DAT 250
Advanced Software Technologies

November 26, 2020

Feed App Project

Western Norway
University of
Applied Sciences

Contents

Abstract

This project was undertaken to use and explore new and useful technologies, such as: Spring Boot, JPA, REST API, NoSQL, Messaging system.

Different software technologies are used to implement the different technologies. One of them is Spring Boot framework. Spring Boot is very fun and easy to use, as it makes coding much easier, because it allows to focus on implementing the vision of the project instead of having to implement all functions from scratch. To learn and explore the different software technologies, a web application was implemented, that would utilize those technologies. A software for testing different implementations was Postman. Postman is a API client allowing for user defined requests against a web application, which make all the testing very easy and simple.

The resulting web application allows a user to create a poll, and other people can vote on it, either by making a account or being anonymous. There is also support for voting on a poll from a IoT voting device.

1 Introduction

1.1 Implementation

While there was a fair amount of freedom associated with creating this prototype application, there was also introduced some requirements that the project had to abide by. In order to meet these requirements, a lot of different methods and technologies were utilized. This includes implementing both relational and non-relational databases, messaging systems, REST API's, security aspects and more. This report will go into detail which technologies and methods were used, why they were used and how they were implemented in correlation with each other.

1.2 Technology stack

The technology stack chosen for the project contains of:

Firebase
RabbitMQ
Spring framework
Dweet.io
H2 Database

The main reasons behind this selection of technologies are that they all run great together, they perform well in the Spring framework. Finding material on these technologies are also a plus, as there are many great examples on how to implement each of them. Spring offers a lot to the application. Since it supports both non-relational and relational databases it is a great choice for developers who need to connect their applications to more than one database.

1.3 Results

The results of this project is an application that allows for users to interact with polls in a number of ways. Either it is creating a poll, deleting one, editing one or voting on one. All of this is done through different CRUD operations that interact with values stored in databases. Voting can also be done by external IoT devices.

1.4 Organization of report

The rest of this report is organized as follows:

Section 2 gives an overview of the key concepts and architecture of the chosen software technologies used in the project. New technologies that are not already introduced in the DAT250 course will have some running examples.
Section 3 gives an architectural overview of the application, with different belonging models such as an architecture diagram, domain model and models for use cases and application flow.
Section 4 describes in greater detail how the different software technologies are implemented, their part in the project and running examples with code snippets for visualization and better understanding of how the code works.
Section 5 explains the progress of the project, how the application has been tested, trial and errors and the results.

2 Software Technology Stack

This section of the report will cover each of the chosen software technologies that are implemented in the final project application. It will introduce some of key concepts of the technologies, with some running examples.

2.1 Firebase

Firebase is a Backend-as-Service (BaaS) platform created by Google that provides helpful tools. Examples of these are:

- **Firebase Database:** Is a NoSQL cloud database used for storing and syncing data. The data is synced in realtime and remains available even when the application goes offline. The data is stored as JSON. [1]

- **Firebase Cloud Firestore:** Cloud database with good scalability for mobile, web and server development. Cloud Firestore offers flexibility, expressive querying, realtime updates and more. [2]

- **Firebase Cloud Messaging:** Used to send notification/data messages. [3]

- **Firebase Remote Config:** Used to quickly update/change behavior and appearance of your application without having to download an update. [4]

As for the poll application that was made in the project it only uses the Cloud Firestore out of these four. In the Cloud database the data is stored in documents with underlying collections where all values are stored in a structural way. In Figure 1 the data is stored in such a way, with IDs as the document name and the values are stored in collections with fields for each value. The database supports many different data types, from integers and strings to more complex data in the form of objects.
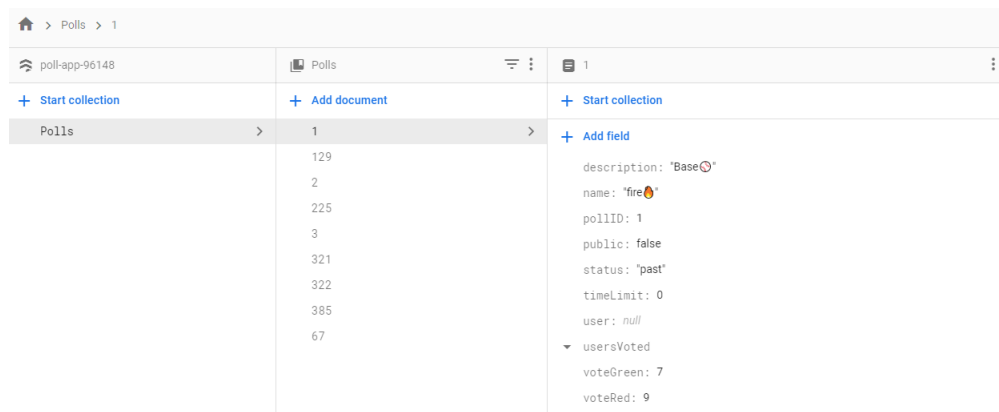

Figure 1: Firebase Cloud Firestore.

2.2 RabbitMQ

RabbitMQ is an open source message broker that implements Advanced Message Queuing Protocol (AMQP). RabbitMQ is an intermediary for messaging. "It gives your applications a common platform

to send and receive messages, and your messages a safe place to live until received." [5] How this works is that a producer publishes messages to the "exchange" with specific type of exchange, the exchange then delivers the messages to different queues depending on exchange types. The recipients will then retrieve messages which they have subscribed to.

2.3 Spring framework

2.3.1 Thymeleaf

Thymeleaf is a template engine written in Java for HTML5/XHTML/XML. It can be used in both web and non-web environments. Thymeleaf implements the concept of Natural Templates, that are described as: "template files that can be directly opened in browsers and that still display correctly as web pages" [6]. The use of Thymeleaf can act as a replacement for using Java Server Pages [7].

2.3.2 Spring Boot

Spring Boot allows for creating Spring based applications in an "easy" way that just runs, with minimal effort. Spring Boot does a lot of the configuration for the user by being a framework built on top of Spring framework which enables the user to quickly "bootstrap" a Spring application from scratch. The Spring initializer makes for an effortless start to any project by simply picking out dependencies according to the type of project.

2.3.3 Spring JPA

Spring JPA is a framework provided by Spring, to make it easier for developers to implement data access layer. By letting the developer only have to write the entities, repository interfaces, and perhaps making some extra custom repository searching methods. While the developer is focusing on programming the vision of the project, Spring JPA will implement all necessary methods for accessing all the data from repositories for the developer.

2.3.4 Spring MVC

Spring MVC is the framework used to build the web application. It provides the Model-View-Controller architecture. This simply means that it separates the application into a model, view, and controller component. The task of these components is to handle different sides of the development in an application. When it comes to creating scalable projects, Model-View-Controller is one of the most commonly used web development frameworks.

2.3.5 Spring Security

Spring Security is a powerful and highly customizable authentication and access-control framework. It is the the most used and a practical standard for securing Spring-based applications, thus also our application. Spring Security is a framework that focuses on providing both authentication and authorization to Java applications.[8] Security is very important and should not be taken for granted. In this aspect, Sprint Security makes it very easy to customize and tweak the settings to meet the developers preferences. Features available to customize is a long list, but protection against attacks and authentication with access-control (AC), are the few features that are featured in our FeedApp application.
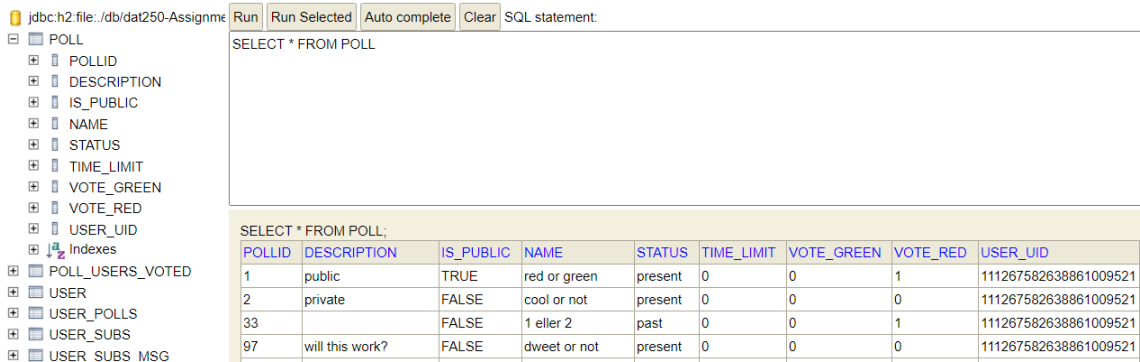
2.4 Dweet.io

Dweet.io is a type of messaging system for machines. It is also referred to as a "Twitter for social machines" [9]. A machine can either publish or subscribe to data, and all data published to dweet.io can be retrieved by simply calling an URL. All messages published to dweet.io will be associated with an unique name. The data is structured in JSON format.

2.5 H2 Database

H2 Database is an open-source SQL database written in Java. The database can be embedded in Java applications, making it a suitable choice for development and testing. The database engine is also exceptionally fast. [10]

The H2 console allows for a full view of the database. It allows for SQL commands for inspecting/or changing values in the database, as is seen in figure 2. After starting an application with H2 embedded in it, the console can be reached in a browser by calling the path URL that is configured. If the application is running on port 8080, and the path URL is set to /h2-console,
simply call "localhost:8080/h2-console". After inputting the correct username and password, one is free to inspect and manage values the database.

POLLID	DESCRIPTION	IS_PUBLIC	NAME	STATUS	TIME_LIMIT	VOTE_GREEN	VOTE_RED	USER_UID
1	public	TRUE	red or green	present	0	0	1	111267582638861009521
2	private	FALSE	cool or not	present	0	0	0	111267582638861009521
33		FALSE	1 eller 2	past	0	0	1	111267582638861009521
97	will this work?	FALSE	dweet or not	present	0	0	0	111267582638861009521

Figure 2: A SQL command is run in the H2 console.

3 Demonstrator Prototype

In this project, several software technologies were used that have already been mentioned and described in section above. For all technologies to communicate with each other, a REST API is used. User can view the application by sending HTTP request to server from web browser and server replying with a handled response. A user has different "interaction" he/she can do with the web application, those are described in section 3.3.

3.1 Architecture

Figure 3 represents the architecture diagram for the FeedApp. A Client (user) uses the web browser to send requests and get responses from the server (application). To use the application the user needs to be authenticated (logged in), at least for the most use cases of the application. Requests gets handled by the REST API on the server which processes the request and sends a response accordingly. REST API communicates with the database to save/get persisted data.

There is also support for IoT devices to communicate with the server, and it sends the requests straight to the REST API which doesn't have to go through the extra steps with authentication and Model view in web browser.

There is also a messaging system that gets poll information from REST API and sends them to a NoSQL database. Where the polls can be used in other use cases, than the web application intends to.
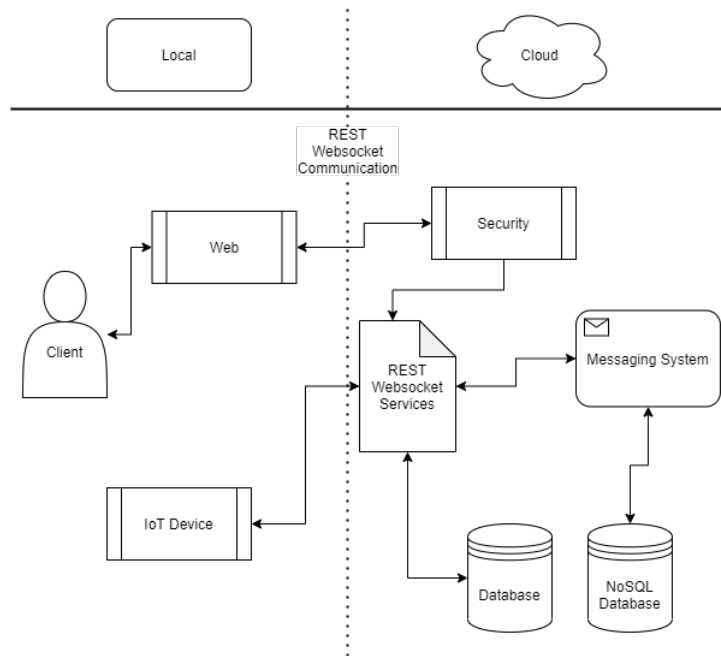

Figure 3: Architecture diagram.

3.2 Application flow

Application flow can be seen in figure 4, which has the thought out interactions on the different web pages and what will be possible for the user to do. A users will be able to vote on a public poll without the need to log in, or have an account at all, or if the poll is private, then the user will be able to create an account to vote, and also create his/her own polls. There will also be a possibility to look-up polls by their names.

All actions will have a success and possibly a failure, these will be handled with proper requests and responses for the different actions.
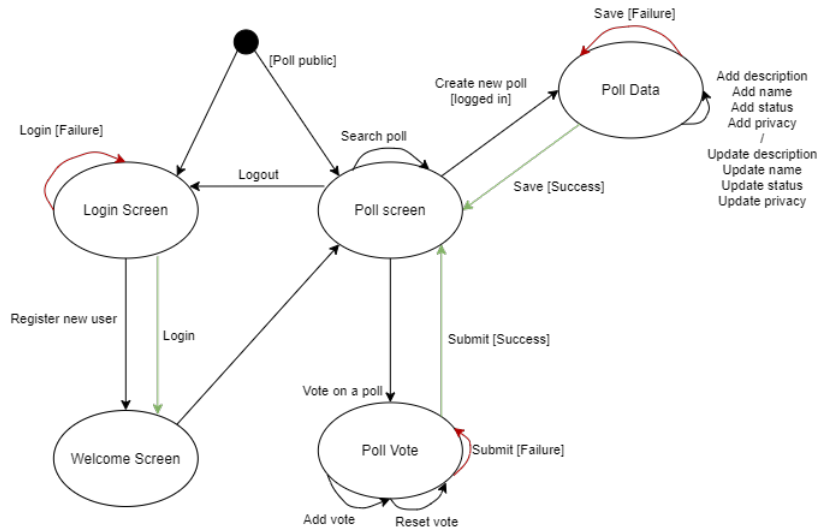
Figure 4: Application flow.

3.3 Use case

In figure 5, user enters the Application via web browser and is able to login or register new account. On main page after logging in; user can create a new poll, delete or edit an already existing poll (if user is owner), search / look-up a poll and vote on it. Since the application also supports IoT devices, the use case of an IoT device is to enter a poll with pollID and vote on it.

- **Vote on a Poll:** Everyone can vote on a public poll, using a web form, but to vote on a private poll the user must be logged in.

- **Register:** Everyone can register a new account to use the web application services, using OAuth.

- **Create Poll:** A logged in user can create a new poll using a web form.

- **Subscribe to Poll:** A logged in user can subscribe to a poll after entering the voting screen with PollID.

- **Show own polls:** A logged in user can view his/her own created polls, including the "live" voting results.

- **Show subscription messages:** A logged in user can view messages from subscriptions, on subscribed polls.

- **Delete Poll:** A logged in user can delete own polls from show own polls screen.

- **Edit Poll:** A logged in user can edit his/her own polls, using another web form from show own polls screen.

- **Look-up a Poll:** Everyone can look-up a public poll by the name, but only logged in users can look-up private polls.

7

- **IoT Device:** An IoT device can vote on all polls with known PollID. IoT can also see Poll data, with-it the results as well.
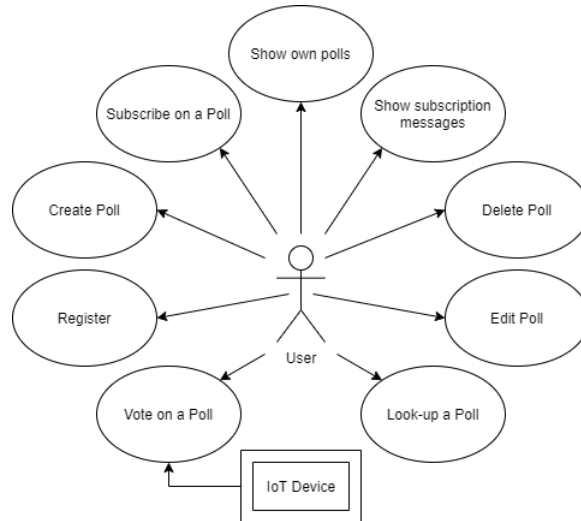

Figure 5: Use cases.

3.4 Domain model

The domain model in figure 6 describes the planned persisted entities. A user with a uID primary key which can have multiple Polls, with a PollID as primary key and user foreign key. The database will also save which user has voted on which poll, so that a logged in user cannot vote multiple times on the same poll. This data is persisted into "User has Voted" table.
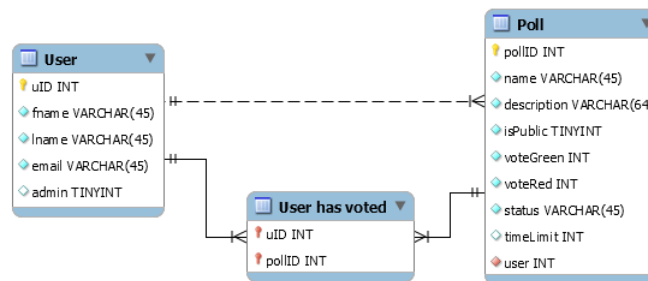

Figure 6: Domain model.

The user also has Lists of Polls he/she subscribed to, to receive updates, and subscribe messages list, with messages of the subscriptions.
In the next section we will see how the prototype was implemented, and the different technologies used.

4 Prototype Implementation

In this section we'll describe how the different software technologies were implemented and how they were used in our project, by briefly mentioning what they do and how we thought by implementing them. Code snippets are also showed to explain and visualize how the code looks in our real-time project.

4.1 JPA

Prototype implementation had few major steps. First and most crucial one, was to create JPA entities and let them communicate with each other, from given relations between them.

Two entities were created, **POLL** and **USER** with appropriate relations stored in a H2 relational database. Relations between those entities are One-To-Many, since one user can create many polls, but only one user can be the creator/owner of a specific poll. Furthermore, there is a possibility to vote on a specific poll, thus there is a need for storing data about that too. This is handled with a "Many-To-Many" relation between Poll and User entities, creating a new table with two attributes **User.uID** and **Poll.pollID** storing "who voted on what".

As shown in the example below, JPA makes it very easy to create new entities. Just by adding annotation @*Entity*, the Java class is then declared to behave as an entity when program runs. In the same way we can use @*Id, @ManyToOne, @ManyToMany, ...* annotations to configure the desired relations and identities for different entity attributes.

```
@Entity
public class Poll {
   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   private int pollID;
   private String name;
   ...

   @ManyToOne
   private User user;

   @ManyToMany(cascade = CascadeType.PERSIST)
   private List<User> usersVoted = new ArrayList<>();
```

JPA doesn't give basic; create, read, update and delete (CRUD) operations, when new entities are created. Therefore, it was necessary to create own Data Access Object (DAO), for every entity created. This is very cumbersome and time consuming. Benefits of using Spring Framework, is that CRUD operation can very simply be added for entities by extending a Spring Framework Class (CrudRepository or PagingAndSortingRepository which extends CrudRepository), we chose PagingAndSortingRepository, as it has the same functionalities as the other class, but with some extra functions which make it simpler to "page and sort" the results obtained.

In the code snippet below, we can see our implementation of User repository extending the spring framework class. With an extra functionality, *findByFname*, which is a read operation to search for a user by his/her first name.

```
import project.dat250.entity.User;
public interface IUserRepository extends PagingAndSortingRepository<User, String> {
   User findByFname(@Param("fname") String fname);
```

```
4  }
```

4.2 REST

With the entities set up and ready to be used, a user needs some way to interact with this data. To accomplish this, a REST API needs to be implemented. To implement a business logic side, Spring Framework was used again. With the benefits of using Java annotations @*Controller*, the Java class is then declared to behave as a RestController when program runs. Creating new methods within this Class and annotating them with desired mapping annotation, all CRUD operations can be accessed from HTTP requests.

```
1   import project.dat250.entity.Poll;
2   import project.dat250.entity.User;
3   import project.dat250.repository.IPollRepository;
4   import project.dat250.repository.IUserRepository;
5
6   @Controller
7   public class PollRestController {
8      @Autowired
9      private IPollRepository pollRepository;
10     @Autowired
11     private IUserRepository userRepository;
12
13     @PostMapping("/pollAdd")
14     public String pollAdd(@RequestParam String name, @RequestParam String
           description, @RequestParam String isPublic, @RequestParam String status,
           Model model) {
15        ...
16     }
17     @GetMapping("/polls")
18     public String searchPolls(@RequestParam String name, Model model,
           OAuth2AuthenticationToken authentication) {
19        ...
20     }
21  }
```

As we can see in the code snippet above, it is very simple to use Spring frameworks annotation to also make a REST controller. The @*Controller* annotation, makes this class a controller when the program runs. To use different repositories to use in this controller class, we need some way to access them, this is accomplished with the annotation @*Autowired*, which wires the repository to the controller when the program launches.

In a controller class we can use different request mappings, such as @*GetMapping* and @*PostMapping*, which we used in our code. In the different request mappings, we need to define the path at which to access the different methods. In the code snippet there are two examples of such mappings, one POST mapping, which at the path */pollAdd*, needs parameters defined with @*RequestParam* to execute the *pollAdd()* method. And a second, GET mapping to search polls at the path */polls* with defined parameters.

While implementing REST API to our prototype; post, get, put, delete mapping was used to make user interact with prototype's business logic. Since this requires the user to send HTTP-request with

different mappings for different operations. Later, this was modified and downgraded to only GET and POST mappings, since those are the only two HTTP requests a HTML form supports.

4.2.1 IoT support

Final project has support for a voting IoT device and display IoT device to connect to a poll with HTTP request. To make this a possibility, more CRUD operation mappings were added to the program. When correct GET and POST mappings were implemented for an IoT device to be able to connect to a poll, a virtual device was created in Java.

A small and easy Java program was created with simple HTTP requests for the IoT to connect to the prototype. In order to create the program, just one Java class was needed and few helping methods to check for any potential errors when "user" input is used to navigate in the program and vote on a poll or look-up its results.

4.3 Security

Since a user must register and login to use most parts of the web page, there needed to be some security implemented. For this we used Spring frameworks Security dependency, which allowed us to simply define which pages could, and which couldn't be accessed without logging in, this can be seen in the code snippet below.

For a "simpler" and more secure login process, we decided to use Google's OAuth 2. By doing this, we don't need to store sensitive user information, and user doesn't need to create a new password for him/her to remember. As the best security is the security you don't need to implement yourself, which can cause unintended errors or follow with security holes.

```
1  @Override
2  protected void configure(final HttpSecurity http) throws Exception {
3     http.csrf().disable().authorizeRequests()
4          //permit all to access some sites
5          .antMatchers("/oauth_login", "/h2-console/*", "/", "/poll", "/pollVote",
                "/polls", "/polls/*/setVotes", "/polls/*/IoT/*", "/styles.css",
                "/fonts/*").permitAll()
6          //Needs authentication to access other sites
7          .anyRequest().authenticated()
8          //Login method, login screen, successful login screen
9          .and().oauth2Login().loginPage("/oauth_login").defaultSuccessUrl("/userAdd",true)
10         //Logout method, successful logout screen
11         .and().logout().deleteCookies("JSESSIONID").logoutSuccessUrl("/oauth_login");
12 }
```

Disclaimer! *All documentations and tutorials we looked at about Spring Framework were taken from [11]*

4.4 Firebase

The project required storing of results in a noSQL database. For this task we chose to use Firebase Cloud Firestore. The way this was implemented was by creating a service component by adding the @Service annotation from Spring Boot. This class handles initializing of the Firebase database, so Spring @Postconstruct is also added to make sure the method is initialized properly before the class is put into service.

For populating of the database with results from the poll we use the @PostMapping and @PutMapping from the controller. When a poll is created, it momentarily gets stored in the database. Same goes for the updates, when for example someone is voting on a poll, it gets updated in the database as the vote is cast. In listing 1 you can see a code snippet for how the poll gets stored in the database, with collection name Polls. It then stores the poll created with document name set as the Polls ID.

Listing 1: Adding poll to Firebase

```
CollectionReference pollCR = fb.getFirebase().collection("Polls");
pollCR.document(String.valueOf((newPoll.getPollID()))).set(fbPoll);
DocumentReference dr =
    fb.getFirebase().collection("Polls").document(String.valueOf(newPoll.getPollID()));
dr.update("pollID",newPoll.getPollID());
```

4.5   Messaging System

4.5.1   RabbitMQ

In the web application, a user can subscribe to a poll and receive its results when the poll closes. This calls for a messaging system implantation. For this we used RabbitMQ with a receiver and a sender class, again using Java annotations to declare a method to be a RabbitListener. We chose to use a topic-based messaging queue, as it made it easy to use poll status (present, past, future) as the topic keys.

But before a user can receive messages from a subscription, the user must subscribe to a specific poll, which is then saved in a subscription list that the user has saved in its entity. From these lists, rabbit listener finds correct receivers and sends the requested message to them.

4.5.2   Dweet.io

As for the implementation of dweet.io, it was desired that the events of starting and closing a poll should be "dweeted". This simply means that whenever a poll is created, information about the poll name and description is published to dweet.io. A poll that is closed will additionally send information about the result of the poll. When implementing code that will send out a dweet, it is important that a unique name/ID is chosen. If a machine or user wants to get information about this dweet, it will use this unique name to get it.

The code snippet in listing 2 illustrates how the application would extract the poll name and description and post it to dweet.io, in the events of creating a new poll. After setting a unique name, it takes the name and description values and puts them into a JSON element. It is lastly published to dweet.io by using the publish function provided by the dweet.io library for Java, with the unique name and JSON element inputted as parameters. If one wished to get this dweet, one could simply type a request URL with the unique name in a browser. In this case, it would be: https://dweet.io:443/get/latest/dweet/for/unique_poll_dweet.

Listing 2: Sending a dweet of a newly created poll with name and description

```
@PostMapping("/pollAdd")
public String pollAdd(@RequestParam String name, @RequestParam String description,
    @RequestParam String isPublic,
        @RequestParam String status, Model model, OAuth2AuthenticationToken
            authentication) {
```

```
4
5   String thingName = "unique_poll_dweet";
6   JsonObject json = new JsonObject();
7   json.addProperty("Poll name: " + name, "Poll description: " + description);
8   DweetIO.publish(thingName, json);
9
10  }
```

4.6   Front End

User interacts with the program via web browser and gets information from database by communicating with business logic via Spring MVC, using Thymeleaf. Thymeleaf is an integrated method to use in HTML code in order to access data from models, generated based on the request sent by user. HTML code was used to create different use case screens and according to what requests can be made, the appropriate thymeleaf code was inserted into HTML code. For the pages to look "better" and be user friendly with what can be done on a web page, CSS was used to decorate the pages and how they look/show up for the client.

5   Test-bed Environment and Experiments

This section will explain and show how this project was tested while implemented as shown in section above. There was lots of trial and error scenarios and all testing was performed before fully implementing. By manually trying if the technology implementation was working as intended to. Therefore the main method of testing in this project was, manually testing with trial and error without any premade testing software's or methods.

5.1   Data Access Object testing

As mentioned in section 4, the first thing implemented was USER and POLL entities. So testing their implementation and relations between them, was the beginning of any testing. To perform tests on entities, entity DAO's were used with some simple CRUD operations; create a user and poll, edit created user and poll, delete created poll and user, and read information of created poll and user.

5.2   REST API testing

Further implementation and development of this project consisted of going from DAO classes to Spring JPA implementation and adding some control over these entities from "outside" of the code, a REST API. This meant a different approach to testing of this implementation. Now some simple code wasn't sufficient enough, thus a software that could send HTTP requests to the project application, which then acted accordingly to the requests was required. To accomplish this testing method, Postman [12] was used.

5.2.1   Postman

Postman is an Open-Source software that can send user defined requests, consisting of JSON body or include path variables to customize requests for own preference. These requests can be of any type (GET, POST, PUT, DELETE, ...). After sending a request from Postman, it will receive the response

according to sent requested. Response will also include HTTP status code for success or error, this was very helpful for testing and to see if implementation of REST API was correctly assembled. This made all testing very simple and the error backtracking was easier. Postman was the main method of testing all client (user) interactions with the program, until security and model view was implemented.
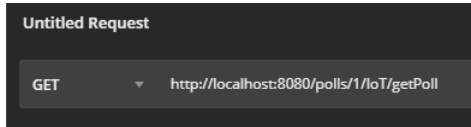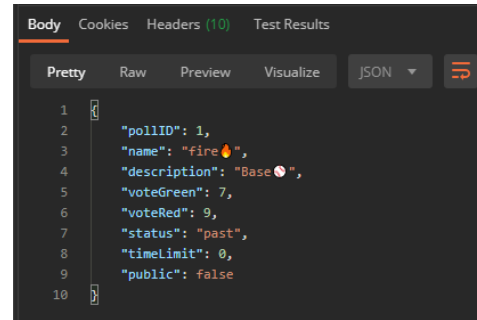

Figure 7: Postman request.

Figure 8: Postman response.

5.2.2 HTML/CSS

After implementing Security, an authorization token was required to use most services. So the only way to test, was to manually go on the web page and see if the service implemented was working properly and as intended or if something could be done differently. Authentication security was therefore turned on after client interaction with FeedApp was behaving as mentioned in prototype design application flow section 3.2.

This method of testing was also used when the CSS was added to the project web application and with CSS, the testing was made more "user friendly". Even though testing was performed on Postman and via the features distributed from it. The whole project after fully developed, does not use or have features available to receive request based on JSON body, such that Postman requests can be performed. All HTML and CSS implementation was based on own education and own preferences, to how things would optimally work and look, but also easy to understand for the client when using the service. When errors occurred during CSS development, some basic tutorials were looked up on w3schools [13] and errors resolved.

5.2.3 IoT testing

IoT used for this project was a simple Java program as mentioned in section 4.2.1. So testing if the implementation was working correctly and error-less, was a tedious process to manually check all possible user input errors. Many tests required many restarts of the Java program, but also many look-ups on the web page. Web page user interaction was already implemented, so the testing was possible to perform without sending request and getting responses through Postman and therefore possible to save time and amount of required clicks needed.

6 Conclusions

Concludes on the project, including the technology, its maturity, learning curve, and quality of the documentation.

The references used throughput the report should constitute a well chosen set of references, suitable for someone interesting in learning about the technology.

References

[1] "Firebase database." [Online]. Available: https://firebase.google.com/docs/database

[2] "Firebase cloud firestore." [Online]. Available: https://firebase.google.com/docs/firestore

[3] "Firebase cloud messaging." [Online]. Available: https://firebase.google.com/docs/cloud-messaging

[4] "Firebase remote config." [Online]. Available: https://firebase.google.com/docs/remote-config

[5] RabbitMQ. [Online]. Available: https://www.rabbitmq.com/features.html

[6] Thymeleaf. [Online]. Available: https://www.thymeleaf.org

[7] Tutorialspoint, "Java server pages." [Online]. Available: https://www.tutorialspoint.com/jsp/index.htm

[8] Spring, "Spring security." [Online]. Available: https://spring.io/projects/spring-security

[9] Dweet.io. [Online]. Available: http://www.dweet.io

[10] "H2 database." [Online]. Available: https://www.h2database.com/html/main.html

[11] R. Johnson, J. Hoeller, K. Donald, C. Sampaleanu, R. Harrop, T. Risberg, A. Arendsen, D. Davison, D. Kopylenko, M. Pollack, T. Templier, E. Vervaet, P. Tung, B. Hale, A. Colyer, J. Lewis, C. Leau, M. Fisher, S. Brannen, R. Laddad, A. Poutsma, C. Beams, T. Abedrabbo, A. Clement, D. Syer, O. Gierke, R. Stoyanchev, P. Webb, R. Winch, B. Clozel, S. Nicoll, S. Deleuze, J. Bryant, and M. Paluch, "Springframework docs," Nov 2020. [Online]. Available: https://docs.spring.io/spring-framework/docs/5.2.x/spring-framework-reference/index.html

[12] Postman, "The collaboration platform for api development." [Online]. Available: https://www.postman.com/

[13] W3schools, "Css tutorial." [Online]. Available: https://www.w3schools.com/css/