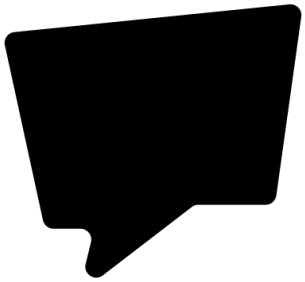
DAT250 Project Report

Group 10 (Eirik Måseidvåg)



feedapp

30th November 2023

Contents

Introduction

This project report presents the development and implementation of a comprehensive feedback and polling system, leveraging IoT principles and modern software architecture. The primary objective of this project is to design and build a versatile system that enables users to create and conduct polls to gather feedback. Users are provided with the flexibility to participate in polls through web or mobile browsers, as well as specialized IoT voting devices.

As part of this assignment, we looked into cutting-edge technologies that were new to our group. The culmination of our efforts will result in a fully functional application capable of creating polls, facilitating votes, retrieving poll results, integrating with third-party services, supporting IoT device interaction, and much more.

We use several technologies that are different from the usual ones recommended in the DAT250 course. On the frontend, we harness the power of Vue.js, accompanied by TailwindCSS for streamlined styling, and leverage supplementary frameworks like Storybook for enhanced component development. On the backend, we rely on NestJS, coupled with TypeORM for efficient database management, while Swagger generates a user-friendly interface to document our API endpoints. To foster code reusability and maintainability, all code resides within a single repository, benefiting from the organizational advantages of Nx monorepos. Automation is facilitated through GitHub workflows, handling tasks such as testing, linting, and seamless deployment to the cloud, where our application finds its hosting home.

This report provides an in-depth exploration of our project's objectives, design, development, deployment processes and IoT device simulation.

Development Methodology

The development of our feedback and polling system relies on a methodology guiding both design and implementation. This chapter offers an overview of the methodology used for this project.

2.1 Architectural Considerations

The architecture is designed with an emphasis on scalability, modularity, and security, particularly focusing on cloud deployment to enhance accessibility and scalability.

2.1.1 Scalability and Extensibility

The architecture is designed for scalability and extensibility, allowing seamless integration of new features and services as the project evolves.

2.1.2 Modularity and Reusability

We prioritize modularity and reusability as core principles, facilitating efficient sharing of logic and definitions across different parts of our codebase.

2.1.3 Security and Privacy

In our commitment to security and data privacy, we have implemented a focused and effective protection strategy. Firstly, we utilize third-party authentication services, specifically Supabase, for user authentication. This approach allows us to forego storing sensitive user data ourselves, thereby significantly lowering the risks of data breaches.

Furthermore, our relational database is hosted by Supabase, and we ensure its security by using a very strong password. This adds an additional layer of protection, safeguarding the data against unauthorized access.

We also place a high priority on robust authentication and authorization guards. These measures are critical in restricting data access, allowing only authorized users to view or modify sensitive information. This layered security approach is key to maintaining the confidentiality and integrity of our user data, providing our users with a secure and reliable experience.

2.1.4 Cloud Deployment

For enhanced accessibility and scalability, our architecture is tailored for cloud deployment. Leveraging platform-as-a-service (PaaS), infrastructure-as-a-service (IaaS), and software-as-a-service (SaaS) solutions meets our hosting needs. The architecture is designed to be highly scalable, allowing seamless integration of additional resources and services as needed. This scalability ensures the system can handle increased workloads and user demands.

It's important to note that for this assignment, we deliberately chose to keep the system relatively small in scale. This decision allows us to efficiently focus on specific features and demonstrates our ability to design and implement a scalable architecture.

Initial Planning and Design

This chapter outlines the decisions made concerning the features and capabilities of the polling application.

3.1 Features

The application's features are designed to meet and, in some aspects, exceed the assignment's requirements.

3.1.1 Polls and Votes

Polls will present a straightforward yes-or-no question. Users can vote as often as they like. When a poll is created, it has an "open" status. It can later be closed, at which point it is considered complete and cannot be reopened. Closing a poll triggers the generation of aggregated results, which are then posted to platforms such as dweet.io, RabbitMQ, and MongoDB. The system supports third-party integration with the RabbitMQ server, allowing subscribers to receive notifications when a poll closes. Polls can be optionally made private, necessitating an invitation for access. Poll owners automatically have access to their own private polls. To participate in a private poll, authorization is required, thus preventing anonymous users from voting.

Votes are straightforward, limited to yes or no responses. In the web application, they will update in real time.

3.1.2 Authenticated Users

The application supports login via Google. Authenticated users can create, read, update, and delete polls, as well as modify their user's display names. They also have the ability to log out.

3.1.3 Anonymous Users

Anonymous users, who are not logged in, can vote on public polls while they are open.

3.1.4 IoT Device

The application will enable IoT devices to connect to polls. Each new device will receive a unique identifier and email through a specific endpoint. This email can be used to invite the device to private polls. Devices are restricted to connecting to, reading, voting on, and disconnecting from polls and cannot create them.

3.2 Domain Models

The domain models represent the core concepts of the polling application and their relationships within the database. This section outlines the entities of our domain model and the rationale behind their design decisions.

3.2.1 Users

The *User* entity uniquely identifies each system user with a `id` and includes attributes such as `email`, `name`, and `avatar`. Both `id` and `email` are unique for each record, ensuring individual identification within the application.

3.2.2 Devices

The *Device* entity represents the IoT devices within the system. It includes a unique `id` and fields for `createdAt` and `updatedAt` timestamps, along with an optional `connectedPollId` to link the device to an active poll.

3.2.3 Polls

The *Poll* entity, essential for the application's functionality, enables users to create and manage polls. Attributes include `title`, `question`, privacy settings, and status, with timestamps to monitor creation and updates. This design accommodates multiple polls per user and manages their lifecycle.

3.2.4 Votes

The *Vote* entity captures users' selections in polls. Linked to the *Poll* entity via `pollId`, it records the `answer` provided by the user, facilitating the analysis of poll results.

3.2.5 Invites

To manage participation, the *Invite* entity associates users with specific polls. It stores the `pollId`, the invitee's `email`, and the `createdAt` timestamp to log when the invitation was issued. A compound primary key composed of `pollId` and `email` forms a unique identifier for each record, ensuring a single invite per email per poll.

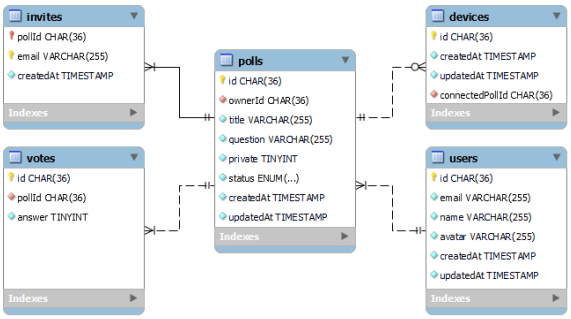


Figure 3.1: EER Diagram of the domain models

The relationships are designed to cascade effectively upon the deletion of records. If a user is deleted, all their polls, along with connected invites and votes, will also be deleted. Devices linked to polls that are deleted will become disconnected.

3.3 System Architecture

The polling application's architecture is a network of interconnected components that communicate through various protocols to ensure a unified user experience. Figure 3.2 illustrates the connectivity and functions of each system component.

3.3.1 Web Browser and Web Application

The user experience begins with the web browser, which communicates with our application via HTTP requests. The cloud-hosted web application, serving as the front-end, utilizes WebSockets to maintain real-time updates of votes with Supabase. It also communicates with the REST API using HTTP requests.

3.3.2 Supabase

Supabase, being a cloud-hosted service, is crucial for user authentication and handling real-time data. It manages user sessions and employs WebSockets, enabling us to subscribe to changes in the database. This functionality will be used to monitor insertions into the votes table, providing real-time updates on the vote page.

3.3.3 REST API

Our REST API is the core of our back-end operations, interfacing with both the Relational Database (RDB) and Non-Relational Database (NDB). It handles HTTP for general operations and uses HTTP requests for interactions with IoT devices, and POST requests to dweet.io. The API facilitates CRUD operations that manipulate database records and integrates with RabbitMQ to communicate the status of polls.

3.3.4 RabbitMQ

The RabbitMQ service, hosted in the cloud, functions as a message broker. It is responsible for disseminating notifications related to poll closures and allows third-party services to subscribe to these events, thereby providing extensibility and integration with external systems.

3.3.5 Databases

The RDB (relational database) is the structured repository for our application, storing all records in accordance with the domain models. The NDB (non-relational database), on the other hand, is designated for the aggregated results of closed polls, providing a flexible schema for this unstructured data.

3.3.6 Dweet.io

dweet.io is utilized as an external platform to publish aggregated poll results. The REST API posts data to this endpoint, making it publicly accessible for retrieval.

3.3.7 IoT Device

IoT devices enhance our application's ecosystem by connecting via HTTP to the REST API. They are constrained to reading poll data and submitting votes, providing a dedicated interface for poll interaction.

Our system aligns with an IoT Level 2 architecture, focusing on real-time data interaction and service integration. It effectively combines web applications and backend services, using RabbitMQ, a non-relational database, and Supabase. The backend manages complex tasks like publishing data when polls close and integrating with PostgreSQL for data storage. Supabase enhances this setup by providing secure authentication and instant updates, especially for live voting on our poll page.

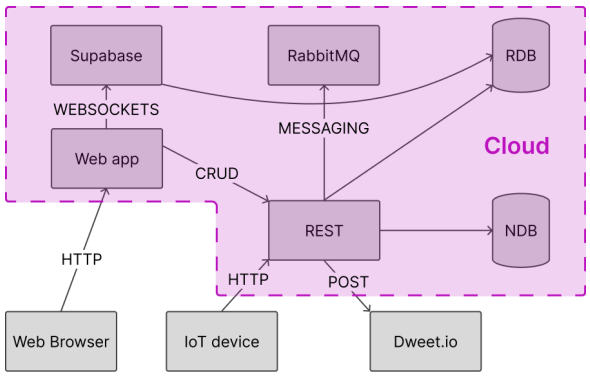


Figure 3.2: System architecture diagram

IoT devices are integrated seamlessly, with the backend supporting their participation in polls. This demonstrates our system's capability to not only handle data but also interact with IoT devices, enriching the user experience. Our architecture ensures reliable and scalable operations, catering to a diverse user base. While it doesn't reach the advanced autonomy and analytics of higher IoT levels, our Level 2 system excels in real-time processing and service integration.

Technologies and Environment

In this chapter we explore the technology stack, coding environment, and automated workflows that form the backbone of the polling application's development process. The selection of technologies was driven by the assignment's criteria, personal proficiency, and the requirement to incorporate a foreign technology, which led to the adoption of NestJS as the backend framework.

4.1 Technology Stack

The technology stack is bifurcated into frontend and backend components, each chosen for its efficiency, ease of development, and community support.

4.1.1 Frontend Technologies

The frontend is built using Vue.js and TailwindCSS. Vue.js was preferred over other frameworks for its simplicity and developer-friendly environment. TailwindCSS was chosen for its utility-first approach, which allows for custom styling without the constraints of predefined component libraries. Additionally, Storybook is utilized to showcase the built components, enhancing the development experience.

4.1.2 Backend Technologies

NestJS, selected as the assignment's foreign technology requirement, serves as the backbone of the backend, working in conjunction with TypeORM for PostgreSQL interaction. MongoDB was chosen as the non-relational database due to its flexibility and ease of use with the Mongoose library for database management. RabbitMQ facilitates messaging services, and Swagger is implemented for both a polished API interface and the generation of services for frontend integration.

The use of Supabase with Google for social login streamlines the authentication process, automatically creating a new user record in PostgreSQL upon the first login. PostgreSQL is favored for its advanced features and support for custom types like UUID (Universally Unique Identifier).

4.2 Testing Framework

Jest is employed for testing both the frontend and backend, selected for its simplicity and effectiveness in setting up and running tests. It will be used alongside other helper libraries to generate mocks and fixtures.

4.3 Development Environment

The entire application is developed within a mono-repository managed by Nx, which simplifies the maintenance and scaling of the code base. GitHub Actions are configured for CI (Continuous Integration) and delivery, automatically testing, linting, building, and deploying code upon commits to the main branch. Deployment targets include Heroku for the REST API, Cloudflare pages for the frontend, and Github Pages for Storybook. Within the Visual Studio Code environment, a designated folder with recommended plugins ensures consistency across development setups.

The technologies and environment chosen for this project not only fulfill the assignment's requirements but also reflect a modern development practice, optimized for efficiency, collaboration, and scalability.

Development and Implementation

This chapter outlines the practical steps taken in turning our project from ideas into a functional system. We'll explore the decisions and tools we used to build our feedback and polling application. From initial planning to solving technical challenges, this chapter offers an overview of our development journey.

5.1 Foundation

The foundation of our project sets the stage for a streamlined and modular development process. In this section, we explore the fundamental aspects that underpin our application's structure and organization.

5.1.1 Monorepo with Nx

We adopted a monorepo structure using Nx to manage our project effectively. This approach allows us to maintain multiple packages within a single repository while sharing code, libraries, and resources seamlessly. Nx simplifies the development workflow, ensuring consistency and modularity across our application.

5.1.2 Dependencies and Development Tools

Ensuring a robust development environment is essential for efficient coding and collaboration. We selected and installed the necessary dependencies, including linters, types, and testing libraries, to enhance code quality and maintainability. We implemented ESLint, Prettier, and Stylelint, defining global configurations for them to be used within each package to ensure uniform code formatting across packages.

5.1.3 Modular Packages

Our project comprises three distinct packages, each serving a specific purpose:

1. **App (Frontend):** The frontend package, responsible for the user interface and user experience.
2. **Backend:** This package houses the business logic and API endpoints, ensuring efficient data management and communication.
3. **Style:** While currently utilized exclusively by the App package, the Style package plays a significant role in maintaining a consistent and visually appealing design. By encapsulating the TailwindCSS configuration, color palette, typography, and font files within this package, we ensure modularity and the potential for easy integration into future projects.

The separation of these packages fosters modularity and code reusability, facilitating potential expansions and making it simpler to integrate our style configuration into new projects.

5.1.4 TypeScript

A crucial consideration in our project's foundation was the technology stack. We intentionally selected technologies that align with the TypeScript ecosystem, ensuring seamless compatibility and promoting a cohesive development experience across all packages. This strategic choice allows us to share common logic and types between the packages in the monorepo.

5.1.5 GitHub Workflows

To streamline our development and deployment processes, we implemented GitHub workflows. These automated workflows handle tasks such as testing, linting, and continuous integration, ensuring code quality and reliability. Our GitHub workflows empower our group to focus on development rather than manual tasks, promoting efficiency and consistency throughout the project. We have also implemented a restriction to the main branch that prevents merging pull requests if either the tests, linters, or builds fail.

5.2 Frontend

Our frontend (https://feedapp.no/) is a crucial part of our application, influencing the user experience significantly. This section explores the various aspects of our frontend development, covering topics such as visual identity, accessibility, error handling, and user-friendly features. Discover how our frontend components and practices contribute to an inclusive, responsive, and user-centric application.

5.2.1 Branding and Visual Identity

We started by crafting a unique logo using Figma and selecting a color palette that not only looks great but also meets Web Content Accessibility Guidelines (WCAG) requirements when combined in various ways. This ensures our application is not only visually appealing but also inclusive.

To further enhance the user experience, we created a comprehensive favicon package featuring our logo. Favicon icons add a polished touch to our application, making it recognizable in browser tabs and bookmarks. We also created a loading spinner using only SVG animations, designed from our logo.

Additionally, we integrated captivating illustrations from unDraw, which we customized to match our color scheme. These illustrations can be seen on our front page, home page, and error pages, adding creativity and vibrancy to the overall user experience. Our dedication to visual branding and illustrations ensures a distinctive and engaging application.

5.2.2 UI Components and Storybook

As we didn't utilize a UI components library, we crafted essential components like buttons, lists, and text inputs manually. For effective development and component inspection, we employed Storybook. Within it, we documented each reusable component through dedicated stories. These stories serve as valuable documentation to understand how the components function and streamline the development process.

Additionally, our Storybook is hosted in the cloud, making it easily accessible at https://storybook.feedapp.no/.

5.2.3 Accessibility

We prioritized accessibility during the development of our UI components. Our aim was to create an inclusive user experience, allowing seamless keyboard navigation throughout the entire application. To achieve this, we ensured that the focused components not only function effectively but also maintain a visually pleasing and user-friendly appearance during keyboard interactions. Additionally, we incorporated ARIA (Accessible Rich Internet Applications) attributes where necessary, enhancing compatibility with screen readers and enabling users with visual impairments to interact effectively with our application. When styling with CSS, we consistently use *REM* values to ensure that the website scales appropriately when users adjust the font size.

5.2.4 Mobile responsiveness

We prioritized mobile responsiveness in our application, ensuring it performs smoothly across different screen sizes and devices. Our design adapts to smartphones, tablets, and desktops, providing an optimal user experience on any device. By using responsive design principles, our application automatically adjusts its layout and content to fit various screen sizes, making it accessible and user-friendly from any location and on any device.

5.2.5 Global notification center

We've integrated a Global Notification Center into our application. This central hub allows us to send notifications with smooth animations from anywhere in the frontend code. It ensures that users receive timely updates and feedback.

5.2.6 Supabase Integration

Our application seamlessly integrates with Supabase for authentication, streamlining the user experience. Here's how we've incorporated this third-party library: We developed a wrapper around Supabase to ensure smooth authentication. When a user logs in, our HTTP request interceptor is updated to include the JWT token acquired during login, which is crucial for authenticating requests in the backend service, providing users with secure access to our platform. Supabase also handles automatic token refreshing using a refresh token, ensuring that users maintain seamless access to our application without the need for repeated logins. Additionally, we leverage Supabase on the vote page to handle real-time updates of poll votes, enhancing user engagement and providing instant feedback on poll results, contributing to an interactive and dynamic user experience.

5.2.7 User-Friendly pages

Navigating our application is user-friendly, even when users encounter non-existent paths or server errors. We've created dedicated "not-found" and "server-error" pages. If users try to access a path that doesn't exist, they are smoothly redirected to the "not-found" page, which offers clear and helpful guidance. Depending on their authentication status, there is a button that directs them either to the home page or the index page.

5.2.8 Form Validation

To improve the user experience and prevent frustrating errors in form submissions, we've implemented robust input validation using the Zod library throughout our application. This ensures that users cannot submit invalid data, leading to a more reliable and accurate application and would at scale ease the load on the backend service.

5.2.9 Locale

Within our application, we've implemented a localization feature that allows users to select their preferred language for an improved user experience. By default, the application adapts to users' language preferences based on their browser settings. When users access our application, it automatically detects and uses the language preference set in their browser.

To ensure that users' language choices persist across different sessions and browser refreshes, we've implemented a straightforward solution using *localStorage*. When users actively change the language, a token is set to remember their preference. This functionality simplifies the user experience, allowing users to comfortably interact with the application in their chosen language.

5.3 Backend

In this chapter, we dive into the backend architecture and implementation of our system. We'll explore how we've structured our code, handle authentication, validate incoming requests, and provide user-friendly API documentation. Our goal is to offer a comprehensive overview of the backend components and their functionality to ensure a deeper understanding of our system's core.

5.3.1 Design Pattern: Controller-Service-Repository Architecture

Our backend architecture adopts the Controller-Service-Repository pattern, offering an effective structure that enhances both maintainability and clarity.

1. **Controllers**: The controllers in our architecture are essential for managing incoming HTTP requests. Their primary function is to parse these requests, delegate processing to services, and handle the resulting responses and errors. This layer acts as the intermediary between the client and the service layer, focusing on request routing rather than business logic.

2. **Services**: This layer encapsulates the core business logic of our application. Services are responsible for executing operational tasks, enforcing business rules, and liaising with the repository layer for data management and access. This separation from the data handling layer underscores their focus on business logic and operational processing.

3. **Repositories**: Our repository layer, separated from the service layer for modularity and maintenance, handles data access and manipulation. We use TypeORM, which offers ready-made repositories based on our entities. These entities are definitions based on our domain models.

Through the Controller-Service-Repository pattern, our backend is structured to facilitate efficient development and maintenance, ensuring a clean and scalable solution for handling our application's business logic and data management needs.

5.3.2 Authentication with JWT

Authentication is a crucial aspect of our backend service, ensuring secure access to protected endpoints and resources. We use JSON Web Tokens (JWTs) for authentication.

1. **Bearer Token Authorization**: To access certain endpoints and protected resources, a request must include a bearer token within the HTTP headers. This token is a valid, digitally signed JWT obtained from Supabase, our chosen third-party authentication provider.

2. **JWT Verification**: For each incoming request to protected endpoints, our backend service verifies the provided JWT. This verification process ensures the token's authenticity and integrity, confirming that it has not been tampered with.

3. **User Identification**: The JWT token includes essential user information, such as the user's unique identifier and email. This information is securely embedded within the token during authentication. By extracting and decoding the JWT, we gain access to this user data, which is crucial for user-based authorization and personalization.

4. **Device Identification**: It's important to mention that for IoT devices, our backend service has the capability to directly sign and deliver JWTs tagged with *isDevice*. This enables us to distinguish between users and devices effectively.

This authentication mechanism guarantees that only authenticated and authorized users can access specific parts of our backend service, ensuring the confidentiality and integrity of user data and interactions while providing a secure user experience.

5.3.3 Request Parameter Validation

We've implemented parameter validation in our backend using the class-validator library. This library automates the validation process, ensuring that the data received in requests meets the required criteria.

To achieve this, we define Data Transfer Objects (DTOs) and Data Access Objects (DAOs) for our endpoints and entities. By applying class-validator decorators to these objects, we can easily specify validation rules. When a request contains invalid parameters, the class-validator library automatically generates human-readable error messages, clearly indicating which parameters are incorrect and why.

This approach simplifies error handling and enhances the user experience by providing informative feedback when invalid data is submitted.

5.3.4 Swagger

We've implemented Swagger (https://api.feedapp.no/docs/) support to create user-friendly API documentation. Swagger helps users understand our REST API by providing clear information about each endpoint. With Swagger, users can easily explore and test our API. We've added Swagger decorators to our REST endpoints, indicating essential details such as authentication requirements, response types, and expected parameter formats. We've also annotated our models and data types for clarity.

Our Swagger setup is user-friendly and comprehensive. It allows users to interact with our API efficiently. Additionally, we can export our Swagger configuration as JSON https://api.feedapp.no/docs-json for integration with tools like Postman, further enhancing the testing and user experience.

5.4 IoT Device Simulation

Our IoT device simulation, built with Python and the "curses" module, offers a streamlined way to interact with IoT devices within our system. This section provides an overview of its core functionalities.

5.4.1 Overview

Our IoT device simulation allows users to create new devices or connect to existing ones, providing insights into how IoT devices interact with our platform.

5.4.2 Authentication

Users can create new devices, which generate JWT authentication tokens. Alternatively, they can connect to existing devices by providing valid JWT tokens. These tokens are tagged with the *isDevice* attribute, allowing the backend to differentiate between devices and real users. This distinction ensures that devices have restricted access compared to regular users.

5.4.3 Functionality

Within the simulation, users can:

- Create new IoT devices, generating JWT tokens.
- Connect to existing devices using valid JWT tokens.
- View their device ID and email. Poll owners can invite devices to private polls with this email.
- View their JWT token

- Connect to available polls.
- Vote (yes or no) in polls.
- Disconnect from polls.

5.4.4 Testing

We've implemented a comprehensive testing strategy to ensure the reliability and robustness of our application. Leveraging the Jest testing framework, we've created a suite of 150 tests that cover various aspects of our codebase, achieving an impressive 90% code coverage. These tests are designed to assess a wide range of scenarios and functionalities.

Our commitment to thorough testing has not only enhanced the quality of our code but also facilitated the development process. Writing tests has allowed us to identify and address issues that might have otherwise gone unnoticed, ensuring a more resilient and dependable application.

5.5 Deployment

We purchased the domain *feedapp.no* and transferred its ownership to Cloudflare. We chose Cloudflare for its excellent DNS services, including features like CNAME flattening, which enables us to host our app at *https://feedapp.no/* instead of *https://www.feedapp.no/*. Additionally, we created two sub-domains: *api* and *storybook*. The *api* sub-domain is designated for the backend service hosted on Heroku, and the *storybook* sub-domain for the GitHub Pages. Since these sub-domains are not hosted by Cloudflare, we had to configure their DNS records to point to different hosts, requiring extra configuration.

Cloudflare Pages cannot host Node.js projects, so we chose to host our application on Heroku. There, we use their web dyno to run our backend with the *node* command. We have also utilized Heroku to host an instance of CloudAMQP, which is integrated with RabbitMQ in our backend. GitHub actions are configured to automatically run tests and linters, and then build and deploy the app, storybook, and backend to their respective hosts.

The relational database is hosted by Supabase, which is a standard part of their service offering. The non-relational database, managed by MongoDB, is hosted on Altas.

Conclusion

In this project, our aim was to develop a unique polling system leveraging the latest IoT technology. Our primary goal was to create an easy-to-use system capable of efficiently handling feedback and polls. We're proud to say that we achieved this by developing a user-friendly platform that integrates seamlessly with IoT devices.

Throughout this project, we faced various challenges, especially when working with new technologies and software. However, these challenges turned into valuable learning experiences. These skills and experiences are something that we will carry forward in our future projects. Moreover, this project opened our eyes to the vast potential of IoT in everyday applications, showing us how technology can be used to make similar tasks more interactive and efficient.

Looking to the future, there are many opportunities to expand and improve this project. The platform we've developed has the potential to be adapted for different uses, making it a versatile tool. As technology continues to evolve, so too can our system, adapting to new trends and requirements. This project represents not only a technical accomplishment but also a valuable repository of coding principles that can be applied to future projects.

Links

- App
- Public poll
- GitHub repository
- Swagger
- Storybook

Bibliography

Altas (2023). URL: https://www.mongodb.com/atlas/database.

CloudAMQP (2023). URL: https://www.cloudamqp.com/.

Cloudflare (2023). URL: https://cloudflare.com/.

dweet.io (2023). URL: https://dweet.io/.

ESLint (2023). URL: https://eslint.org/.

Figma (2023). URL: https://www.figma.com/.

GitHub (2023). URL: https://github.com/.

Heroku (2023). URL: https://heroku.com/.

Jest (2023). URL: https://jestjs.io/.

JWT (2023). URL: https://jwt.io/.

MongoDB (2023). URL: https://mongodb.com/.

NestJS (2023). URL: https://nestjs.com/.

Nx (2023). URL: https://nx.dev/.

PostgreSQL (2023). URL: https://www.postgresql.org/.

Postman (2023). URL: https://postman.com/.

Prettier (2023). URL: https://prettier.io/.

RabbitMQ (2023). URL: https://rabbitmq.com/.

Storybook (2023). URL: https://storybook.js.org/.

Stylelint (2023). URL: https://stylelint.io/.

Supabase (2023). URL: https://supabase.com/.

Swagger (2023). URL: https://swagger.io/.

TailwindCSS (2023). URL: https://tailwindcss.com/.

TypeORM (2023). URL: https://typeorm.io/.

TypeScript (2023). URL: https://www.typescriptlang.org/.

unDraw (2023). URL: https://undraw.co/illustrations.

Vue.js (2023). URL: https://vuejs.org/.

WebSockets (2023). URL: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API.

Zod (2023). URL: https://zod.dev/.