MIDDLE EAST TECHNICAL UNIVERSITY
DEPARTMENT OF COMPUTER ENGINEERING

SUMMER PRACTICE REPORT
CENG 300

STUDENT NAME:	Mustafa Sezgin
ORGANIZATION NAME:	SRDC Software Research & Development and Consultancy Corp.
ADDRESS:	K1–16 Silikon Bina, ODTÜ Teknokent, Çankaya, Ankara, TURKEY 06800
START DATE:	26/07/2021
END DATE:	20/08/2021
TOTAL WORKING DAYS:	20 days

STUDENT SIGNATURE

[illegible]

ORGANIZATION APPROVAL

Contents

1 INTRODUCTION

I have done my summer practice at SRDC, which was conducted online due to the pandemic. In this 20-day time period, I worked as a full-stack web development intern and built web-based user management and messaging applications. I developed three projects using three different technologies, explained in detail in the Information About the Projects section.

Mustafa Yüksel, principal solutions architect and associate director at the company, was our supervisor, and Alper Teoman and Doğukan Çavdaroğlu, software engineers at the company, were assistant supervisors. During the internship, we held weekly meetings where we discussed the project of the week and our supervisor presented the concepts and the technologies related to it. We worked individually on the projects until the deadline and tested them together with our supervisors.

2 ORGANIZATION

Software Research & Development and Consultancy Corp. (SRDC) was founded in 2007 as a spin-off company of METU Software Research and Development Center, which dates back to the early 90s. Located in METU Teknokent, SRDC has a strong relationship with our university.

Prof. Dr. Asuman Dogac is the general manager of the company. The team consists of 15 people, the majority of whom are graduates of METU Computer Engineering Department with Ph.D. and M.Sc. degrees.

The company does research and development work and produces products and services in the areas of eHealth and eGovernment solutions, semantic web technologies, data analytics, interoperability and conformance testing, and more. It has been and continues to be a part of the projects funded by the European Union.

3 INFORMATION ABOUT THE PROJECTS

There are three projects. All are user management and messaging applications, which allow users to send and receive messages and admins to perform simple CRUD (Create, Read, Update, Delete) operations. However, the main difference among them is the technologies used and the architecture they are based on.

3.1 FIRST PROJECT

We started from scratch, so the first project is a command-line interface based on multi-threaded socket programming. After running the application, users log in with their usernames and passwords and use commands to perform operations. For example, `SENDMSG` is for sending a message, and `CREATEUSER` is for creating a new user, which requires authorization.

3.1.1 ANALYSIS PHASE

First, Java is used for this project since it is an object-oriented programming language, and it has built-in libraries that make things easier while dealing with multithreaded socket programming.

Client and server sockets transfer data between them via input and output streams, and the data is plain text without any regulation. Therefore, I needed to create my own protocol to give meaning to what is being transferred.

A relational database is required to store the user and message data, so PostgreSQL is a reasonable choice for this purpose. To execute queries on this database using Java, the JDBC driver for PostgreSQL is used.

We were not required to design a graphical user interface, so this project was planned to be a command-line interface application.

3.1.2 DESIGN PHASE

In this project, there are user and message models. The user model has a user ID and other fields for username, password, first name, etc. Since the project has two user roles, which are admin and regular, there is also the admin field that determines the role of the user. The message model has message ID, timestamp, usernames of the sender and the receiver, and the message content.

The main structure of the project is based on a server, a database, and clients. Users run the client and use commands to perform their actions. The client program asks the user for the necessary information line by line in order to create the proper string that will be sent to the server as a request. Then, the server takes the request, does the requested work communicating with the database, and sends a response back to the client.

To determine what a request and a response look like, a protocol was created. It requires the transferred data to be a string composed of substrings separated by the delimiter "`|`", where the first substring identifies the type of the string, and the others are for the necessary information.

For example, when the user with the username `sarah03` wants to send a message to her friend `eric.roy`, the request the client sends to the server would look like the following.

```
SENDMSG|sarah03|eric.roy|Hi Eric! How are you?
```

And the server's response would be

```
SUCCESSFUL|MSG_SENT|509785
```

where `509785` is the message ID created by the database, or

```
ERROR|NOT_FOUND|User not found.
```

if `eric.roy` does not exist.

3.1.3 IMPLEMENTATION PHASE

The server initially connects to the socket on the configured port and to the database and waits for a client connection. When the client connects to the same port, the server accepts it, opens a new thread to run the client handler, and starts waiting for another client. The client handler is the class where the actual job is done.

Both the client and the client handler have `while` loops to keep the request-response cycle.

1. the user uses a command
2. the client creates a request and sends it to the server
3. the server serves for the request
4. the server sends a response back to the client
5. the client takes the response and prints it for the user

If the user tries to perform an action that requires admin authorization, there go two more steps before the 3^{rd} one. First, the server sends a response that indicates authorization is required, and second, the client asks the user to reenter the password and sends it to the server.

The client uses a `switch` case for the user's commands, and there is a method for each command to create the corresponding request. The client handler also uses a switch case for the requests. It uses its own methods and the database class to serve for the request and create the response.

The database class depends on the JDBC driver to execute queries on the PostgreSQL database. It has methods like `createUser`, `updateUser`, `showMessage`, etc. For instance, the `createUser` function takes a `User` object as an argument, creates an `INSERT INTO` statement and executes the query on the database, and returns the user ID of the newly created user if successful.

3.1.4 TESTING PHASE

I logged in with the username `superuser` and password `0000`. This is a user with admin privileges that is created by default when the connection is established with the database for the first time. I created two more regular users and logged in with their credentials at the same time. I tried to test all of the commands in all possible cases. Some of the major problems were as follows.

- Admins can change users' username, password, or other data even when they are online, which causes confusion and incompatibilities until they log out.
- When a user's username changes, their messages keep the old one.
- Users need to know the exact usernames of others to send a message to them.

The solutions to these problems are implemented in the second project.

3.2 SECOND PROJECT

The second project is a WSDL-based SOAP web service. In this time, a user interface is implemented as a website using HTML, CSS, and JavaScript. Users enter their usernames and passwords into the related fields of the login page, and they are able to see their profile information, inbox and sent messages, and create new messages. If the user is an admin, there is also an admin panel where they can see all the users and make changes to them.

3.2.1 ANALYSIS PHASE

For the server-side of this project, I used the Spring framework, which is one of the most popular frameworks for building web services and allows us to produce a SOAP web service.

SOAP and WSDL are XML-based, so the JAXB plugin is used to generate Java classes from the XSD schema that I created to define the domain for WSDL.

Similar to the first project, PostgreSQL and the corresponding JDBC driver are used as the database to store user and message data.

For the client-side, I did not use any framework or library and created a web client using HTML, CSS, and JavaScript.

In the first project, there was no information about the clients for the server to respond to the requests properly; therefore, I considered keeping a list of online users on the server. Additionally, I decided to no longer use usernames in any place where I could use user IDs instead, such as sender and receiver fields of messages.

3.2.2 DESIGN PHASE

There are user and message models in this project, and their structures are the same as before. Unlike the first project, they are defined in an XSD schema that will be used by the JAXB plugin to generate the corresponding Java classes. There is also a client model that consists of a session ID and a user ID. It gets assigned to a client when a user logs in, and it is required to exist in the header part of any SOAP request the client makes.

The types of all the requests and responses that go into the body part of a SOAP request are also defined in this schema file that will be exported by Spring as a WSDL.

Let us consider the same situation as in the example I gave in the first project. When `sarah03` sends a message to `eric.roy`, the client, in this case, will send a SOAP request that looks as follows.

sendMessageRequest

```xml
<?xml version="1.0" encoding="utf-8"?>
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/"
          xmlns:tns="http://sezgin.com/MyWebServer/namespace">
<Header>
  <tns:clientSoapHeaders>
    <tns:client>
      <tns:sessionID>178688221</tns:sessionID>
      <tns:userid>604176</tns:userid>
    </tns:client>
  </tns:clientSoapHeaders>
</Header>
<Body>
  <tns:sendMessageRequest>
    <tns:message>
      <tns:msgid/>
      <tns:when/>
      <tns:from>604176</tns:from>
      <tns:to>431910</tns:to>
      <tns:subject>Hello</tns:subject>
      <tns:body>Hi Eric! How are you?</tns:body>
    </tns:message>
  </tns:sendMessageRequest>
</Body>
</Envelope>
```

`178688221` is the session ID created when `sarah03` logged in, and `604176` and `431910` are the user IDs of `sarah03` and `eric.roy`, respectively. `msgid` and `when` fields are empty because they are assigned by the server when the message is sent.

The server's response would be

```
sendMessageResponse
<?xml version="1.0" encoding="utf-8"?>
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/"
          xmlns:tns="http://sezgin.com/MyWebServer/namespace">
<Header/>
<Body>
  <tns:sendMessageResponse>
    <tns:msgid>509785</tns:msgid>
  </tns:sendMessageResponse>
</Body>
</Envelope>
```

where `509785` is the message ID created by the database.

The server side is based on an endpoint, a repository, and a database. When a request comes to the server, the endpoint captures it, extracts the header and the body from it, and uses the repository to do the requested work. The repository communicates with the database via the database class.

On the other hand, the client-side is a single-page web client built with HTML, CSS, and JavaScript. It consists of

- a login form,
- a user panel where users can see their profile information and their messages, and write a new message,
- and an admin panel where admins can see and make changes on all the users.

When a user logs in, the session ID and the user ID returned by the server are stored in global variables, and the user profile gets initialized. After that, the login form gets hidden, and the user panel appears. If the user is an admin, the admin panel also appears.

3.2.3 IMPLEMENTATION PHASE

`MyWebServerEndpoint` class is the endpoint that has a method to handle each SOAP request type defined in the XSD schema. Several annotations provided by Spring such as `PayloadRoot`, `ResponsePayload`, and `RequestPayload` are used in the declaration of these functions to make the conversion between XML-based request types and corresponding Java objects.

These methods take the request object and the SOAP header as arguments and return the corresponding response object. They extract the required information from the request, use the related function in the repository to do the actual work, and create the proper response to be sent to the client.

MyWebServerEndpoint.java

```java
@PayloadRoot(
    namespace = NAMESPACE_URI,
    localPart = "getUserByUserIDRequest"
)
@ResponsePayload
public GetUserByUserIDResponse getUserByUserID(
    @RequestPayload
        GetUserByUserIDRequest request,
    @SoapHeader(value = "{" + NAMESPACE_URI + "}clientSoapHeaders")
        SoapHeaderElement header
) {
    GetUserByUserIDResponse response = new GetUserByUserIDResponse();
    response.setUser(repository.showUser(
        this.getClientSoapHeaders(header).getClient(),
        request.getUserid()
    ));
    return response;
}
```

`UserMessageRepository` class is the repository class that has methods used by the `MyWebServerEndpoint` class, such as `showMessage`, `addUser`, `listUserIDs`, etc. It also has a list named `onlineUsers` that is of type `HashMap`, and private methods `generateSessionID` and `validateSession` that are responsible for the authentication of the users.

When a client sends a login request, the `login` function, after verifying the username and the password, calls `generateSessionID` to generate a session ID and adds it to `onlineUsers` together with the user ID. Any other function than the `login` function first calls `validateSession` to recognize who is making the request and check whether the session ID exists in `onlineUsers`. After the session is authenticated, the requested action gets done and the data is returned to be used by the endpoint while creating the response.

`MyDB` class is the database class similar to the one in the first project, and it also depends on the JDBC driver for PostgreSQL. Its methods are used by the repository when the data in the database is needed, e.g., `db.insertUser(newUser)`.

On the client-side, I used a single JavaScript file that is simply a collection of all the functions for making each request and handling the response. These functions take the required inputs from the user and prepare the request according to the definitions in the XSD schema. They, then, call the `commonRequest` function by giving the request and a callback function as arguments. `commonRequest` is used as a service that creates the full SOAP request by putting its request argument into the body part of a SOAP request draft it has, sends it to the server, and finally calls the callback function when the response is ready.

3.2.4 TESTING PHASE

I used SoapUI, a web-service testing application for SOAP as well as other APIs like REST, GraphQL, etc., for testing the SOAP requests while working on the server.

Initially, I made a test request with `XMLHttpRequest` before starting the implementation of the application running on the browser. At this time, I encountered an error that says "Access to `XMLHttpRequest` at '`http://localhost:8080/ws`' from origin '`null`' has been blocked by CORS policy."

This was because the server and the client were running on different ports, therefore on different origins, and the browser was looking for additional headers in the server's response that indicate that any request from a different origin is allowed, but there were none.

A solution that does not solve but instead eliminates the problem was to serve the resources from the same origin, which is proposed by our supervisor. To do so, I created a directory named `public` and put the HTML, CSS, and JS files into it. The client was now running on the same host and port as the server and was able to receive the server's responses without any problem.

When the client application was done, I tested all the actions in all possible cases both as an admin and as a regular user. Some of the improvements other than the ones that come with using a powerful framework and providing a user interface are as follows.

- When an admin updates the data of an online user, the user's session gets invalidated, and the client automatically logs out.
- Messages keep the user IDs of the sender and the receiver to prevent keeping the old username in case of a username change.
- Users have options to reply, forward, and delete a message.

However, things are still not perfect. The major problems of this project are as follows.

- Passwords of users are stored as plain text in the database.
- Keeping a list of online users increases the workload and the memory usage of the server and would be a terrible way to authenticate users in a real-world application with millions of users.
- All of the message IDs are loaded into the inbox and sent lists at once without any limit and paging, which may result in large data transfers.
- There is no registration option for newcomers.

- Inbox and sent lists are composed only of message IDs. Consequently, users see nothing but some numbers instead of brief information about the messages, such as sender, timestamp, etc., until they click on one of them.
- Users need to know the exact usernames of others to send a new message to them.

The solutions to these are implemented in the third project.

3.3 THIRD PROJECT

The third and final project is a full-stack web application. Using modern and powerful web frameworks and libraries allowed me to produce an application that is similar to real-world applications in less time and with less effort. This project has additional features that are discussed in the next parts, and it is more secure compared to the previous ones. I tried to find and implement a solution for all the problems in previous ones.

3.3.1 ANALYSIS PHASE

We are required by our supervisor to build our applications on the MEAN stack, which is one of the most popular web development stacks and is completely based on JavaScript and TypeScript languages. The components of MEAN are the following.

MongoDB a NoSQL database based on JSON documents

Express.js a Node.js framework for back-end development

Angular a front-end web development framework

Node.js a JavaScript runtime environment

The server is now a REST API and uses JSON format to transfer data. While using MEAN stack, JSON is more preferable to XML because it directly maps to JavaScript objects and MongoDB documents, and is less verbose, which makes it faster to parse and serialize and lighter to transfer.

There are other libraries I used while building the server and the client.

First, `Mongoose`, an object modeling library, is used to define schemas and make queries on MongoDB in my JavaScript application.

The server and the client run on different ports in this project, which means that the application will probably run into an error because of the CORS policy. To prevent any issues, the `cors` library is used to allow the client to make `GET`, `POST`, `PUT`, and `DELETE` requests.

Storing the passwords of users as plain texts in the database is a big security hole. Therefore, I used the `bcrypt.js` library to encrypt the passwords before storing them and compare the passwords users enter while logging in with the encrypted ones in the database.

Another bottleneck of the previous project is keeping a list of online users with a session ID assigned to them. A more effective way to recognize and authenticate the users is to use JSON Web Tokens. So, I used the `jsonwebtoken` library to create and check tokens.

On the front-end, I used the `Bootstrap` framework to quickly design the web pages and `Moment.js` library to display timestamps relative to the moment.

3.3.2 DESIGN PHASE

The main structure of the project is based on a server, a client, and a database, and the models used are user, message, and log. The main differences between these models and the previous ones are as follows.

- user
  - user ID is an object ID assigned by MongoDB
  - two new fields `createdAt` and `updatedAt`
  - no more inbox and sent lists
- message
  - message ID is an object ID assigned by MongoDB
  - two new fields `updatedAt` and `seen`
  - `from` and `to` fields now consist of the subfileds `userId`, `username`, and `deletedThis`
- log
  - a new model for logging the user actions like register, login, logout, etc.
  - has fields like `when`, `action`, `success`, etc.

The API has five main paths, which are `/register`, `/login`, `/logout`, `/admin`, and `/user`. Admin actions, which are reading, creating, updating, and deleting the users, messages, and logs are handled by the related endpoints of the admin path. Similarly, user path has its endpoints to handle user actions such as viewing inbox and sent lists, reading and creating a message, and searching for a user. The endpoints of the API are as follows.

- `/register`
- `/login`
- `/logout`
- `/admin/users`

- `/admin/users/:userID`
- `/admin/messages`
- `/admin/messages/:messageID`
- `/admin/logs`
- `/admin/logs/:logID`
- `/user/messages`
- `/user/messages/inbox`
- `/user/messages/sent`
- `/user/messages/:messageID`
- `/user/search`

This API follows REST principles, which means that reading, creating, updating, and deleting data is done with the `GET`, `POST`, `PUT`, and `DELETE` requests respectively. For example, when an admin wants to make changes to the first name, last name, and email of the user with user ID `507f1f77bcf86cd799439011`, the client application will make a `PUT` request to the endpoint `/admin/users/507f1f77bcf86cd799439011` together with the following JSON data in the request body.

HTTP Request Body

```
{
  "firstname": "Eric",
  "lastname": "Roy",
  "email": "example@mail.com"
}
```

In this project, there are no server-side sessions, which is another principle of a RESTful service. The server expects a JWT token in the `Authorization` header of the request to authenticate the client before responding to any request. I decided the token to include the user ID and the information of whether the user is admin or not, as well as the creation and the expiration date of the token, which are by default.

The client application built on the Angular framework has login, register, user, and admin components.

The login component consists of a form where the user can enter their username and password and a link to the registration page. This is the default screen that is redirected to by the application if the user has not logged in before and tries to go to the other pages.

Register component similarly consists of a form with fields such as first name, last name, password, etc.

The user component has subcomponents for listing, reading, and creating messages. Inbox and sent message lists are tables with columns such as when, from, subject, etc., and every row corresponds to a message. The user can see the message body by clicking on a row, which also marks the message as seen, and can create a new message using the "New Message" button. I used the same component for reading and creating a message, where the only difference is that the input fields are disabled in reading mode.

The admin component is only visible to admins, and it also has subcomponents that allow admins to monitor and manipulate users, messages, and logs. The structure of the components is the same for all three, and they are similar to the user components. For example, the user list is again a table with related columns, and there is a single component for viewing, updating, and creating a user.

3.3.3 IMPLEMENTATION PHASE

This project is a combination of two Node.js applications: the server and the client. Just like all other Node.js applications, these ones have their own `package.json` files in which all the dependencies and other metadata are kept.

Let us start with the server. Dependencies of the application are

- `bcrypt.js`,
- `body-parser`,
- `cors`,
- `Express.js`,
- `jsonwebtoken`,
- `Mongoose`,

which are listed in the `package.json` file.

`server.js` is the main file in which the configurations are done, the connection to the database is established via Mongoose, and the application is set up and run on the configured port. I made the application use the dependencies such as `body-parser` and `cors` with the `use` method of Express.

User, message, and log models are defined under the `models` directory. I used Mongoose schema to define the properties of each of them and created Mongoose models depending on those schemas that allow me to make queries on the corresponding MongoDB collections.

I created a route file under the `routes` directory for each of the paths `/register`, `/login`, `/logout`, `/user`, and `/admin`, the last two of which has also subroutes under separate directories. These route files contain the endpoints that are responsible for handling `GET`, `POST`, `PUT`, or `DELETE` requests. I also created a middleware that extracts the JWT token from the `Authorization` header of the request and determines whether the user is allowed to use the related resource depending on their role.

Let us consider the same example as in the design phase, where there is a `PUT` request from the client to the endpoint `/admin/users/507f1f77bcf86cd799439011`. By the server, this request gets directed to the admin route, which also redirects it to the users subroute that is under the `admin_routes` directory. Then, the endpoint in this subroute that is responsible for handling `PUT` requests and expecting a user ID captures the request, applies the updates specified in the request body on the user with ID `507f1f77bcf86cd799439011`, and sends a result message to the client as a response.

```
server.js
```

```
app.use('/admin', adminRoute);
```

```
routes/admin.route.js
```

```
router.use('/users', usersRoute);
```

```
routes/admin_routes/users.route.js
```

```
router.put('/:userId', (req, res, next) => {
  // update the user and send a response
});
```

The server divides the data into fixed-size pages when the data is a list with no upper limit on its size, e.g., inbox messages of a user. To implement this, I created a pagination middleware that takes the page size and the page number from the query parameters and passes them to the endpoint that uses itself. Query parameters may also include the `sort` parameter by which the data is requested to be sorted.

For example, a `GET` request to the URL `/user/messages/inbox?page=3&pageSize=10&sort=-when` will receive the user's inbox messages 21 through 30, sorted by the time they were sent in descending order.

The client application is the second half of this project. Its `package.json` file includes the following dependencies.

- `Angular`
- `Bootstrap`
- `MomentJS`
- `RxJS`

The main ingredients of this application are models, components, services, and helpers.

A model is the definition of a data type used by the app. The models in this app are user, message, and log, which are rewrites of the ones in the server app.

A component of Angular, together with its template HTML document, defines a web view, which is what humans see on the browser. The main component of an Angular project is `AppComponent`, and the other components are injected into it by the framework. As described in the design phase, this application has the register, login, logout, user, and admin components. I created an additional component `AlertComponent` to show success and error messages, such as `"Message sent successfully"` and `"Invalid username or password"`.

For example, `InboxSentListComponent` is the component used when the user wants to view their inbox messages or sent messages. The template of this component, which is a piece of HTML that is injected into `AppComponent`, defines a sortable table to list the messages and provides pagination. The TypeScript file includes the variables and the functions providing reactivity and functionality to the component. `getAllMessages` function, for example, uses `UserService` to get the messages from the server and is called when the component is created or whenever the user changes the page, page size, or the sort parameter.

There are other components in this application such as `UserInfoComponent`, which defines a form and allows admins to view and manipulate users, `LoginComponent`, which also defines a form for users to enter their usernames and passwords and navigates to the home component after login, etc.

A service provides values and functions the application needs. Services in this app include user, admin, alert, and authentication services. `UserService` and `AdminService` are the services that communicate with the server by making HTTP requests for user or admin actions.

For example, the `reqGetMessageByMessageID` method of the user service is called by the related user component when the user wants to view a message. This function takes the message ID as an argument, makes a GET request to the endpoint `/user/messages/:messageID`, and returns an observable, which is a data type that allows us to pass data asynchronously, of the message type. The definition of the function is as follows.

user.service.ts

```
reqGetMessageByMessageID(_id: string) {
  return this.http.get<Message>(
    `${Config.apiURL}/user/messages/${_id}`
  );
}
```

Other services are `AlertService`, which emits alert messages to the `AlertComponent`, and `AuthenticationService`, which provides login, logout, and register functions, and stores the JWT token and the other data of the logged-in user.

Helpers in this project can be listed as follows.

`JwtInterceptor` is a request handler that puts the JWT token into the `Authorization` header of any request before it is sent to the server.

`ErrorInterceptor` is another interceptor that catches any error in the incoming responses and can be used to provide a separate handler for each error type. I used the interceptor to log the user out automatically when the response has the status code `401`, which means the JWT token is invalid and the session has expired.

`AuthGuard` is a class that redirects the application to the login screen when there is no user logged in. It also determines whether the user is allowed to navigate to a route and use the corresponding component depending on their role.

`MomentPipe` is a utility function that allows me to easily format dates and times, and display them relative to the moment using the `Moment.js` library. For example, `2021-05-02T09:43:26` is converted to `May 2nd 2021, 9:43:26`, or `12 minutes ago`.

3.3.4 TESTING PHASE

I simply run both the back-end and the front-end applications and try to test all the actions in all possible cases manually.

I created two new profiles using the register page and logged in to the application. I sent messages between the two users and tried deleting them from one of the users and then both of the users to see whether they are also deleted from the database or not.

I used an online random data generator to create fake users, messages, and logs and manually saved them to the database in order to increase the amount of data to be transferred and see how the pagination and sorting mechanism work.

I also temporarily decreased the time the JWT tokens are valid to see the behavior of the application when the session expires.

Some key points from the test results can be listed as follows.

- If the user is not logged in and tries to access another page, the client application sets the URL as the return URL and navigates to the login screen. After a successful login, the user is redirected to the return URL.
- If the user tries to access a page that they are not allowed to or does not exist, the application redirects them to the home page.
- When the JWT token expires, the user gets logged out automatically and the application navigates to the login screen with a return URL.
- Login and register operations are being logged in the database. However, though it works properly on the client application, the logout operation is not being logged

on the server due to some error I did not have enough time to figure out.

- Unread messages are highlighted with green color, and a message gets marked as seen when the receiver reads it.
- While creating a new message, users can search for other users. As they type into the "To" field, some of the usernames that match the input will be listed below.
- Deleting a message marks it as deleted, and it will get deleted from the database when both the sender and the receiver delete it. This is applied to all of the messages of a user an admin deletes.
- Pagination and sorting work properly even when the page number is out of bounds. In such cases, the server returns the first or the last page.
- Components that are reused for read, create, and update actions work without confusion.
- The passwords of the users are stored encrypted, which increases the security.
- The front-end is now more user-friendly.

4 CONCLUSION

In summary, my job in this summer practice was to develop an application in three stages with different architectures and technologies.

- A command-line interface based on multithreaded socket programming
- A SOAP web service using Spring framework and a web client based on pure HTML, CSS, and JavaScript
- A REST API based on Node.js and a web application using Angular framework

While working on those projects, I met some concepts that are new for me, such as socket programming, multithreading, asynchronous programming, and network protocols like TCP/IP and SOAP, as well as HTTP. Other than that, JavaScript and TypeScript were the programming languages that I had to learn first before building my projects using them.

Unfortunately, I was unable to be in a real workplace setting since I worked remotely from home. However, I was still able to experience a software developer's work-life full of rush and unending learning.