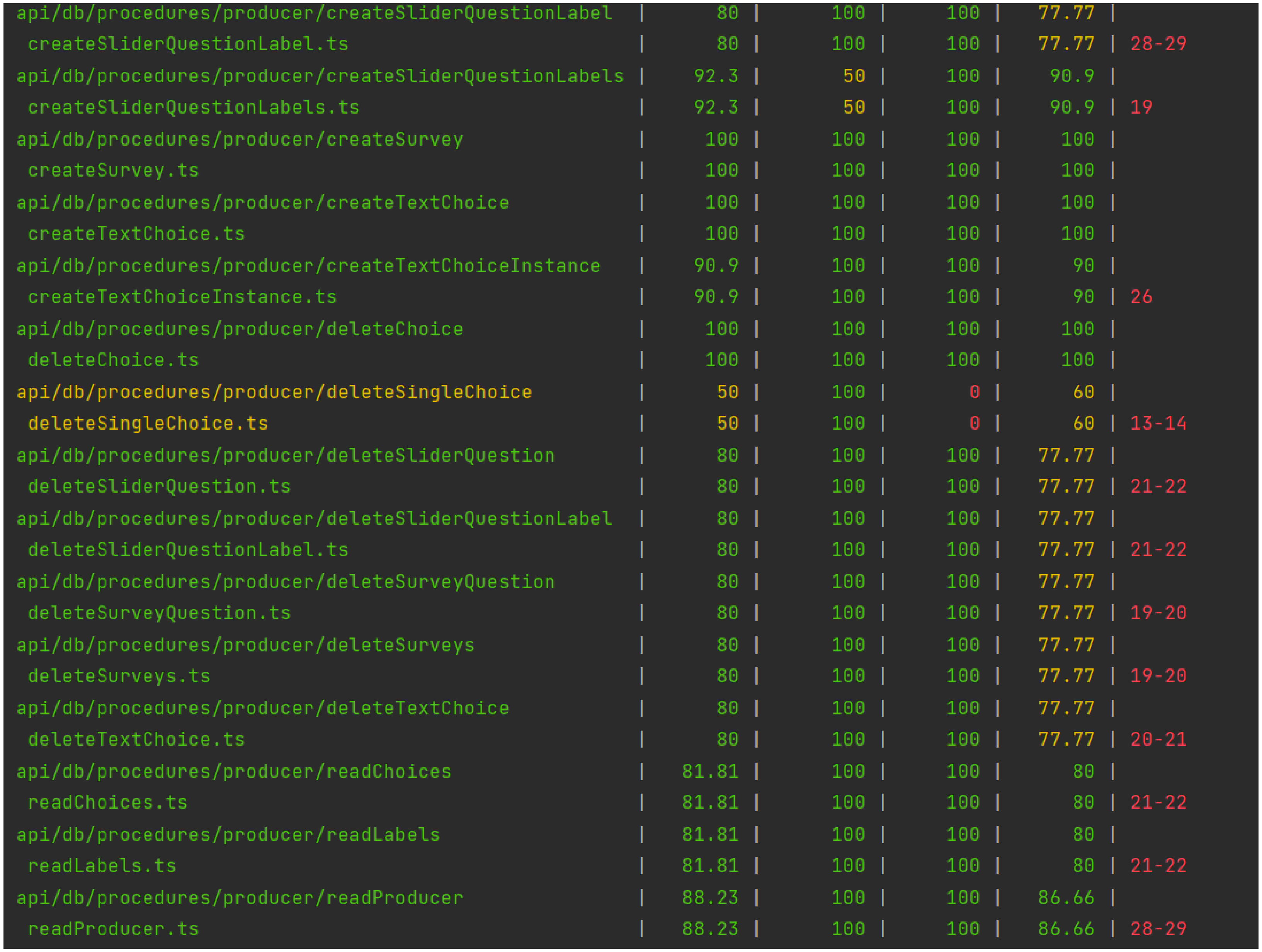
1. Code coverage



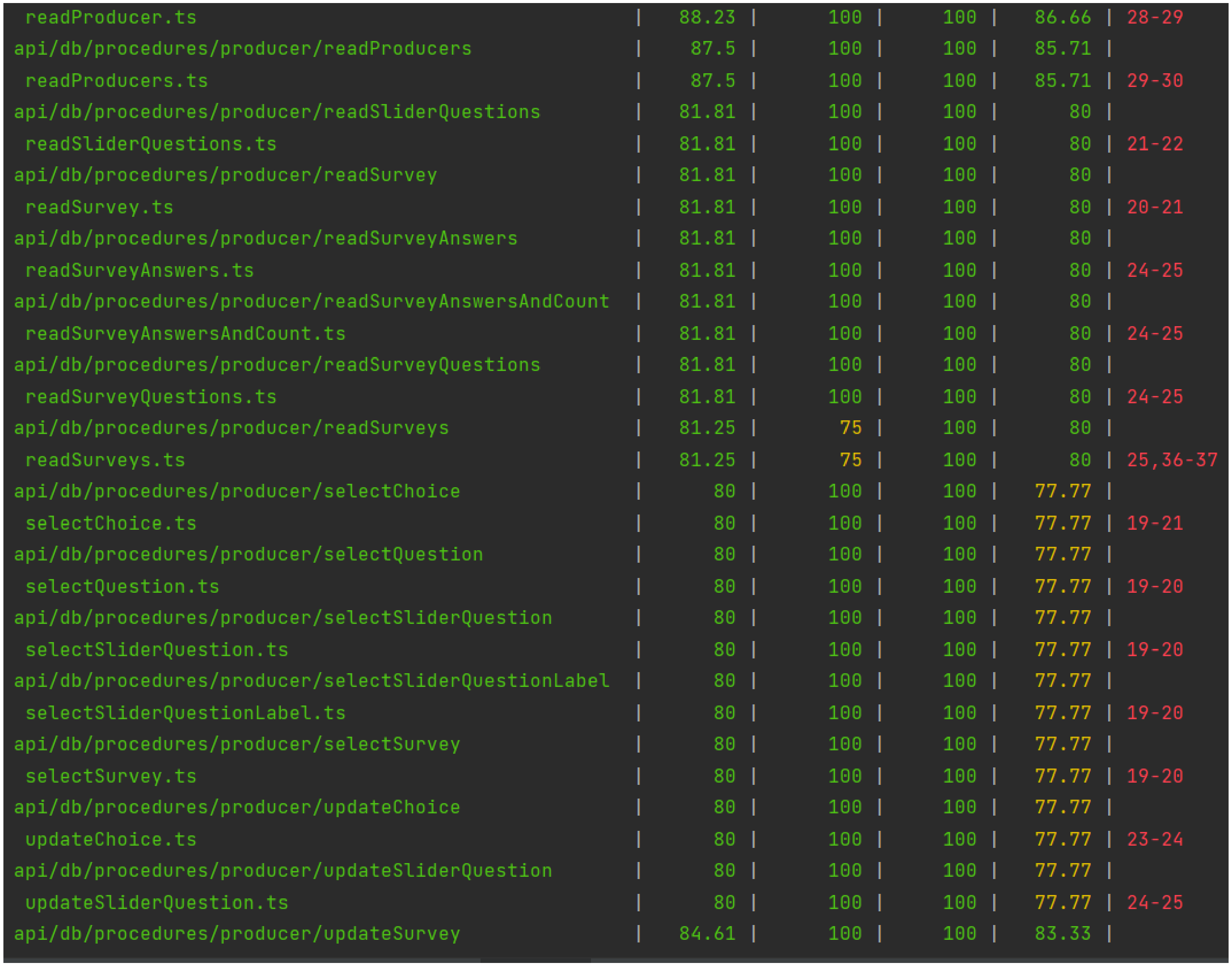
```
-------------------------------------------------------|---------|----------|---------|---------|-------------------
File                                                   | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
-------------------------------------------------------|---------|----------|---------|---------|-------------------
All files                                              |   68.09 |    44.78 |   77.62 |   67.77 |
 api                                                   |   91.17 |       75 |   33.33 |   93.93 |
  start.ts                                             |   91.17 |       75 |   33.33 |   93.93 | 69,80
 api/db                                                |   85.18 |       75 |   66.66 |      84 |
  config.ts                                            |      60 |    83.33 |     100 |      60 | 2-3
  connect.ts                                           |   71.42 |      100 |      50 |   66.66 | 6-7
  dbFormatter.ts                                       |     100 |       50 |     100 |     100 | 35
 api/db/procedures/auth                                |     100 |      100 |     100 |     100 |
  procedures.ts                                        |     100 |      100 |     100 |     100 |
 api/db/procedures/auth/register                       |   94.73 |     62.5 |     100 |   94.73 |
  register.ts                                          |   94.73 |     62.5 |     100 |   94.73 | 20
 api/db/procedures/auth/selectUserByEmail              |     100 |      100 |     100 |     100 |
  selectUserByEmail.ts                                 |     100 |      100 |     100 |     100 |
 api/db/procedures/consumer                            |     100 |      100 |     100 |     100 |
  procedures.ts                                        |     100 |      100 |     100 |     100 |
 api/db/procedures/consumer/answerChoiceQuestion       |     100 |      100 |     100 |     100 |
  answerChoiceQuestion.ts                              |     100 |      100 |     100 |     100 |
 api/db/procedures/consumer/answerNumQuestion          |     100 |      100 |     100 |     100 |
  answerNumQuestion.ts                                 |     100 |      100 |     100 |     100 |
 api/db/procedures/consumer/answerOpenQuestion         |     100 |      100 |     100 |     100 |
  answerOpenQuestion.ts                                |     100 |      100 |     100 |     100 |
 api/db/procedures/consumer/createAnswer               |     100 |      100 |     100 |     100 |
  createAnswer.ts                                      |     100 |      100 |     100 |     100 |
 api/db/procedures/consumer/createSurveyAssignment     |     100 |      100 |     100 |     100 |
  createSurveyAssignment.ts                            |     100 |      100 |     100 |     100 |
 api/db/procedures/consumer/deleteUserAccount          |   77.77 |      100 |     100 |      75 |
```



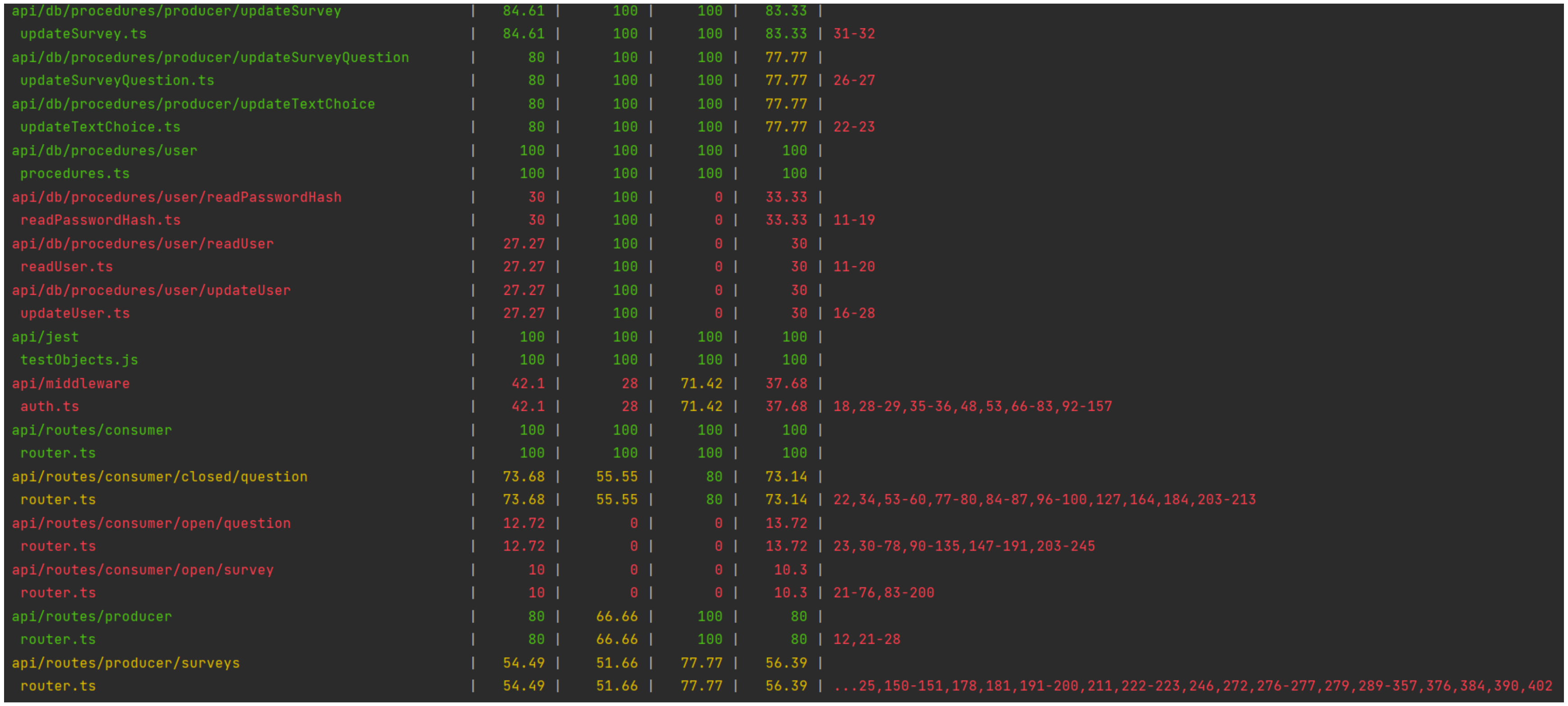
```
 createSurveyAssignment.ts                     |     100 |      100 |      100 |     100 |
api/db/procedures/consumer/deleteUserAccount   |   77.77 |      100 |      100 |      75 |
 deleteUserAccount.ts                          |   77.77 |      100 |      100 |      75 | 17-18
api/db/procedures/consumer/readQuestionId      |   81.81 |      100 |      100 |      80 |
 readQuestionId.ts                             |   81.81 |      100 |      100 |      80 | 14-15
api/db/procedures/consumer/selectAssignedQuestions |   88.88 |      100 |      100 |    87.5 |
 selectAssignedQuestions.ts                    |   88.88 |      100 |      100 |    87.5 | 19
api/db/procedures/consumer/selectChoices       |   88.88 |      100 |      100 |    87.5 |
 selectChoices.ts                              |   88.88 |      100 |      100 |    87.5 | 21
api/db/procedures/consumer/selectSurveyQuestions |     100 |      100 |      100 |     100 |
 selectSurveyQuestions.ts                      |     100 |      100 |      100 |     100 |
api/db/procedures/consumer/skipAssignedQuestion |   77.77 |      100 |      100 |      75 |
 skipAssignedQuestion.ts                       |   77.77 |      100 |      100 |      75 | 13-14
api/db/procedures/producer                     |     100 |      100 |      100 |     100 |
 procedures.ts                                 |     100 |      100 |      100 |     100 |
api/db/procedures/producer/createChild         |   91.66 |      100 |      100 |    90.9 |
 createChild.ts                                |   91.66 |      100 |      100 |    90.9 | 24
api/db/procedures/producer/createChoice        |     100 |      100 |      100 |     100 |
 createChoice.ts                               |     100 |      100 |      100 |     100 |
api/db/procedures/producer/createLabelInstance |   88.88 |      100 |      100 |    87.5 |
 createLabelInstance.ts                        |   88.88 |      100 |      100 |    87.5 | 19
api/db/procedures/producer/createLabelInstances |    92.3 |      100 |      100 |    90.9 |
 createLabelInstances.ts                       |    92.3 |      100 |      100 |    90.9 | 28
api/db/procedures/producer/createQuestion      |     100 |      100 |      100 |     100 |
 createQuestion.ts                             |     100 |      100 |      100 |     100 |
api/db/procedures/producer/createQuestionInstance |    90.9 |      100 |      100 |      90 |
 createQuestionInstance.ts                     |    90.9 |      100 |      100 |      90 | 19
api/db/procedures/producer/createSliderQuestion |     100 |      100 |      100 |     100 |
 createSliderQuestion.ts                       |     100 |      100 |      100 |     100 |
api/db/procedures/producer/createSliderQuestionLabel |      80 |      100 |      100 |   77.77 |
```



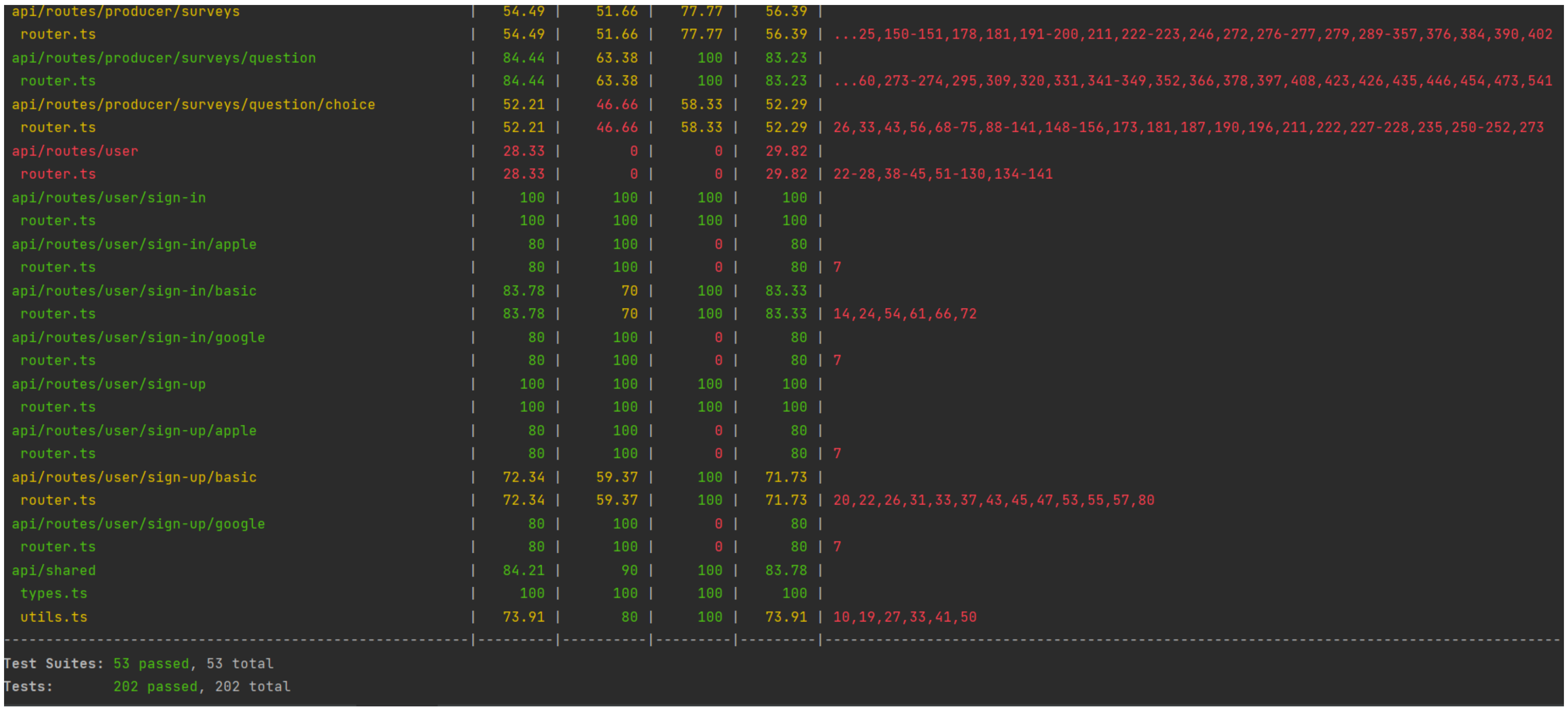
```
api/db/procedures/producer/createSliderQuestionLabel  |      80 |      100 |     100 |   77.77 |
 createSliderQuestionLabel.ts                         |      80 |      100 |     100 |   77.77 | 28-29
api/db/procedures/producer/createSliderQuestionLabels |    92.3 |       50 |     100 |    90.9 |
 createSliderQuestionLabels.ts                        |    92.3 |       50 |     100 |    90.9 | 19
api/db/procedures/producer/createSurvey               |     100 |      100 |     100 |     100 |
 createSurvey.ts                                      |     100 |      100 |     100 |     100 |
api/db/procedures/producer/createTextChoice           |     100 |      100 |     100 |     100 |
 createTextChoice.ts                                  |     100 |      100 |     100 |     100 |
api/db/procedures/producer/createTextChoiceInstance   |    90.9 |      100 |     100 |      90 |
 createTextChoiceInstance.ts                          |    90.9 |      100 |     100 |      90 | 26
api/db/procedures/producer/deleteChoice              |     100 |      100 |     100 |     100 |
 deleteChoice.ts                                      |     100 |      100 |     100 |     100 |
api/db/procedures/producer/deleteSingleChoice        |      50 |      100 |       0 |      60 |
 deleteSingleChoice.ts                                |      50 |      100 |       0 |      60 | 13-14
api/db/procedures/producer/deleteSliderQuestion      |      80 |      100 |     100 |   77.77 |
 deleteSliderQuestion.ts                              |      80 |      100 |     100 |   77.77 | 21-22
api/db/procedures/producer/deleteSliderQuestionLabel  |      80 |      100 |     100 |   77.77 |
 deleteSliderQuestionLabel.ts                         |      80 |      100 |     100 |   77.77 | 21-22
api/db/procedures/producer/deleteSurveyQuestion      |      80 |      100 |     100 |   77.77 |
 deleteSurveyQuestion.ts                              |      80 |      100 |     100 |   77.77 | 19-20
api/db/procedures/producer/deleteSurveys             |      80 |      100 |     100 |   77.77 |
 deleteSurveys.ts                                     |      80 |      100 |     100 |   77.77 | 19-20
api/db/procedures/producer/deleteTextChoice          |      80 |      100 |     100 |   77.77 |
 deleteTextChoice.ts                                  |      80 |      100 |     100 |   77.77 | 20-21
api/db/procedures/producer/readChoices               |   81.81 |      100 |     100 |      80 |
 readChoices.ts                                       |   81.81 |      100 |     100 |      80 | 21-22
api/db/procedures/producer/readLabels                |   81.81 |      100 |     100 |      80 |
 readLabels.ts                                        |   81.81 |      100 |     100 |      80 | 21-22
api/db/procedures/producer/readProducer              |   88.23 |      100 |     100 |   86.66 |
 readProducer.ts                                      |   88.23 |      100 |     100 |   86.66 | 28-29
```



```
 readProducer.ts                                 |   88.23 |      100 |      100 |   86.66 | 28-29
api/db/procedures/producer/readProducers         |    87.5 |      100 |      100 |   85.71 |
 readProducers.ts                                |    87.5 |      100 |      100 |   85.71 | 29-30
api/db/procedures/producer/readSliderQuestions   |   81.81 |      100 |      100 |      80 |
 readSliderQuestions.ts                          |   81.81 |      100 |      100 |      80 | 21-22
api/db/procedures/producer/readSurvey            |   81.81 |      100 |      100 |      80 |
 readSurvey.ts                                   |   81.81 |      100 |      100 |      80 | 20-21
api/db/procedures/producer/readSurveyAnswers     |   81.81 |      100 |      100 |      80 |
 readSurveyAnswers.ts                            |   81.81 |      100 |      100 |      80 | 24-25
api/db/procedures/producer/readSurveyAnswersAndCount |   81.81 |      100 |      100 |      80 |
 readSurveyAnswersAndCount.ts                    |   81.81 |      100 |      100 |      80 | 24-25
api/db/procedures/producer/readSurveyQuestions   |   81.81 |      100 |      100 |      80 |
 readSurveyQuestions.ts                          |   81.81 |      100 |      100 |      80 | 24-25
api/db/procedures/producer/readSurveys           |   81.25 |       75 |      100 |      80 |
 readSurveys.ts                                  |   81.25 |       75 |      100 |      80 | 25,36-37
api/db/procedures/producer/selectChoice          |      80 |      100 |      100 |   77.77 |
 selectChoice.ts                                 |      80 |      100 |      100 |   77.77 | 19-21
api/db/procedures/producer/selectQuestion        |      80 |      100 |      100 |   77.77 |
 selectQuestion.ts                               |      80 |      100 |      100 |   77.77 | 19-20
api/db/procedures/producer/selectSliderQuestion  |      80 |      100 |      100 |   77.77 |
 selectSliderQuestion.ts                         |      80 |      100 |      100 |   77.77 | 19-20
api/db/procedures/producer/selectSliderQuestionLabel |      80 |      100 |      100 |   77.77 |
 selectSliderQuestionLabel.ts                    |      80 |      100 |      100 |   77.77 | 19-20
api/db/procedures/producer/selectSurvey          |      80 |      100 |      100 |   77.77 |
 selectSurvey.ts                                 |      80 |      100 |      100 |   77.77 | 19-20
api/db/procedures/producer/updateChoice          |      80 |      100 |      100 |   77.77 |
 updateChoice.ts                                 |      80 |      100 |      100 |   77.77 | 23-24
api/db/procedures/producer/updateSliderQuestion  |      80 |      100 |      100 |   77.77 |
 updateSliderQuestion.ts                         |      80 |      100 |      100 |   77.77 | 24-25
api/db/procedures/producer/updateSurvey          |   84.61 |      100 |      100 |   83.33 |
```



```
api/db/procedures/producer/updateSurvey           |   84.61 |      100 |     100 |   83.33 |
 updateSurvey.ts                                  |   84.61 |      100 |     100 |   83.33 | 31-32
api/db/procedures/producer/updateSurveyQuestion   |      80 |      100 |     100 |   77.77 |
 updateSurveyQuestion.ts                          |      80 |      100 |     100 |   77.77 | 26-27
api/db/procedures/producer/updateTextChoice       |      80 |      100 |     100 |   77.77 |
 updateTextChoice.ts                              |      80 |      100 |     100 |   77.77 | 22-23
api/db/procedures/user                            |     100 |      100 |     100 |     100 |
 procedures.ts                                    |     100 |      100 |     100 |     100 |
api/db/procedures/user/readPasswordHash           |      30 |      100 |       0 |   33.33 |
 readPasswordHash.ts                              |      30 |      100 |       0 |   33.33 | 11-19
api/db/procedures/user/readUser                   |   27.27 |      100 |       0 |      30 |
 readUser.ts                                      |   27.27 |      100 |       0 |      30 | 11-20
api/db/procedures/user/updateUser                 |   27.27 |      100 |       0 |      30 |
 updateUser.ts                                    |   27.27 |      100 |       0 |      30 | 16-28
api/jest                                          |     100 |      100 |     100 |     100 |
 testObjects.js                                   |     100 |      100 |     100 |     100 |
api/middleware                                    |    42.1 |       28 |   71.42 |   37.68 |
 auth.ts                                          |    42.1 |       28 |   71.42 |   37.68 | 18,28-29,35-36,48,53,66-83,92-157
api/routes/consumer                               |     100 |      100 |     100 |     100 |
 router.ts                                        |     100 |      100 |     100 |     100 |
api/routes/consumer/closed/question               |   73.68 |    55.55 |      80 |   73.14 |
 router.ts                                        |   73.68 |    55.55 |      80 |   73.14 | 22,34,53-60,77-80,84-87,96-100,127,164,184,203-213
api/routes/consumer/open/question                 |   12.72 |        0 |       0 |   13.72 |
 router.ts                                        |   12.72 |        0 |       0 |   13.72 | 23,30-78,90-135,147-191,203-245
api/routes/consumer/open/survey                   |      10 |        0 |       0 |    10.3 |
 router.ts                                        |      10 |        0 |       0 |    10.3 | 21-76,83-200
api/routes/producer                               |      80 |    66.66 |     100 |      80 |
 router.ts                                        |      80 |    66.66 |     100 |      80 | 12,21-28
api/routes/producer/surveys                       |   54.49 |    51.66 |   77.77 |   56.39 |
 router.ts                                        |   54.49 |    51.66 |   77.77 |   56.39 | ...25,150-151,178,181,191-200,211,222-223,246,272,276-277,279,289-357,376,384,390,402
```



```
api/routes/producer/surveys                            |   54.49 |    51.66 |   77.77 |   56.39 |
 router.ts                                             |   54.49 |    51.66 |   77.77 |   56.39 | ...25,150-151,178,181,191-200,211,222-223,246,272,276-277,279,289-357,376,384,390,402
api/routes/producer/surveys/question                   |   84.44 |    63.38 |     100 |   83.23 |
 router.ts                                             |   84.44 |    63.38 |     100 |   83.23 | ...60,273-274,295,309,320,331,341-349,352,366,378,397,408,423,426,435,446,454,473,541
api/routes/producer/surveys/question/choice            |   52.21 |    46.66 |   58.33 |   52.29 |
 router.ts                                             |   52.21 |    46.66 |   58.33 |   52.29 | 26,33,43,56,68-75,88-141,148-156,173,181,187,190,196,211,222,227-228,235,250-252,273
api/routes/user                                        |   28.33 |        0 |       0 |   29.82 |
 router.ts                                             |   28.33 |        0 |       0 |   29.82 | 22-28,38-45,51-130,134-141
api/routes/user/sign-in                                |     100 |      100 |     100 |     100 |
 router.ts                                             |     100 |      100 |     100 |     100 |
api/routes/user/sign-in/apple                          |      80 |      100 |       0 |      80 |
 router.ts                                             |      80 |      100 |       0 |      80 | 7
api/routes/user/sign-in/basic                          |   83.78 |       70 |     100 |   83.33 |
 router.ts                                             |   83.78 |       70 |     100 |   83.33 | 14,24,54,61,66,72
api/routes/user/sign-in/google                         |      80 |      100 |       0 |      80 |
 router.ts                                             |      80 |      100 |       0 |      80 | 7
api/routes/user/sign-up                                |     100 |      100 |     100 |     100 |
 router.ts                                             |     100 |      100 |     100 |     100 |
api/routes/user/sign-up/apple                          |      80 |      100 |       0 |      80 |
 router.ts                                             |      80 |      100 |       0 |      80 | 7
api/routes/user/sign-up/basic                          |   72.34 |    59.37 |     100 |   71.73 |
 router.ts                                             |   72.34 |    59.37 |     100 |   71.73 | 20,22,26,31,33,37,43,45,47,53,55,57,80
api/routes/user/sign-up/google                         |      80 |      100 |       0 |      80 |
 router.ts                                             |      80 |      100 |       0 |      80 | 7
api/shared                                             |   84.21 |       90 |     100 |   83.78 |
 types.ts                                              |     100 |      100 |     100 |     100 |
 utils.ts                                              |   73.91 |       80 |     100 |   73.91 | 10,19,27,33,41,50
-------------------------------------------------------|---------|----------|---------|---------|-------------------------------------------------------------------------------------
Test Suites: 53 passed, 53 total
Tests:       202 passed, 202 total
```

2. Functional requirements

2.1. Required Features (must-haves)

The system must be able to register accounts with different roles
These are the survey producer and consumer roles. As the application will give different functions based on the users' needs, the users should be able to indicate this when they register.

The system must store user profile data (such as login credentials).
Because users must be able to log in, so this information should be stored.

Producers must be able to make surveys for consumers
This is a core functionality, without surveys, there is nothing for consumers to interact with.

The system must store surveys in a database
Surveys must be stored, in order to later serve them to consumers.

Producers must be able to review answered surveys
As the goal for the producer is to get feedback, they must be able to review their survey.

Producers must be able to receive their data in an aggregated and visualized form
Survey makers must get their data in a clear and summarized form, such as a graph, to save time and effort. It should also be possible to download data in an aggregated form, such that the survey producers can use tools of their choosing to further analyse and visualize the data.

Producers must be able to create branching surveys
Branching surveys allow producers to select different subsequent questions based on answers given by consumers. As some questions will become more relevant based on previous answers, this might give producers more relevant data at the end.

Producers should be able to invite consumers to a survey through an URL
This allows producers to share the survey with their target group.

Survey consumers must be able to answer survey questions
This is an absolute necessity, as without answers there is no data.

The system must store consumer answers in a database
Answers are stored to serve as feedback for the producers.

Survey progress of consumers is saved automatically
To improve user experience, consumers must be able to leave the application and continue answering surveys whenever they want. Every answer is stored in the database.

The final product can be hosted in production with Amazon Web Services (AWS)
Dunk currently uses AWS to host their website and has decided that it would like to continue using their services. AWS offers, among other cloud solutions, several ways to host servers, databases, file storage solutions, routing and load balancing of server instances.

2.2. Low Priority Features (should-haves)

Producers should be able to invite others to a survey through a QR code.
This allows producers an alternative method of sharing the survey with their target group. QR codes are especially useful in scenarios where survey consumers are reached via non-electronic means such as posters, flyers, letters, etc.

Producers should be able to mark their survey as private
A password is generated for private surveys and only consumers with an URL or QR code, and the correct password, gain access to the survey. This way, a producer can ensure that only consumers with the password can access the survey.

Consumers should be able to answer surveys without creating an account
This improves the privacy situation, as consumers are not forced into making an account and giving their personal data.

Consumers should be able to provide feedback about the survey.
With this, surveys can be further improved upon.

Consent pop up & Terms of Service should be added for doing surveys and registering accounts.
Users should be aware of their privacy rights and what happens with their data.

Accounts should be deletable, including the user data.
Any user should be able to be erased from the database, to implement the right to be forgotten from the GDPR.

2.3. Optional Features (could-haves)

The system could keep track of consumers that filled in surveys and reward the user with points based on a reward system.
Consumers could be incentivized into giving as many answers as possible. This can be done by implementing a reward system, which tracks points. The consumer could see their progress, how many surveys were completed, how many questions were answered and how close they were to certain rewards.

The system should be able to spot bots or suspicious behaviour.
This is necessary to stop people from using bots to participate in a large number of surveys to get rewards without actually taking part in them.

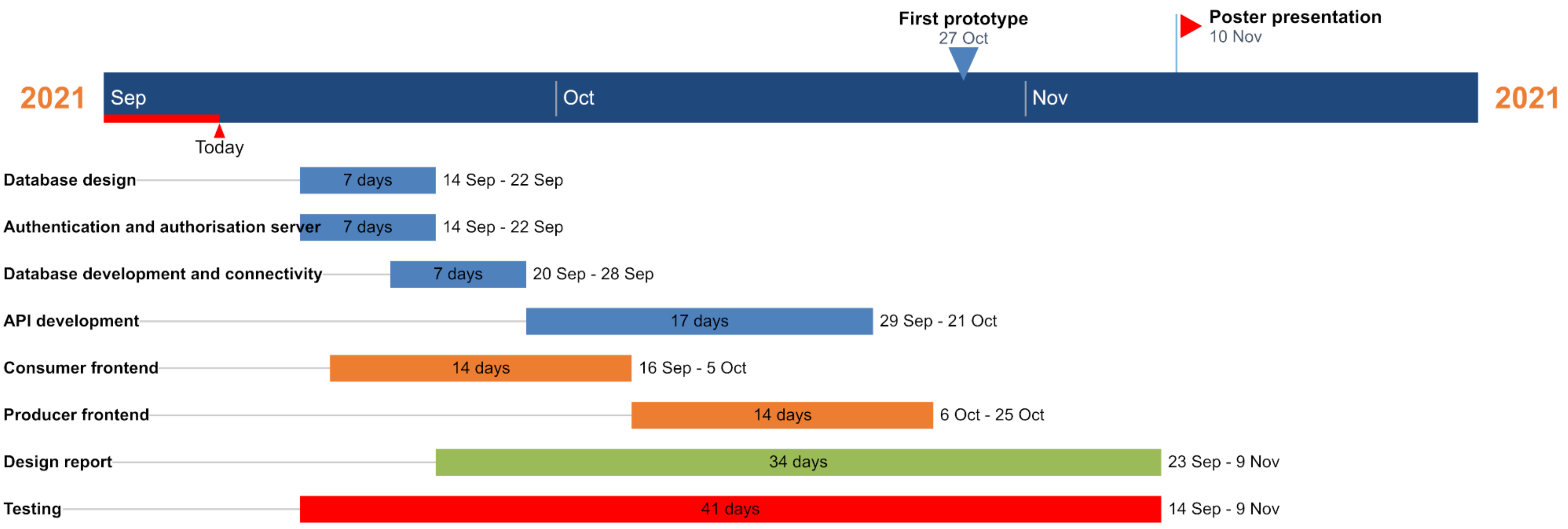
The producer could receive automatically generated feedback based on the statistics from the answered surveys.
If enough data is available, models could be used to generate (simple) conclusions that could be given as feedback, so the producers do not have to manually look at the data and spend time on the feedback.

Utilizing machine learning, consumers could have more specific surveys based on previous data that they submitted.
This leads to more relevant surveys given to consumers, which means they do not have to spend time looking for relevant surveys.

The consumer could earn extra rewards for giving feedback.
As producers would receive more valuable information on how to create their surveys in the future, this should be rewarded if it is useful.

Database security and measures against leaks could be taken.
As this project involves a lot of personal data that is stored, correct security measures should be taken to prevent leaking of this data. As this will take more time and expertise, this is left as an optional feature.

3. Planning


First prototype
27 Oct
Poster presentation
10 Nov
2021
Sep
Oct
Nov
2021
Today
Database design
7 days
14 Sep - 22 Sep
Authentication and authorisation server
7 days
14 Sep - 22 Sep
Database development and connectivity
7 days
20 Sep - 28 Sep
API development
17 days
29 Sep - 21 Oct
Consumer frontend
14 days
16 Sep - 5 Oct
Producer frontend
14 days
6 Oct - 25 Oct
Design report
34 days
23 Sep - 9 Nov
Testing
41 days
14 Sep - 9 Nov