Open Source Voting Web Application

Tobias Prasser, Max Martin Hierzer, Gernot Fasching

May 18, 2025

Abstract

This thesis details the design and development of an open-source web application for secure and flexible online voting, developed in collaboration with the Austrian political party Liste Madeleine Petrovic (LMP). The application's backend is built using Node.js, Express, and Sequelize, with a PostgreSQL database, while the frontend is implemented with React and enhanced with Progressive Web App (PWA) functionality for cross-platform compatibility. The project was developed using agile methodologies, with continuous feedback loops through regular meetings with the LMP to ensure the system met real-world requirements. Key features include a secure role-based login system, email-based user registration, poll creation with customizable question types - including single, multiple, and weighted choice - demographic data collection, and multiple voting modes such as anonymous, disclosed, and public voting. Additionally, accessibility and responsive design were prioritized to support a broad user base. The platform is published under the MIT license to promote transparency, reusability, and further development by other organizations in need of digital voting solutions.

Kurzfassung

Diese Diplomarbeit beschreibt die Konzeption und Entwicklung einer Open-Source-Webanwendung für sicheres und flexibles Online-Voting, die in Zusammenarbeit mit der österreichischen politischen Partei Liste Madeleine Petrovic (LMP) realisiert wurde. Das Backend der Anwendung basiert auf Node.js, Express und Sequelize in Kombination mit einer PostgreSQL-Datenbank, während das Frontend mit React umgesetzt und durch Progressive-Web-App-Funktionalität (PWA) für plattformübergreifende Kompatibilität erweitert wurde. Die Entwicklung erfolgte nach agilen Methoden mit kontinuierlichem Feedback durch regelmäßige Meetings mit der LMP, um sicherzustellen, dass das System realen Anforderungen gerecht wird. Zu den zentralen Funktionen zählen ein sicheres, rollenbasiertes Login-System, eine benutzerfreundliche Registrierung per E-Mail, die Erstellung von Umfragen mit anpassbaren Fragetypen – darunter Single-, Multiple- und Weighted-Choice-Fragen –, die Erhebung demografischer Daten sowie verschiedene Wahlmodi wie anonyme, offene und öffentliche Abstimmungen. Darüber hinaus wurden Barrierefreiheit und responsives Design gezielt berücksichtigt, um eine breite Nutzerbasis zu unterstützen. Die Plattform wird unter der MIT-Lizenz veröffentlicht, um Transparenz, Wiederverwendbarkeit und Weiterentwicklung durch andere Organisationen zu fördern, die eine digitale Abstimmungsplattform benötigen.

Table of Contents

Chapter 1

Introduction

1.1 Description of performed work

The aim of this thesis project was to develop an open-source web application that enables organizations to create and mange secure online polls for their members. The platform is designed to be flexible and reusable, so that any group or association interested in running online votes, can benefit from the developed system.

The Authors of the thesis partnered with Liste Madelein Petrovic(LMP), a political party in Austria, who expressed the need for a platform that offers features not currently available in existing solutions. Their input was especially valuable, as it allowed the team to tailor the application to the real-world needs of a user organization.

The application includes core features such as poll creation, vote casting, result calculation and a group system. While under active development, the source code is published in a public GitHub repository under the MIT License, ensuring transparency and long-term reusability. [**opensourceinit**]

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.

1.2 Methodology of the thesis

This chapter outlines the methodological approach taken during the planning and development of the open-source voting web application. The work was structured around agile software development principles, with a Scrum-inspired workflow that emphasized flexibility, continues feedback and iterative improvement. The development period spanned from early November 2024 to mid-April 2025 with small improvements afterwards, following a planning phase that began in September 2024.

1.2.1 Planning and organization

The organizational backbone of the project was a GitHub Project board, which was used to manage tasks in the form of issues. Each development sprint was aligned with concrete short-term goals, allowing the team to maintain focused and manageable workload. Some issues were assigned to individual team members, while others, especially more complex features, were shared among multiple developers. The initial planning phase also included the design of the database schema and the selection of a suitable technology stack.

After careful evaluation, the team chose a backend architecture based on Node.js, Express and Sequelize in combination with a PostgreSQL database. The frontend was built with React and enhanced with Progressive Web App (PWA) capabilities. Before feature development began, the team implemented a basic end-to-end data flow, from frontend input to backend processing, to establish a shared understanding of the system's architecture. Features were then modularized into distinct components to enable parallel development. For example, the single-choice voting feature was split into poll creation, poll display, voting, and result visualization.

1.2.2 Internal and External Communication

To support ongoing coordination, the development team held weekly internal meetings via Discord. These meetings served to review current progress, identify technical or organizational blockades and plan subsequent tasks. Each session was documented in the form of written meeting protocols ensure traceability and clarity.

In addition, the team met with representatives of Liste Madelein Petrovic (LMP) approximately every two to three weeks via Google Meet. These meetings focused on presenting development progress, discussing evolving requirements and jointly prioritizing upcoming tasks. Although the GitHub repository was publicly accessible, the management of the GitHub Project board remained in the hands of the development team. However, during meetings with LMP, open issues were reviewed together and priorities were adapted based on their feedback.

1.2.3 Feedback and testing

One member of LMP regularly tested the application independently and provided direct feedback to the development team. Communication about setup or encountered issues was conducted via a Signal group. This informal yet effective feedback loop allowed the team to adjust functionality in response to user stories. Feedback was especially valuable in evaluating usability and identifying edge cases that might not have been apparent during internal testing.

1.2.4 Benefits and challenges

The adopted methodology brought several advantages. The regular meeting schedule allowed the team to detect and resolve issues early and adapt flexibly to changing requirements. The transparent task management via GitHub enabled clear structuring and prioritization of tasks, which improved project oversight and goal alignment.

However, the collaborative nature of the project also introduced challenges. In some cases, coordination between the development team and the client led to misunderstandings regarding specific requirements or expectations. These were generally resolved through clarification in subsequent meetings, but they underscored the importance of clear and continuous communication in stakeholder-driven projects.

Chapter 2

Tech Stack

2.1 PostgreSQL

2.1.1 General

PostgreSQL was selected as the relational database management system for this project due to its balance of performance, reliability, and advanced feature support. Unlike alternatives such as MySQL or SQLite, PostgreSQL offers robust support for complex data types, full ACID compliance and extensibility through custom functions and indexing methods. ACID refers to four key properties of reliable database transactions: Atomicity (ensuring all steps of a transaction are completed or none at all), Consistency (guaranteeing that data remains valid before and after transactions), Isolation (preventing concurrent transactions from interfering with each other) and Durability (ensuring that once a transaction is commited, it remains in the system even in the case of a crash). [**postgresdocs**] [**acid**]

While MySQL is widely used, its default configuration prioritizes speed over strict transactional consistency, which may be a drawback for applications requiring high data integrity. SQLite was excluded from consideration because it is more suitable for lightweight or embedded applications and does not support concurrent access by multiple users efficiently. [**postgresvsMysql**]

The installation was also simple because of the detailed install guide provided by multiple sites like 'w3schools.com'. To create the first database the SQL shell of PostgreSQL was used in the beginning. But due to its excellent tooling support, the open-source tool pgAdmin 4 was later used, because it provided a powerful and user-friendly interface for managing and visualizing the database. Furthermore, the availability of extensive official documentation and community support made it easier to troubleshoot and adapt the database to the project's needs. [**pgAdmin**] [**pgsinstallation**]

2.1.2 Database Schema

The initial design of the database schema evolved significantly during the project as client requirements changed. However, the "Polls" and "Users" tables were identified early on as the foundational entities in the system, with many other tables referencing or depending on them.

To design and visualize the schema, tools such as 'draw.io' and 'dbdiagram.io' were used, enabling direct import of SQL scripts for entity-relationship diagrams. The final scheme consists of 17 interrelated tables, designed to support complex relationships between users, polls, quesions and answers. Key intermediary tables such as "UserPolls", "UserAnswers", "PollGroups" and "QuestionAnswers" implement many-to-many associations between core entities. In addition to standard poll-specific questions, the schema includes support for reusable public questions and answers through tables like "PublicQuestions", "PublicAnswers" and their connectors. Group and role management is implemented via the "Groups", "UserGroups" and "Roles" tables, enabling structured access control. The inclusion of metadata fields such as timestamps ensures extensibility and auditability across the entire system.

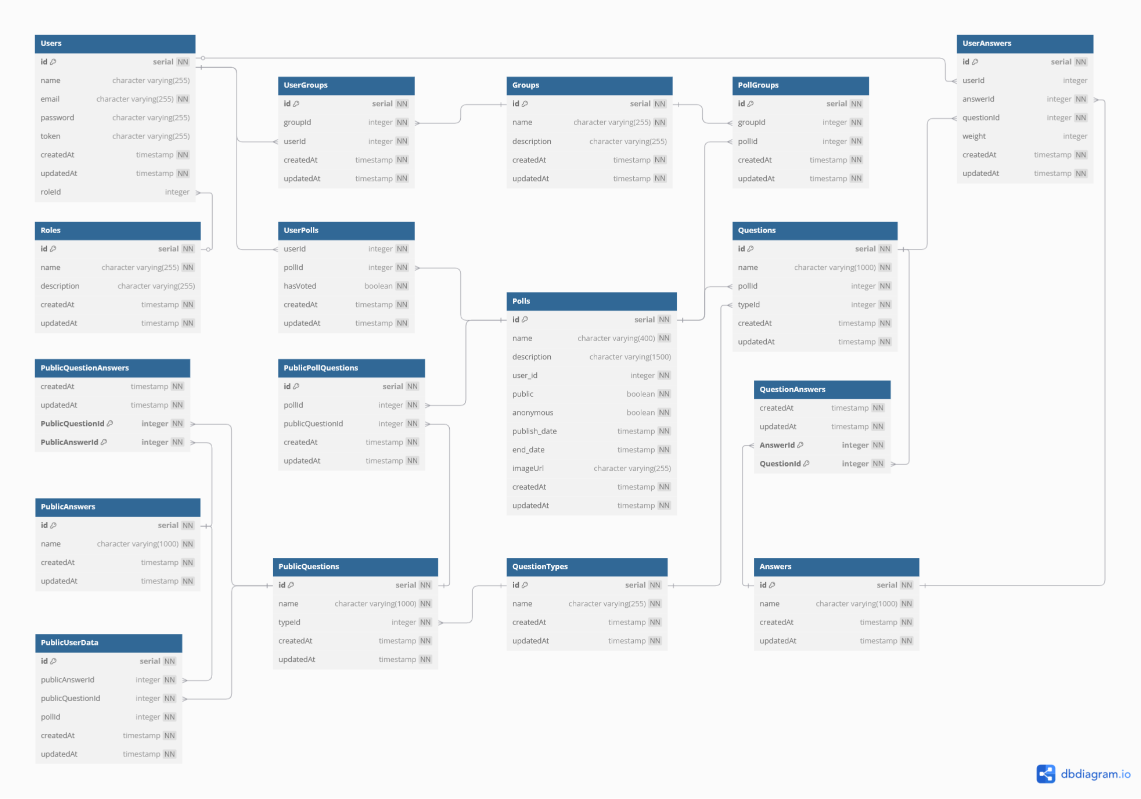


Figure 2.1: Entity Relationship Diagram

2.2 Node.js

For the backend Node.js was used. It is a JavaScript runtime environment, that allows the programming language to run on a server instead of the browser. Node.js is based on the V8 engine developed by Google, which compiles JavaScript into machine code for faster execution. [**nodejs-intro**]

2.2.1 Advantages

One of the main advantages of Node.js is its non-blocking I/O model. This means it does not wait for certain processes, such as reading from a file or database, to finish before continuing with other operations. Instead, it starts a process and immediately listens for the next event. When the initial process finishes, a callback function is executed. This makes it possible for Node.js to handle multiple operations concurrently, even though it is inherently single threaded. This results in high performance and responsiveness.

[nodejs-nonblocking]

Another advantage of Node.js is that it creates a unified language environment because both the frontend and the backend can be programmed in JavaScript. This simplifies the development process, improves code reusability, and reduces the learning curve, since developers do not need to learn a new programming language for the backend.

Node.js also has the npm (Node Package Manager) which provides access to a vast collection of reusable packages and modules. These include prebuilt code solutions to simplify common requirements like encryption and also sending emails, both of which were used in this project. [**mdn-express-nodejs**]

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 a 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.

When compared to other ORMs like Prisma, Sequelize offers a mature and stable solution with wide database compatibility and a strong community. While Prisma provides a modern, type-safe approach with auto-generated types, Sequelize's straightforward API and extensive features make it a reliable choice for this project, because of its simplicity and flexibility. [**Ormcomparemedium**] [**Ormcomparedhiwise**]

2.3.1 Sequelize Seeders

In Sequelize, seeders (seed files) are scripts used to populate database tables with initial or sample data. Each seed file typically exports **up** and **down** functions that perform bulk inserts and deletes. Seeders are especially helpful during development and testing phases to ensure consistent and predictable database states. [**sequelizemigrations**]

In this project, the Sequelize-CLI tool was used to generate and execute seeders. For example, the command **npx sequelize-cli seed:generate –name demo-user** generates a default seed file as seen in the figure 2.2. The file is named using timestamp format like `20250516163426-demo-user.js` to guarantee the correct execution order. To insert data, the command **npx sequelize-cli db:seed:all** was used, while **npx sequelize-cli db:seed:undo:all** removed the data from the database.

```
'use strict';
/** @type {import('sequelize-cli').Migration} */
module.exports = {
  async up (queryInterface, Sequelize) {
  },
  async down (queryInterface, Sequelize) {
  }
};

```

Figure 2.2: Generated seeder structure

2.3.2 Used Seeders

The seed data was created in the following sequence:

1. **Roles:** Three roles were inserted, admin, poweruser and normal, each with a description to reflect different leves of access. These were seeded first to to satisfy foreign key constraints in later data.

2. **Questiontypes:** The types `Single Choice`, `Multiple Choice` and `Weighted Choice` were inserted. These were essential foundational entries required before any question could be created in the database.

3. **User:** A default administrator user was seeded. The password was securely hashed using bcrypt. and the user was associated with the admin role, allowing login access to the application.

4. **Group:** A single group named *Admin Group* was created, which could then be associated with polls. The seeder for this data inserted the group in the corresponding table and a entry into the "UserGroups" table, as shown in Figure 2.3 to establish the relationship to the admin user.

5. **Test Poll:** One sample poll was created, including metadata such as publication dates and visibility settings. This poll featured six questions covering all seeded question types. Each question had associated answers. Additionally, the intermediate tables "QuestionAnswers" and "PollGroups" were populated to establish the correct relationship between questions, answers and groups.

```
// seeders/20250317-seed-group.js
module.exports = {
up: async (queryInterface, Sequelize) => {
  const group = await queryInterface.bulkInsert('Groups', [{
  name: 'Admin Group',
  description: 'This is a test group created by Seeder',
  createdAt: new Date(),
  updatedAt: new Date()
  }], { returning: true });
  const groupId = group[0].id;
  await queryInterface.bulkInsert('UserGroups', [{
  groupId: groupId,
  userId: 1,
  createdAt: new Date(),
  updatedAt: new Date()
  }]);
},
down: async (queryInterface, Sequelize) => {
  await queryInterface.bulkDelete('UserGroups', null, {});
  await queryInterface.bulkDelete('Groups', null, {});
}
};

```

Figure 2.3: Seeder for groups and user-group assignment

The purpose of this seeded data was to simulate realistic application scenarios, support frontend development and allow for consistent testing. Automating the seeding process improved reproducibility across development environments and enabled immediate access to meaningful application data during testing and debugging.

2.4 Express.js

Express.js is a flexible web framework for Node.js which is widely used for building RESTful APIs. It provides a middleware-based architecture that allows for clear routing, request handling, and modular code organization. In this application. Express serves as the backend framework that handles HTTP requests to handle communication between the frontend and the database. [**expr-base**]

There are multiple alternative frameworks that were evaluated in the selection process:

- **NestJS** provides a structured and scalable framework based on TypeScript. The ideal use for NestJS would be a large enterprise project, which this thesis is not. This adds complexity and a way steeper learning curve, which is why this option was not chosen. [**expr-comp**] [**expr-comp2**] [**nest-js**]

- **Fastify** is a performance-orientated alternative to Express.js, which even outperforms it. However, since this framework is less mature, there is less documentation and a smaller community for support in comparison to what Express.js is offering. Since the developers of this thesis had no real experience with APIs the more developer-friendly Express was chosen over Fastify. [**expr-comp**] [**expr-comp2**]

- Other options were the following two. Hono, which is a new ultra-lightweight framework, but is still niche and lacks in support and documentation, which is why Express was favored. FeatherJS is built on Express, designed for real-time applications. Since real-time features were not needed for this platform, FeatherJS was not chosen. [**expr-comp**] [**expr-comp2**]

2.4.1 Architecture

The backend is organized into the following layers:

- **Routing Layer:** This layer defines the application's HTTP endpoints and maps incoming requests to the appropriate controller functions. Routes are grouped thematically using **express.Router()** to maintain modularity and logical organization.

- **Controller Layer:** Controllers serve as intermediaries between the routing layer and the business logic. They are responsible for extracting relevant data from incoming requests, invoking the necessary service functions, and constructing structured responses. This ensures that request handling remains concise and focused.

- **Service Layer:** This layer contains the main application logic and performs database operations using Sequelize. Each service function handles one use case, such as creating a user or poll. Data access is performed directly within the service layer.

2.4.2 Error Handling

In the Express.js backend, the error handling for the application is integrated directly into the controllers. The controller interacts with the service layer and handles HTTP requests.

```javascript
const editController = {
  async updatePoll(req, res) {
    try {
      const updatedPoll = await editService.updatePoll(req.body);
      res.json(updatedPoll);
    } catch (error) {
      res.status(500).json({ error: error.message });
    }
  },
};

```

Figure 2.4: Example of error handling in controller

In the code above, **updatePoll** attempts to handle errors using a try/catch. When an error occurs during the execution of the updatePoll service function, the error is caught in the catch block. The server then responds with a `500 Internal Server Error` status, including the error message in the response body. This ensures that the client is informed about the failure. This is the most basic form of error handling and can be improved in the future e.g. by using specific errors like `400 Bad Request` or `404 Not Found` instead of the inaccurate `500 Internal Server Error` to give a more detailed description to the client.

2.4.3 CORS

In the application, Cross-Origin Resource Sharing (CORS) is managed using the cors middleware in Express.js. CORS is a security feature implemented by web browsers that controls how web pages from one domain can make requests to a backend server hosted on another domain. It is particularly important when the frontend and backend are hosted on different origins, such as when they are running on different ports during development or on different subdomains in production.

In this case, CORS is configured with the default settings, meaning that the backend allows cross-origin requests from any domain. This default setup permits all HTTP methods (like GET, POST, PUT, DELETE, etc.) and headers from any origin, making it very permissive. This configuration is helpful during the development phase, as it avoids the need for specifying which domains or methods are allowed.

However, while the default configuration is useful in development, it is not recommended for production environments due to security concerns. Allowing any origin to interact with the backend can expose the server to potentially malicious requests from untrusted sources. It is generally better to restrict CORS to specific trusted domains and limit the allowed HTTP methods and headers to only those necessary for the application. Therefore, this should be improved in the future. [**mdn-cors**]

2.5 React.js

React.js is a declarative. component-based JavaScript library, which is widely adopted in modern web development due to its modular structure and virtual DOM optimization. In this application, React is used as the frontend framework responsible for rendering the user interface, managing local and global state, and communicating with the backend API. Several alternative frameworks were taken into consideration during the selection process. [**react-docs**]

Vue.js offers a simple and intuitive syntax that would lower the learning curve for new developers. However, since React has a larger ecosystem, widespread community support and a better compatibility with third-party libraries, it was chosen over Vue. [**react-docs**] [**vue-docs**]

Angular is a comprehensive frontend framework with built-in support for routing and forms. While powerful, Angular is way more complex and is harder to learn which is why React was preferred for the scope of this project. [**angular-docs**]

Besides these two, smaller frameworks like Svelte were also considered, but due to smaller communities and less comprehensive documentation they were not chosen. Another reason why React was the preferred framework is because the developers were already familiar with the Syntax and used it for small exercises on various platforms like 'freecodecamp.org'.

2.5.1 Architecture

React.js is based on a component oriented architecture, where the user interface is broken down into small, reusable building blocks called components. Each of these is typically a function that returns a declarative description of how the UI should look like, using JavaScript XML (JSX). Every component also manages its own internal logic and state, responds to user interaction and renders the content dynamically.

React components receive configuration and data through props, which are passed from their parent components. This allows for a unidirectional data flow and promotes component reusability. In addition to props, components can manage internal dynamic data using React's Hooks API. Hooks are special functions that enable state management and side effect handling in functional components capabilities that were previously only available in class components. [**react-docs**] [**mdn-react**]

The most commonly used hooks are **useState** and **useEffect**. The **useState** hook allows components to declare and update local state variables, while **useEffect** lets developers perform side effects such as data fetching, DOM manipulation, or setting up subscriptions. For example, **useEffect** can be configured to run after the component mounts or whenever specific values change. Beyond these, React provides a range of built-in hooks such as **useContext** for accessing context values, **useReducer** for complex state logic, and **useRef** for referencing DOM elements or persisting values across renders without causing re-renders. Overall, hooks offer a more expressive and modular approach to handling component logic, encouraging better separation of concerns and improved code maintainability. [**react-hooks**]

2.6 PWA

“A progressive web app (PWA) is an app that’s built using web platform technologies, but that provides a user experience like that of a platform-specific app.” [**mdn-pwa**]

This application is PWA, compatible. PWAs combine the best features of both websites and platform-specific applications. [**mdn-pwa**]

2.6.1 Platform-Specific

Platform-Specific applications are developed only for one particular operating system in one the supported programming languages, which means that most of the code cannot be reused for other systems, because they most likely require a different programming language. The most common target platforms for such applications are Android and IOS.

Because these applications are tailored to a specific operating system, they can fully utilize the device’s hardware and software capabilities. This increases the applications performance and user experience as well as enabling more customization and personalization. [**native-apps**]

Additionally, platform-specific applications can also run in the background, when they are not being interacted with. This enables them to perform tasks such as automatic updates or background synchronization. This helps to keep the application up to date, even when the device is offline. [**mdn-pwa**]

2.6.2 Websites

Unlike platform-specific applications, websites are not restricted to a single operating system. They are accessed through a browser and not installed on the device. Therefore they appear and function the same across all platforms. [**mdn-pwa**]

However, websites have limited access to platform-specific features and also do not showcase the same level of performance. They also do not provide offline functionality, but rely heavily on network connectivity.

2.6.3 Function PWA

PWAs are essentially websites, but they also offer important features typically unique to platform-specific applications. They can be installed to the device directly on a device from the browser and can even be published to the native app store. Once installed, they appear like native applications as they have an app name and icon, and can run in a standalone display. This is made possible with the manifest. PWAs can also provide offline functionalities through a service worker, however this feature is not a major focus in this project. [**mdn-pwa**]

Manifest

The manifest makes it possible to install the PWA from the browser. It is a json file and must contain information about the name, icon, starting url and display. It can also include many additional configuration options like the background color and theme color. [**mdn-pwa-installation**]

```
{
  "short_name": "Survey-Tool",
  "name": "Survey-Tool - LMP",
  "icons": [{
    "src": "favicon.ico",
    "sizes": "64x64 32x32 24x24 16x16",
    "type": "image/x-icon"
  },{
    "src": "logo192.png",
    "type": "image/png",
    "sizes": "192x192"
  },{
    "src": "logo512.png",
    "type": "image/png",
    "sizes": "512x512"
  }],
  "start_url": ".",
  "display": "standalone",
  "theme_color": "#000000",
  "background_color": "#ffffff"
}

```

Figure 2.5: This project's manifest file

Chapter 3

Features

3.1 Login

The login feature is essential for the 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, user-friendly error handling was implemented 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 the `/api/login` API route that forwards the data to the **handleFetchLogin** function in the `userController.js`. 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. It was intentionally chosen not to 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

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. They 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 the following 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 routed to the **sendEmail** function in the backend through the `/api/email` API route using 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 `userController.js` file via the `/api/user` API route using a PUT request. The password is sent in plain text over HTTPS, as explained in the login section above.

The controller 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 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, two different approaches were implemented. 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, the secure option and also authentication credentials.

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

```

Figure 3.1: Example transporter using mailersend as the SMTP server

After setting up the transporter, the email options are defined. These include the sender and recipient addresses along with the subject and the emails general content, as shown in Figure 3.2. 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: Example email options for transporter

MailerSend API

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

To use the MailerSend API, a transporter is created with, like in Figure 3.3. Just like int the first method mail options are created specifying the sender, recipient, subject and message, as shown in Figure 3.4. Then the email is sent using **mailerSend.email.send(mailOptions)**.

```
const mailerSend = new MailerSend({
  apiKey: process.env.MAILERSEND_API_KEY,
});

```

Figure 3.3: Example MailerSend transporter

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

```

Figure 3.4: Example MailerSend email options

3.2.4 Roles

In the application, each user record includes a *roleId* that indicates the user's assigned role. This reflects the Role-Based Access Control (RBAC) model in which "each user is assigned one or more roles, and each role is assigned one or more privileges". [**rbac**]

In this project three roles with increasingly broader permissions were defined:

1. **Admin (roleId 1:)** Can register users, create and manage polls and view all results.
2. **Poweruser (roleId 2:)** Can create and manage polls and vote, but cannot manage users.
3. **Normal (roleId 3:)** Can only view available polls and vote on them.

The React front-end uses conditional rendering so that UI components appear only if the current user's role permits them. For instance, an Admin user will see the navigation element for registration that are hidden from Powerusers and Normals. Similarly, a Normal user will only see the voting and result option and not the administrative views. In this way the UI dynamically enforces the role permissions stored in the database, aligning with the RBAC principle of tying access decisions to user roles. [**rbac**]

Implementing a role-based system with three distinct roles is crucial for the functionality and security of the application. By assigning permissions flexibly, a clear hierarchy is established, enhancing both user experience and data integrity. This structure facilitates efficient task delegation and 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 managed exclusively within the database. Role-based access is determined solely based on these database-level associations.

Advantages and Limitations of RBAC: RBAC is widely adopted because it simplifies access management in large systems. By aggregating permissions into organizational roles, RBAC "provides a powerful mechanism for reducing the complexity, cost and potential for error" in assigning user privileges. In practice, studies have shown that RBAC can yield significant efficiency and security benefits: a NIST analysis found that implementing RBAC led to savings from reduced downtime and more efficient provisioning and policy administration, in addition to the security gains. At the same time, RBAC has recognized limitations. In particular, defining a good set of roles up front can be difficult, especially in dynamic enviroments. Researchers note that a pure RBAC scheme may be inflexible to context-dependent rules and may suffer a "role explosion" if many narrowly-scoped roles must be created to cover all cases.. Consequently, purely role-based systems can struggle with fine-grained or rapidly changing access requirements. For these reasons, practitioners sometimes augment RBAC with additional models when highly dynamic or context-sensitive policies are needed. In this project, however, a group-based model was sufficient to meet the applications requirements.

For these reasons, practitioners sometimes augment RBAC with additional models (such as attribute-based controls) when highly dynamic or context-sensitive policies are needed. In this project, however, a group-based model was sufficient to meet the application's access control requirements. [**rbac**]

3.3 Group System

The group system was implemented to simplify user management. By grouping users together, poll creators can efficiently control access to polls. Instead of managing individual users, a poll owner can assign entire groups to a poll, ensuring that only selected users can participate in voting.

Beyond restricting access, the system also allows assigning different permission levels. Users with elevated privileges, such as power users or administrators, can be granted editing rights, enabling collaborative management of polls. Additionally, poll results are only visible to users belonging to groups that have access to the poll.

To support the group system, a new navigation option was added to the interface. The page for creating groups is shown in Figure 3.5. At the top of the page, a toggle button labeled *Gruppen Bearbeiten* allows switching between group creation and editing. Below that, users can enter a group name and provide a short description explaining the group's purpose and membership.

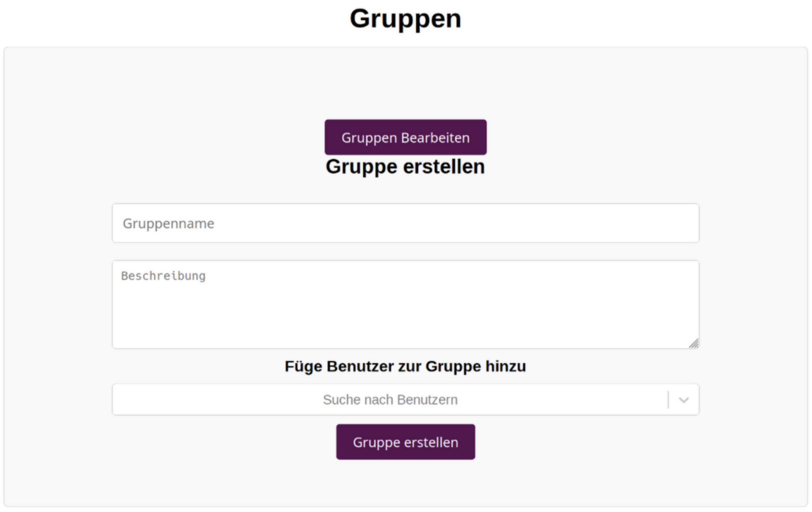


Figure 3.5: Create Groups Page

At the bottom of the form, a React Select component is used to assign users to the group. The group editing interface is visually similar to the creation interface, with the addition of a group selection dropdown and a removal option for existing members. An additional feature of the editing interface is the ability to export group members, allowing external tools to be used for planning and organization.

3.3.1 Backend

To support the group functionality on the backend, three new database tables were introduced:

- "UserGroups": Represents a many-to-many relationship between "Users" and "Groups". It stores user assignments to groups and is essential for frontend group management.
- "Groups": This table is the core of the group system, storing group metadata such as name and description.

- "PollGroups": Establishes a many-to-many relationship between "Groups" and "Polls". It defines which groups have access to which polls and is directly used in the poll creation and editing workflows.

When displaying polls to a user, the system queries all groups the currently signed-in user belongs to and checks whether any of those groups are assigned to a poll. This determines the user's access to view or interact with a poll.

3.3.2 Frontend

The frontend for the group system on the group page includes several key elements already mentioned:

- **Toggle Mode:** A button labeled *Gruppen Bearbeiten* allows switching between group creation and editing modes without navigating away from the page.
- **Form Inputs:** The interface includes input fields for entering the group name and description. These are controlled components in React, ensuring real-time validation and state management.
- **User Selection:** The user assignment is implemented using a React Select component, which allows searching and selecting users from a dropdown. In editing mode, an additional select dropdown is displayed to remove existing users from the group.
- **User Export:** In editing mode, an export function is available, allowing the list of group members to be downloaded. This can be useful for external planning or documentation.

In addition to group management, a React Select component is also integrated into the poll creation and editing sections. Here, users with editing privileges can assign groups to a poll or remove them. This ensures that only selected groups have access to vote or view the poll.

Overall, these features provide a smooth and efficient user experience while significantly reducing the administrative effort required to manage poll access.

3.4 Create Polls

The Create Polls section represents the poll creation interface of the system. It allows users to create customized polls based on their needs. Customization options include entering a poll name and description, selecting different poll types, assigning the poll to specific groups, defining a start and end date for the voting period, uploading an image and dynamically adding questions with chosen question types and a variable number of answers per question.

3.4.1 Error Handling

To ensure that polls are created correctly and without missing information, the Create Polls section includes validation in the frontend and transaction in the backend.

Frontend

In the frontend, validation is handled by the **validatePollData** function located in a separate file, `ValidatePolls.js`. This function checks wether any required input fields are left empty. If a required field is empty, a specific message about which field is afflicted is returned, which is then displayed to the user. This improves the user experience and also prevents unnecessary API calls to the backend.

```
validatePollData(poll, publishDate, endDate, questions) {
  if (!poll) return `Poll name is required`;
  if (!publishDate) return `Publish date is required`;
  if (!endDate) return `End date is required`;

  for (let questionIndex = 0; questionIndex < questions.length; questionIndex++) {
    if (!questions[questionIndex].name) return `Question ${questionIndex + 1} requires text`;
    for (let answerIndex = 0; answerIndex < questions[questionIndex].answers.length; answerIndex++) {
      if (!questions[questionIndex].answers[answerIndex].name) return `Answer ${answerIndex + 1} in Question ${questionIndex + 1} requires text`;
    }
  }
  return null;
}

```

Figure 3.6: Frontend validation

Backend

Even though validation already occurs in the frontend, the backend still needs to ensure that all database insertions are executed correctly. This is handled in the **createPoll** function. To maintain data integrity, database transactions are used.

A transaction is created and used for all data operations involved in poll creation, including inserting the poll, its questions and all corresponding answers. Once all operations are completed successfully, the transaction is committed. If any error occurs during the process, the transaction is rolled back, undoing all previous changes made during that transaction. This ensures that no incomplete data is saved into the database.

```
    try {
      const transaction = await sequelize.transaction();
      const createdQuestion = await Question.create({
        name: question.name,
        pollId: createdPoll.id,
        typeId: questionType.id,
      }, { transaction });
      await transaction.commit();
    } catch (error) {
      await transaction.rollback();
    }

```

Figure 3.7: Example backend validation

3.4.2 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 selection. 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

These time-based conditions are the core feature used to differentiate between the various states of a poll: edit, voting, and results. To apply these conditions, three arrays are created - each corresponding to one of the states - and passed as props to their respective React components. Figure 3.8 shows the code that performs this classification. The `data` variable contains all polls retrieved from the database via an API call. The `current_datetime` is set using **new Date().toISOString()**. The function iterates through each poll, comparing its `publish_date` and `end_date` to the current time.

A poll is:

- pushed into the `edit` array if the current time is earlier than its `publish_date`,
- added to the `vote` array if the current time is between `publish_date` and `end_date` and the poll is not public,
- moved to the `results` array if it is public and its `end_date` has already passed.

An important detail is how public polls are handled. Since they are not intended to appear in the standard voting view, they are excluded from the vote array. In contrast, public polls may appear in the results view even before their end date, allowing creators and interested parties to observe the vote development in real time.

```
data.forEach((poll) => {
  if (poll.publish_date > current_datetime) edit.push(poll);
  else if (poll.publish_date <= current_datetime
  && poll.end_date >= current_datetime
  && poll.public === false) vote.push(poll);
  else if (poll.public === true && poll.end_date < current_datetime) results.push(poll);
})

```

Figure 3.8: Sorting the polls depending on time

3.4.3 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**]

```
const storage = multer.diskStorage({
  destination: (req, file, cb) => {
    cb(null, 'uploads');
  },
  filename: (req, file, cb) => {
    cb(null, Date.now() + path.extname(file.originalname));
  }
});

```

Figure 3.9: 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:

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

```

Figure 3.10: 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.

```
const handleImageUpload = async (event) => {
  const file = event.target.files[0];
  if (!file) return;
  const previewUrl = URL.createObjectURL(file);
  setImage(previewUrl);
  const formData = new FormData();
  formData.append("image", file);
  try {
    const res = await fetch(`${process.env.REACT_APP_API_URL}/api/upload-image`, {
      method: 'POST',
      body: formData,
    });
    const data = await res.json();
    if (res.ok) {
      setImageUrl(data.imageUrl);
    }
}};

```

Figure 3.11: 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 feedback to the user by displaying the image in the form before it is uploaded.

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 `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.

3.4.4 Questions

To support a variety of survey use cases, the system offers three types of questions: single choice, multiple choice, and weighted choice. Each type serves different purposes and allows for more flexible poll creation depending on the information being collected. The differences and use cases for each question type are described in the following section.

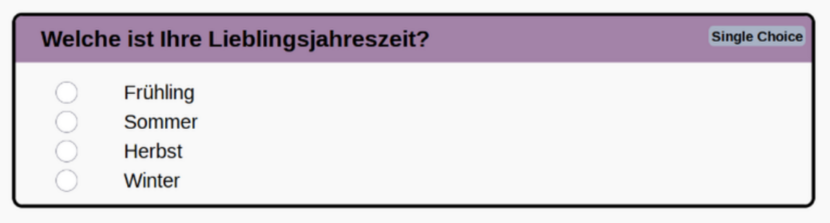
Single Choice

Single choice questions are the simplest type of question in the system. They allow users to select exactly one option from a list of possible answers. One major benefit of this question type is the minimal time required to answer. Deciding on a single option is usually faster than selecting multiple answers or assigning weights.

Another advantage is the simplicity of data analysis. Since each user provides only one answer, the resulting data is easier to aggregate and interpret compared to other question types. This format is particularly effective when the poll creator wants to highlight the most important user preferences. A single choice question forces participants to choose the most relevant option, thereby focusing the collected data on the most impactful areas.

However, limiting users to one response can also lead to incomplete or skewed data, especially in cases where the question is ambiguous or multiple answers would apply. In such scenarios, forcing a single choice may result in user frustration or less accurate feedback. [**singlevsmultiple**]

Figure 3.12 shows an example of a single choice question in the voting section. The options are presented using radio buttons, ensuring that only one answer can be selected at a time.



Welche ist Ihre Lieblingsjahreszeit?

○ Frühling
○ Sommer
○ Herbst
○ Winter

Figure 3.12: Single Choice Vote

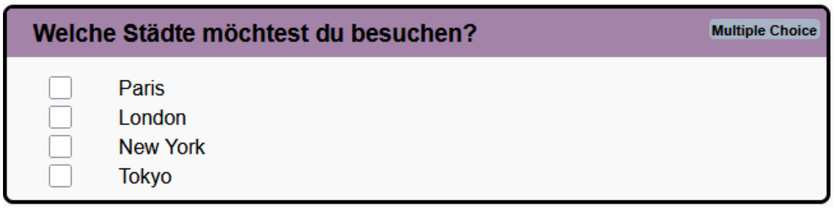
Multiple Choice

Multiple Choice questions, also known as multiple response questions allow users to select one or several answers from a predefined list of options. This format offers flexibility in collecting diverse user options, as participants are not restricted to a single choice. It is especially usefull when multiple options may be valid or relevant from a participant's perspective.

One key advantage of multiple choice questions is their ability to reflect the complexity of real-world decision-making. Participants are not forced to prioritize a single answer, which can improve satisfaction and reduce frustration in situations wherer limiting to one choice would feel overly restrictive. Additionally, allowing multiple ansers can help surface trends or common combinations of opinions that would be lost in single-choice formats. [**singlevsmultiple**]

Even so, this question type also introduces certain challenges. From a data analysis standpoint, interpreting multiple selections can be more complex, particularly when attempting to identify the most influential or decisive factors. There is also a risk of over-selection, where users may choose all options to avoid making a decision, which can dilute the clarity of the result. Furthermore, response time may be longer, as users take more time to evaluate all possible answers. [**singlevsmultiple**] In the application's voting interface, multiple selections are implemented by enabling checkboxes for each answer option. This allows users to select any combination of responses, while still enforcing that at least one option must be selected to proceed.

Figure 3.13 illustrates how multiple choice questions are presented in the voting interface, highlighting the use of checkboxes to support flexible input.



Welche Städte möchtest du besuchen?

Multiple Choice

☐ Paris
☐ London
☐ New York
☐ Tokyo

Figure 3.13: Multiple Choice Vote

Weighted Choice

The `Weighted Choice` question type implemented in this application is a variation of the Likert scale. It combines a basic `Single Choice` question with an additional importance scale. This way users can assign a weight to the answers they have given.

Likert scales are commonly used to measure opinions or attitudes by offering a range of answer options from "strongly disagree" to "strongly agree". This type of question lets people show not just what they think, but also how strongly they feel. The Likert scale is helpful because it turns personal opinions into numbers that can be analyzed. As a result, it becomes easier to decide whether an issue should be prioritized or not.[**likert-scale**] In this application, the importance scale is implemented as a custom-designed component consisting of nine buttons, each representing a value from one (least important) to nine (most important). When a button is clicked, the selected value is passed to the parent component via the `handleSelectImportance` function, and the visual style of the selected button is updated to reflect the user's choice.

```
const ImportanceScale = ({ questionId, onImportanceChange }) => {
  const [importance, setImportance] = useState(null);
  const scaleValues = [1, 2, 3, 4, 5, 6, 7, 8, 9];

  const handleSelectImportance = (value) => {
    setImportance(value);
    onImportanceChange(questionId, value);
    console.log(value);
  };

  return (
  <div className="scale-container">
  <div className="button-container">
  {scaleValues.map((value) => (
    <button
    key={value}
    onClick={() => handleSelectImportance(value)}
    className={importance === value ? 'scale-button-selected' : 'scale-button'}
    >
    {value}
    </button>
    ))}
  </div>
  </div>
  );
};

```

Figure 3.14: Importance scale implementation

3.4.5 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 the polls creator would have the data already. If the user 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 3.15 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-choice` is not an option, since demographic data is more like a fact less an opinion.

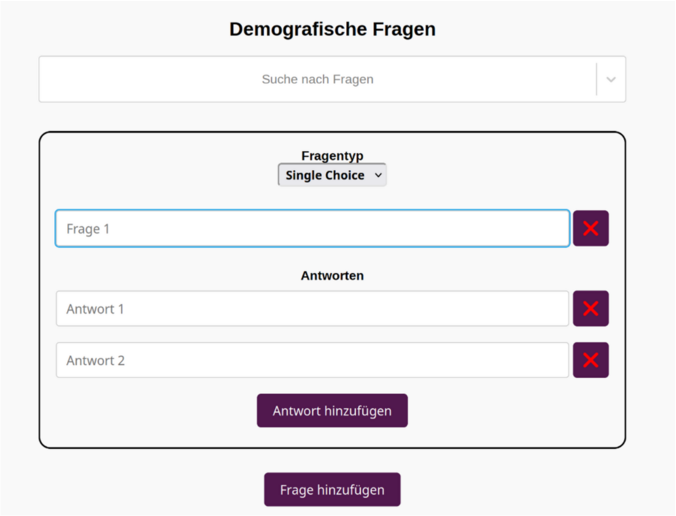


Figure 3.15: Create Demographic Question

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** 3.16 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**]

```
let [createdQuestion, created] = await PublicQuestions.findOrCreate({
  where: {
    name: question.name,
    typeId: questionType.id,
  }
});
```

Figure 3.16: findOrCreate PublicQuestions

3.5 Edit Polls

Editing a poll is essential for maintaining a smooth administrative workflow on the platform. Editing is only permitted before the poll is published, the publish time is already described in the creation section. Without this feature, power users and administrators responsible for creating polls would encounter significant limitations.

For instance, simple issues such as typos or unclear instructions could not be corrected, requiring the entire poll to be deleted and recreated. Additionally, the edit functionality enables collaborative poll creation: one user can create the initial poll and assign a group with editing privileges. These users can then revise the poll, allowing for a structured and collaborative workflow. Furthermore, setting the publish time to a future date ensures that the team has ample time to discuss and refine the poll before it goes live.

3.5.1 Backend

The backend for editing uses the same structural approach described in the creation section. However, specific logic is implemented to handle operations such as adding, updating, and removing elements of an already created poll.

The complete poll data is sent from the frontend via the API, and the backend separates and processes it accordingly.

Before applying any changes, the backend checks whether the poll has any recorded votes to ensure data integrity is preserved.

The process involves the following steps:

- Validating general poll data (name, description, start and end date).
- Fetching all existing questions from the database and comparing them with the updated data.
- Fetching the associated answers for each question and performing a similar comparison.

Adding To add new questions, the backend compares the existing questions with those in the received data. If a question does not already exist, it is created with its name and type. Then, each answer is added individually, followed by the corresponding "QuestionAnswers" entries.

Changing For existing questions and answers, the backend uses Sequelize's **update** functionality on each relevant entry. This ensures that unchanged elements remain untouched, while modified data is accurately updated.

Deleting To remove obsolete questions, the system combines all existing and newly added questions into a single array. This array is then compared against the incoming data. If a question exists in the database but not in the new data, it is deleted using Sequelize's **destroy** function.

3.5.2 Frontend

The frontend for editing polls reuses the layout and structure of the poll creation interface. The key difference is that a poll must first be selected for editing. Once selected, the poll data is fetched and pre-filled into the corresponding input fields.

The data is managed through a poll object, which is dynamically updated using various state manipulation functions depending on the type of edit being performed. Figure 3.17 shows an example where an answer is removed from a question. Depending on whether the poll is public or not, the system updates either the *publicQuestions* or the standard *questions* array. The answer is removed using JavaScript's **splice** method.

```
const deleteAnswer = (questionIndex, answerIndex, isPublic = false) => {
  if (isPublic) {
    const newQuestions = [...publicQuestions];
    newQuestions[questionIndex].PublicAnswers.splice(answerIndex, 1);
    setPublicQuestions(newQuestions);
  } else {
    const newQuestions = [...questions];
    newQuestions[questionIndex].answers.splice(answerIndex, 1);
    setQuestions(newQuestions);
  }
};

```

Figure 3.17: Example of manipulating the poll object in the frontend

3.6 Voting

Voting is a core feature of any survey application. To provide users with flexibility, the system supports multiple voting modes: disclosed, anonymous, and public voting. The advantages and differences between these modes are explained in the following section.

3.6.1 Disclosed Voting

In disclosed voting, each voter can see who voted for which option. The votes are stored in the "UserAnswers" table in the database. Here the corresponding *answerId*, *questionId* and *userId* are stored. 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 comprehensive analysis and reporting, making it easier to identify patterns, understand 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 contrast, anonymous voting may raise concerns about the system's integrity, since there's no way to independently verify the authenticity of each vote. [**disvsanon**]

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**]

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. Trust, benefits of the demographic data and the ability to abstain are important to a poll. All of these factors have to be taken into ones account when creating these questions. 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 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**]

Figure 3.18 shows how the backend handles the CAPTCHA request. The token, the site key, is sent from the frontend through the request body, while the secret Key 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 Googles response indicates how likely a user is human. Therefore, before returning a successful response to the frontend, the score is checked. The code handles the cases where the request results in an error or the score is too low. [**recaptcha**]

```
app.post('/verify-recaptcha', async (req, res) => {
  const { token } = req.body;
  const secretKey = process.env.RECAPTCHA_SECRET_KEY;
  const url = `https://www.google.com/recaptcha/api/
  siteverify?secret=${secretKey}&response=${token}`;
  try {
    const response = await fetch(url, {
      method: 'POST',
    });
    const data = await response.json();
    if (data.success && data.score > 0.5) {
      res.json({ success: true });
    } else {
      res.json({ success: false, message: 'Verification failed' });
    }
  }
});

```

Figure 3.18: reCAPTCHA backend

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:

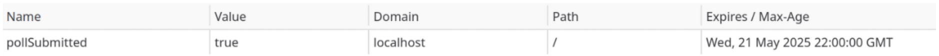
1. Session Management: Allows a website to remember user behavior and preferences across sessions.

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 3.19 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.



Name	Value	Domain	Path	Expires / Max-Age
pollSubmitted	true	localhost	/	Wed, 21 May 2025 22:00:00 GMT

Figure 3.19: Cookie stored in the browser

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 cookie was 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

Results are arguably the most important feature of the application. Here, the voting data is displayed in a comprehensible way. This section of the platform includes the display of results depending on the different voting types, disclosed and anonymous, as well as an export option for further analysis of the voting data.

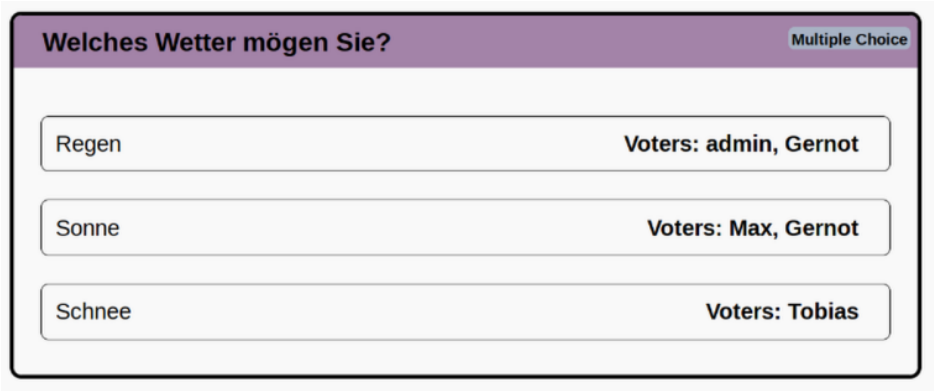
3.7.1 Backend

The API for results consists of two main components. First, the total number of voters is calculated. This is done by counting the entries in the "User-Polls" table using Sequelize's **count** function. This number represents the total number of unique users who have participated in a specific poll.

The second part handles the calculation of votes per question. For each question, the backend first determines the total number of answers submitted using the "UserAnswers" table. Then, for each answer within the question, it calculates the percentage of total votes that the answer received. These values are stored in an object named *questionVotes*, where each question id maps to another object containing answer ids and their corresponding vote percentages as strings formatted to two decimal places.
Weighted choice questions are also supported in the database and export. However, since each user can assign different weights to answers, it was not feasible to display this clearly in the frontend. Various methods were considered, such as showing the average or total weight per answer, but these either led to confusion or failed to reflect individual user input accurately. As a result, weight values are excluded from the visual results and are only available in the exported data.

3.7.2 Frontend

Figure 3.20 shows an example of a disclosed multiple choice question in the results view. For each question, the results section displays the list of possible answers along with the corresponding voting data. Depending on the polls settings, the view can either show the percentage of total votes each answer received or a list of the users who voted for each answer.



Welches Wetter mögen Sie?

Regen — **Voters: admin, Gernot**

Sonne — **Voters: Max, Gernot**

Schnee — **Voters: Tobias**

Figure 3.20: Example of result

The component uses two asynchronous API calls. The first fetches the overall voting results, using the API described in the backend part. The second fetches a list of voters per answer, if the poll is disclosed. This voter data is stored in an object where the keys are answer IDs and the values are arrays of usernames.

A toggle button allows switching between the two display modes: percentages and voter names. This functionality is controlled by a boolean variable named *showVotersMode*. When *showVotersMode* is enabled, the result view shows the vote percentage for each answer. When disabled, it lists the names of the voters for each answer, or displays a fallback message if no one voted for that answer.

The single-choice questions are displayed in the same format. Weighted choice questions are visually identical to multiple-choice ones, but their weight values are not shown due to the reasons mentioned earlier. Additional features include the optional display of the polls image and description if such data is available. This ensures that the result view remains consistent with the polls context.

3.7.3 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 Microsoft Excel. [**json2csv**]

3.8 MyPolls

3.8.1 Purpose

The MyPolls section is a central location in which 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 link used for sharing the polls is generated by encoding key information into a hash. This is done using the **btoa** function, which creates a Base64-encoded string. In order to ensure compatibility with the browser URLs, the hash is also encoded using the **encodeURIComponent** function. The generated hash contains information about whether the poll is public or anonymous, the poll's mode and its ID. [**mdn-btoa**] [**mdn-encodeUriComponent**]

When a poll link is opened, the hash is first decoded using **decodeUriComponen** and then **atob**. Depending on the mode specified in the url, the corresponding section in the dashboard is opened and the poll with the matching ID is loaded. Should the poll ID not exist in this specific mode, or in general, the system still opens the correct mode but in the default state without any poll selected. [**mdn-atob**] [**mdn-decodeUriComponent**]

To make sharing of the surveys easier, a QR-code generation was implemented. 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 transaction. This guarantees atomicity, meaning that either all steps of the deletion process succeed or none do - preventing partial data deletion and potential 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.

```
const deletePoll = async (pollId) => {
  const transaction = await sequelize.transaction();
  try {
    ...
    await PollGroups.destroy({
      where: { pollId },
      transaction,
    });
    await Polls.destroy({
      where: { id: pollId },
      transaction,
    });
    await transaction.commit();
    return { pollId, questionsDeleted: questionIds.length };
  } catch (error) {
    await transaction.rollback();
    throw error;
  }
```

Figure 3.21: Example for a Sequelize transaction

3.9 Accessibility

3.9.1 Tooltips

To enhance the usability and user experience of the application, tooltips were integrated into all input fields, 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 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.

3.10 Styling

3.10.1 General styling

At the beginning of the development process, a rough design was implemented to help visualize the overall concept of the future application. In order to maintain a clean and organized project structure, general styling files such as `App.css`, `dashboard.css`, `voting.css`, and others were placed within a dedicated styles directory.

The official 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". In addition to the existing project structure, custom components such as the header with navigation and the footer were introduced, each of which can be individually styled. These were integrated to establish a layout that conforms to standard design practices.

To ensure consistency of the palette user defined characteristics from CSS where used to assign variables for each main color. [**csscolorvariables**] Those variables where then used in each element of where it was needed.

```css
    :root {
      --primary-color: #51184e;
      --secondary-color: #F9BB03;
      --primary-hover-color: rgb(163, 131, 168);
    }

```

Figure 3.22: Variables of the main colors

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 the contents background 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.

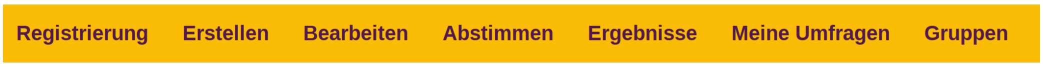
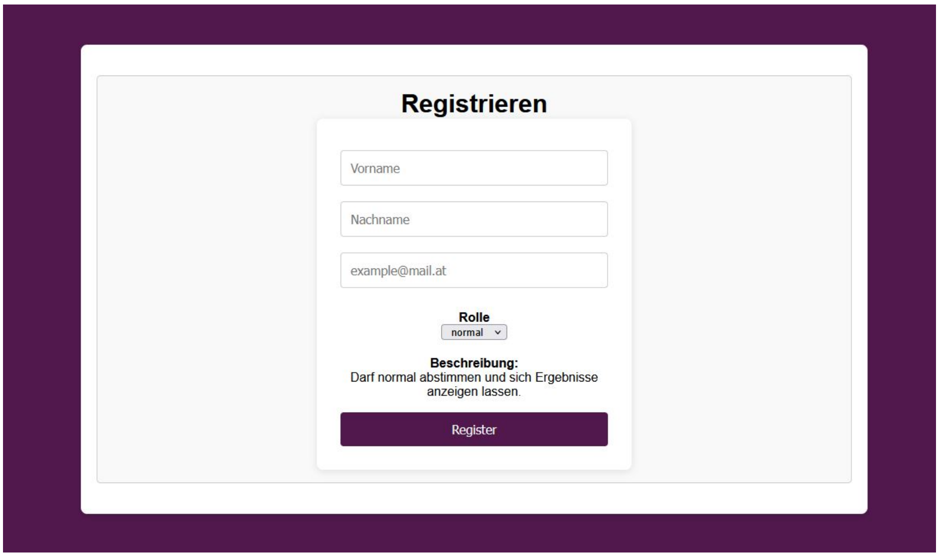


Figure 3.23: Navigation

3.10.3 Individual styling

Some areas of the web application required additional styling efforts beyond the general layout to enhance usability, visual clarity and the overall aesthetic appearance. These areas were identified based on both internal testing and feedback from the client and were adapted to meet specific functional and visual needs.

The Register Page is a dedicated interface accessible exclusively to administrators. As seen in Figure 3.24, the registration form is designed for clarity and ease of use. It features all necessary input fields to create a new user, alongside a dropdown for role selection. The description of the role is dynamically shown directly below depending on the selection, providing transparency about the permissions and purpose of each role. Styling is handled in a seperat `register.css` stylesheet for modularity and maintainability.



Registrieren

Vorname

Nachname

example@mail.at

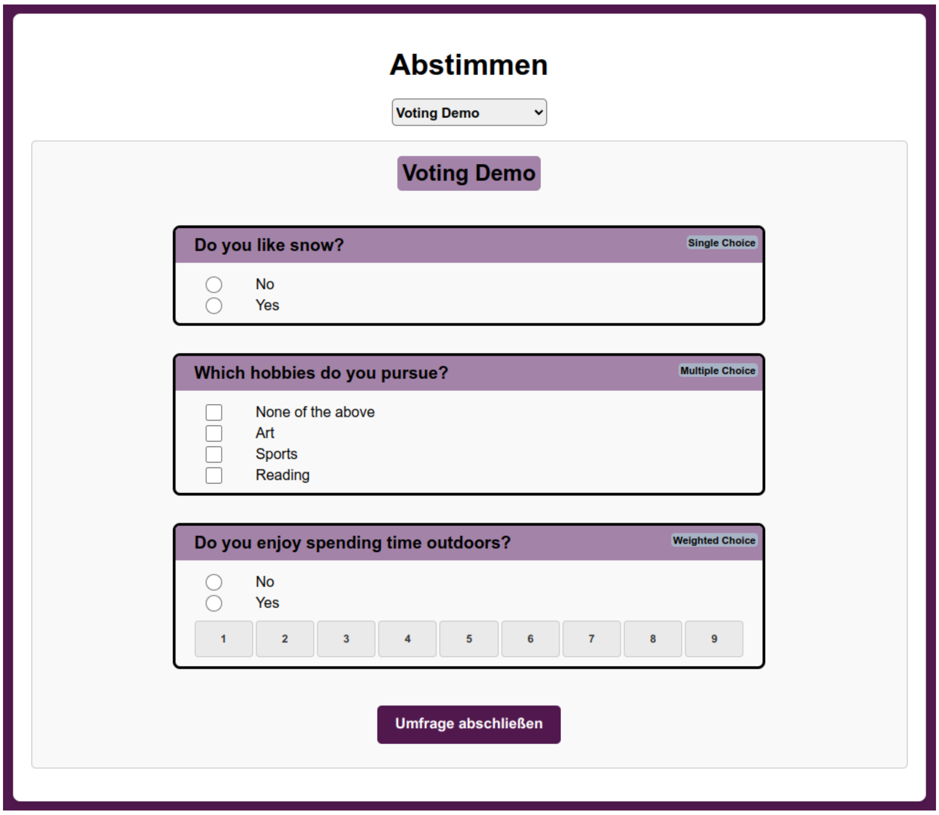
Rolle: normal

Beschreibung:
Darf normal abstimmen und sich Ergebnisse anzeigen lassen.

Figure 3.24: Registration form

Create Polls allows users to configure and launch customized surveys. Due to the flexibility and number of inputs required, the styling and layout needed careful design considerations. The whole page has been divided into smaller parts, which are the metadata of the poll, survey type and group selection, image upload and the questions at the very end as a separate section. The Styling also had to be modular to support reusability in both the creation and editing interfaces.

Voting is the section where users can submit their votes on polls they have permission to participate in. As shown in Figure 3.25, the selected poll's name is displayed below the section's heading and the poll selection dropdown, highlighted using the site's primary color. Each question is presented in a separate block, labeled with the question type (e.g. Single Choice, Multiple Choice, Weighted Choice) in the top right corner. Within each block, the question title is emphasized and all answer options are listed along with radio boxes for Single Choice and Weighted Choice questions, and check boxes for multiple choice questions. Weighted Choice questions also include an importance scale, which consists of multiple buttons that users must interact with to assign an importance value to their answer.



Do you like snow?

○ No
○ Yes

Which hobbies do you pursue?

□ None of the above
□ Art
□ Sports
□ Reading

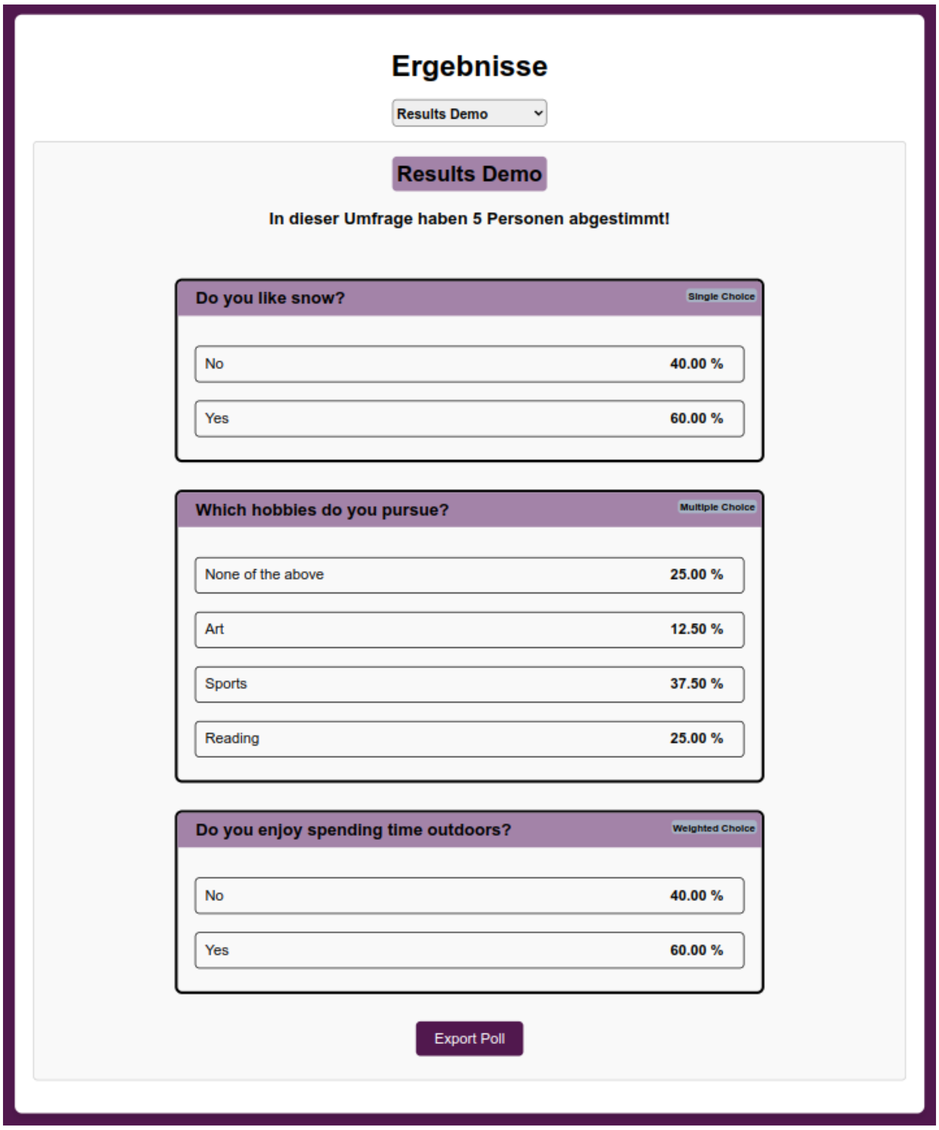
Do you enjoy spending time outdoors?

○ No
○ Yes

Figure 3.25: Registration form

Public voting enables anonymous users to participate in designated public polls. The layout mirrors the sturcture of the standard voting interface. But after submitting their responses, a follow-upp section automatically appears, prompting users to complete a set predefined of demographic questions. These are presented in a similar format ensuring a consistent and user-friendly experience throughout the public voting process.

Results allows users to see the outcome of completed polls they have access to. As illustrated in Figure 3.26, the interface follows the same layout as the voting section. The number of participants is displayed beneath the poll title. Inside each question block, the title is again highlighted and the answer options are listed with the percentage of votes they received.



Ergebnisse

Results Demo

Results Demo

In dieser Umfrage haben 5 Personen abgestimmt!

Do you like snow? Single Choice

No	40.00 %
Yes	60.00 %

Which hobbies do you pursue? Multiple Choice

None of the above	25.00 %
Art	12.50 %
Sports	37.50 %
Reading	25.00 %

Do you enjoy spending time outdoors? Weighted Choice

No	40.00 %
Yes	60.00 %

Figure 3.26: Registration form

The PollImage enables users to create surveys with a title picture. It was styled to maintain visual harmony and responsiveness across Create, Voting, and Results views. The image container uses a flexbox layout to center the image horizontally, ensuring it looks balanced on all screen sizes. The image itself is limited to a maximum width of 90 percent of its container and a maximum height of 300 pixel to prevent oversized images from disrupting the layout. To preserve the original aspect ratio and avoid distortion, the *object-fit: contain* property is applied. [**cssobjectfit**]

3.11 Deployment

This project was deployed to test how it behaves in a live environment and to ensure that the setup can be easily replicated on other systems. The deployment also helped to verify that all components work together correctly outside of the development environment.

3.11.1 Deployment Host

For hosting the deployment, DigitalOcean was chosen. It allows the creation of so-called "droplets", which are virtual machines that can be configured with custom storage, RAM and other specifications. These droplets function like personal servers but are hosted on the cloud and can be accessed publicly.

Configuration

In our case, Ubuntu 22.04 LTS was used as the operating system. The server was configured with 25 GB of storage and 1 GB of RAM. After creating the "droplet", DigitalOcean assigned an IP address to it. Through this IP the server is publicly reachable. To connect to the "droplet", SSH was used, allowing for safe configuration and management over the Terminal.

```
    ssh -i ~/.ssh/id_rsa_digitalocean root@1.2.3.4

```

Figure 3.27: SSH connection to "droplet" with example IP

3.11.2 Domain

In order for the application to be reachable over HTTPS, a domain was needed. After careful consideration the domain “togema.eu” was acquired from the provider Namecheap.

Configuration

On the Namecheap dashboard, a DNS record was added to point the domain name to the IP address of the "droplet" from DigitalOcean. With this configuration, all internet traffic directed to the domain “togema.eu”, is redirected to the IP address of the DigitalOcean “droplet”.

3.11.3 Deployment Process

This section describes the deployment process for this application. It is composed of multiple components, including PostgreSQL, the Node.js backend, the React frontend, as well as additional tools such as Nginx, PM2 and Certbot. All steps are executed with the terminal, which is connected to the “droplet” using SSH. The setup is done step by step to ensure everything is configured correctly.

First the server is updated using the standard update and upgrade command.

```
    sudo apt update && sudo apt upgrade -y

```

Figure 3.28: Ubuntu update command

Project Cloning

Before setting up the individual components, the project should be cloned from the Github repository, ’https://github.com/Max-Hierzer/Development-Diplomarbeit’.

PostgreSQL

PostgreSQL is installed using the system package manager. After successful installation, a database as well as a user are created manually.

```
sudo -u postgres psql
CREATE DATABASE db_name;
CREATE USER username WITH PASSWORD 'password';
GRANT ALL PRIVILEGES ON DATABASE db_name TO username;
\q

```

Figure 3.29: Example PostgreSQL database and user creation

When the database as well as the user are created, the production configuration in the `config.json` file in the backend needs to be adjusted to the correct values.

```
"production": {
  "username": "username",
  "password": "password",
  "database": "db_name",
  "host": "1.2.3.4",
  "dialect": "postgres"
}

```

Figure 3.30: Example config.json

Backend and Frontend

Node.js needs to be installed and all dependencies in the backend and frontend are installed using npm. Then a `.env` file is created to store sensitive data like credentials. Once all dependencies are installed, a production build is created in the frontend.

```
npm install
npm run build

```

Figure 3.31: Dependencies installation and production build creation commands

PM2 - Backend Process Manager

To run the backend of the application on the server PM2, a Node.js process manager was used. PM2 ensures that the backend application continues running in the background even after SSH session is closed, which is crucial for a web application, as it needs to stay online. PM2 also offers automatic restarts in case of crashes due to errors or unexpected shutdowns.

PM2 is installed and then used to run the backend with the production configuration.

```
NODE_ENV=production pm2 start app.js --name backend-app
pm2 startup
pm2 save

```

Figure 3.32: PM2 commands

Nginx - Web Server and Reverse Proxy

To handle incoming web traffic and route it properly, Nginx is used as a reverse proxy server. It is configured to listen on port 80 and forward all incoming HTTP requests to the backend service running on localhost.

```
server {
  listen 80;
  server_name domain_name www.domain_name;

  location / {
    proxy_pass http://localhost:5000;
    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection 'upgrade';
    proxy_set_header Host $host;
    proxy_cache_bypass $http_upgrade;
  }
}
```

Figure 3.33: Example Nginx configuration

HTTPS Setup with Certbot

To enable HTTPS, Certbot was used in combination with Nginx. It automatically requests and installs SSL certificates from "Let's Encrypt" and configures Nginx to use them. Additionally, Certbot configures Nginx to redirect all incoming HTTP traffic to HTTPS. This ensures that all traffic between the client and the server is encrypted and secure.

```
certbot --nginx -d togema.eu -d www.togema.eu
```

Figure 3.34: Certbot command

Chapter 4

Summary

List of Figures