DAT 250
Advanced Software Technologies

Wojciech Pasiak
Jaroslaw Pasiak
Cristoffer Brandhaug
Erlend Meling Torsvik

November 24, 2020

Feed App Project

Western Norway
University of
Applied Sciences

Contents

Abstract

10-15 lines with the software technology and the highlights from the project that has been undertaken.

1 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 1.3.

1.1 Architecture

Figure 1 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.

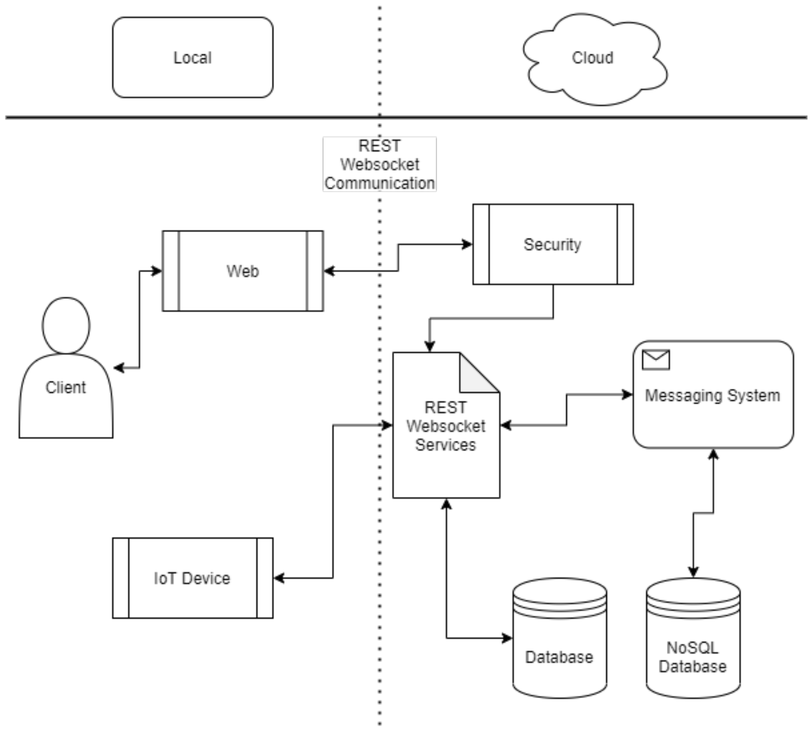

Figure 1: Architecture diagram.

1.2 Application flow

Application flow can be seen in Figure 2, 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.

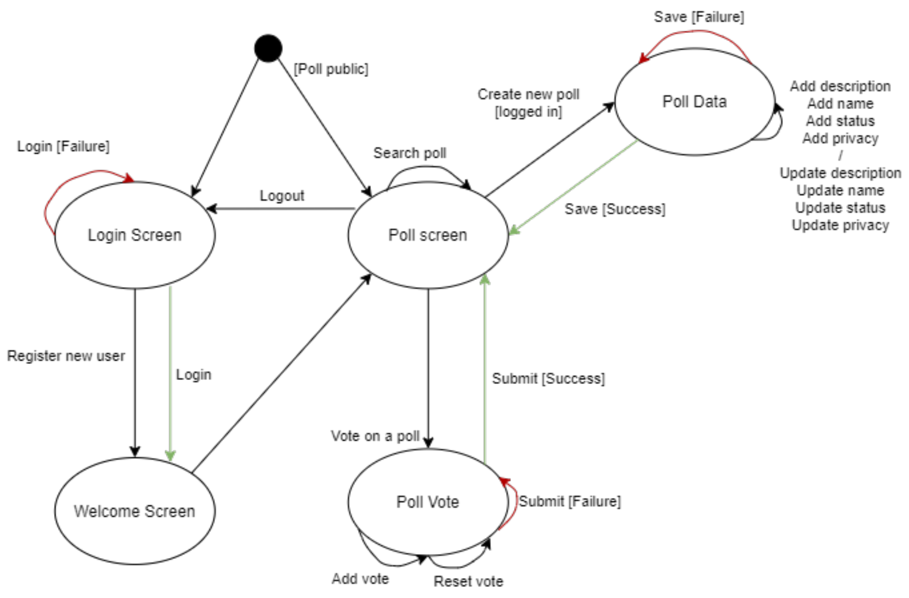

Figure 2: Application flow.

1.3 Use case

In Figure 3, 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.
- **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.

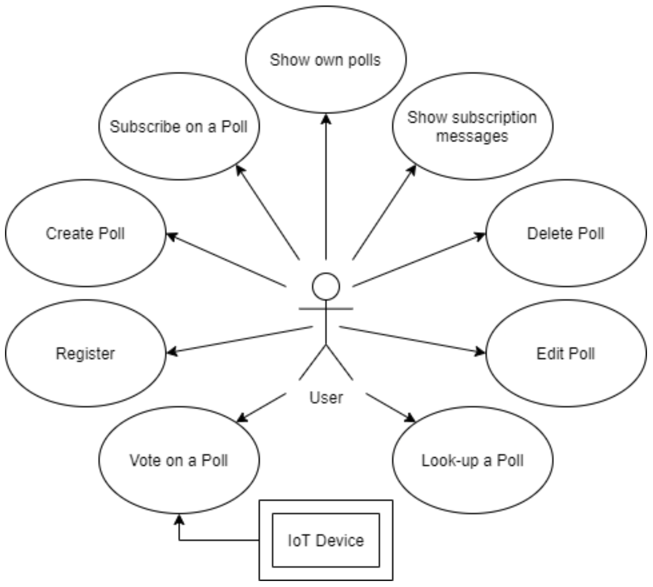

Figure 3: Use cases.

1.4 Domain model

The domain model in Figure 4 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.

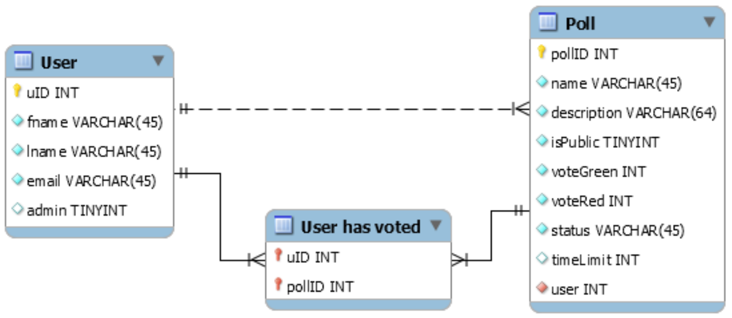

Figure 4: Domain model.

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.

2 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.

2.1 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 Derby 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.

```java
@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.

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

2.2 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.

```java
import project.dat250.entity.Poll;
import project.dat250.entity.User;
import project.dat250.repository.IPollRepository;
import project.dat250.repository.IUserRepository;

@Controller
public class PollRestController {
   @Autowired
   private IPollRepository pollRepository;
   @Autowired
   private IUserRepository userRepository;

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

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.

2.3 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.

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

Disclaimer! *All documentations and tutorials we looked at about Spring Framework were taken from [1]*

2.4 Messaging system

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[2] 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.

2.5 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 [3]. 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.

References

[1] 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

[2] RabbitMQ, “Rabbitmq tutorials.” [Online]. Available: https://www.rabbitmq.com/getstarted.html

[3] R. Borowiec. [Online]. Available: https://www.thymeleaf.org/doc/articles/springmvcaccessdata.html