Report SpringApp Project Group 4

Jacek Antoni Wegrzynowski, Rashaad Wells Iversen, Veronicha C. T. Pettersen

November 30, 2023

Abstract

This document describes the journey of a group of students at Western Norway University of Applied Sciences took to develop a web-based application prototype to create poll and receive votes. The prototype's creation involved a new learned technology stack, integrating Angular, Spring Boot, JPA, Hibernate, H2, MQTT and Java. We achieved significant functionalities, including user registration and authentication, poll and question creation, poll searching, and versatile voting mechanisms via both web interface and IoT integration. Remarkably, these milestones were reached despite the team's initial lack of Java expertise, demonstrating a commendable learning curve and adaptability. The successful implementation of these features showcases the prototype's potential to deliver a final product for polling. While the complexity of our work may not align with every expectation, we take considerable pride in the achievements we have realized. Our efforts reflect significant dedication and learning, particularly in areas initially unfamiliar to us.

1 Introduction

This document outlines the development of a web-based application designed to facilitate interactive polling and voting processes. The application prototype was developed to meet specific requirements for the group project.

- Web Front-End: The application will feature a user-friendly web front-end, enabling users to interact with the system via a web browser. This interface will serve as the primary point of user interaction for conducting polls and voting.
- Business Logic Implementation: Essential business logic will be embedded within the application to handle the processing of polls, votes, user interactions, and other related functionalities.
- REST API: The application will include a RESTful API, exposing a select subset of the business logic. This API will facilitate external interactions and integrations with the application.
- Database Integration: Persistent storage of polls, voters, users, and other relevant entities will be managed through a relational database, ensuring data integrity and consistency.
- Voting and Display Device: The project encompasses the development of a voting and display device, which may be either virtual or physical. This device will interact directly with the application's business logic and contribute to the polling process.
- Publish-Subscribe System Integration: Results and outcomes of polls will be disseminated through a publish-subscribe messaging system, enhancing the reach and accessibility of polling data.

- Dweet.io Integration: Key events such as the initiation and conclusion of polls, along with pertinent information, will be published to dweet.io using RESTful services, ensuring real-time updates and public accessibility.
- Analytics Component: An analytics subsystem will be implemented, tasked with subscribing to poll data and storing results in a NoSQL database. This component will enable advanced data analysis and insight generation.
- Security Considerations: The application will be developed with a strong focus on security aspects, including confidentiality, authentication, and authorization, to protect user data and ensure system integrity.
- Cloud Deployability: While the application will be designed for cloud deployment, actual deployment to a cloud environment is not a mandatory requirement of this project.

The development of this application aims to integrate these requirements into a single system allowing an interactive polling and voting environment.

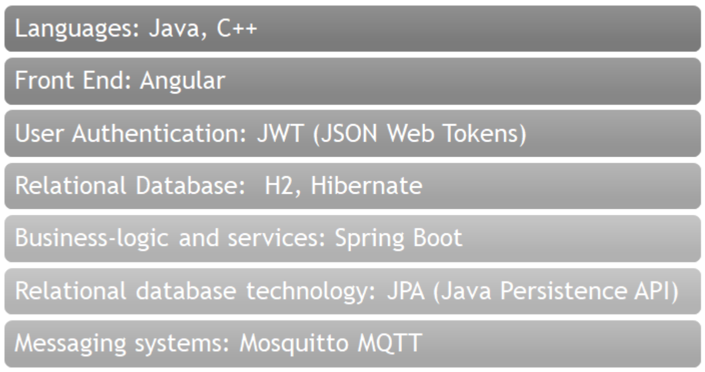
The technology stack for this project includes a variety of frameworks and systems, each in response to specific aspect of the application's functionality. For the front-end development, we have used Angular to create a responsive and interactive user interface. The business logic and backend services are powered by Spring Boto. To manage the data, we utilize JPA for our relational database needs, ensuring effective data management and retrieval, while H2 is employed for non-relational database operations, offering flexibility and scalability on a Hibernate database. Additionally, Mosquitto is the chosen messaging system, facilitating reliable and efficient message handling across different parts of the application.

In the prototype implementation of our application, we successfully achieved several key functionalities, crucial for its operation and user engagement. The prototype allows for user registration, enabling new users to create accounts and gain elevated access. Once registered, users can seamlessly log in and log out, ensuring secure access to their profiles and activities. A significant feature of the prototype is its capability to create polls, allowing users to initiate new voting sessions. In addition to this, users can also create unlimited questions within these polls. The system includes a feature to search for and find specific polls.

Voting mechanisms have been implemented in two distinct forms: users can cast their votes through the web interface, providing ease of access and convenience and/or have the functionality to cast votes via an IoT device, showcasing the application's capability to interface with physical hardware for a more interactive polling experience. The prototype successfully demonstrates a few aspects of the creation of a web application with the ability to perform CRUD operations and handle user authentication. API testing with Postman, demonstrates the application to be responsive and functional. The prototype's functionality, achieved under these circumstances, thus not only showcases the application's capabilities but also reflects the team's dedication and skill in overcoming technical challenges.

In this document, we have started with this introductory section where we explain the background and context of the assignment, setting the stage for an understanding of the objectives of the project. Second, provide an overview of our technology stack, including a breakdown of each technology's role and purpose in section 2. Next, we provide an in-depth explanation of the system architecture and design, detailing how we planned to build our project in section 3. This is followed by section 4, an explanation on how we actually implemented the our ideas, including code snippets on topics we found specifically interesting to describe in depth. Furthermore in section 5, we describe testing methodologies and experiments used to ensure the functionality of the application performed as expected. Section 6 concludes the report, presenting a reflective overview of our journey through this project. This section boasts our achievements, discusses the challenges and lessons learned from this experience, offers a candid appraisal of our overall teamwork, and expresses our sentiment towards missed opportunities.

2 Software Technology Stack


Languages: Java, C++

Front End: Angular

User Authentication: JWT (JSON Web Tokens)

Relational Database: H2, Hibernate

Business-logic and services: Spring Boot

Relational database technology: JPA (Java Persistence API)

Messaging systems: Mosquitto MQTT

Technology Stack

2.1 Angular

We chose Angular as our web application framework to develop the frontend of the FeedApp prototype. Angular is a popular, open source TypeScript based framework[1] that is used to create Single Page Applications (SPAs). SPAs are web applications that only loads one single page, and then changes the content of that page depending on how the user interacts with the page.

2.1.1 Fundamental Consepts of Angular

Components: Components are many places described as the main building blocks for Angular applications. Each component will contain one HTML-file with the UI-template of the component and one TypeScript- file that contains the components logic. It can also contain CSS or SCSS files, defining the styles of the component, as well as test files and configuration files. A component defines a specific view, as well as the functionality that goes into that view. In other words, it contains both what the user sees in the UI and the logic that goes into the component.

Services: It is also possible to share different functions and logic between different components. This can be done by creating a service, which is a TypeScript file that is used for tasks such as for example business logic and handling of data. In our FeedApp implementation, we created services dedicated to managing authentication and handling poll data.

Dependency Injection (DI): Services can also be used to inject dependencies into multiple components, with the use of DI. This is useful for connecting the different parts of the application.

Routing: The Angular Router handles the navigation between different views as users performs different tasks. This is a key element in SPAs since instead of reloading the page every time the view is changed.

One of the main advantages of creating the application as a SPA is that it speeds up the development process.

[1]https://angular.io/guide/architecture

2.2 Spring Boot

We have chosen Spring Boot as our enterprice software framework when developing our application. Spring Boot is an extension of the Spring Framework that simplifies the development process, making it possible to create a functioning web application fast. Considering the time limitation we had on our project, Spring Boot seemed like a good choice when choosing a framework.

2.2.1 Spring Framework

To fully understand the benefits of the Spring Boot extension, we are first going to briefly go through some core concepts in the Spring Framework[2]:
Bean Definition: Beans are often described as the backbone of a Spring application. They are managed by the Spring Inversion of Control (IoC) Container. How they are configured, and how their lifecycle is managed plays an important role when having a smooth and proper running application.

Dependency Injection (DI): Spring's DI mechanism manages dependencies among application components. This setup is crucial for injecting required services and modules into different parts of the application.

Aspect-Oriented Programming (AOP): In Spring, AOP makes it possible to handle tasks that are common in multiple parts of an application efficiently by defining the function and then applying it to the sections that need to use it.

How a Spring application is deployed:

- Packaging: The application is compiled and packaged, typically into a JAR or WAR file, ready for deployment.
- Running on a Server: The packaged application is deployed on a web server. Spring Boot, with its embedded server capability, simplifies this by allowing the application to run independently without needing a separate server setup.

2.2.2 Spring Boot's Role in FeedApp:

Auto-Configuration: Spring Boot automatically configures the application based on the included libraries, reducing the need for extensive manual configuration.

Simplified Deployment: The embedded server feature of Spring Boot allows our application to be deployed as a standalone unit, enhancing ease of deployment and portability.
In the implementation of our FeedApp prototype, the use of Spring Boots auto configuration and deployment functionalities has made it possible for us to spend more time on the applications business logic.

2.3 JSON Web Tokens

We used an open standard called JSON Web Tokens (JWT), that provides a condensed, self-contained method for securely exchanging data as a JSON object between parties. This data is digitally signed,

[2]https://docs.spring.io/spring-framework/reference/index.html

so it can be validated and trusted. RSA or ECDSA can be used to sign JWTs with a secret key or a public/private key pair. [3]

JWTs are essential to our FeedApp's authentication procedure. A JWT is given to a user upon login. Subsequent requests to authenticate the user and grant access to restricted routes, services, and resources within the application will then utilize this token. Because of their short size and ease of use, JWTs are the ideal format for our application and work well in our web-based environment.

2.4 H2 Database

We used a quick and easy approach to store and manage data with the lightweight H2 database. Because of its cheap overhead and ease of setup, it's especially helpful throughout the development and testing phases. H2 can operate embedded within a Java program or in client-server mode.

We are using the H2 database in FeedApp development for testing and prototyping. It eliminates the requirement for a complicated database setup and enables us to simulate database interactions and run unit tests. The H2 database is the perfect option for our needs for rapid development because it interfaces with our Spring Boot platform.

2.5 Hibernate

For Java applications, Hibernate is a powerful Object-Relational Mapping (ORM) framework. It offers a framework for converting a conventional relational database to an object-oriented domain model. Hibernate resolves the mismatch issues by substituting high-level object handling routines with direct, permanent database accesses. [4]

Hibernate is used in our FeedApp to make it easier for our Java application to communicate with the database. The CRUD (Create, Read, Update, Delete) procedures are made simpler and more effective by managing database operations and data persistence. Our team is able to concentrate more on the business logic and less on database management because Hibernate abstracts the database logic.

2.6 Java Persistence API

A Java specification for managing, retrieving, and storing data between Java objects and a relational database is called the Java Persistence API (JPA). JPA, which is a component of the Java EE platform, makes it easier to create Java apps that communicate with databases.

JPA is implemented into FeedApp to manage our application's relational data. It offers a platform for managing relational data in Java programs and executing database operations on Java Entities. Hibernate and JPA work together to make database administration in our application more error-free and efficient.

[3] https://datatracker.ietf.org/doc/html/rfc7519

[4] https://hibernate.org/orm/

2.7 Mosquitto MQTT

A simple publish-subscribe network protocol called MQTT (Message Queuing Telemetry Transport) is used to send messages between devices. Mosquitto is portable and works with a wide range of devices, including powerful servers and single-board computers with minimal power consumption. [5]

Mosquitto MQTT is essential to our FeedApp since it manages real-time messages and facilitates effective connectivity with IoT devices. MQTT is a good solution for our IoT-related capabilities since it guarantees efficient and reliable message delivery even in limited contexts. Maintaining the application's responsiveness and performance requires minimal resource usage, which is ensured by its lightweight design.

[5] https://mosquitto.org/

3 Design of the FeedApp Prototype

3.1 Architectural Overview

Our fundamental idea of the FeedApp application was to mimic some of the Kahoot application capabilities while remaining simplistic, keeping alignment with good practices in software development and prototyping. Our archtiecutre aims to meet requirements, demonstrate core functionality, and balance the effort put into development of a prototype.

The application is accessible through web browsers, ensuring compatibility with devices such as smartphones, tablets, and computers. The application provides registered users with the ability to create and participate in various polls. It offers the flexibility for users to cast their votes using either a physcial IoT device or through a web browser. The application incorporates a user authentication system, ensuring secure access and protecting user identities and information.

3.2 Domain Model

Our domain model of the FeedApp, is represented in a Unified Modeling Language (UML) class diagram representing the fundamental objects and their relationships. By doing this we describe behavior and functions in a clear way so that we are all on the same page. Throughout the development lifecycle of the FeedApp, this domain model has undergone iterative modifications. These adaptations were essential to maintain alignment with changes made in our objectives. A primary focus during these modifications has been the preservation of simplicity within the system's architecture. By continuously refining the domain model, we have ensured that the FeedApp remains functionally efficient.

3.2.1 Users

The domain model describes an entity of a user and categorizes it into two distinct types based on account registration status. Users with registered accounts are granted full access within the FeedApp including the capability to create and participate in both public and private polls. This category of user must undergo an authentication process which is described in TODO:Section. Users without registerd accounts are only allowed to participate in publicly accessible polls. This restriction is a delibarate choice, allwoing our development efforts to maintain simpliciy in implementation.

3.2.2 Polls

A key feature of the system is the poll entity, characterized by distinct attributes such as a title, an identification number, and the option to be set as private or public. Each poll is required to have a time limit and are also designed to integrate with Internet of Things (IoT) devices. Furthermore, polls are designed to be flexible, allowing an unrestricted number of questions. The application does not impose a limit on the quantity of questions per poll. In addition to this, for private polls, there is a feature to specify a list of authorized users who are permitted to vote. In the design of polls, the questions must be structured as binary-choice and closed-ended. This format restricts responses to one of two predetermined options, exemplified by pairs such as True/False, Yes/No, or Pancakes/Waffles. While the application does not impose time limits on individual questions, it integrates time restrictions at the poll level. This design choice, made for ensuring simplicity in implementation, focuses on the entire poll rather than individual questions.

3.2.3 IOTDevice

For this project, a physical IoT device was chosen to add the interactive dimension to the polling process. This device is intentionally simplistic, capable only of transmitting voting data without knowledge of the specific poll or question involved. It operates via a Mosquitto broker, which relays messages from the IoT device to the feed application. The integration with the IoT device is minimalistic. The device's primary function is to acknowledge a voting action and communicate this to the feed application. It is not equipped to discern details about the poll or the voting options. The absence of an on-device display is compensated for by using a command-line interface to indicate when a vote has been cast. This setup was demonstrated in the submitted video, showcasing the Mosquitto broker's role in facilitating communication between the IoT device and the feed application.

3.2.4 IOTDisplay

The project's infrastructure includes an Internet of Things (IoT) device, functioning independently on a dedicated hardware platform. This device establishes a wireless connection with a Windows-based HP laptop, which serves a dual purpose: as a control unit and as a display for the IoT device's operations. The design stipulates that both the IoT device and the laptop must share the same Wi-Fi network for seamless communication.

3.2.5 IOTDevice Integration with FeedApp

Further enhancing the system's architecture is the incorporation of a Linux virtual machine (VM) on the HP laptop. This VM is designated as the application server, managing the core functionalities of the feedback feed application. Notably, the VM's connectivity is not restricted to the same Wi-Fi network as the IoT device, offering flexibility in network configurations. It communicates with the IoT device through a Mosquitto broker, which acts as an intermediary in message transmission. The IP address of the Mosquitto broker is hardcoded into the system for consistent connectivity.

The application leverages a REST API to handle incoming data from the IoT device. When a user interacts with the IoT device, such as by pressing a button, this action triggers a message that is relayed by the Mosquitto broker to the feed application. The application's REST API is configured to recognize this input, determining the active poll and the specific question being addressed by the IoT device. While the mechanism for transitioning between different poll questions was not fully established, the conceptual framework suggests that the REST API would facilitate this progression by incrementally recording votes and navigating to subsequent questions.

3.2.6 Analytics

Once the a vote is cast on the last quesiton in the poll, the application is designed to immediately reflect the analytical data offering insight into collective responses. The system is designed to facilitate the display of analytics for all polls, regardless of their current status (active or inactive). This functionality is implemented through a standardized display template, which is accessible via web browsers. The template ensures uniformity in the visualization of analytics, providing a coherent and consistent user experience. In addtion, the system's architecture allows for third-party applications to access poll-related data. This capability enables external entities to conduct their own analytical assessments. However, it is important to note that the scope of data accessible to third-party applications is confined to poll-level information. Granular data pertaining to individual questions within the polls is not available for external analysis.

3.3 FrontEnd Design

To get an overview of how we wanted to implement the user interface, an application flow diagram was modeled. The diagram displays how the user navigates through the different frames in the front end of the application:

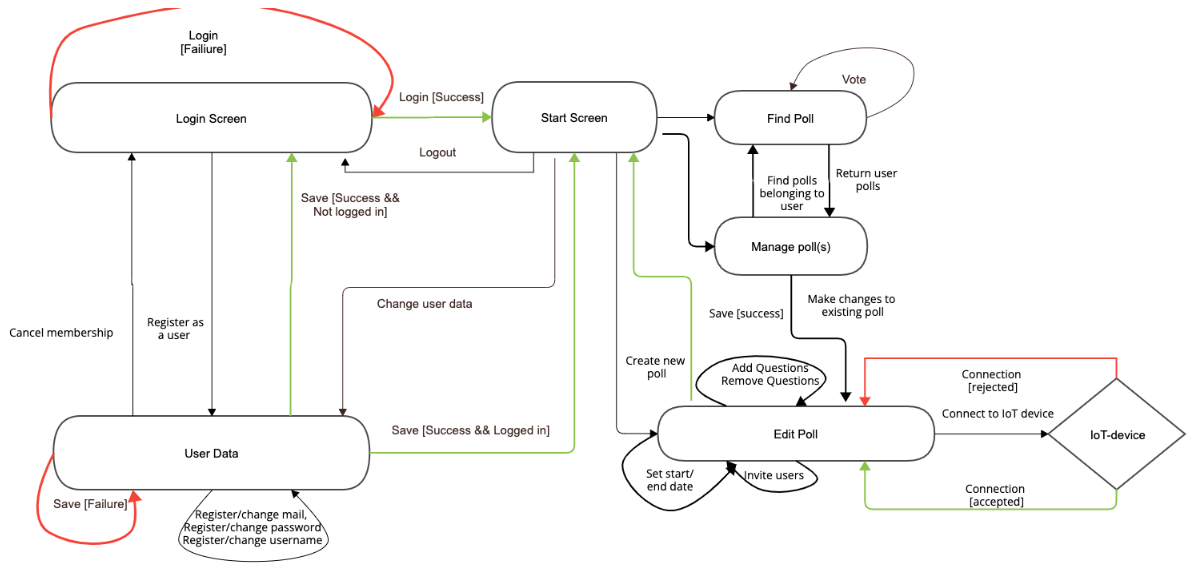


Figure 1: Application Flow Diagram

Each screen has in the diagram been modelled as a state, and for every state, the user is able to do some action or provide some input. These actions or inputs are modelled as transitions, and are in the figure above illustrated with arrows. Transitions that results in errors are colored red, and transitions that does not result in errors are colored green. Our application consists of six screens. We have a login screen, where the user can either login, or create a new user. Once the user is logged in, he is directed to a start screen, where he gets the option to do multiple actions. He can search for a poll, which then can be voted on, or he can view and manage his own polls. He can create new polls, and in this state he gets the option to pair the poll with an IoT device.

4 Prototype Implementation

4.1 Frontend

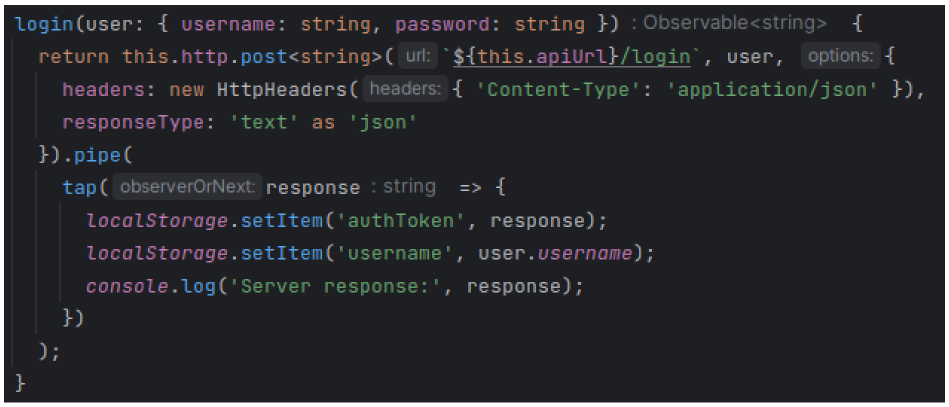
The following components have been implemented for the front end:

- Login: this is the first view that the user are presented with. It's main purpose is to authenticate users, and help them access the application. It contains input fields for username and password. When the user presses "Log in", he is taken to the main-page of the application. If the user is not registered, he can easily do so by pressing the "Register"- button.
- Register: here the user is presented a schema, where he can register and that way access the application. He needs to add an email address, a username and a password. Once the information is submitted, the user is sent back to the login page.
- Main-page: This component contains buttons that lets the user easily navigate to all parts of the application quickly.
- Find-poll: Here the user can search for a poll via its id, or by a poll-name. All the polls that matches the input will then be displayed with the option to vote on it.
- Create-poll: Here the user can create a new poll. He can decide the questions that should be displayed, when the poll is active, if it is private and invite users to participate in it. Once created, a poll-id is given to the user. This can for example be given to other users and used to search for the poll.
- Vote: Here the user gets to submit votes to active polls.

A authentication service, a poll service and a voting service has also been created to handle logic that can be applied to multiple components. The authentication service handles operations such as logging the user in to the application and searching for users. The poll service handles logic depicting the polls such as finding polls, creating polls and changing polls. The Voting service handles the logic connected to the voting. The following subsection describes more detailed how the logic behind the authentification process has been implemented, and displays how the frontend is connected to the REST API.

4.1.1 Authentication Process

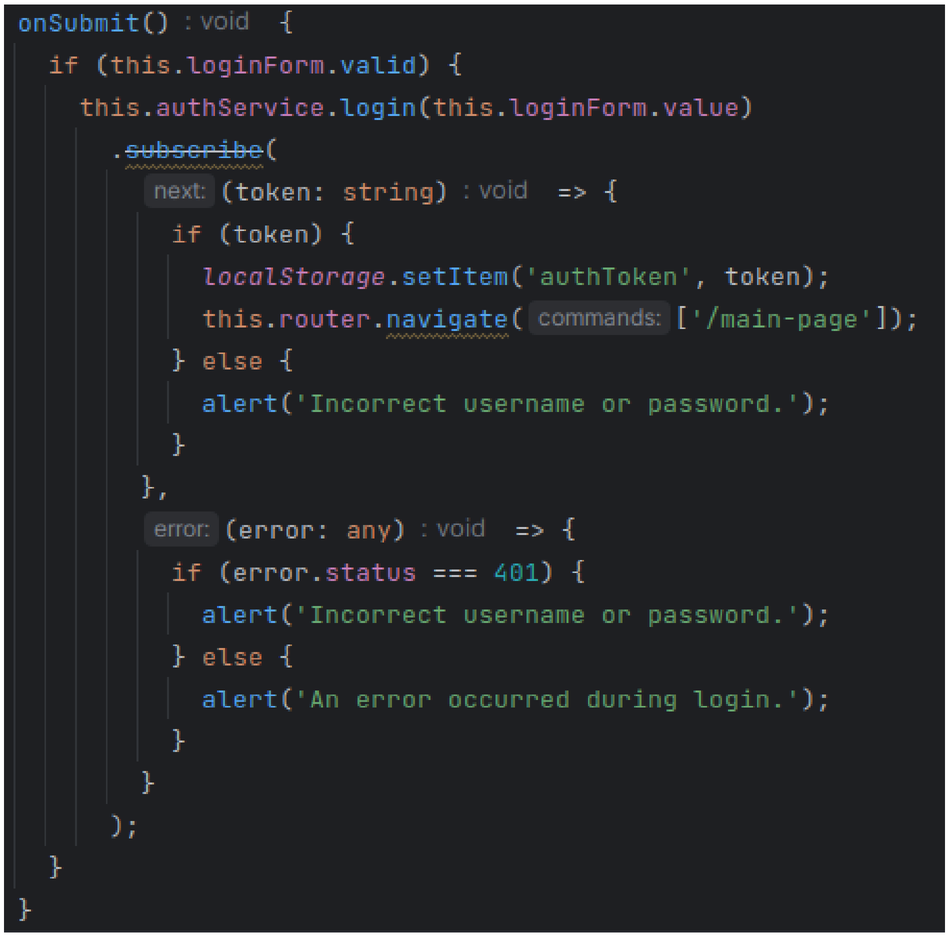
A crucial part of the frontend implementation is the user authentication process. This is managed through the `login` method in the `auth.service` file and the `onSubmit` method in the `LoginComponent`. The `login` method is defined as:



```
login(user: { username: string, password: string }) : Observable<string>  {
  return this.http.post<string>( url: `${this.apiUrl}/login`, user,  options: {
    headers: new HttpHeaders( headers: { 'Content-Type': 'application/json' }),
    responseType: 'text' as 'json'
  }).pipe(
    tap( observerOrNext: response : string  => {
      localStorage.setItem('authToken', response);
      localStorage.setItem('username', user.username);
      console.log('Server response:', response);
    })
  );
}
```

This method handles user login credentials, sending them to the backend for verification and storing the received token and username in the browser's local storage for session management. The token is stored in the local storage to maintain the user sessions.

The `LoginComponent` uses this service in its `onSubmit` method:



```
onSubmit() : void  {
  if (this.loginForm.valid) {
    this.authService.login(this.loginForm.value)
      .~~subscribe~~(
        next: (token: string) : void  => {
          if (token) {
            localStorage.setItem('authToken', token);
            this.router.navigate( commands: ['/main-page']);
          } else {
            alert('Incorrect username or password.');
          }
        },
        error: (error: any) : void  => {
          if (error.status === 401) {
            alert('Incorrect username or password.');
          } else {
            alert('An error occurred during login.');
          }
        }
      );
  }
}
```

Figure 2: Verification Service Example

This function makes sure that only users with valid credentials are able to access the main page of the application, while also handling different error scenarios. If a user gives an incorrect credential or any other login error occurs, an error message is displayed.

4.2 Backend

The architecture of our program has modular architecture that includes multiple important entities and their corresponding repositories.

4.2.1 Entities

We will begin by describing the structural components of the application which is object-oriented by design. The primary class, `AppUser`, implements properties such as username, email, and password. It also holds a collection of polls owned by the AppUser. A critical feature of this class is the implementation of two-factor authentication, validated through a verification code to ascertain user authenticity.

The `Poll` class, a core entity in our application, represents polls created by users. It includes details like title, open/close status, and duration, and can contain multiple questions. The poll implements general properties for storing privacy status, activity status, duration, and IoT device pairing. It also implements specific properties like the poll title, vote tallies, a list of authorized users (for private polls only) and funttion implementation to perform question management such as add, delete and edit. Polls are also linked to IoT device and therefore, holds information on which device it is paired with.

The `Question` class constitutes individual items within polls for users to vote on. Records responses and associates with specific polls.

A `Vote` class records user votes in polls, indicating 'yes' or 'no' choices. It contains the logic that is responsble for determining which question in a poll the vote should be submitted towards.

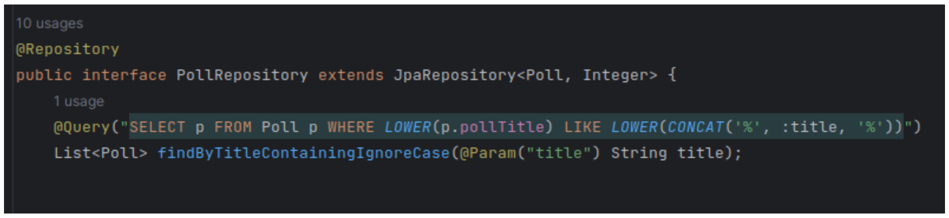
The `IoTDisplay` is a singleton class only aware of its associated IoT device by the property pariedDevice, which contains a string value of of the device name.

`ThirdPartyApp` interface class provides logic to allow third party entties to retreive data from the application.

4.2.2 Repositories

JPA repositories helps with performing create, read,update and delete (CRUD) database operations by reducing the need to manually code complex SQL queries. We use these repositories with JPA EntityManager to interact with the database. We have defined various repositories in the FeedApp responsible for handling specific data operations.

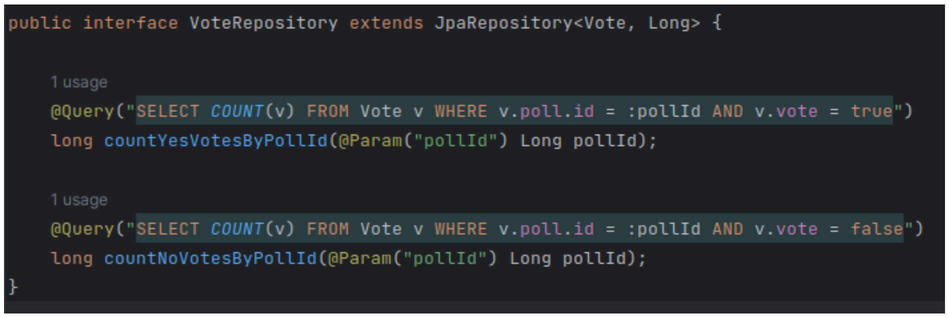
The `AppUserRepository` manages the `AppUserentity` data operations, including user lookup by username. `PollRepository` handles data operations for the `Poll` entity, with custom queries for poll title searches.



```
10 usages
@Repository
public interface PollRepository extends JpaRepository<Poll, Integer> {
    1 usage
    @Query("SELECT p FROM Poll p WHERE LOWER(p.pollTitle) LIKE LOWER(CONCAT('%', :title, '%'))")
    List<Poll> findByTitleContainingIgnoreCase(@Param("title") String title);
```

Figure 3: Custom Poll Title Query

The `QuestionRepository` provides data access for `Question` entity. `VoteRepository` manages `Vote` data, including vote counting in polls.



```
public interface VoteRepository extends JpaRepository<Vote, Long> {

    1 usage
    @Query("SELECT COUNT(v) FROM Vote v WHERE v.poll.id = :pollId AND v.vote = true")
    long countYesVotesByPollId(@Param("pollId") Long pollId);

    1 usage
    @Query("SELECT COUNT(v) FROM Vote v WHERE v.poll.id = :pollId AND v.vote = false")
    long countNoVotesByPollId(@Param("pollId") Long pollId);
}
```

Figure 4: CustomVoting Query

4.2.3 Integration with Spring Boot, Hibernate, and JPA

In the implementation of our voting application, we employed several key technologies and tools including Spring Boot as the primary framework to simplify configuration and setup, Hibernate for Object-Relational Mapping (ORM) translation of Java object to database representations and JPA interfaces such as `JpaRepository` for efficient data access and manipulation.

4.2.4 Functionality

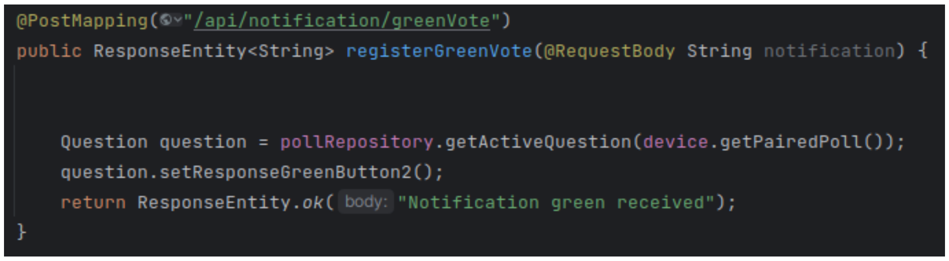
In our application, we implemented several key functionalities including the he ability to register and manage user accounts, features for creating and managing interactive polls, voting mechanisms within these polls and Integration with a physcial IoT devices.

4.3 REST API

The web application consists of three key controllers, each handling different aspects of the voting system:

- The AppUserController is responsible for user-related functions, including user authentication, registration, and the verification process. It implements the AppUser entity using depencency injection by the @Autowired annotation to reference AppUserRepository and AppUserService.

- The PollController oversees the management of polls, specifically handling the publication of poll results and controlling their status, whether open or closed.
- The QuestionController introduces and manages a voting mechanism for individual questions, reflecting a finer level of interaction and user engagement within the application. It is mapped
- The IOTController class in annotated with @RestController which is part of a RESTful web service in a Java application using the Spring Framework. It handles HTTP requests specifically related to IoT device notifications. The controller is mapped to handle requests at the /api/notifications path. It defines two endpoints for processing notifications sent from the IoT device, /api/notification/greenVote and /api/notification/redVote.



```
@PostMapping("/api/notification/greenVote")
public ResponseEntity<String> registerGreenVote(@RequestBody String notification) {


    Question question = pollRepository.getActiveQuestion(device.getPairedPoll());
    question.setResponseGreenButton2();
    return ResponseEntity.ok( body: "Notification green received");
}
```

Figure 5: Green Vote Endpoint

All controllers are designed following REST principles, offering a specific base URLs and using HTTP methods for standard CRUD operations.

4.3.1 Spring Boot and Spring Data Integration

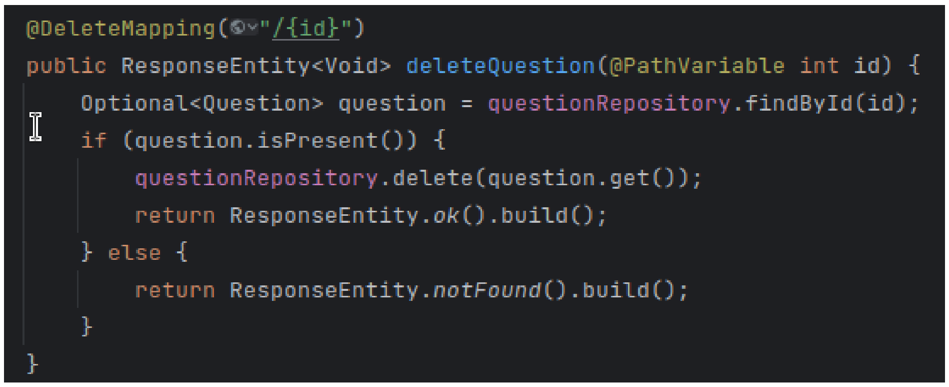
They utilize Spring Boot's capabilities for creating standalone applications and Spring Data's repository pattern for data access. Dependency injection with `@Autowired` is also used.

4.3.2 Error Handling and Response Management

The controllers use `ResponseEntity` class for HTTP response handling, allowing them to manage different scenarios (even those ending with an error). Method `deleteQuestion` in `QuestionController` that is responsible for deleting a question in a poll returns an error response if the question wasn't found:

4.3.3 Security and User Management

The `AppUserController` focuses on security and user identity management through authentication and JWT token management. There are also a few features that we modelled into our application in the first phase of this project, but due to the time constraints are yet to be implemented. These include Poll management and User management. Currently the application allows users to create and participate in polls. However, the management of the polls is not yet functional. The same goes for the management of the user settings. The buttons has been created in the main-page component, but they are not directing the users to new views yet.

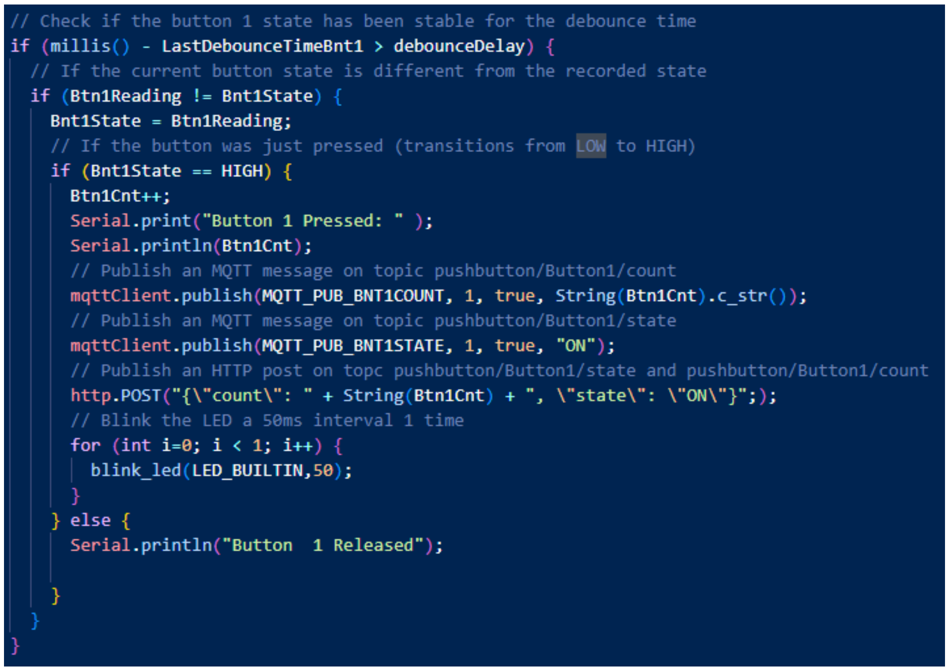


```java
@DeleteMapping("/{id}")
public ResponseEntity<Void> deleteQuestion(@PathVariable int id) {
    Optional<Question> question = questionRepository.findById(id);
    if (question.isPresent()) {
        questionRepository.delete(question.get());
        return ResponseEntity.ok().build();
    } else {
        return ResponseEntity.notFound().build();
    }
}
```

Figure 6: Error Handling Response

4.3.4 IoT Device

The IoT device in our system is a physical unit and is implemented using C++ and the Arduino library[6] and is based on the ESP8266 [7] module which includes Wi-Fi connectivity. This device is programmed with hardcoded Wi-Fi credentials which in our case is a cellar hotspot. Its design is straightforward, featuring just two buttons, referred to as the red and green buttons. The device is configured to publish a message to the MQTT broker whenever a button is pressed; however, it does not register continuous presses but only single clicks. These messages are then relayed by the MQTT broker, post distinct messages via REST interfacing with the IoTController.



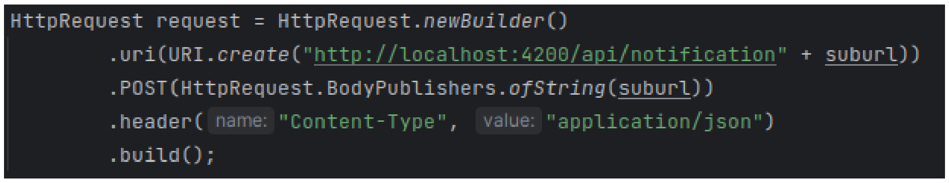
```cpp
// Check if the button 1 state has been stable for the debounce time
if (millis() - LastDebounceTimeBnt1 > debounceDelay) {
  // If the current button state is different from the recorded state
  if (Btn1Reading != Bnt1State) {
    Bnt1State = Btn1Reading;
    // If the button was just pressed (transitions from LOW to HIGH)
    if (Bnt1State == HIGH) {
      Btn1Cnt++;
      Serial.print("Button 1 Pressed: " );
      Serial.println(Btn1Cnt);
      // Publish an MQTT message on topic pushbutton/Button1/count
      mqttClient.publish(MQTT_PUB_BNT1COUNT, 1, true, String(Btn1Cnt).c_str());
      // Publish an MQTT message on topic pushbutton/Button1/state
      mqttClient.publish(MQTT_PUB_BNT1STATE, 1, true, "ON");
      // Publish an HTTP post on topc pushbutton/Button1/state and pushbutton/Button1/count
      http.POST("{\"count\": " + String(Btn1Cnt) + ", \"state\": \"ON\"}";);
      // Blink the LED a 50ms interval 1 time
      for (int i=0; i < 1; i++) {
        blink_led(LED_BUILTIN,50);
      }
    } else {
      Serial.println("Button  1 Released");

    }
  }
}
```

[6]https://www.arduino.cc/reference/en/
[7]https://www.nabto.com/esp8266-for-iot-complete-guide

4.3.5 Messaging Receiver

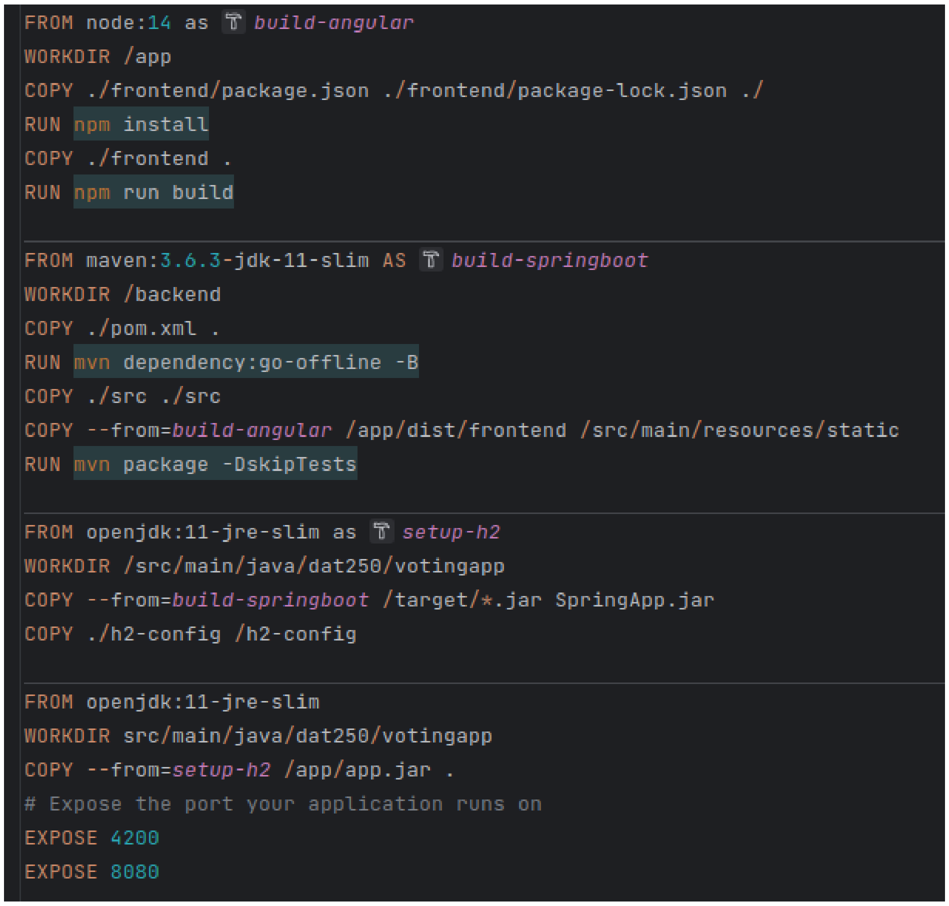
The MQTT message receiver is provided as a service in the voting application's Spring Boot backend. It uses HTTP requests within the registerVote method to send votes to the voting applciation. HTTP requests are asynchronous to ensure the other messages are not blocked. While the current setup is sufficient for the purposes of a prototype application, it's important to note that for a product intended for release, a more secure implementation would be necessary.



```
HttpRequest request = HttpRequest.newBuilder()
        .uri(URI.create("http://localhost:4200/api/notification" + suburl))
        .POST(HttpRequest.BodyPublishers.ofString(suburl))
        .header( name: "Content-Type",  value: "application/json")
        .build();
```

4.3.6 Docker Container

We created a Dockerfile in preparation for cloud deployment. This docker file invokes several events. It builds the front end Angular application, build the Spring Bood back end application, sets an H2 database and then runs the applications.



```
FROM node:14 as build-angular
WORKDIR /app
COPY ./frontend/package.json ./frontend/package-lock.json ./
RUN npm install
COPY ./frontend .
RUN npm run build

FROM maven:3.6.3-jdk-11-slim AS build-springboot
WORKDIR /backend
COPY ./pom.xml .
RUN mvn dependency:go-offline -B
COPY ./src ./src
COPY --from=build-angular /app/dist/frontend /src/main/resources/static
RUN mvn package -DskipTests

FROM openjdk:11-jre-slim as setup-h2
WORKDIR /src/main/java/dat250/votingapp
COPY --from=build-springboot /target/*.jar SpringApp.jar
COPY ./h2-config /h2-config

FROM openjdk:11-jre-slim
WORKDIR src/main/java/dat250/votingapp
COPY --from=setup-h2 /app/app.jar .
# Expose the port your application runs on
EXPOSE 4200
EXPOSE 8080
```

5 Test-bed Environment and Experiments

5.1 Frontend testing

When developing the frontend of our application, multiple testing strategies were performed to make sure that the application's functionalities were behaving as expected. One of the advantages of using Angular, is that it provides the option to do real-time compilation of the application. This feature made it possible to view the results immediately after code changes. This shortened the time spent on troubleshooting errors. The inspect tool in the browser were also a great tool that quickly let us inspect the applications. The console feature were handy when inspecting the HTML and CSS, and the Network analysis tool gave us a great insight into how the frontend and the backend communicated. Alerts were also created to notify the user whenever errors and issues occurred. of errors

5.2 JUnit testing

The purpose of the JUnit tests is to verify our application's functionality. Using a mocking framework, they imitate particular application components, enabling targeted testing of individual functionalities without requiring the entire system. Important components analyzed are:

- The ability of the application to successfully retrieve all items from a list, such as all users or polls.
- Capability to find a specific item by its unique identifier.
- Correct behavior of the application when an item cannot be found.
- The functionality to add, update, or remove items effectively.

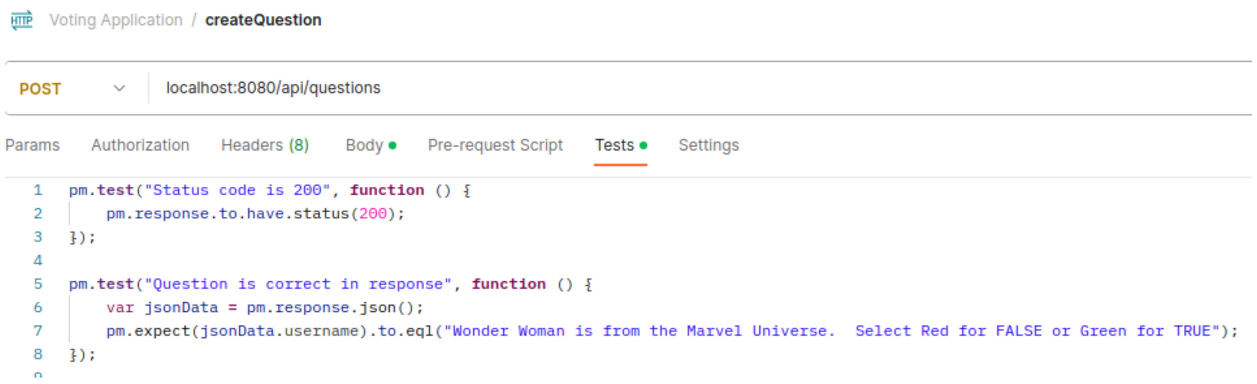
Expected results, such as the quantity of objects recovered or the reaction in the event that an item is missing, are confirmed in each test. These tests are essential to ensure the application functions as expected.

5.3 Postman testing

Postman is a popular tool for testing REST APIs. It allows us to query the API endpoints of our application and receive back responses. The most important features tested using Postman include:

- Sending Different Types of Requests: Postman test various request types like GET, POST, PUT, and DELETE. These are essential for operations in our application such as getting, creating, updating, or deleting data.
- Verifying Responses: The tool can verify whether the API returned the correct data and response codes. For instance, confirming that a GET request fetches the right list of items, or a 404 status code is returned when an item is not found.
- Parameter Testing: Postman allows testing how the API handles different parameters, such as specific item searches or result filtering.
- Error Handling: By sending incorrect or incomplete requests, we can test the API's error handling and check if it returns appropriate error messages.

These tests make sure that our application's API is reliable, and performs as expected under different scenarios.


Voting Application / createQuestion

POST localhost:8080/api/questions

Params Authorization Headers (8) Body Pre-request Script Tests Settings

```
pm.test("Status code is 200", function () {
    pm.response.to.have.status(200);
});

pm.test("Question is correct in response", function () {
    var jsonData = pm.response.json();
    pm.expect(jsonData.username).to.eql("Wonder Woman is from the Marvel Universe.  Select Red for FALSE or Green for TRUE");
});
```

Figure 7: Sample Postman Test

5.4 IoT Device testing

Testing of IoT device was testing using an informal and dynamic approach. Instead of defining specific test cases, we relied on our domain knowledge, experience and creativity to find defects. Using this type of testing allowed us to design, implment and test simultaneiously, enabling us to uncover design and development defects continiously in the software development loop.

6 Conclusions

The overall design framework for the FeedApp project was pre-established at the beginning of the assignment. This design was further amplified through the course of various lectures that looked into different aspects of the project. Given our initial unfamiliarity with many of the technologies outlined in the project's scope, we proceeded to adopt those specified in the project guidelines.

The project's brief did offer the option to incorporate one additional technology, which, while seemingly providing a degree of flexibility, also imposed certain constraints on our technological explorations. For example, questions such as the feasibility of implementing the project in Python, the use of a Python persistence model, or Python's capability to interface effectively with a database, remained unexplored. These potential avenues of inquiry could have provided a valabue alternative allowong a more complex design and quicker development approach.

However, the intensive nature of the assignments precluded a thorough investigation into these alternative technological possibilities. As a result, the project proceeded within the confines of the pre-specified technology stack, leaving some questions about potential alternative implementations and technologies unanswered.

References

1. Angular Architecture Guide: `https://angular.io/guide/architecture`
2. Spring Official Documentation `https://docs.spring.io/spring-framework/reference/index.html`
3. JSON Web Tokens `https://datatracker.ietf.org/doc/html/rfc7519/`
4. Hibernate ORM `https://hibernate.org/orm/`
5. Mosquitto MQTT `https://mosquitto.org`
6. Arduino `https://www.arduino.cc/reference/en/`
7. ESP8266 for IoT `https://www.nabto.com/esp8266-for-iot-complete-guide/`
8. Rapid Software Testing `https://rapid-software-testing.com/`