Report: Project 3 - CS334

**Team-Nr:** 37

**Members:**

|  |  |
| --- | --- |
| Cassandra Vellema (22543899) | Marc Hamilton (22848231) |
| Damian de Freitas (22600949) | Michelle Khanyi (23028874) |
| Christiaan Slabbert (21810354) | Marvin Lüdtke (25765744) |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

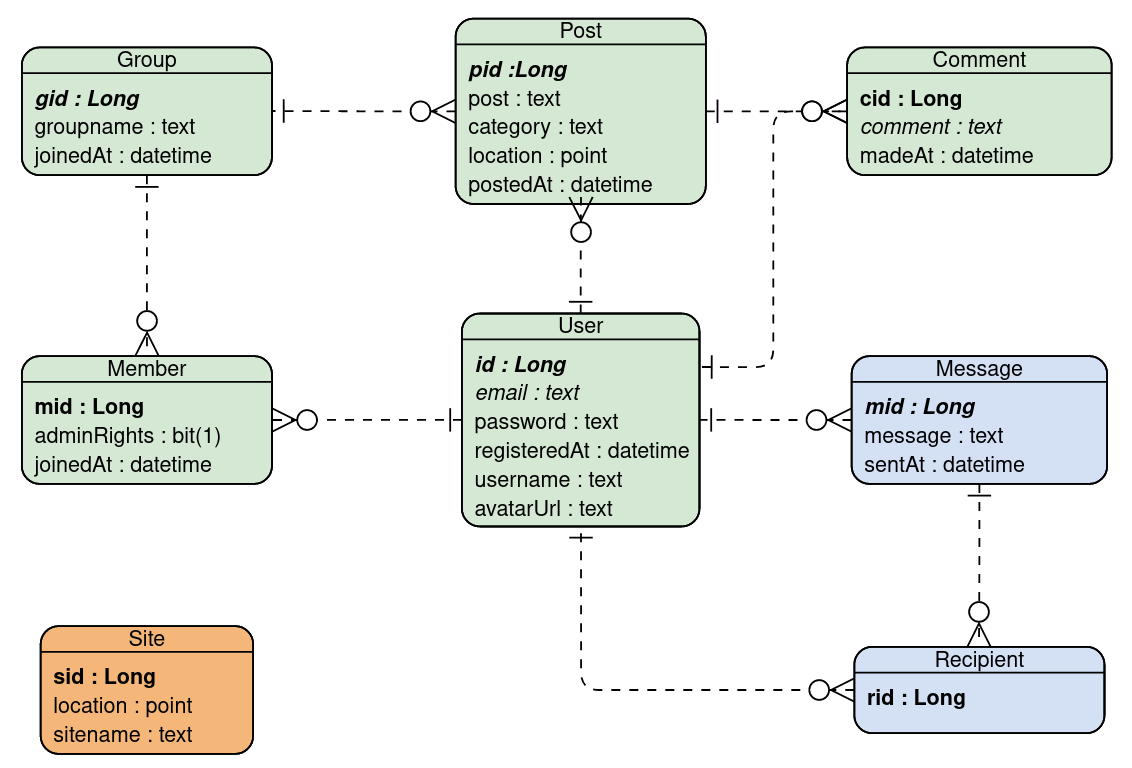
# Introduction

The application name "Propinquity" is a play on words as propinquity refers to close kinship between people but also means physical proximity. We thought this was fitting given the app is a social network with geo-tagged messages.

# Data Modelling

This section explains the implemented data model used in the relational MySQL-database in more detail. To describe the defined conceptual objects and draw the boundaries as well as connections between them, the data model will be visualized in the form of an Entity-Relationship Diagram (short: ERD). The ERD has the advantage of giving an intuitive approach to data while being a functional complete method.

The following picture shows the ERD of the database. Its representation differs slightly from the standard as relationships are not covered in an extra element but rather are displayed in the form of vertices between entities. This simplification is since a relationship in our case does not hold further information apart from making the connection between the entities. As a result, keeping the diagram as concise and compact as possible.



Entity-Relationship Diagram of the implemented data model (Generated with VisualParadigm).

In the following all entities and the relationships between them will be briefly discussed. Due to the self-explanatory nature of the ERD only outstanding attributes will be further described.

## Network

The data model is centered around the entity of a User as the main reference and structuring point. The *avatarUrl* holds a path to the avatar image a user has uploaded or by default to the default avatar. The password is not stored in plain text but is encrypted via the BCrypt algorithm provided by the springboot security framework. A Group is the platform and location entity where users interact with each other. Organizing group memberships of users and managing their admin rights is realized with the Member entity. A member connects a user with a group, which is represented by the *ManyToOne* relationship to user and group. The attribute *adminRights* specify if a user is admin of a group.

## Posts

A Post is the essential entity for the content of the application. The *category* holds the tag of a post, which implies the constraint that a post can only have one tag. *Location* is the geo point at which the user created the post. The attribute *post* is the written content from the user. For ordering purposes, the time of creation is automatically added by the database. Posts have a *ManyToOne* Relationship with both users and groups as it can only be published in one group and created by one user. The entity Comment enables further user interaction by commenting on a post. A comment is posted by a user under a post. Therefore, it holds a *ManyToOne* relationship to users and posts. The attribute set is reduced compared to a post for simplicity reasons. The Site entity is unrelated as its only purpose is to ease a location-based search by providing predefined locations.

## Messaging

The entities Message and Recipient allow server-side private messaging between users. A message is sent by a user resulting in a *ManyToOne* relationship with a user. The receiver of a message is stored separately in the recipient entity to have the option of sending a message to multiple users and keeping the 3NF. It therefore holds the *ManyToOne* relationship to a message and a user.

Operating Environment

The term solution stack simply refers to the collection of software required to develop our web application. There are three aspects of the solution stack for this project. They are the client-side system, the server-side API and the chosen database. We were given a wide range of options for each aspect of the stack, varying in both complexity as well as familiarity. In a pursuit to achieve in the upper bracket of marks we decided to take on a challenge with the technologies chosen for each aspect of our solution stack.

The chosen solution stack for this project was the following:

* The client-side system made use of Angular.
* The server-side API made use of SpringBoot.
* The chosen database engine was MySQL.

A detailed description of and the reasoning for using each technology will be given in the section below this one. For now, a general description of the stack will be given.

The operating system used during development of the entire project was Ubuntu, which is a Linux distribution based on Debian.

There are multiple programming languages involved across the solution stack. The backend was built using SpringBoot which runs on the Java Virtual Machine, JVM for short. This means that we were able to develop the server-side API exclusively using Java. Since the backend is Java-based, it allows us to make use of Apache Maven as a project management tool which is very helpful to automate the build and manage dependencies. The frontend makes use of a variety of languages. The technology used was Angular which primarily makes use of TypeScript. TypeScript is built upon JavaScript so there is some JavaScript involved along with some html and CSS used for styling the client-side system.

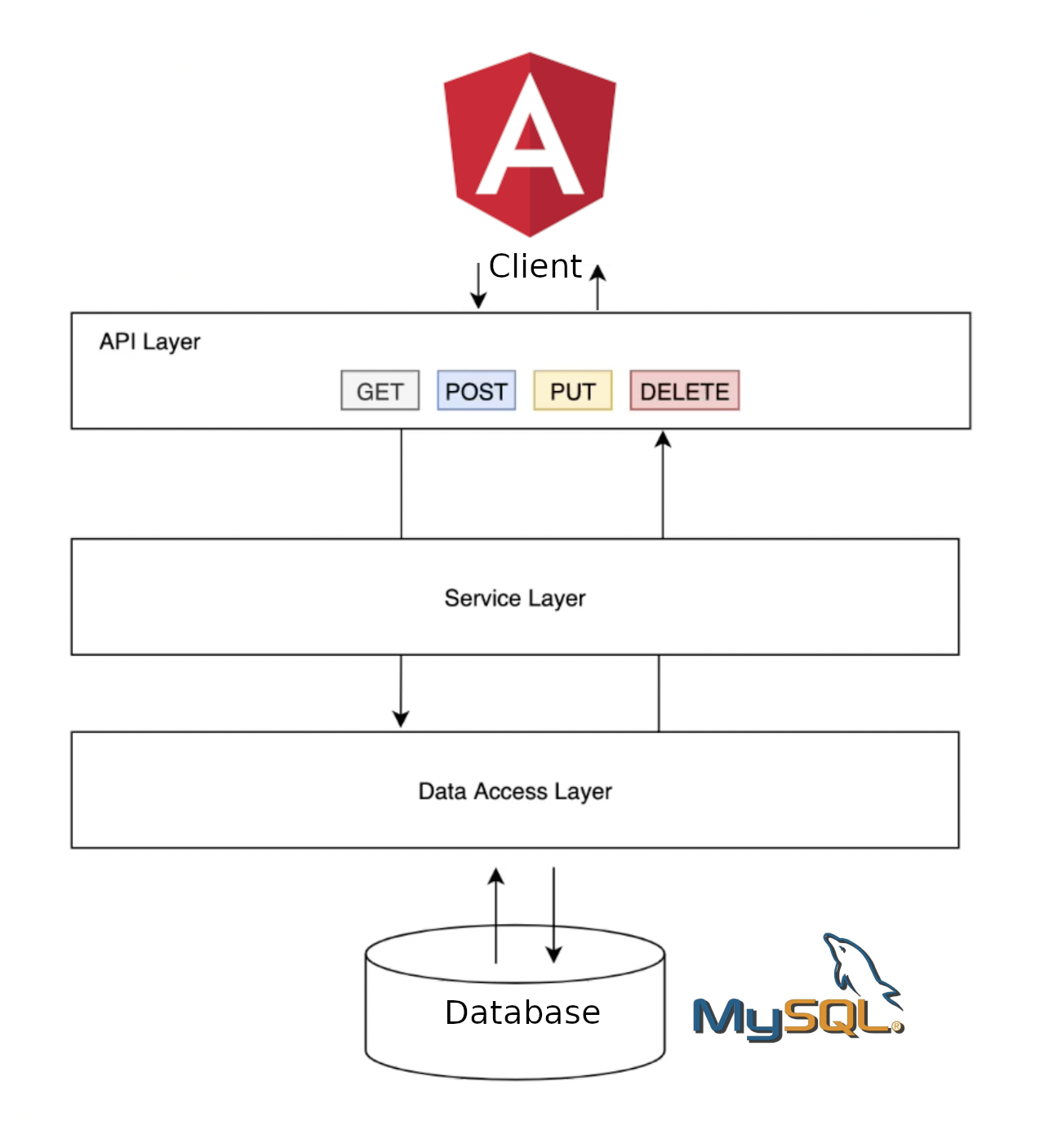
Deployment of the web application

Our web application was not deployed in the cloud and thus only runs locally.

The database which holds all information regarding our social network is run on a MySQL server on the local machine. The server-side API is also run on the local machine. SpringBoot makes use of Cross Origin Resource Sharing, which is an HTTP-header based mechanism that allows a server to indicate origins other than itself which a browser should allow to load resources. This allows us to run the client-side system on a different domain, scheme or port while still being able to access the API. This would be similar to a real-life scenario in the sense that the database and API are run on hardware completely separate from that of the client-side system.

# Solution

# Architecture descriptions



Application architecture

## Front-end client:

Angular 12, a TypeScript-based application design framework and development platform for creating efficient and sophisticated apps. As a platform, Angular includes a component-based framework for building scalable web applications, a collection of integrated libraries that cover a wide variety of features, including routing, forms management, client-server communication, and a suite of developer tools to aid in developing, building, testing, and updating of code.

The core ideas of using Angular include the use of:

1. Components, which are the building blocks that compose an application. A component includes a TypeScript class with a @Component() decorator, an HTML template, and styles.
2. Templates, in which every component has an HTML template that declares how that component renders. You can define this template either inline or by file path. Angular extends HTML with additional syntax that lets you insert dynamic values from your component.
3. Dependency injection, which allows for the declaration of dependencies of TypeScript classes without having to take care of their instantiation. Instead, Angular handles the instantiation. This design pattern allows for the developer to write more testable and flexible code.

## Back-end API:

An API (application program interface) for a website defines the way in which requesting systems should access data. It is a set of definitions and protocols used for building and integrating application software. It can be seen as the link between the consumer that wants to access content (the call) and the system that is providing the content (the response).

The architectural style used for our API can be described as RESTful. Representational State Transfer (REST) is a software architectural style that defines a set of constraints used to create Web services. For an API to be considered RESTful it must use a client-server architecture made up of clients, servers, and resources, with resource access managed through HTTP requests. The client-server communication used to access resources is said to be stateless, meaning no client information is stored between get requests and each request is separate and unconnected. RESTful Web services are preferred over other Web services, such as SOAP Web services, because they use less bandwidth and are more conducive for efficient Internet usage.

Different HTTP requests used in this project include GET, POST, PUT and DELETE which were used to read, create, update, and delete resources, respectively. This is the API layer. The HTTP requests of the different entities, their usage and routes are shown in the following tables.

**Users**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Request** | **Route** | **Header** | **Body** | **Explanation** |
| GET | <http://localhost:8080/api/users> | Authorization: Bearer token |  | Get all users |
| GET | <http://localhost:8080/api/users/user?username={{username>}} | Authorization: Bearer token |  | Get user with a specific username |
| POST | <http://localhost:8080/api/users/login> |  | Content-Type: application/json {"username": "...", "password": "...", "rememberMe": "true/false"} | Login user. Returns user token. |
| POST | <http://localhost:8080/api/users/register> | file: multipart/form-data | Content-Type: application/json  { "username": "...", "email": "...", "password": "..." }  Content-Disposition: form-data; name="file"; filename="default.png"  Content-Type: image/png | Register a new user |
| DELETE | <http://localhost:8080/api/users> | Authorization: Bearer token |  | Delete user account |
| PUT | <http://localhost:8080/api/users?username={{username}}&email={{email}}&password={{password>}} | file: multipart/form-data  Authorization: Bearer token | Content-Disposition: form-data; name="file"; filename="default.png"  Content-Type: image/png | Update user account. The desired new user details are provided as parameters in the route. Returns new user token. |

**Groups**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Request** | **Route** | **Header** | **Body** | **Explanation** |
| GET | <http://localhost:8080/api/groups> | Authorization: Bearer token |  | Get all groups |
| GET | <http://localhost:8080/api/groups/group?groupId={{groupId>}} | Authorization: Bearer token |  | Get group with groupId |
| GET | <http://localhost:8080/api/groups/group?groupName={{groupName>}} | Authorization: Bearer token |  | Get group with groupName |
| POST | <http://localhost:8080/api/groups> | Authorization: Bearer token | Content-Type: application/json { "groupName": "..." } | Create new group |
| DELETE | <http://localhost:8080/api/groups?groupId={{groupId>}} | Authorization: Bearer token |  | Delete group with groupId |
| PUT | <http://localhost:8080/api/groups?groupId={{groupId}}&groupName={{groupName>}} | Authorization: Bearer token |  | Update the groupName of group with a specific groupId |

**Members**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Request** | **Route** | **Header** | **Body** | **Explanation** |
| GET | <http://localhost:8080/api/member> | Authorization: Bearer token |  | Get all entries in members\_info table for currently logged in user. |
| GET | <http://localhost:8080/api/members?groupId={{groupId>}} | Authorization: Bearer token |  | Get all members of a specific group with groupId |
| GET | <http://localhost:8080/api/members/{{groupId>}} | Authorization: Bearer token |  | Get info regarding currently logged in user in a specific group with groupId |
| POST | <http://localhost:8080/api/members?groupId={{groupId>}} | Authorization: Bearer token |  | Join a group with a specific groupId |
| DELETE | <http://localhost:8080/api/members?userId={{userId}}&groupId={{groupId>}} | Authorization: Bearer token |  | Delete a user with userId from a group with groupId (for group admins) |
| DELETE | <http://localhost:8080/api/members?groupId={{groupId>}} | Authorization: Bearer token |  | Leave a group with groupId |
| PUT | <http://localhost:8080/api/members?userId={{userId}}&groupId={{groupId}}&adminRights={{adminRights>}} | Authorization: Bearer token |  | Update the adminRights of user with userId in group with groupId |
| PUT | <http://localhost:8080/api/members/{{groupId}}?adminRights={{adminRights>}} | Authorization: Bearer token |  | Update own adminRights (if user has adminRights to do so) |

**Posts**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Request** | **Route** | **Header** | **Body** | **Explanation** |
| POST | <http://localhost:8080/api/posts?groupId={{Id>}} | Authorization: Bearer token | Content-Type: application/json  {"post":"...","category":"...","location":{"type":"Point","coordinates":[Lon,Lat]}} | Post is mapped added to Group with user as author |
| GET | <http://localhost:8080/api/posts/allPosts> | Authorization: Bearer token |  | Get all Posts from Groups the user is in ordered by timestamp |
| GET | <http://localhost:8080/api/posts/allPosts/location> | Authorization: Bearer token | Content-Type: application/json  {"longitude":"65","latitude": "80","radius": "1"} | Get all Posts from Groups the user is in around a location ordered by timestamp |
| GET | <http://localhost:8080/api/posts/allPosts/site> | Authorization: Bearer token | Content-Type: application/json  {"sitename":"Site1","radius": "1"} | Get all Posts from Groups the user is in around a site within radius in km ordered by timestamp |
| GET | <http://localhost:8080/api/posts/allPosts/tag?tag={{tagstring>}} | Authorization: Bearer token |  | Get all Posts from Groups a user is in having the specific tag |
| GET | <http://localhost:8080/api/posts/allPosts/tags> | Authorization: Bearer token |  | Get all Tags from all Posts in the Groups a user is in |
| GET | <http://localhost:8080/api/posts/allPosts/group?groupId={{Id>}} | Authorization: Bearer token |  | Get all Posts from a Group with groupId, check if user is in that group |
| GET | <http://localhost:8080/api/posts/byUser?userId={{userId>}} | Authorization: Bearer token |  | Get all Posts from a User identified by userId |
| GET | <http://localhost:8080/api/posts/byUserOrGroup?userId={{userId}}&groupId={{groupId>}} | Authorization: Bearer token |  | Get all Posts from a Group or User or both |
| GET | <http://localhost:8080/api/posts/byTime> | Authorization: Bearer token | Content-Type: application/json  {"time": "2021-06-05T07:26:59.529","operation": "greater"/"less"} | Get all Posts after / before certain Timestamp |
| GET | <http://localhost:8080/api/posts/byLocation> | Authorization: Bearer token | Content-Type: application/json  {"longitude":"65","latitude": "80","radius": "1"} | Gives all Posts around GPS Coord within Radius in km |
| GET | <http://localhost:8080/api/posts/bySite> | Authorization: Bearer token | Content-Type: application/json  {"sitename":"Site1","radius": "1"} | Get all Posts around a certain Site which is already stored in SiteTable |
| DELETE | <http://localhost:8080/api/posts/{{postId>}} | Authorization: Bearer token |  | Delete a Post identified by its ID, check if written by user |
| PUT | <http://localhost:8080/api/posts/{{postId>}} | Authorization: Bearer token | Content-Type: application/json  {"text":"Update"} | Update the text of a Post identified by its ID, check if written by user |

**Comments**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Request** | **Route** | **Header** | **Body** | **Explanation** |
| POST | <http://localhost:8080/api/posts/{{postId}}/comment> | Authorization: Bearer token | Content-Type: application/json  {"commentText":"Text"} | Adds Comment to a Post from a User |
| GET | <http://localhost:8080/api/posts/{{postId}}/comment> | Authorization: Bearer token |  | Get all Comments to a Post |
| DELETE | <http://localhost:8080/api/posts/comment/{{commentId>}} | Authorization: Bearer token |  | Delete a Comment from a Post, check if written by user |
| PUT | <http://localhost:8080/api/posts/comment/{{commentId>}} | Authorization: Bearer token | Content-Type: application/json  {"updatedText":"Text"} | Update the Text of a Comment from a Post |

**Sites**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Request** | **Route** | **Header** | **Body** | **Explanation** |
| POST | <http://localhost:8080/api/site> | Authorization: Bearer token | Content-Type: application/json  {"siteName":"Site1","location":{"type":"Point","coordinates":[Lon,Lat]}} | Adds Site |
| GET | <http://localhost:8080/api/site/id?sId={{Id>}} | Authorization: Bearer token |  | Get Site by its ID |
| GET | <http://localhost:8080/api/site/name?name={{Name>}} | Authorization: Bearer token |  | Get Site by its Name |
| GET | <http://localhost:8080/api/site/all> | Authorization: Bearer token |  | Get all Sites |
| DELETE | <http://localhost:8080/api/site/{{siteId>}} | Authorization: Bearer token |  | Delete a Site |

The API layer communicates with the service layer to access data. It calls functions in the service layer. The service layer serves as business logic for managing the different entities in the data access layer. In the data access layer, we create different entities (java classes) that each map to and control a table in the database.

# Additional packages

## Front-end client:

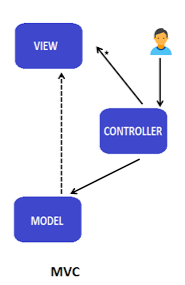
Angular Material, JavaScript, TypeScript

## Back-end API:

Spring data JPA, MySQL connector for Java, Hibernate, Spring security, Jsonwebtoken, Jackson, Persistence

# Design pattern

## Model View Controller (MVC)



For this project the MVC design pattern is used. MVC design pattern divides an application into three major aspects: Model, View, and Controller.

**Model**

The model aspect is the data that needs to be displayed by the view. It consists of a collection of classes that make up the business logic (service layer and the data access layer in the application architecture). It also defines the constraints for working with the data.

**View**

The view aspect is our front-end client, and it displays the data obtained from the controller as output.

**Controller**

The controller represents the API layer in the application architecture. It handles the requests from the view and sends the results back. It is the mediator between the view (front-end client) and the model (back-end API).

# Technologies used

## Front-end client: Angular 12

The Angular 12 front-end framework was chosen as it proved easy to create and maintain customizable components and route between them, either making them independent or dependent on one another.

Angular was also chosen for its ease of testing, as the Angular.js modules contain the application parts, which are easy to manipulate. Angular ensures easy development as it eliminates the need for unnecessary code. It has a simplified MVC architecture.

Angular is also our choice due to its modular nature - Angular organizes code into modules. Modules make application functionality organization easy, segregating it into features and reusable parts.

The popularity of Angular has resulted in the appearance of many additional tools and components that can be used in Angular apps, some of which we investigated in our research.

When it came to our implementation of Angular 12 for this project, the division of work was as follows:

* Login and register components,
* Homepage feed and post creator/search,
* User profile and modification.

Thus, Angular components were created with routes linking them for navigation from the user’s perspective. The general flow of the client-side stack:

Login/register -> Homepage landing page

-> Post creator/searching

-> User profile and modification

## Back-end API: SpringBoot

SpringBoot is an open-source Java based framework that simplifies creating stand-alone, production-grade Spring based micro services. SpringBoot is by far one of the most popular frameworks. It gives you about everything you need to build applications and makes your life as a developer much easier. With SpringBoot it is easy to connect to any database, whether it be MySQL, PostgreSQL, etc., and it has modules such as spring data JPA, which simplifies writing queries and working with databases. There are security modules available as well such as spring security. SpringBoot is also quite easy to learn as a beginner and has great community support.

## Database: MySQL 8.0.25

MySQL was a good choice as our database, because of its many advantages. MySQL is a reliable database system that has high performance when it comes to heavy reading operations. Most of its developers contend that MySQL has a speed advantage over other database systems. It is also a relatively simple database to use and was also quite easy to obtain and set up. Because it supports the SQL Query language; the database was easily set up by writing a simple sql script that was run in the MySQL shell. Another reason MySQL was a good choice, is because of its large community support.