DB Exam Project: The Social Network

Martin Høigaard Cupello - cph-mr221@cphbusiness.dk
Simon Schønberg Bojesen - cph-sb339@cphbusiness.dk
Kenneth Leo Hansen - cph-kh415@cphbusiness.dk
Frederik Blem - cph-fb114@cphbusiness.dk

May 2021

Contents

1 Introduction

The Social Network is a Databases for Developers exam project that simulates a typical social network with user accounts, the ability to create and read posts with or without tags referencing other users, a page for changing profile images with an option to select a previously used image and a live chat. This subject was chosen, because the group enjoyed working with the database types that the group agreed could handle this type of project.

In this paper, the system architecture along with the functional and non-functional requirements will be described. Additionally, the paper will contain an explanation on the installation of the project through Docker composition as well as a reflection upon The Social Network.

2 Installation

Before beginning the installation process make sure to have docker installed.
To install the program, first clone the repository from the attached GitHub or extract the zip folder.
After cloning, open a terminal in the root project folder and run this command “docker-compose up”. Then wait for docker to install all the required components.
When docker is done installing, run the main called TheSocialNetworkApplication.java. It is now possible to access the program by typing in

```
localhost:8080
```

in a browser.
There are 4 premade users: ”martin”, ”kenneth”, ”simon”, ”frederik”, all with the pass ”1234”. These show some example posts and chat, and follow-user functionality.
It is also possible to make your own user.

3 Planning and Development

3.1 Choice of Language

The team chose to write in java, since the team members have the most experience in this language. Also available are the convenient features of maven and spring boot. Previous experience with spring web security, which is used for the user login, were also a factor.

3.2 Development Process

The group opted primarily for a pair programming approach to this project, even though it meant slower progress. Since this was an exam task, it was decided that it was better for everyone to have a general knowledge of the different classes, files and queries. A few parts were done individually, like the chat system, loading of posts and images, but overall everyone was involved. We decided on a MVC setup using Spring controllers and Thymeleaf, with only a few JPA repositories, because it allowed us to write the queries directly.
The team used GitHub for versioning control, to ensure everyone had up to date code.
It was decided to host the application in Docker containers, however there were difficulties with locating the connection between the Neo4j database and the

main program when everything was containerized. It was therefore decided to run the main program locally, while keeping the databases containerized. Also difficulties with storing files locally in a docker container affected this decision.

4 System Architecture

4.1 Functional Requirements

- User can register an account - Implemented
- User has to be able to login - Implemented
- User can chat with another user - Implemented
- User can see history of chat messages - Implemented
- User gets notification when receiving a message - Implemented
- User can follow another user - Implemented
- User can unfollow another user - Implemented
- User can make a post with tagged users - Implemented
- User can upload a profile picture - Implemented
- ~~User can see who follows him/her~~ - This was dropped due to time constraints
- User can change among his old profile pictures - Implemented

4.2 Non-Functional Requirements

- There is a limit of 128 active connections. - Implemented - Redis is set up to allow 128 clients.
- Password are always hashed so they are not immediately readable. - Implemented
- The application will work in Chrome and Firefox browser. - Tested with these two browsers
- The system will run in docker containers. - implemented for databases. Main program still runs locally.

4.3 System Architecture Explained

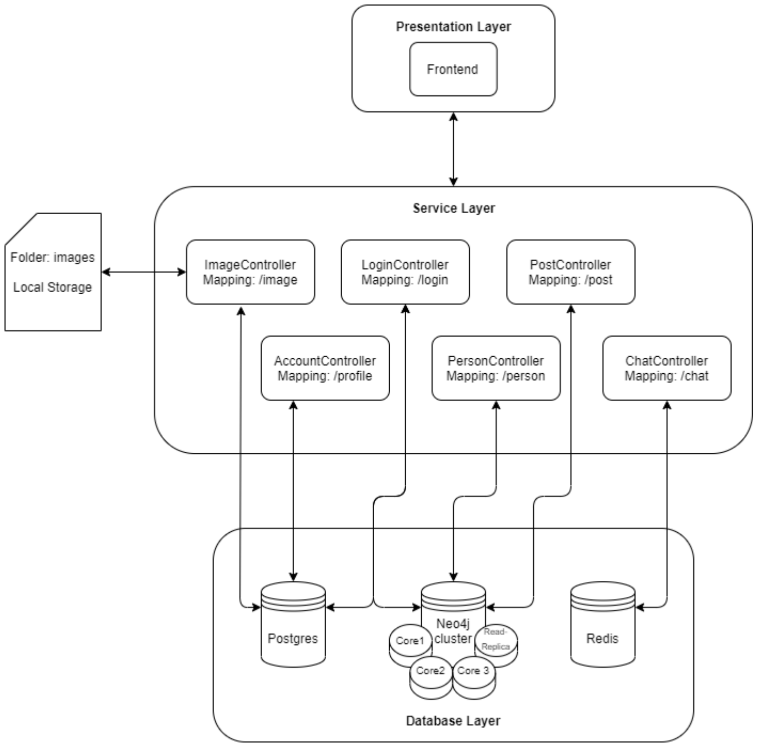


Figure 1: System Architecture

The three layers in the above figure split the system into categories depending on their designated jobs.

For the Presentation Layer the users consume The Social Network using an internet browser and access the available functions of the service layer on the mapped routes.

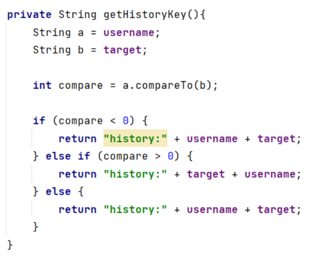
In the Service Layer the controllers are tasked with receiving the calls of the users, validating input & access and handling errors if appropriate, querying the databases through the repositories in their respective languages to retrieve data and serving it for the users in the front-end.
Each controller typically only communicates with one database, with two exceptions. The first being the LoginController that stores public access level user data in Neo4j and private user data in Postgres. The second exception is the ImageController that stores images in local storage space, that is made discoverable by file path URLs stored in Postgres.

In the Database Layer we have our three databases with our data. It is possible to infer the purpose of the database by looking at the controllers each is communicating with.
As mentioned earlier, the Postgres database contains sensitive user information and file path URLs for images.
Redis is responsible for containing chat messages and conversation histories of users with keys to sort between which chat history to access.
Neo4j handles public access level connected data, detailing relationships between user follows, posts and tags. In the case of Neo4j, The Social Network has a Neo4j database cluster, with 3 cores and 1 read replica. The main job of a core is to safeguard data, which they do by replicating all transactions. Another important function of the core is to balance out the workload. If one core fails, the other two will still remain functional and the database won't break down.The read-replica on the other hand are disposable, and have no impact on the database availability. The main job of the read replica is to scale out the workload of read only queries.

4.4 Databases

4.4.1 Redis - Chat Functionality

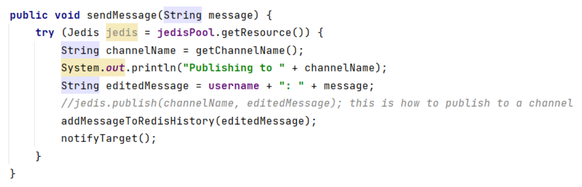
It was decided to use Redis for a chat database, as it was lightweight, easy to use and supported lists and sets. Originally thought to be a subscription (PubSub) setup as Redis supports this, however this was quickly moved away from, as the current setup is simpler to implement, and does not require juggling multiple blocking threads.
It works by posting to a list of strings in Redis with a key shared between the writer and the target. The key is the same both ways of the conversation, and is decided by running a compareTo on the writers username and the targets username:



```
private String getHistoryKey(){
    String a = username;
    String b = target;

    int compare = a.compareTo(b);

    if (compare < 0) {
        return "history:" + username + target;
    } else if (compare > 0) {
        return "history:" + target + username;
    } else {
        return "history:" + username + target;
    }
}
```

Figure 2: getHistory method

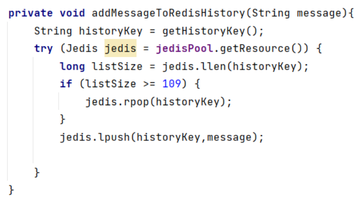
When a user sends a message from the frontend chat interface, it goes through the ChatController which creates an instance of ChatClient through the ChatClientHandler with a username, a target name, and a jedisPool object, which ensures that the calls to Redis are thread safe and can be used from multiple threads at the same time. It then calls sendMessage with the message as the parameter.



```
public void sendMessage(String message) {
    try (Jedis jedis = jedisPool.getResource()) {
        String channelName = getChannelName();
        System.out.println("Publishing to " + channelName);
        String editedMessage = username + ": " + message;
        //jedis.publish(channelName, editedMessage); this is how to publish to a channel
        addMessageToRedisHistory(editedMessage);
        notifyTarget();
    }
}
```

Figure 3: sendMessage method

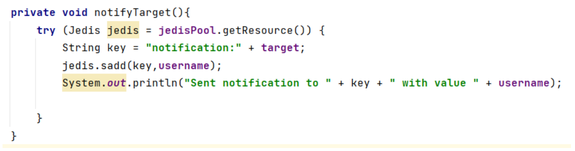
sendMessage calls two methods: First addMessageToRedisHistory is called with the message as parameter. It then left-pushes the message to the history key and if the list is of a certain size, it right-pops from the list. This ensures that the chat history never gets too big. (a much larger number could probably have been chosen as Redis supports list lengths of more than four billion).



```
private void addMessageToRedisHistory(String message){
    String historyKey = getHistoryKey();
    try (Jedis jedis = jedisPool.getResource()) {
        long listSize = jedis.llen(historyKey);
        if (listSize >= 109) {
            jedis.rpop(historyKey);
        }
        jedis.lpush(historyKey,message);

    }
}
```

Figure 4: addMessageToRedisHistory method

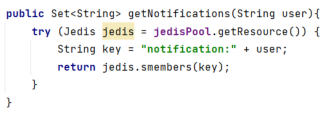
Secondly, sendMessage calls notifyTarget. This method is responsible for updating the notifications list in our frontend. This method adds to a set instead of a list as seen above in addMessageToRedisHistory.



```
private void notifyTarget(){
    try (Jedis jedis = jedisPool.getResource()) {
        String key = "notification:" + target;
        jedis.sadd(key,username);
        System.out.println("Sent notification to " + key + " with value " + username);

    }
}
```

Figure 5: notifyTarget method

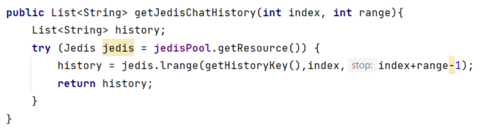
ChatClientHandler has a method for retrieving notifications for the current user, which are then shown in the frontend, allowing a user to see the history of an ongoing conversation.



```
public Set<String> getNotifications(String user){
    try (Jedis jedis = jedisPool.getResource()) {
        String key = "notification:" + user;
        return jedis.smembers(key);
    }
}
```

Figure 6: getNotifications method

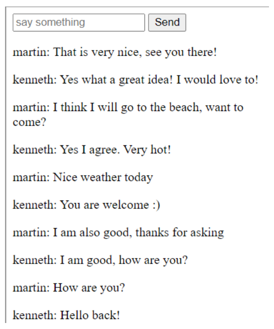
Lastly, the chat controller calls the clients getJedisChatHistory to load the chat history in order to show it to the user. Notice that it takes an input index and a range. This allows for pagination of the chat history!



```
public List<String> getJedisChatHistory(int index, int range){
    List<String> history;
    try (Jedis jedis = jedisPool.getResource()) {
        history = jedis.lrange(getHistoryKey(),index, stop: index+range-1);
        return history;
    }
}
```

Figure 7: getJedisChatHistory method

The end result looks like this in the frontend:



martin: That is very nice, see you there!

kenneth: Yes what a great idea! I would love to!

martin: I think I will go to the beach, want to come?

kenneth: Yes I agree. Very hot!

martin: Nice weather today

kenneth: You are welcome :)

martin: I am also good, thanks for asking

kenneth: I am good, how are you?

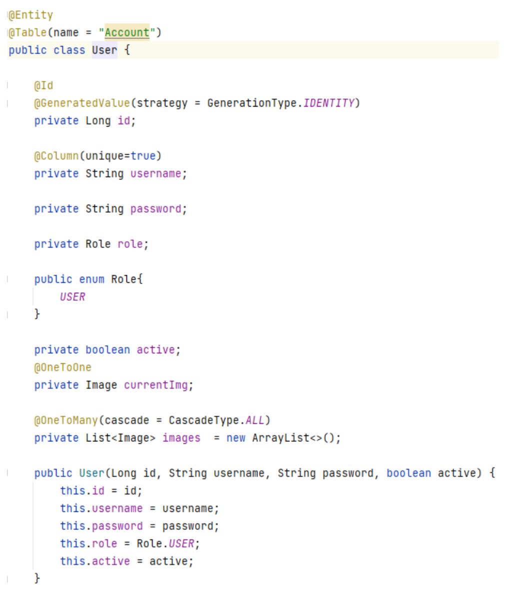
martin: How are you?

kenneth: Hello back!

Figure 8: a conversation between two users

4.4.2 PostgreSQL : Accounts and Filesystem

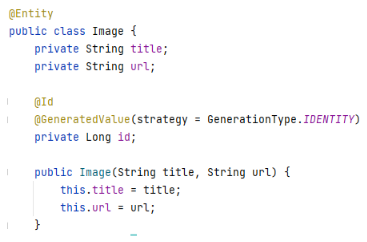
For the relational database the group chose PostgreSQL as all the team members already had a working environment for it with DBeaver installed. It was decided early on to keep only a few things in this database, login information and profile pictures saved through relative path URL to a local folder.
For this we have 2 entities: First, the User class entity for mapping between the table "Account" and Java. A user has an id generated with IDENTITY, a UNIQUE username, a password that gets encrypted with BCrypt, a boolean defining if the user is active or inactive, a OneToOne relation to an image which is the user's current profile picture, a OneToMany list of images which contains all undeleted old profile pictures and lastly a role which is supposed to define the



```java
@Entity
@Table(name = "Account")
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(unique=true)
    private String username;

    private String password;

    private Role role;

    public enum Role{
        USER
    }

    private boolean active;
    @OneToOne
    private Image currentImg;

    @OneToMany(cascade = CascadeType.ALL)
    private List<Image> images  = new ArrayList<>();

    public User(Long id, String username, String password, boolean active) {
        this.id = id;
        this.username = username;
        this.password = password;
        this.role = Role.USER;
        this.active = active;
    }
```

Figure 9: User class

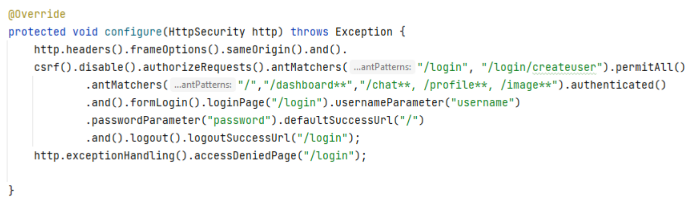
user's access level. In the current implementation all users have the same access level, but the role was added to show our thoughts of future implementation.
Secondly, the Image class entity contains a title, which is defined by the user when uploading a new picture, and the relative path to which the picture has been saved.



```java
@Entity
public class Image {
    private String title;
    private String url;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    public Image(String title, String url) {
        this.title = title;
        this.url = url;
    }
```

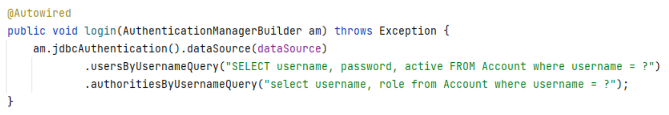
Figure 10: Image class method

For the implementation it was chosen to use Spring Repositories, and Spring Web Security to set up Login, Password Encryption and Access Layers. The class used for this is the SecurityConfiguration file. It contains 2 methods and a BCryptEncoder bean. The configure method defines which endpoints can be accessed with and without being logged in, what endpoint contains login and more things like which pages to redirect to on Login and Logout.
The login function is autowired through Spring and first it finds the user by the username and after that it finds the user's role by the username.



```java
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.headers().frameOptions().sameOrigin().and().
    csrf().disable().authorizeRequests().antMatchers( ...antPatterns: "/login", "/login/createuser").permitAll()
            .antMatchers( ...antPatterns: "/","/dashboard**","/chat**, /profile**, /image**").authenticated()
            .and().formLogin().loginPage("/login").usernameParameter("username")
            .passwordParameter("password").defaultSuccessUrl("/")
            .and().logout().logoutSuccessUrl("/login");
    http.exceptionHandling().accessDeniedPage("/login");

}
```

Figure 11: Security configuration



```java
@Autowired
public void login(AuthenticationManagerBuilder am) throws Exception {
    am.jdbcAuthentication().dataSource(dataSource)
            .usersByUsernameQuery("SELECT username, password, active FROM Account where username = ?")
            .authoritiesByUsernameQuery("select username, role from Account where username = ?");
}
```

Figure 12: Login method

To go over the endpoints in the 3 controllers: ImageController, LoginController and AccountController, that Postgres has a hand in, it will be quickly summarized here:

ImageController

- getActiveImage: gets the currently logged in users current profile picture.
- getImage: gets another account current image by username - note: if the user does not have any images, an anonymous image will be shown.

LoginController

- getLoginPage: returns the HTML page.
- createUser: Transactional endpoint as when creating an account in Postgres it must also create a user in Neo4j.

AccountController

- getProfileView: returns the profile CRUD page. Currently only image related.
- changeProfilePicture: update endpoint for changing current profile picture back to older picture.
- uploadImg: this one saves a new image as your profile picture with a unique filename to avoid overwrite conflicts.

4.4.3 Neo4j: Posts and User Relations

Neo4j was chosen for everything that revolved around connecting people. This would be Posts, Followers and further down the line, Likes. When these things were chosen for Neo4j it was because Neo4j is much better at querying connected data (relationships). This is perfect for the application of posting messages on a social network, as the post references the author and the author can tag people by writing for example "@username". In the Neo4j database this post would have a relationship:

```
(post)-[:POSTED_BY]->(user)
```

Also tagged users would have relationships to the posts they are tagged in:

```
(user)->[:TAGGED_IN]->(post).
```

"POSTED_BY" relationship is used to know who published a post, and "TAGGED_IN" is used to define tags as of right now. This relationship should be used to notify or show a person he/she has been tagged in a post, even if it's a post of someone they don't follow.
This example shows a person found by handleName, then all related posts can easily be queried out.

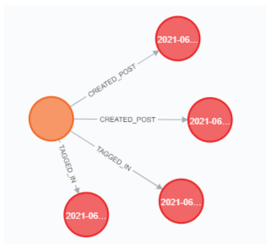


Figure 13: Person node with Post nodes

For followers this also makes sense as 2 relationships are created when a user follows another user:

`(user)-[:FOLLOWS]->(targetUser)`

and

`(targetUser)<-[:FOLLOWED_BY]-(user)`

These 2 relationships make it easier to query a newsfeed for the user as he/she would want to get updates from the persons they follow. In future revisions, the "FOLLOWED_BY" relationship could be used for counting the number of followers a person has.
In this example 2 people are queried out by their handleName to find out if they follow each other or not, which in this case they do.

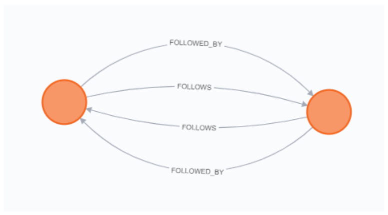
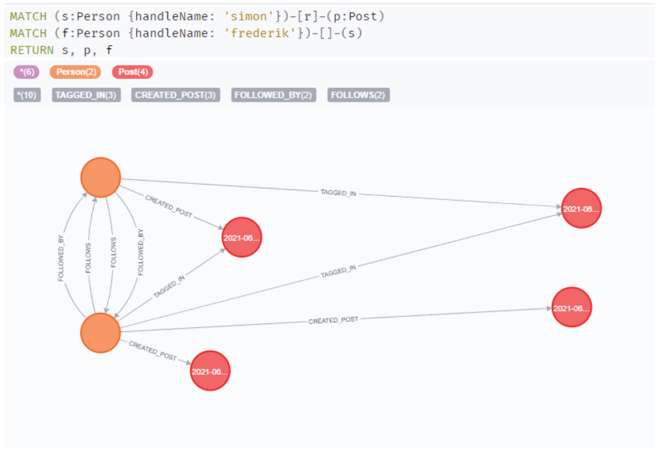


Figure 14: Two person nodes who follows each other

Lastly this example query can be shown which finds all related nodes and relationships between the same 2 people shown above.



```
MATCH (s:Person {handleName: 'simon'})-[r]-(p:Post)
MATCH (f:Person {handleName: 'frederik'})-[]-(s)
RETURN s, p, f
```

Figure 15: Full relation graph

Entities: For Neo4j there exists 2 entities, a Person and a Post. These were first made to visualize how a person and his relations should look like in Neo4j, and later used as the objects when creating Posts. The relations on a person or a post are made as lists that are initialized as empty lists when creating a Person, as it is not needed to have all of a Person's relations in all cases, and can therefore just leave them empty. On a Post everything is used, but the list of People that has liked the Post as this was not yet implemented.

Repositories and DTO's: For Cypher the group made custom repositories and a number of Data Transfer Objects. Even though the repositories could be easily made using Spring and Extending Neo4jRepository, it was decided not to do this as writing queries should be done in a DB exam. This means 2 repository classes were made: PersonRepository for getting followers functions, creating new persons, getting handle name, creating and deleting follower relations, and checking if database is empty to create dummy data.
The second repository: PostRepository has the responsibilities of creating posts and adding tagged people from the text as relations in Neo4j, finding tagged people, getting posts from people you follow so the program can show you your newsfeed and getting your created posts to show you your post history.

DTOs: 3 were created for different endpoint use cases.

- FollowsDTO which is used in the endpoints for creating and deleting follower relations.
- CreatePostDTO which is used solely for the createPost endpoint.
- LikedPostDTO which is intended for use in liking post implementation, it is not implemented yet.

Endpoints: To go over the endpoint it will be quickly summarized here

PersonController:

- createRelationShipPerson: makes a follower relation between user and target.
- deleteRelationShipPerson: deletes a follower relation between user and target.
- getPeople: gets a list of people the user is not yet following.
- getFollowing: gets a list of people the user is following.

PostController:

- getNewPostView: returns a view for creating a new post.
- getNewsFeed: finds relevant new posts and sends to newsfeed view for display.
- createPost: takes a text and creates a post and all TAGGED_IN and CREATED_BY relationships.
- likePost: unused. Supposed to be for liking posts.
- unlikePost: unused. Supposed to be for unliking posts.

5 Reflection

Development Process Our development process was a smooth experience, with only minor bumps on the road (like docker containers). Though probably not very practical in regular work life, we find that pair programming is well suited for a learning experience, albeit a bit slow progress.

Alternative Databases If we have had more programming experience with HBase, we could have replaced Neo4j with HBase. HBase is better suited for cases with big data, while Neo4j is better suited for data with many relationships. As social networking applications tend to become rather big with a lot of data, HBase might be better suited for this specific case.

Future Work Our resulting project is what could be considered a functional prototype. Given more time we would have like to have included the following:

- Hosting in cloud - We tried to put our program into a docker container, but our application had problems with routing and finding the Neo4j cluster inside the container. If we had been successful with this, and had more time we would have hosted the containerized application on droplet, so you could access the program anywhere.
- Able to like post - We would also have liked to fully implement the like and unlike feature. We created the Cypher queries and endpoints to do this, but we never implemented the front-end serving this feature.
- Group Chat - Another avenue of expansion would be the possibility to chat with multiple persons at the same time. This should still be possible to have been in Redis database.
- Server/admin messages in chat - Messages like message of the day and admin messages to users would also be an interesting feature.