Open Source Voting Web Application

Tobias Prasser, Max Martin Hierzer, Gernot Fasching

April 18, 2025

Abstract

Kurzfassung

Contents

Chapter 1

Introduction

Link Gliederung: https://www.diplomarbeiten-bbs.at/durchfuehrung/gliederung-der-diplomarbeit-und-formale-vorgaben

1.1 Short description

The topic of this diploma thesis is creating a platform which supports different voting options like single, multiple or weighted choice. Additionally there should be a Login system with different roles to administer and create or delete polls and one where the user can simply vote for the polls he's included in. Furthermore there an option to disclose the results and who voted for which answers. The database should run on a remote server and be accessed by an API.

The reason we chose this topic is because our supervisor is part of the LMP party and they couldn't find an appropriate platform to vote on party intern problems and topics. Hence he approached us and suggested we write our diploma thesis on a voting platform.

1.2 Description of performed work

Our aim is to provide a website where different organizations can create and publish polls for their members. Since our finished work will be open source, everyone who wants to create polls will benefit from our work.

We chose to accept the LMP as our partner, because they brought up that there isn't a platform that supports all the features they need. Moreover can they give us feedback of the real life application so we can adjust the features to a user organization. During the development of our work we had monthly meetings with the LMP to discuss the progress. Because we decided

to develop our software in an agile way the discussions we had with them also helped so we could focus on the more important features first and implement elements of lesser importance later.

1.3 Methodology of the thesis

At first we had to decide on a tech stack. After careful consideration we decided upon a PostgreSQL database, a backend of node.js, sequelize to perform database operations and express to write APIs so we can connect with our frontend. Our frontend is based on React and we also included a PWA. After this decision we began with a simple input and output from front- to backend so ensure we all understood how each part is connected to each other. The next step was implementing the first features. We split the elements in different components so we could work separately and efficiently, e.g the single choice is split in create the poll, display the poll, vote, and show the results. Reasons we chose this tech stack and a thorough description of each function our work has will be in the main part.

Chapter 2

Tech Stack

2.1 PostgreSQL

2.1.1 Generel

PostgreSQL was chosen as the relational database management system for this project due to its robustness, scalability and strong support for advanced data types and queries. The decision to utilize PostgreSQL was further supported by its user-friendly design, extensive and well-maintained documentation, and its widespread adoption as one of the leading open-source relational databases.

The installation was also simple because of the detailed install guide provided by multiple sites like **w3schools**. Before the installation certain configuration had to be done such as: specify the storage directory, select a password, set the port the server should listen on. To create the first database the SQL shell of PostgreSQL was used in the beginning. But after that the preferred tool was **pgAdmin 4**, an popular and feature rich Open Source administration and development platform for PostgreSQL, to manipulate data or view schemes of the database.[**pgsinstallation**]

For this project it was essential to have a stable and robust system in the backend that can handle a bigger amount of data efficiently.

2.1.2 Scheme of the database

The initial database scheme differed significantly from its current structure, primarily due to evolving client requirements. However, it was evident from the outset that the "Polls" and "Users" tables would serve as central components of the schema, as the majority of the data entities and relationships would reference or depend on these two core tables. For the Entity

Relationship Diagrams tools like **draw.io** and **dbDiagramm** where used to create the visualization directly from SQL. The database consists of 13 tables, with "UserPolls", "UserAnswers", "PollGroups" and "QuestionAnswers" functioning as intermediary tables that facilitate many-to-many relationships between the core entities.
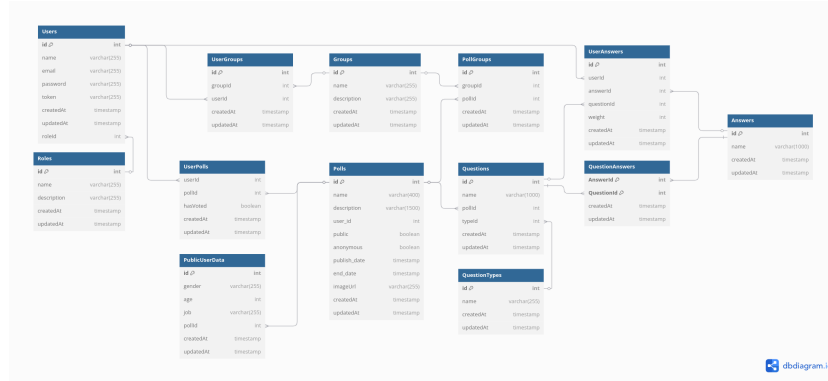

Figure 2.1: New Entity Relationship Diagram

2.2   node.js

2.3   Sequelize

Sequelize was selected as the Object-Relational-Mapping(ORM) tool for this project due to its seamless integration with Node.js and PostgreSQL, offering a robust and declarative approach to database interaction. By abstracting raw SQL queries into model-based definitions, Sequelize enables a more maintainable and readable codebase, especially in project with complex data relationships and evolving schema requirements.

Its support for model associations, transaction management and migrations provides a powerful foundation for implementing scalable and consistent backend logic. Additionally, Sequelize's promised-based architecture aligns well with asynchronous programming paradigms common in modern JavaScript applications, improving performance and responsiveness.[**sequelizegit**]

The comprehensive documentation, active community support, and compatibility with various SQL dialects, as in our case PostgreSQL, further contributed to the decision to use Sequelize as the ORM in this project.

2.4 express

2.5 React

2.6 PWA

Chapter 3

Features

3.1 Login

The login feature is essential for our application, as users need a secure and reliable way to authenticate themselves and gain access to their accounts. It consists of two main functionalities: the login itself and also the logout. Additionally, we have implemented user-friendly error handling to enhance user experience as well as security.

3.1.1 Frontend

The frontend component of the login feature uses React's built-in useState hook in order to manage the input username and password and store them in the component's state temporarily. After submitting the login form, the input data is sent to the backend API via a POST request.

The password is sent to the backend in plaintext over HTTPS. It is not encrypted on the client side, because HTTPS already provides built-in encryption and also protects against interception or man-in-the-middle attacks.

3.1.2 Backend

The backend receives the login request through a predefined API route that forwards the data to the handleFetchLogin function in the user controller. It extracts the username and password from the request body and passes them to the fetchLogin function in the user service.

This function first checks if a user with the provided username even exists in the database. If no user is found, it returns an error message stating

that either the username or password is invalid. We intentionally chose to not specify which of the two parts is incorrect in order to prevent attackers from being able to determine whether a username exists in the system. This provides protection against user enumeration attacks.

If a user is found, the provided password is then compared to the hashed password stored in the database using bcrypt's comparison method, bcrypt.compare(). In case the passwords do not match, the same error message mentioned above is returned to the frontend.

However, if the password is correct, the backend returns a response containing a success indicator, the user's unique ID (userId), their username, the assigned role ID (roleId), and the name of the role (roleName).

3.1.3 Error Handling and Security

Error handling is implemented consistently across both frontend and backend to ensure security and user-friendliness. If a login attempt fails, whether due to an incorrect username, password, or both, the system always returns the same general error message. This prevents attackers from determining whether a specific username exists in the system.

Sensitive data like passwords is handled securely. Passwords are not stored in plain text. Instead they are hashed using bcrypt before being inserted into the database. Furthermore, communication between frontend and backend is encrypted through HTTPS. This ensures that all transmitted data remains private and protected from unauthorized access.

3.2 Registration

The application features a controlled and secure registration. In order for users to be able to join the system, they first have to be invited. The registration consists of two main functionalities.

3.2.1 Admin Registration Process

Only administrators are allowed to invite new users. They start the process by navigating to the registration tab and entering the user's email as well as their intended role. After submitting, the data gets sent to the "sendEmail" function in the backend through a POST request.

The function checks if a user with the provided email address already exists in the database. If so, it returns a message indicating exactly that. In case the email is not found, an invitation email is sent and a new user entry with the provided email, role ID and a randomly generated token is created. The token is later used for verifying the legitimacy of the registration. If the process is successful the administrator is informed that the email was sent and the user was created successfully.

3.2.2   User Registration Process

The email sent to the user contains a registration link consisting of the URL of the application and the previously mentioned token. By clicking the link, the user is redirected to a registration page on which they can enter a username and password.

The page contains three fields. One for the username, one for the password and one for password validation. This approach was chosen in order to ensure a correct password input. If the contents of the two password fields do not match, an error message telling them about the mismatch is displayed. Similarly, if any of the fields are left blank, submission is disabled and the user is notified of the problem. If all inputs are valid the token extracted from the link, as well as username and password are sent to the backend using a PUT request. The password is sent in plain text over HTTPS, as explained in the login section above.

The backend then passes the data to the "createUser" function. It first checks whether the token exists in the database and also if the username is already being used. In both cases, no token or existing username, the user receives an error message telling them the specific problem. If the username is the issue they can just change it and try again, but if it is the token it will never work no matter how often they retry. If both checks pass, the password is hashed using bcrypt's hash function. The database is then updated with the new username and hashed password. The token is set to null to prevent reuse. If everything is successful the function returns a success message is returned telling the user that the registration is complete. After five seconds they are redirected to the login page, where they can then log in and use the application.

3.2.3 Sending Invitation Emails

For automatic sending of the invitation emails, we implemented two different approaches. Therefore both will be explained below.

Nodemailer (SMTP)

The first approach uses "nodemailer" to send emails over SMTP. Firstly a transporter is created using nodemailer's "createTransport" function. There several options can be defined including the host, like gmail or a custom SMTP server, the port, secure option and also authentication credentials. After setting up the transporter, the email options are defined. These include

```
let transporter = nodemailer.createTransport({
  host: "smtp.mailersend.net",
  port: 587,
  secure: false,
  auth: {
    user: test.email@domain.at,
    pass: Password1,
  }
});
```

Figure 3.1: This is an example for the transporter using mailersend as the SMTP server

the sender and recipient addresses along with the subject and general content of the email. Finally the email is sent using "transporter.sendMail(mailOptions)".

```
const mailOptions = {
  from: "Tool" <test.email@domain.at>,
  to: user@email.at,
  subject: "Polling tool",
  text: "Hello this is your invitation."
};
```

Figure 3.2: This is an example for the email options

MailerSend API

The second approach uses the MailerSend API to send the invitation emails. This method was implemented because our deployment host blocks all SMTP ports.

To use the MailerSend API, a transporter is created with "new MailerSend" and the API key. Just like int the first method mail options are created specifying the sender, recipient, subject and message. The email is then sent

```
1    const mailOptions = new EmailParams()
2    .setFrom(new Sender(test.email@domain.at, "Tool"))
3    .setTo([new Recipient(user@email.at)])
4    .setSubject("Polling tool")
5    .setText("Hello this is your invitation.");
6
```

Figure 3.3: This is an example for the email options for MailerSend

using "mailerSend.email.send(mailOptions)".

3.2.4 Roles

Implementing a role-based system with three distinct roles - "Admin," "Poweruser," and "Normal" - is crucial for te functionality and security of the application. By assigning permissions flexibly, a clear hierarchy is established, enhancing both user experience and data integrity. Admins are granted full control over the application, while Poweruser enjoy extended privileges for managing polls. Normal users can seamlessly participate in polls and view results without jeopardizing sensitive functionalities. This structure facilitates efficient task delegation an scalability, allowing the application to be easily expanded with additional roles in the future. The role system thus significantly contributes to the security, organization, and user-friendliness of the polling application.

This functionality was implemented by introducing a "Roles" table, which maintains a one-to-many relationship with the "Users" table. Each user is associated with a specific role through the foreign key **roleId**. The roles are defined and manged exclusively within the database, with the following

ID assignments: 1 for administrators, 2 for powerusers and 3 for standard users. Role-based access is determined solely based on these database-level associations.

3.3 Group System

3.4 Create Polls

3.4.1 Function

3.4.2 Validation

3.4.3 Start-/ Endtime

To make Polls more manageable, users can define a voting period by setting a start and end time during poll creation. This feature consists of two main parts.

Setting The Times

The first part involves setting the start and end time in the poll creation interface. Initially, the HTML native input type "datetime-local" was used. However it was not fully supported by every browser, including Firefox which could only depict the date but not the time. For this reason, after some consideration, the "react-datetime" component was chosen instead.

When a user clicks on the input field, a graphical selector appears, which allows the user to select both the date and time easily.

Once the poll is submitted, the selected start and end times are sent to the "createPoll" function in the backend via a POST request. There, the received times are converted to JavaScript "Date" objects before being inserted into the database.

Utilizing The Times

Max

3.4.4 Poll Image

In response to client requirements, a feature was implemented allowing users to upload a title image for each poll. This image serves a purely atmospheric and aesthetic purpose, aiming to enhance the visual appeal of the poll and provide users with a more engaging and intuitive introduction to the survey topic. By incorporating visual context, the feature helps draw user attention and encourages participation, particularly in cases where the subject matter benefits from visual reinforcement.

Backend Implementation

On the backend, the image upload is handled using the **multer** middleware in the **imageRoutes.js** file. The storage engine is configured to save uploaded files in the **uploads** directory with a unique filename based on a timestamp and the original file extension. The file types are restricted to JPEG, PNG and GIF formats through a custom file filter to ensure consistency and security.[**expressmulter**]

```
1    const storage = multer.diskStorage({
2      destination: (req, file, cb) => {
3        cb(null, 'uploads');
4      },
5      filename: (req, file, cb) => {
6        cb(null, Date.now() + path.extname(file.originalname));
7      }
8    });
9
```

Figure 3.4: Custom storage engine configuration for multer

The route /**api**/**upload-image** is exposed via the Express router and returns the public URL of the uploaded image on success. The image can later be used as part of the poll metadata.

In **app.js**, static file serving is enabled for the **uploads** directory to make the images accessible to the frontend:

```
1    app.use('/uploads', express.static('uploads'));
2
```

Figure 3.5: Static file serving enabled

Frontend Integration

On the frontend, the image upload is handled within the **CreatePolls.js** component. When a user selects an image file, the **handleImageUpload** function is triggered.

```
1    const handleImageUpload = async (event) => {
2      const file = event.target.files[0];
3      if (!file) return;
4      const previewUrl = URL.createObjectURL(file);
5      setImage(previewUrl);
6      const formData = new FormData();
7      formData.append("image", file);
8      try {
9        const res = await
     ↪   fetch(`${process.env.REACT_APP_API_URL}/api/upload-image`,
     ↪   {
10          method: 'POST',
11          body: formData,
12        });
13        const data = await res.json();
14        if (res.ok) {
15          setImageUrl(data.imageUrl);
16        } else {
17          setResponse('Error uploading image');
18        }
19      } catch (error) {
20        console.error('Error uploading image:', error);
21        setResponse('Error uploading image');
22      }
23    };
24
25
```

Figure 3.6: handleImageUpload function

This function performs the following steps.

1. A local preview of the selected image is created using the browser's
   URL.createObjectURL() method. This provides immediate feed-
   back to the user by displaying the image in the form before it is up-
   loaded.

2. To prepare the image for upload to the server, a new instance of the
   FormData object is created. This object is specifically designed to
   handle form submissions that include binary data such as files. The
   use of **FormData** is necessary because standard JSON-based payloads

18

application/json do not support file transfers. The browser automatically sets the **Content-Type** to **multipart/form-data** when **FormData** is used, which is the appropriate MIME type for file uploads.

3. The image is uploaded to the backend by making a **POST** request to the endpoint **api/upload-image**. If the upload is successful, the backend responds with the relative path to the stored image, which is then saved in the component state **imageUrl** for use in the final poll submission.

4. The uploaded images's URL is included as part of the final poll payload when the user submits the form. This allows the backend to store the image reference in the database alongside the poll data.

5. Upon successful poll creation, the image preview, image URL and file input are all reset to ensure a clean state for subsequent poll creation actions.

By combining real-time preview functionality with reliable backend integration, the image upload feature significantly improves the overall usability and aesthetic appeal of the application.

3.4.5   Questions

Single Choice

Multiple Choice

Weighted Choice

3.4.6   Demographic Questions

To gather the data of our public voters, the implementation of demographic questions was crucial. This feature is only available for public polls since the created user would be part of the organization using our project and therefore have the data already. If the users is not part of the organization there is still the option to contact them via the e-mail used for the registration. Since most of these questions are similar for every poll, a modular system where questions can be created, added, removed and changed is the best solution. Figure 3.7 shows the demographic question in create polls. The options and functionality of these questions are the like ones described in the previous sections, with the difference that weighted is not an option, since demographic data is more like a fact less an opinion.

Figure 3.7: Create Demographic Question

The new part for this feature is the search bar. For this the "Select" component of "react-select" is used. The components' controllable state props and modular architecture allows "isMulti" to select multiple options or "isSearchable" to search. These features, allow easy implementation and an already styled search bar in the project. [**reactselect**]

The database structure for the demographic questions is similar to the standard ones. The table "PublicQuestions" is used to store the question specific data like name and type. To enable the reuse of questions on different polls "PublicQuestions" is in an many-to-many relationship with "Polls" through "PublicPollQuestions". Answers are stored within the table "PublicAnswers", which is connected to "PublicQuestions" via "PublicQuestionAnswers. This relation is also many-to-many since the options yes or no for example would be used in multiple questions.

For the select every existing question with its answers is fetched from the database and stored within an array. The options are then mapped with the value being id and the label as name of each element. To display the selected options they are mapped through and use the same functions as the standard ones used for the poll. When saving these questions a problem arises, as the selected ones are already stored in the database and possibly changed. To handle this a findOrCreate 3.8 is used to get the existing questions and answers or create new ones. This function also returns each instance found or the created one. With this the question and answer ids can be used for

the many-to-many relationship. [**sequelizedoku**]

```
1  let [createdQuestion, created] = await
   ↪  PublicQuestions.findOrCreate({
2    where: {
3      name: question.name,
4      typeId: questionType.id,
5    }
6  });
```

Figure 3.8: findOrCreate PublicQuestions

3.5 Edit Polls

3.6 Voting

3.6.1 Disclosed Voting

In disclosed voting, each voter can see who voted for which option. This level
of openness can significantly shape how people vote, as choices are no longer
private. As such, poll creators should carefully consider the implications and
advantages of this approach.

Transparency: Can be a powerful tool especially in contexts where open
discussion is encouraged. When voters are aware that their choices will be
visible, and everyone has access to the same information, it creates a shared
starting point for meaningful discussion.

Analytics and Reporting: Unlike anonymous voting, disclosed voting
provides access to detailed, voter-specific data. This allows for more compre-
hensive analysis and reporting, making it easier to identify patterns, under-
stand voter behavior, and repurpose the results for future decision making.

Voter Trust: Disclosed voting can also help increase trust in the voting
process. Because individual votes can be traced back to their sources, it
becomes much harder to commit fraud or manipulate the outcome. In con-
trast, anonymous voting may raise concerns about the system's integrity,

since there's no way to independently verify the authenticity of each vote. [**disvsanon**]

Frontend

Backend

3.6.2 Anonymous Voting

This voting style keeps individual choices completely hidden, making it impossible to trace votes back to specific participants. While this limits the amount of data available for analysis, it comes with several key advantages that protect voters and promote fairness.

Voter Freedom: Because votes are not linked to identities, participants can express their opinions freely, without fear of judgment or consequences from others. This anonymity encourages honest responses, even on sensitive or controversial topics.

Protect Against Intimidation: In a disclosed voting system, participants could become targets of pressure, bullying, or even blackmail based on how they vote. Anonymous voting helps mitigate these risks by shielding voters from external influence, creating a safer environment for participation.

Eliminate "Herd Mentality": Herd mentality refers to the tendency of individuals to follow the majority, sometimes at the expense of their true opinions. While the platform already helps reduce this by showing results only after voting ends, anonymous voting further minimizes this effect. Even if a voter ends up being in the minority, their identity remains protected, encouraging more authentic and independent choices. [**disvsanon**]

Frontend

Backend

In an anonymous voting system, preventing the ability to trace votes back to individual users is crucial. In contrast, for disclosed voting, the UserAnswers table stores each answer along with the corresponding user's ID.

To support anonymous voting, it's important that the user's ID is not traceable. Therefore, the userId field is configured with "allowNull: true". However, this creates a challenge: the system cannot verify whether a user has

already submitted a vote.

To solve this, a many-to-many relationship is introduced between the Polls and Users tables, using an intermediate table that includes a hasVoted boolean field with a default value of true. When a user submits a vote, an entry is created in this table with hasVoted set to true. This allows the system to track whether a user has already voted in a specific poll without linking their vote to their identity.

3.6.3 Public Voting

The public voting allows users without an account to vote. With this feature a wide range of people can be questioned in street surveys or through a shared link. This poll type has two section, the normal and demographic questions. The order of these play a major role. Römermann mentions trust, benefits of the demographic data and the ability to abstain. All of these factors have to be taken into ones account when creating these questions. The article also mentions, that at the beginning of a survey the motivation is high and the demographic data are answered, but the trust in full anonymity is decreased. Therefore the author states it is best to put these questions at the end. [**demographicdata**]

Poll Questions

In public voting, since users cannot select a poll themselves, it's important to handle cases where the poll is accessed via a direct link outside the allowed time frame. To keep things simple, a short message — "Poll not available" — is displayed in such cases. If the site is accessed within the designated start and end dates, the questions are shown, similar to those in disclosed and anonymous voting. The main difference in this voting method lies in the submit button. Its function is only to change the display so that demographic questions are shown.

Demographic Questions

After the poll is completed, the demographic questions linked to it are displayed. These questions follow the same visual style as those in the other sections. Once submitted, the submit button becomes greyed out and disabled. A short thank-you message, along with a link to the organization's website, is then shown. Similar to the poll questions, the key difference here lies in how the submit button behaves. Before sending the data, the

system checks whether the user has already voted. If not, the responses to both the poll and demographic questions are sent to the backend API via a POST request. The service processes these answers and inserts them into the corresponding database tables.

Vote Integrity

A major problem when having anonymity and no accounts is the data integrity. Without the ability to store information about a voter in the database to check for multiple votes, it is important to prevent them from voting multiple times. Completely avoiding this problem is nearly impossible, but to ensure no problems arise we chose two different security measures.

To prevent fraudulent activity, spam, and abuse with bots, Google reCAPTCHA is integrated into the application. To implement this a key pair is generated, one for the site and a secret key. The site key is used to integrate the reCAPTCHA service into the frontend. The secret key facilitates secure communication between the backend server and the reCAPTCHA system to validate user responses. To maintain security the keys are stored in an env file. For this whole process the invisible option is checked to prevent the flow of the voting being disturbed.[**recaptcha**]

3.9 shows how the backend handles the CAPTCHA request. The token, the site key, is sent from the frontend through the request body, while the secretKey is accessed via the prossess.env environment variable. To maintain readability, the URL is defined and the query string includes both keys. Then these parameters are sent to Googles endpoint to validate the user interaction. The score in Google's response indicates how likely a user is human. Therefore, before returning a successful response to the frontend, the score is checked. The code also handles the cases where the request results in an error or the score is too low. [**recaptcha**]

Cookies are small pieces of data stored locally on a user's device by their browser. They are commonly used to save user-specific information, such as usernames or passwords, to enhance the web browsing experience.

Other common use cases include:
1. Session Management: Allows a website to remember user behavior and preferences across sessions.

```
1  app.post('/verify-recaptcha', async (req, res) => {
2    const { token } = req.body;
3    const secretKey = process.env.RECAPTCHA_SECRET_KEY;
4    const url = `https://www.google.com/recaptcha/api/
5    siteverify?secret=${secretKey}&response=${token}`;
6    try {
7      const response = await fetch(url, {
8        method: 'POST',
9      });
10     const data = await response.json();
11     if (data.success && data.score > 0.5) {
12       res.json({ success: true });
13     } else {
14       res.json({ success: false, message: 'Verification failed'
          ↪  });
15     }
16   } catch (error) {
17     console.error('Error verifying reCAPTCHA:', error);
18     res.status(500).json({ success: false, message: 'Server error'
          ↪  });
19   }
20 });
21
```

Figure 3.9: reCAPTCHA backend

2. Personalization: Enables websites to tailor content, such as language settings or recommended items, to individual users.

3. Tracking: Often used in e-commerce to maintain a shopping cart while users navigate through different pages of a site.

In all these scenarios, the data is stored locally on the user's device [**cookies**]. On this platform, however, cookies serve a more specific purpose: to store a boolean flag indicating whether a user has already voted. Figure 3.10 illustrates how the cookie is stored in the browser. The cookie's expiration is set to the poll's end date, ensuring it is automatically removed once voting closes.

The cookie is set only after the voting request is successfully processed. This order is crucial, as vote submissions can occasionally result in errors. If the

Figure 3.10: Cookie stored in the browser

cookie were set before confirmation, users could be wrongly prevented from resubmitting. By storing the cookie only after a successful vote, users can correct and resubmit their answers when needed.

This method is not entirely secure. If someone knows where the cookie is stored and how to access it, they can delete it and vote again. Still, this setup addresses most common threats to the integrity of the poll.

3.7 Results

3.7.1 CSV-Export

To allow for the further analysis and documentation of the poll results, a CSV export functionality was implemented. This export privides a structured overview of all user responses, including additional metrics such as the number of votes per answer and the average weight for weighted questions.

The logic is handled in the **csvExportController.js** controller. Upon receiving a request, the system retrieves all relevant data for a given poll by its ID, including associated question, answers, and question types. The raw voting data is processed to count the number of responses per answer and to calculate the average weight if applicable. Special care is taken to distinguish between different types: for instance, "Single Choice" questions ensure that only one answer per user is counted, while "Weighted Choice" questions include the weight vale in the computation.

The processed data is then converted into a structured CSV format using the json2csv library. The final file includes columns such as Poll Name, Question, Question Type, Answer, Vote Count and Average Weight, and can be downloaded directly by the user. This feature ensures transparency and enables further statistical evaluation using external tools like Excel.[**json2csv**]

3.8 MyPolls

3.8.1 Purpose

The "MyPolls" section central location where users can view and manage all
polls they have created. It provides access to the links for the polls as well
as a delete option for them.

3.8.2 Poll Link

The poll link used for sharing the Polls is generated by encoding key infor-
mation into a hash. This is done using the "btoa" function, which cre-
ates a Base64-encoded string. In order to make the hash browser navi-
gation compatible it is also encoded as an URI component with the same
name(encodeURIComponent). The hash contains information about if the
poll is public, if it is anonymous, about the mode and the poll's ID.

When opening the Polllink the hash gets URI and "atob" decoded and de-
pending on the mode the corresponding section in the dashboard and the
poll with the right ID gets opened. Should the ID not exist in this specific
mode, or in general, it just opens the right mode in the default state.

To make sharing of the Surveys easier, a QR code generation was imple-
mented. This was done using the React component "qrcode.react" which
requires the link to the poll.[**qrcode**]

3.8.3 Delete Polls

The deletion of polls is crucial for two main reasons: first, to ensure that
all associated data of a survey can be safely removed when necessary and
second, to determine when a poll is still eligible for deletion. In this case, the
deletion of polls is strictly limited to those that have not yet received any
user votes.

This constraint ensures the integrity of the data and avoids loss of user-
generated information, which could distort statistical analysis or transparency
within the system.

To maintain data consistency, the deletion process is implemented as a trans-
action. This guarantees atomicity, meaning that either all steps of the dele-
tion process succeed or none do - preventing partial data deletion and po-

tential corruption. Sequelize's transaction management is used here to wrap the entire process in a rollback-safe structure. [**sequelizedoku**]

The deletion logic performs the following steps

1. Validate that the poll exists.

2. Fetch all related questions and their IDs.

3. Check if any user answers exist for these questions. If so abort the operation with an error.

4. Remove the many-to-many relations between answers from the "QuestionAnswers" table.

5. Delete the associated answers.

6. Delete the questions.

7. Delete any group relations in the "PollGroups" table.

8. Finally, delete the poll itself.

Using Sequelize's transaction mechanism not only helps to avoid data inconsistency, but also ensures that no information is accidentally deleted once users have participated in a poll. This feature is particularly important in environments where transparency and trust are key, such as in political or organizational voting systems.

3.9 Accessibility

3.9.1 Tooltips

To enhance the usability and user experience of the application, tooltips were integrated into all input field, providing users with contextual guidance and reducing the likelihood of input errors. For this purpose, the React library **react-tooltips** was utilized, offering a high degree of customizability in both design and behavior. This allowed for the implementation of consistent and informative tooltip elements throughout the interface.[**tooltips**]

Each from element is assigned a unique **data-tool-id**, which serves as a reference key within the central **Tooltips.js** configuration file. This file

```
1   const deletePoll = async (pollId) => {
2     const transaction = await sequelize.transaction();
3     try {
4       ...
5       await PollGroups.destroy({
6         where: { pollId },
7         transaction,
8       });
9       await Polls.destroy({
10        where: { id: pollId },
11        transaction,
12      });
13      await transaction.commit();
14      return { pollId, questionsDeleted: questionIds.length };
15    } catch (error) {
16      await transaction.rollback();
17      throw error;
18    }
```

Figure 3.11: Example for a Sequelize transaction

defines the content, styling, and interaction logic for each tooltip, ensuring that the visual presentation and timing behavior are consistent across the application. To prevent the tooltips from becoming distracting during regular use, a delay of 1500 milliseconds was introduced before displaying the tooltip on hover.

The overall purpose of this component was, to improve the intuitiveness of the application, especially for first-time users or those unfamiliar with specific input requirements.

3.9.2 Screenreader

3.10 Styling

3.10.1 Generel styling

The styling of the application was done later in the making process, because only then it was demanded by the customers. The same applies to the color

scheme which has two main colors: Yellow and Blackberry.

To ensure consistency of the palette user defined characteristics from CSS where used to assign variables for each main color. [**csscolorvariables**]

```
1    :root {
2      --primary-color: #51184e;
3      --secondary-color: #F9BB03;
4      --primary-hover-color: rgb(163, 131, 168);
5    }
6
```

Figure 3.12: Variables of the main colors

Those variables where then used in each element of where it was needed. Therefore changing the color scheme to a different one is less afford for the end user.

The basic structure of the page is split into three sections: header with navigation, main with content and footer with imprint and privacy policy. The style is based on the client's homepage therefore the standard background is purple but of the content it still is white to ensure readability. [**lmppage**]

3.10.2 Responsive design

Responsive design is essential nowadays because users access websites from a wide variety of devices - phones, tablets, laptops and desktops - each with different screen sizes and resolutions. It ensures a seamless, user-friendly experience across all platforms, which improve engagement and accessibility.

To ensure responsiveness in each section of the application, "@media CSS at-rule" was used to apply different styles based on the screen size or device characteristics. This approach allows the layout, font sizes and element spacing to adapt dynamically, creating a smoother experience for users on any device. The navigation bar was designed with media queries to transform into so-called "burger menu" when the screen width falls below a certain threshold. This ensures usability on smaller devices like smartphones because the elements within the navigation get displayed in the menu now.

Registrierung	Erstellen	Bearbeiten	Abstimmen	Ergebnisse	Meine Umfragen	Gruppen

Figure 3.13: Navigation

Figure 3.14: Burger menu

3.10.3   Individual styling

Chapter 4

Summary

List of Figures