Software Technology Evaluation Project

November 29, 2023

Abstract

The software technology stack for our voting application is an integration of React, Tailwind CSS, Spring Boot, JPA with Hibernate, and an H2 database. React's declarative UI paradigm combined with Tailwind CSS's utility-first styling affords a seamless and responsive front-end experience. The Spring Boot framework underpins the back-end, facilitating rapid development and deployment with its embedded server and 'convention over configuration' philosophy. JPA, implemented via Hibernate, abstracts the complexity of database interactions with an ORM layer, promoting a robust data access strategy. The H2 database provides a lightweight, file-based database, perfect for development and testing phases due to its simplicity and speed. The project's highlights include a secure, scalable architecture with a focus on clean, maintainable code and an emphasis on user experience. Authentication is streamlined through Firebase, ensuring secure access and data integrity. The application's design, leveraging containerization with Docker, ensures consistent environments across development, testing, and production. The result is an intuitive voting platform that is both developer-friendly and scalable.

1 Introduction

1.1 Development of the Feedback App

In this project we have developed a feedback app for voting on polls. We have implemented register and login functionality. A user can create polls that are either private or public, and both users and voters can vote on polls. We have also created a pollinfo display where users can look at results of their own polls. On this pollinfo page, you can also find a shareable link so that people that dont have user accounts can vote on public polls. Users can also manage their polls by ending them, starting them again, and deleting them.

1.2 Technology stack

The technology stack chosen for this project is both robust and dynamic, covering a range of modern development tools and frameworks. We have employed React for our front-end development, capitalizing on its component-based architecture for a responsive and interactive user interface. The back-end is powered by Spring Boot [3], providing a strong and scalable foundation for our application services. For message handling and asynchronous operations, RabbitMQ [13] has been integrated, ensuring efficient communication and process management.

Further enhancing our application's capabilities, we have incorporated Tailwind CSS [11] for streamlined styling, JPA (Java Persistence API) [14,18] for efficient data handling, and Hibernate [12] for object-relational mapping, ensuring a seamless data management experience. Firebase [10] is used for our authentication solutions, fortifying the app's security. Additionally, Dweet.io has been utilized for its simplicity in real-time messaging, and an H2 database for its lightweight and easy-to-use nature.

1.3 Results

During this project, we have gained a better insight into different software technologies, frameworks and tools. We now have a working prototype of a voting app which can easily be extendable to include more features in the future.

1.4 Overview

The rest of this report is organised as follows: Section 2 gives an overview of the software technology stack used. Section 3 describes the design of the prototype. Section 4 details how the prototype has been implemented. Section 5 explains how the prototype has been tested and what experiments have been done. In Section 6 we provide conclusions regarding different aspects of the project.

2 Software Technology Stack

2.1 React and Tailwind: Efficient Frontend Development

We have chosen React and Tailwind for our web application's frontend, aligning with Goadrich and Rogers' findings that React Native excels across different operating systems [2]. React enables us to build dynamic web applications where data changes without page reloads. Tailwind CSS, a utility-first framework, has been rising in popularity for its streamlined approach to styling.

React stands out for its component-based architecture, allowing developers to create self-contained components managing their own state, leading to reusable, complex UIs. It simplifies development and reduces redundancy. React's lifecycle management offers fine control over component behavior, and its declarative nature means developers describe the UI while React handles DOM updates efficiently. This is especially beneficial for large application maintainability.

Tailwind CSS complements this by enabling in-HTML styling, reducing the need to switch between HTML and CSS files. Its utility-first approach and self-explanatory classes enhance readability and speed up development. Tailwind's focus on utility classes often results in smaller CSS files, boosting performance. Its Just-In-Time (JIT) compiler ensures minimal CSS generation, enhancing load times and overall web application performance.

2.2 Spring Boot: Back-End Infrastructure

Spring Boot is an extension of the Spring framework that simplifies the initial setup and development of new Spring applications.[3] It favors convention over configuration and is designed to get you up and running as quickly as possible. Here's a detailed explanation of why Spring Boot is an excellent choice for our back-end infrastructure:

Spring Boot simplifies the management of application configurations. It eliminates the need for defining boilerplate configuration with its auto-configuration feature. This feature automatically configures your application based on the libraries present on the classpath. This can significantly reduce development time and increase productivity because developers can focus on the unique aspects of their application rather than on infrastructure setup.

Spring Boot makes it easy to create standalone, production-grade Spring-based applications that can be run directly from the command line without requiring an external web server. This is made possible by embedding servers like Tomcat, Jetty, or Undertow directly in the application. The ability to run as a standalone app simplifies both the development and the deployment processes.

To streamline the configuration process, Spring Boot provides a set of 'starter' dependencies that bundle the necessary libraries to get a feature working. These starters cover many Spring Boot features, such as data access, messaging, and web services. Using starters, you can avoid library version conflicts and ensure that you're using a set of dependencies that Spring Boot has tested and approved.

Spring Boot has excellent integration with Spring Data, which simplifies database access and provides support for various data access technologies, including JPA, JDBC, and NoSQL. Spring Boot also works well with Hibernate - one of the most popular object-relational mapping (ORM) tools - making it easier to work with databases using an object-oriented paradigm.[19]

2.3 RabbitMQ and Dweet.io: Message Systems

RabbitMQ, an open-source message broker, is chosen for its reliability, scalability, and flexibility in handling application messaging systems.[13] It serves as a platform-neutral intermediary, enabling asynchronous communication between services with support for various messaging protocols, intricate queuing, and multiple exchange types. This ensures flexible routing and enhances system robustness by allowing producers and consumers to scale independently. Additionally, RabbitMQ's features like message durability, persistent queues, and capability to operate in high-load environments with consistent hashing and sharding make it ideal for maintaining application integrity and managing high message volumes.

Complementing RabbitMQ, Dweet.io offers a simplified solution for real-time messaging, especially beneficial for Internet of Things (IoT) applications[20]. It enables devices to "dweet" messages for instant access by subscribed clients, emphasizing timely data delivery with minimal setup requirements. This makes Dweet.io highly accessible for immediate deployment and integration.

Together, RabbitMQ and Dweet.io provide a comprehensive messaging solution. RabbitMQ handles the heavy-duty, reliable queuing and processing of messages, while Dweet.io contributes with its

real-time messaging capabilities, ensuring a responsive and interconnected application ecosystem.

2.4 JPA, Hibernate and H2: Data Management and Storing

Java Persistence API (JPA) is a Java specification providing an Object-Relational Mapping (ORM) standard. This abstraction simplifies the interaction between Java objects and relational databases. JPA's primary advantage lies in its database-agnostic nature, offering flexibility in database choice and minimal application code changes when switching databases.

Hibernate, as an implementation of JPA, enhances its capabilities. It's renowned for its performance optimization features, such as lazy loading and sophisticated caching mechanisms. Hibernate also supports a rich set of querying capabilities through the Hibernate Query Language (HQL) and Criteria API, enabling more complex and efficient data retrieval operations. According to a study presented at the 2021 23rd International Conference on Control Systems and Computer Science, "ORM (object/relational mapping) has now widespread. This is mainly due to Hibernate, an open-source ORM, and Spring Data, an umbrella project from the Spring family whose purpose is to unify and facilitate access to different kinds of persistence stores, including relational database systems and NoSQL databases"[19]. Together, JPA and Hibernate offer:

- **Simplified Database Interactions**: By abstracting complex JDBC operations, they reduce boilerplate code and streamline database interactions.
- **Advanced Query Capabilities**: Enhanced querying and retrieval options facilitate sophisticated data handling.
- **Performance Optimizations**: Features like caching and batching improve application performance and database interaction efficiency.

The H2 database is a Java-based database known for its speed and simplicity. Primarily used in development and testing environments, H2 stands out for its ease of setup and rapid execution. It offers a web console for direct database interactions, enhancing its usability during development.

2.5 Firebase: Authentication (New Technology)

Firebase Authentication provides a full backend service that can authenticate users through multiple methods, including passwords, phone numbers, popular federated identity providers like Google, Facebook, and Twitter, and more. The decision to use Firebase for authentication in the application is based on its comprehensive suite of features that enhance user security and improve the overall user experience.

Firebase Authentication offers a seamless integration with applications, providing a complete identity solution supporting email/password auth, social media login, and phone authentication. Its SDKs and ready-made UI libraries allow for quick implementation of secure user authentication. The service not only handles user authentication but also manages user accounts and sessions with ease. It's equipped

with features such as email and password reset, account verification, and sign-in link capabilities, which contribute to a robust authentication flow.

According to Pramono and Javista, Firebase Authentication ensures that user credentials are securely managed and stored, preventing users from manipulating data or using the identity of another user due to its security and strict data verification process [21]. The platform is compliant with identity standards such as OAuth 2.0 and OpenID Connect, so user data is handled in a secure and standardized manner. Designed to handle large-scale applications, Firebase Authentication can effortlessly scale to accommodate millions of users. The flexibility offered by Firebase allows developers to focus on the user experience while Firebase takes care of the authentication backend.

Firebase Authentication is part of the larger Firebase ecosystem, which means it can be easily integrated with other Firebase services like Firestore, Firebase Realtime Database, and Firebase Cloud Functions. This integration provides a seamless development experience and allows for the creation of sophisticated, authenticated workflows. By incorporating Firebase Authentication, the application streamlines the user sign-in process and ensures that authentication is handled safely and efficiently, allowing developers to focus on building features that add value to the user experience rather than the intricacies of user security management.

3 Design

The application is designed using modern scalable technologies with clear separation of concerns across its various layers. The system is built upon a microservices [16] architecture pattern, enabling independent scaling and flexibility in deployment

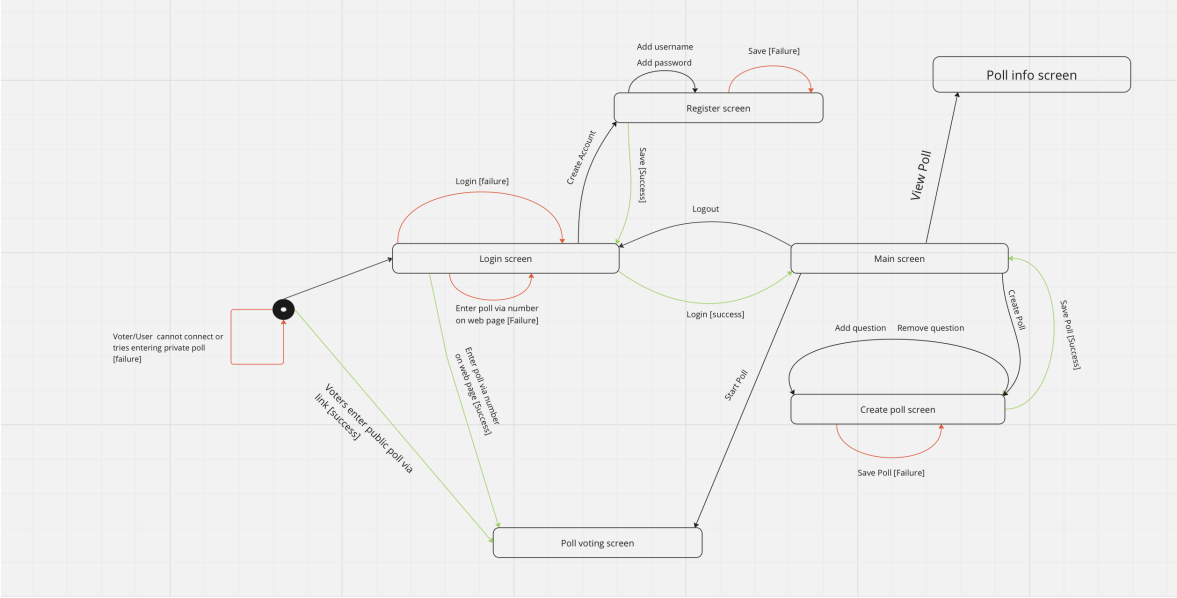

Figure 1: Application flow diagram

The application flow diagram, as seen in Figure 1, provides a high-level overview of the user's journey

through the application. According to Lucidchart, process flow diagrams help to ”show unnecessary steps, bottlenecks, and other inefficiencies” [5]. It begins with the Login screen, where users can either log in or register. In case of registration, the user is prompted to add a username and password before being redirected to the main screen upon successful account creation. The main screen serves as a navigation hub, allowing users to view polls, create new polls, or log out.

Upon selecting a poll, the user is taken to the Poll Voting screen, where they can cast their vote. Both successful and unsuccessful actions within the application are accounted for, with clear pathways leading the user to appropriate screens or error messages, ensuring a robust user experience. The diagram distinctly marks the paths taken during normal operation (success) and exceptions (failure), reflecting the applications error-handling capabilities.

The flow for creating a poll is similarly user-friendly, with the Create Poll screen allowing users to add or remove questions before saving. The user is provided with immediate feedback on the success or failure of their actions, with the system guiding them accordingly. In addition to the primary user flows, the diagram also highlights potential failure states, such as login failures or issues entering private polls, indicating comprehensive error handling throughout the application.

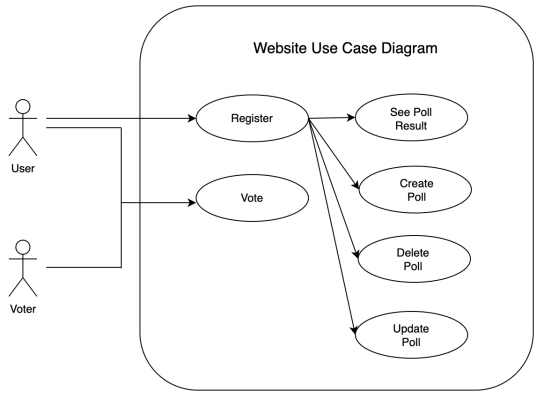

Figure 2: Website use case diagram

Figure 2 shows the use case diagram for the website, detailing the interactions that both ’User’ and ’Voter’ actors have with the system. The ’User’ actor, representing registered individuals, has the capabilities to register, vote, see poll results, create, delete, and update polls. The distinction between a general ’User’ and a ’Voter’ highlights the specialized role that a ’Voter’ plays in the context of the application, primarily focused on the voting functionality. The use case diagram clearly identifies the system’s boundary and the primary functions available to the users, providing a straightforward visualization of the different ways the users can interact with the system. ”A key concept of use case modeling is that it helps us design a system from the end user’s perspective” [6].

Figure 3 presents the domain model of the application, illustrating the primary entities and their relationships. ”A domain model is a visual representation of real situation objects in a domain. A domain is an area of concern. It’s used to refer to the area you are dealing with” [7]. The model defines a User entity with attributes for username, password, a unique Firebase user ID, administrative rights, and an identifier. This entity is central to the model and is associated with Voter, Vote, and IoT-Device entities.

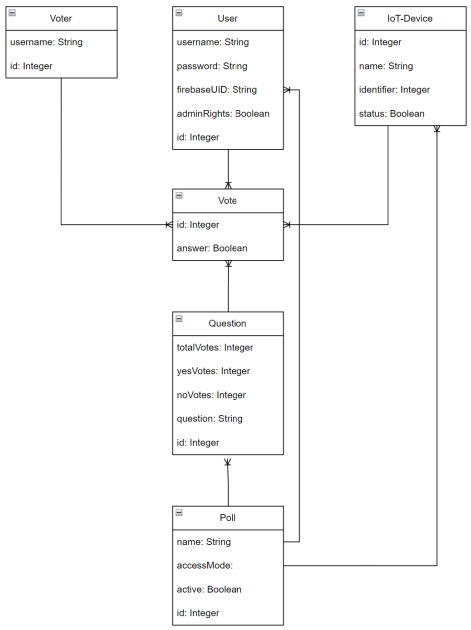

Figure 3: Domain model

The Voter is a specialized type of User, indicated by a shared id attribute, encapsulating only the necessary attributes for voting purposes, such as username and voter identifier. The Vote entity represents an individual vote, with a boolean answer attribute and a unique identifier, which is associated with a Question. Each Question contains attributes for the total number of votes, counts of yes and no votes, the text of the question, and an identifier.

The model also includes a Poll entity, which aggregates Questions. It has attributes for the poll name, access mode, an active status indicator, and an identifier. The IoT-Device entity represents devices that may interact with the system, characterized by an identifier, a name, another unique identifier, and a status boolean, which could be used for integrating IoT capabilities within the application. This domain model serves as a blueprint for the systems structure, outlining the key data elements and their interconnections, which are crucial for understanding the relationships and data flow within the application.

3.1 Design Patterns

3.1.1 Model-View-Controller (MVC)

Our application employs the **Model-View-Controller (MVC)**[8] design pattern, a paradigm for creating software with separated concerns, enhancing code reusability, and facilitating parallel development. The implementation of MVC within our voting application is outlined as follows:

Model: The model layer manages data and business logic. It encompasses domain entities corresponding to database tables, abstracting database access through Spring Data JPA with Hiber-

nate. The models encapsulate the application's data structures and business rules.

View: Implemented using React.js, the View is a dynamic user interface for client-side interactions. Tailwind CSS is integrated for responsive and aesthetic styling. The frontend operates independently, interfacing with the server through API calls.

Controller: Controllers in the 'controller' package mediate between Model and View, handling HTTP requests with '@RestController' annotations. They direct user inputs to model operations and return appropriate responses, whether as rendered views or data payloads.

Advantages of MVC in Our Application:

- **Separation of Concerns:** MVC facilitates maintenance and scalability with its independent component operation.
- **Parallel Development:** Developers can work on the Model, View, and Controller concurrently, expediting development cycles.
- **Testability:** The separation allows for precise automated testing of business logic and user interfaces.
- **Adaptability:** Business logic or UI changes incur minimal impact on the overall architecture.

The MVC design pattern is integral to our application's design, ensuring a clean separation between user interface, business processing, and data management components, making the system robust, maintainable, and adaptable.[17]

3.2 Architectural Layers Overview

Our web application's architecture is thoughtfully organized into distinct layers, each serving a specific function, yet collectively contributing to a cohesive and efficient system.[9]

The **Frontend Layer** features React.js for crafting the user interface, where components are meticulously designed to present the application's UI, facilitating data presentation and user interaction. Tailwind CSS is intricately used within this layer to style the React components, offering an enhanced user experience through a customized look and feel.

In the **Application Layer**, controllers, integral to our Spring Boot backend, manage HTTP requests. They process incoming data, trigger business logic, and return responses to the frontend, functioning as a pivotal connection point in the application flow.

The **Business Logic (Service) Layer** houses the core of our application's operations. It contains service classes that encapsulate the business rules and computations, bridging the gap between the presentation layer and the data access layer.

The **Data Access (Persistence) Layer** is key in managing data interactions. It comprises JPA repositories that abstract data layer operations and models that represent the application's business data,

mapped to database tables. Hibernate, as our chosen JPA implementation, seamlessly handles the object-relational mapping, translating between the object model and the relational database representation.

Our **Configuration Layer** encompasses Spring configuration classes, which define beans and configure database and security settings, ensuring a robust and secure application infrastructure.

The **Database Layer** utilizes the H2 Database thats ideal for persisting application data, accessed via the JPA repositories.

The **Authentication Layer** features Firebase Authentication services, crucial for user authentication. It is integrated into the application layer to secure endpoints and manage user sessions effectively.

Finally, the **Build and Deployment Infrastructure** employs Gradle for dependency management, compilation, and packaging, while Docker is used to create a containerized application environment, promoting consistency across various development and production settings. This modular layering not only enhances the maintainability and scalability of the application but also aligns perfectly with agile and DevOps practices, supporting continuous development and deployment.

4 Prototype Implementation

4.1 JPA Repositories

A key component of our application's persistence layer is the use of the Java Persistence API (JPA)[18], which provides a specification for managing relational data in Java applications. We leverage JPA in our project to streamline database operations and facilitate object-relational mapping. As part of our implementation, we utilize Spring Data JPA, which extends the standard JPA repository interfaces with powerful data access methods, reducing the need for boilerplate code. The UserRepository is a prime example of this implementation. It is annotated with @Repository, indicating that it is a JPA repository and making it a candidate for Spring's component scanning to detect it as a Spring Bean. We extend the 'JpaRepository' interface, providing CRUD operations for User. This extension allows us to utilize pre-built methods such as save, findById, and delete without explicitly defining them. The method declarations is custom query methods that Spring Data JPA will automatically implement.

```
1
2  @Repository
3  public interface UserRepository extends JpaRepository<User, Long> {
4
5      Optional<User> findByFirebaseUID(String firebaseUID);
6
7      Optional<User> findByUsername(String username);
8  }
```

4.2 Controller Implementation

In the heart of our backend's HTTP request handling is the PollController, a class dedicated to managing poll-related operations. The controller is annotated with @RestController, indicating that it's ready to handle web requests. REST stands for REpresentational State Transfer [15]. We've implemented two main request mappings in this controller.

The first mapping, @GetMapping("/id"), is designed to retrieve a poll by its unique identifier. When a GET request is made to this endpoint with a poll's ID, the pollService.findById(id) method is called. This service method returns an Optional-Poll, which is a container that may or may not contain a non-null value, depending on if a poll with the given ID exists. If the poll is found, it is returned with an HTTP 200 status, otherwise, an HTTP 404 status is returned to indicate that the resource is not found.

```
@GetMapping("/{id}")
public ResponseEntity<Poll> getPollById(@PathVariable Long id) {
    Optional<Poll> poll = pollService.findById(id);
    if (poll.isPresent()) {
        return ResponseEntity.ok(poll.get());
    } else {
        return ResponseEntity.notFound().build();
    }
}
```

The @PostMapping method in our application serves to create a new poll based on user input. When a POST request is received at this endpoint, the method createPoll is invoked, expecting a JSON representation of a Poll object in the request body and a Firebase ID token in the 'Authorization' header. Utilizing a custom utility method FirebaseFunctions.getUidFromToken, we decode this ID token to extract the user's Firebase UID, which is essential for associating the poll with the correct user account. The action dweetioController.sendToDweet publishes the newly created poll, making it accessible for real-time updates and interactions.

```
@PostMapping
public Poll createPoll(@RequestBody Poll poll, @RequestHeader(name =
    "Authorization") String idToken) {
    // Extract the Bearer token
    idToken = idToken.replace("Bearer ", "");

    // Decode the ID token to get the Firebase UID
    String firebaseUID = FirebaseFunctions.getUidFromToken(idToken);

    // Save the question first
    Question savedQuestion = questionService.save(poll.getQuestion());
    poll.setQuestion(savedQuestion);

    // Find the user by Firebase UID
    User user = userService.findByFirebaseUID(firebaseUID)
            .orElseThrow(() -> new RuntimeException("User not found with Firebase
                UID: " + firebaseUID));
```

```
16
17      // Associate the poll with the user
18      poll.setUser(user);
19
20      // Generate a unique code for the poll
21      String uniqueCode = PollCodeGenerator.generateCode();
22      poll.setCode(uniqueCode);
23
24      Poll savedPoll = pollService.save(poll);
25
26      dweetioController.sendToDweet(savedPoll);
27
28      return savedPoll;
29  }
```

4.3 Frontend Implementation

The frontend of our application is implemented using React, a declarative, efficient, and flexible JavaScript library for building user interfaces. Here is how we have structured the frontend to interact with our backend services and third-party integrations:

React and Tailwind CSS

React's component-based architecture allows us to build encapsulated components that manage their own state and compose them to make complex UIs. Tailwind CSS is used alongside React to style our components with utility classes, ensuring a responsive and modern user interface.

The App function in this React code snippet is the main component that sets up the client-side routing for our application using React Router. This function returns a Router component, which defines the navigation context for the entire application. Within the Router, a Routes component is used to declare different URL paths and associate them with corresponding React components that should be rendered when the application navigates to those paths.

```
1   function App() {
2       return (
3           <Router>
4               <Routes>
5                   <Route path="/" element={<Welcome />} />
6                   <Route path="/login" element={<Login />} />
7                   <Route path="/register" element={<Register />} />
8                   <Route path="/home" element={<PrivateRoute><Home /></PrivateRoute>}
                        />
9                   <Route path="/createPoll" element={<CreatePoll />} />
10                  <Route path="/poll/:code" element={<PollVoteDisplay />} />
11                  <Route path="/pollPage/:pollID" element={<PollData />} />
12                  <Route path="/ThankYou" element={<ThankYou />} />
13              </Routes>
```

```
        </Router>
    );
}
```

Login component, as shown below, is a functional component in React used for user authentication. It employs React's `useState` hook to manage the states for user email and password, and `useNavigate` for navigation post-login. This component integrates with Firebase to authenticate users. The 'handleLogin' function manages the login process, handling both successful and failed login attempts. The UI is crafted using a combination of React and Tailwind CSS, which aids in creating a responsive design. Key elements include input fields for email and password, and a custom `Button` component for submitting the form. Additionally, a `BackButton` is included to enhance navigation.

```
const firebaseInstance = new Firebase();

function Login() {
    const [email, setEmail] = useState('');
    const [password, setPassword] = useState('');
    const navigate = useNavigate();

    const handleLogin = async (e) => {
              e.preventDefault();
        try {
            await firebaseInstance.signIn(email, password);
            const currentUser = firebaseInstance.auth.currentUser;
            if (currentUser) {
                const idToken = await currentUser.getIdToken();
                console.log("ID Token:", idToken);
            } else {
                throw new Error('No user is currently signed in.');
            }
            navigate('/home');
        } catch (error) {
            alert('Error logging in: ' + error.message);
        }
    };

    return (
        <div className="welcome-background min-h-[80vh] flex items-center
            justify-center bg-gray-200">
            <div className="beigeBox relative md:w-1/2 lg:w-1/3 xl:w-1/4 p-4 md:p-6
                lg:p-8 rounded-xl shadow-lg flex flex-col justify-center
                items-center">
                <div className="absolute top-0 left-0 p-4">
                    <BackButton />
                </div>
                <h2 className="text-xl md:text-2xl lg:text-3xl font-bold
                    mb-4">Login</h2>
                <form onSubmit={handleLogin}>
                    <div className="mb-4">
                        <label className="block text-sm md:text-base lg:text-lg
                            font-medium text-gray-600 mb-2">Email</label>
```

```
                        <input
                            type="email"
                            placeholder="Email"
                            value={email}
                            onChange={(e) => setEmail(e.target.value)}
                            className="w-full p-2 md:p-2.5 lg:p-3 border rounded-md
                                text-sm md:text-base lg:text-lg"
                        />
                    </div>
                    <div className="mb-6">
                        <label className="block text-sm md:text-base lg:text-lg
                            font-medium text-gray-600 mb-2">Password</label>
                        <input
                            type="password"
                            placeholder="Password"
                            value={password}
                            onChange={(e) => setPassword(e.target.value)}
                            className="w-full p-2 md:p-2.5 lg:p-3 border rounded-md
                                text-sm md:text-base lg:text-lg"
                        />
                    </div>
                    <div className="flex items-center justify-center">
                        <Button text="Login" type="submit"/>
                    </div>
                </form>
            </div>
        </div>
    );
}

export default Login;
```

5 Test-bed Environment and Experiments

The web application's testing strategy involved a blend of manual testing in the Integrated Development Environment (IDE) and using Postman for HTTP request evaluations. This approach aimed to ensure the application's reliability and functionality.

5.1 Testing With Postman

For backend validation, Postman was the tool of choice. It helped test HTTP requests, ensuring backend stability before integration with the frontend. The process began by identifying the types of requests the frontend would make, including those for polls, results, users, and votes. These requests were defined and executed in Postman, allowing for the observation and verification of output in JSON format. This approach resulted in a set of repeatable tests to continually verify backend performance, especially when modifications were made.

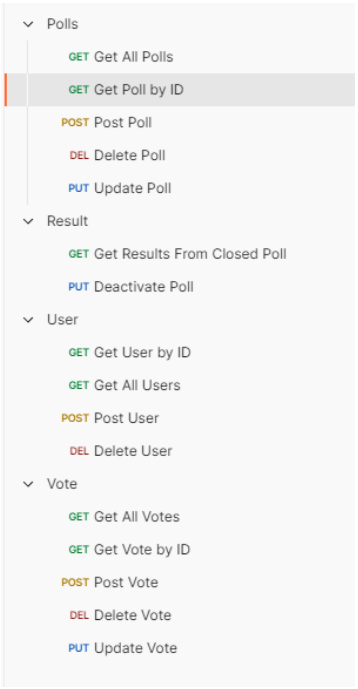

Figure 4: List of Postman requests

Initially, our focus was to determine the range of requests that the front end of the application would execute. This investigation culminated in the identification of a series of key requests, specifically targeting Polls, Results, Users, and Votes. This comprehensive list of requests is illustrated in Figure 4, providing a clear overview of the interactions expected between the front end and the back end of our system.

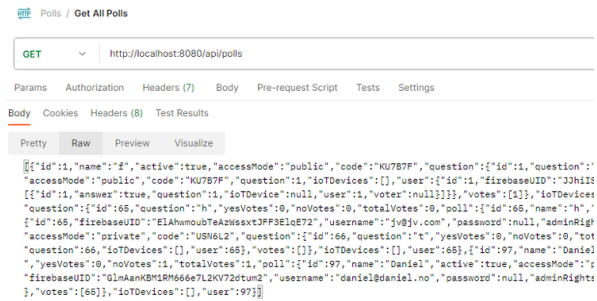

Figure 5: The request for all Polls in Postman

Subsequently, these identified requests were meticulously formulated and executed to monitor their respective outputs, which were primarily in JSON format. This process is exemplified in Figure 5 and Figure 6, showcasing the outcomes of different requests. The execution of these requests not only allowed for real-time observation of system responses but also led to the creation of a suite of tests. These tests could be readily deployed to evaluate the backend's functionality at any given time. This approach ensured a robustly tested and verified backend, establishing a solid foundation prior to initiating the development of the frontend.

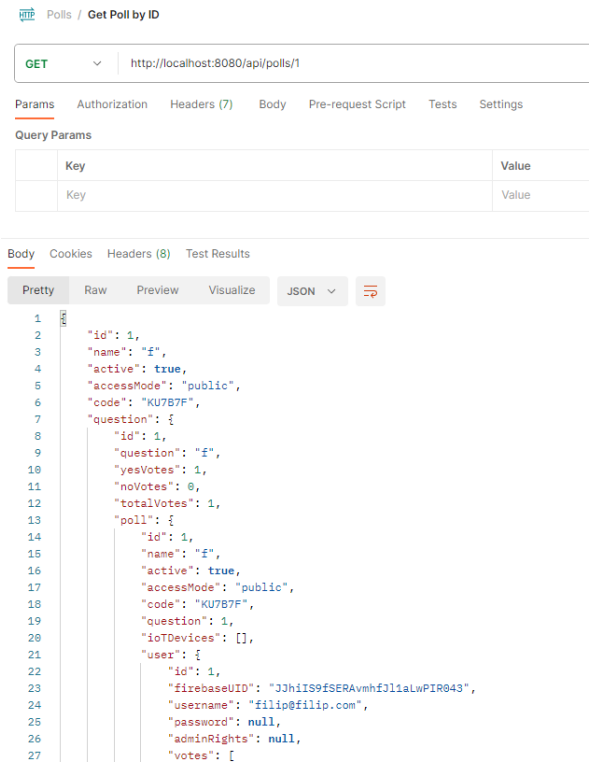

Figure 6: Request for a specific Poll by ID, and the resulting JSON

5.2 IDE-Based Manual Testing

With the backend nearly complete and validated, frontend testing was conducted by running the application in the IDE. This involved using the web page to assess functionality and utilizing the IDE's debugger for in-depth testing. Regular checks and console printing were integral to this process.

5.3 Testing Experience

Overall, the testing experience was highly effective. Testing the backend with Postman before moving to the frontend streamlined the integration process, reducing the need for extensive backend revisions. The saved Postman requests also facilitated periodic retesting, ensuring continued backend integrity.

Frontend testing was also successful, with new features being checked alongside existing ones to maintain overall functionality. However, this manual approach to testing may become cumbersome as the application scales, indicating a potential need for automated testing in future development phases.

Regular presentations required us to thoroughly test all the old and new parts of our project, so everything ran smoothly when we showed it off. This approach really highlighted how important it is to keep testing our work carefully and consistently as we develop it.

6 Conclusion

In our project, we adeptly leveraged several advanced technologies, each contributing significantly to our application's success. The front end was built using React and Tailwind CSS [11], offering a dynamic and responsive user interface. Despite the learning challenges, their extensive documentation aided our development process.

For the backend, we chose Spring Boot [3], appreciating its robustness and scalability. The learning curve was steep, but its comprehensive features and industry maturity justified our choice. We integrated RabbitMQ [13] for efficient message handling and asynchronous tasks, which, while demanding a deeper understanding of message queuing, proved invaluable.

Our data handling was streamlined using Hibernate [12] and JPA, enhancing our interactions with the database and integrating smoothly with Spring Boot. Lastly, Firebase [10] provided a secure and easy-to-implement authentication framework. This project stands as a testament to the effectiveness of combining modern, robust technologies with sound design and implementation practices to create scalable, efficient, and maintainable software solutions.

References

1. Brown, A.W., Wallnau, K.C. (1996). A Framework for Evaluating Software Technology. *IEEE Software*, *13*(5), 39-49.

2. Goadrich, M., Rogers, M. (2011). Smart Smartphone Development: iOS versus Android. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education* (pp. 607-612).

3. *Spring Boot*. (2023). Accessed: November 24, 2023, https://spring.io/projects/spring-boot.

4. *AMQP Version 0-9-1 Specification*. (2023). Accessed: November 25, 2023, https://www.rabbitmq.com/resources/spe 9-1.pdf.

5. *Process Flow Diagrams*. (2023). Accessed: November 25, 2023, https://www.lucidchart.com/pages/process-flow-diagrams.

6. *What is Use Case Diagram?*. (2023). Accessed: November 25, 2023, https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-use-case-diagram/.

7. Ensing, N. (2018). What is a domain model? - Stack Overflow. Answered November 30, 2018, at 14:39. Accessed: November 25, 2023, https://stackoverflow.com/questions/1863537/what-is-a-domain-model.

8. Sandesh. (2018). MVC Design Pattern. *Medium*. Accessed: November 25, 2023, https://medium.com/@sandesh$_{30/m}$

9. *Domains*. (2023). Accessed: November 25, 2023, https://enterprisearchitecture.harvard.edu/domains.

10. *Firebase*. (2023). Accessed: November 25, 2023, https://firebase.google.com/.

11. *Tailwind CSS*. (2023). Accessed: November 25, 2023, https://tailwindcss.com/.

12. *Hibernate*. (2023). Accessed: November 25, 2023, https://hibernate.org/.

13. *RabbitMQ*. (2023). Accessed: November 25, 2023, https://www.rabbitmq.com/.

14. *JPA Introduction - JavaTpoint*. (2023). Accessed: November 25, 2023, https://www.javatpoint.com/jpa-introduction.

15. Fielding, R.T., Taylor, R.N. (2002). Principled design of the modern web architecture. *ACM Transactions on Internet Technology (TOIT)*, *2*(2), 115-150. doi:10.1145/337180.337228. Accessed: November 25, 2023, https://doi.org/10.1145/337180.337228.

16. Dragoni, N., Giallorenzo, S., Lafuente, A., Mazzara, M., Montesi, F., Mustafin, R. (2017). Microservices: Yesterday, Today, and Tomorrow. In *Present and Ulterior Software Engineering* (pp. 195-216).

17. *MVC Framework Tutorial*. (2020). *Tutorialspoint*. Available online. Accessed: November 25, 2023, https://www.tutorialspoint.com/mvc$_f ramework/mvc_f ramework_t utorial.pdf$.

18. *Java Platform, Enterprise Edition: The Java EE Tutorial*. (2014). Release 7, *E39031-01*. September.

19. Tudose, C., Odubăşteanu, C. (2021). Object-relational Mapping Using JPA, Hibernate and Spring Data JPA. In Proceedings of the 2021 23rd International Conference on Control Systems and Computer Science (CSCS) (pp. 1-2). DOI: 10.1109/CSCS52396.2021.00076. Publisher: IEEE.

20. Guinard, D., Trifa, V.M. (2016). Building the Web of Things. Manning Publications Co.

21. Pramono, L.H., Javista, Y.K.Y. (2021). Firebase Authentication Cloud Service for RESTful API Security on Employee Presence System. In *2021 4th International Seminar on Research of Information Technology and Intelligent Systems (ISRITI)* (pp. 1-15). IEEE. DOI: 10.1109/ISRITI54043.2021.9702776.