Assignment 1 Documentation

Utiu Monica-Iulia
30433

Table of contents

1. Requirement Analysis

1.1. Introduction

We will design and implement a simple version of StackOverflow.
This is a Java Web application desired to hold questions and answers posted by programmers .
The application has two types of users (user and moderator) which must provide a username/email and a password in order to use the application.
These users can add questions and answers which are stored in a database. The database is implemented in MySQL Workbench. There are 6 different tables in the database: users, questions, answers, votes (one for questions, one for answers), tags
Users table is populated with data about application users. The data about them are: a unique ID, first name, last name, email, phone. We have also added the columns roles (user/moderator) and rating (initialized with 0).

1.2. Functional Requirements

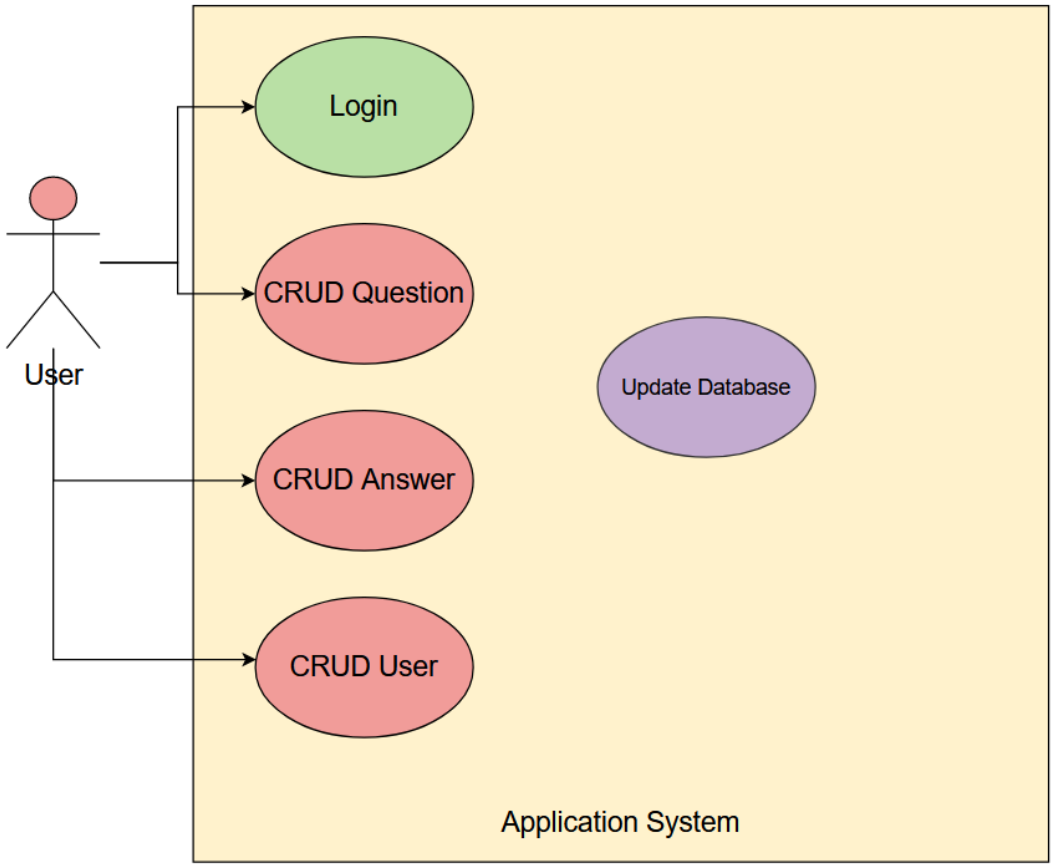
- The admin user can perform the following operations:
  CRUD on questions (their questions)
  CRUD on answers of questions (their answers)
- The system can perform the following operations:
  CRUD on users
- Data is stored in a relational database.
- Entity mapping.
- Layers architectural pattern is used to organize the application and OOP concepts are respected.
- Endpoint setup.

1.3. Non-functional Requirements

2. Technologies

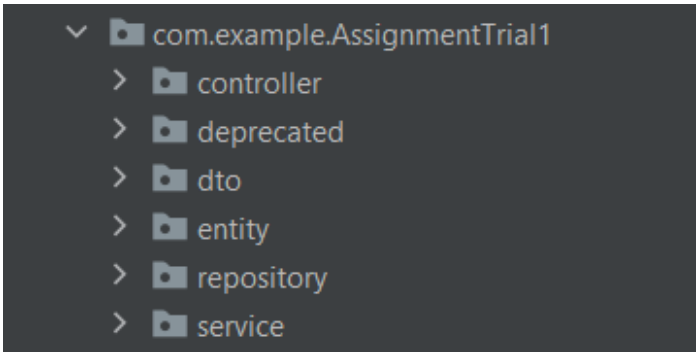
For web development, we are using Intellij as IDE, Spring Boot Initializr to add the dependencies for us, Spring Boot and Hibernate for the backend.
For the database, we are using MySql and MySql Workbench.
For the endpoints, we are using Postman to send requests and view responses.

3. Use-case Model

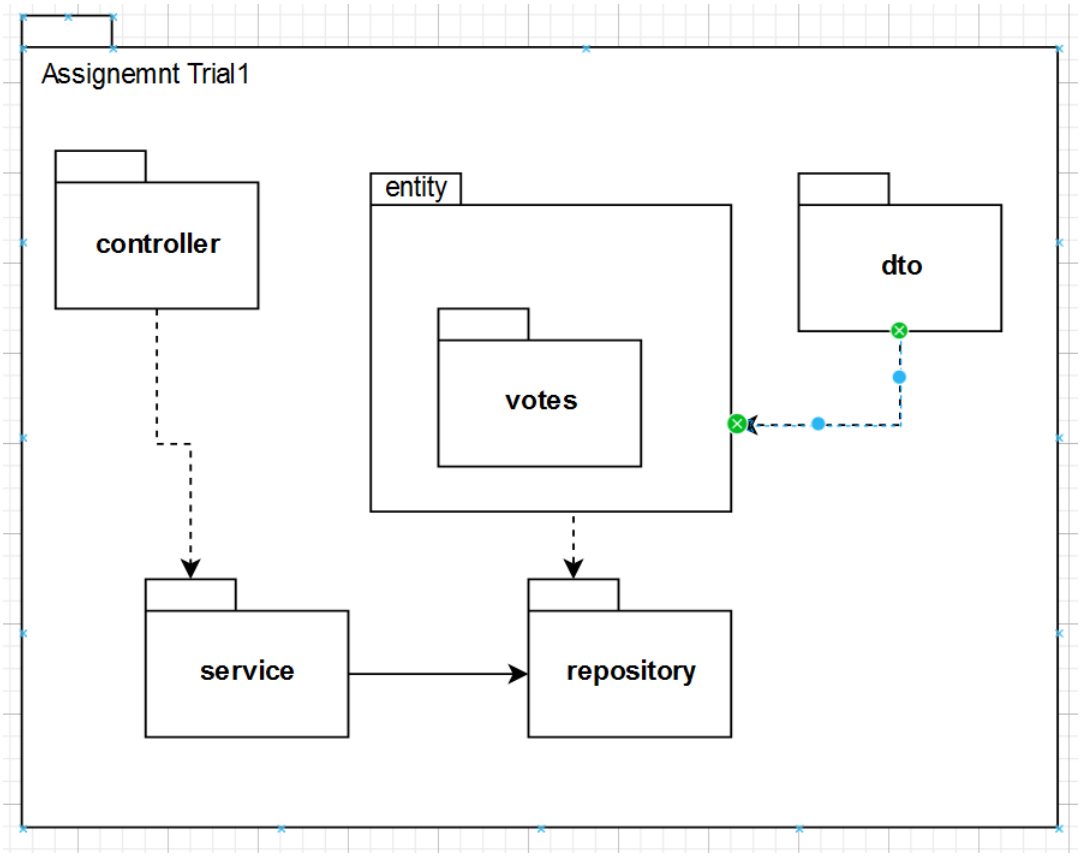


4. Architectural Design

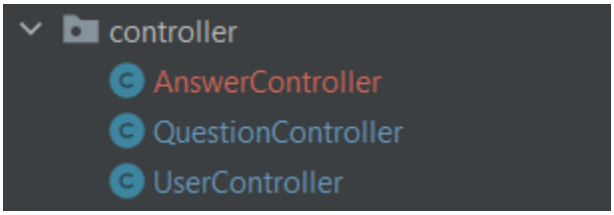
4.1. Packages



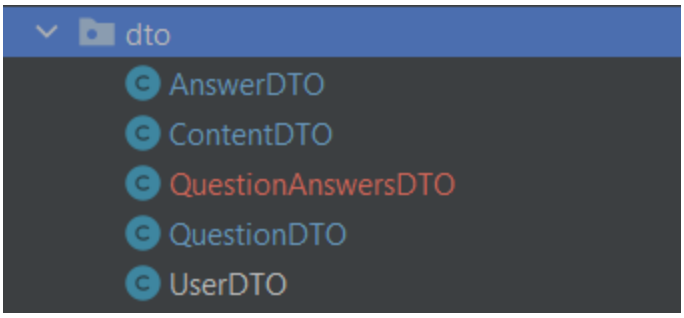
The application respects the Model-View-Controller model, therefore the packages are split into corresponding entities: entity, dto, service, repository – not implemented yet – controller.



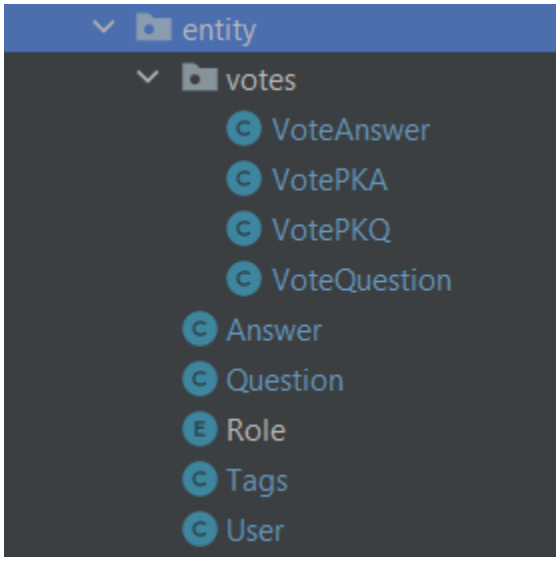
- The controller package contains a RestController class for each entity – user, question, answer. They contain the corresponding services which are Autowired and the request mappings for the endpoints, which call those services methods.



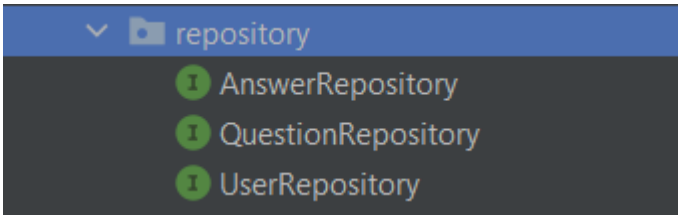
- The dto package has some Data Tranfer Objects for the classes used with the endpoints: UserDTO, QuestionDTO, AnswerDTO, QuestionAnswerDTO and ContentDTO which is a superclass for QuestionDTO and AnswerDTO. These classes hold only specific fields of their initial entities to be displayed after an endpoint request.



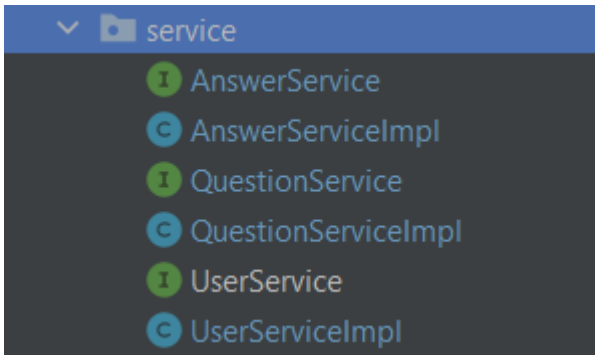
- The entity package has an Entity Class for each table in the database, an enumeration for the User class to use and another package called votes, which contains the Entity classes for votes, but also their Embedded Id classes (VotePKA, VotePKQ).



- The repository package has an interface which implements CrudRepository<> for each class on which we implemented the CRUD operations.



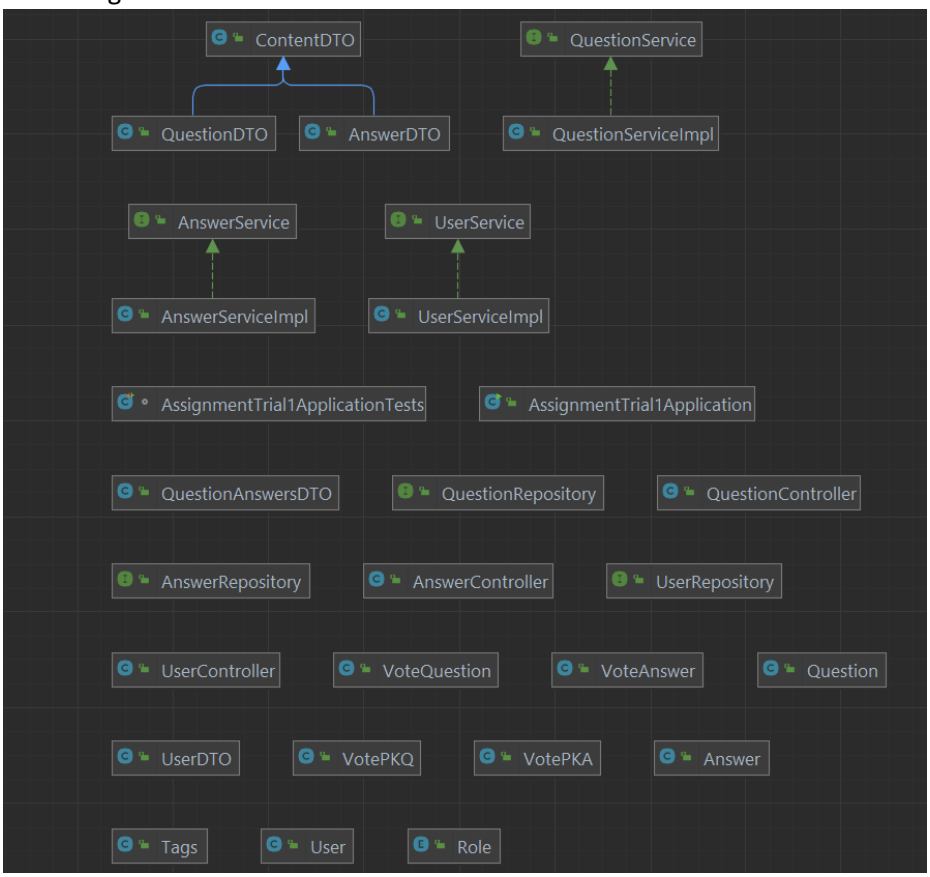
- The service package holds a service interface and a service implementation (class) for each class that has a controller (has CRUD operation). The service implements the service interface and has the corresponding CrudRepository as an autowired field that it calls in its methods.

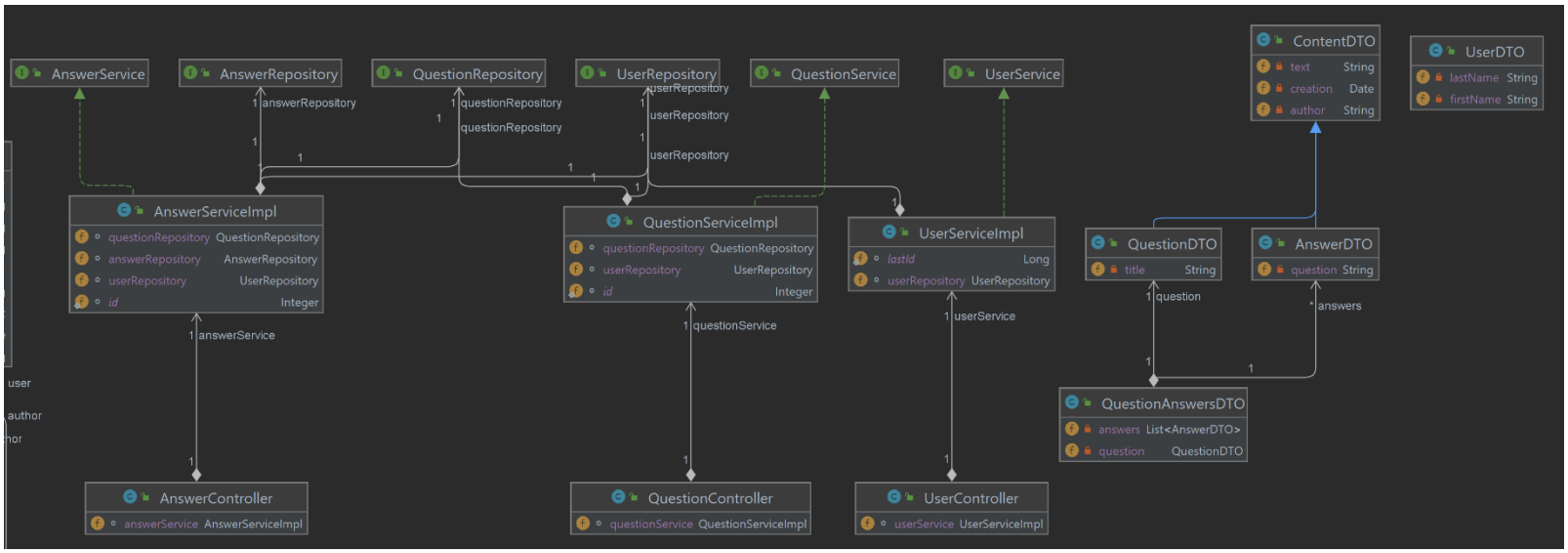


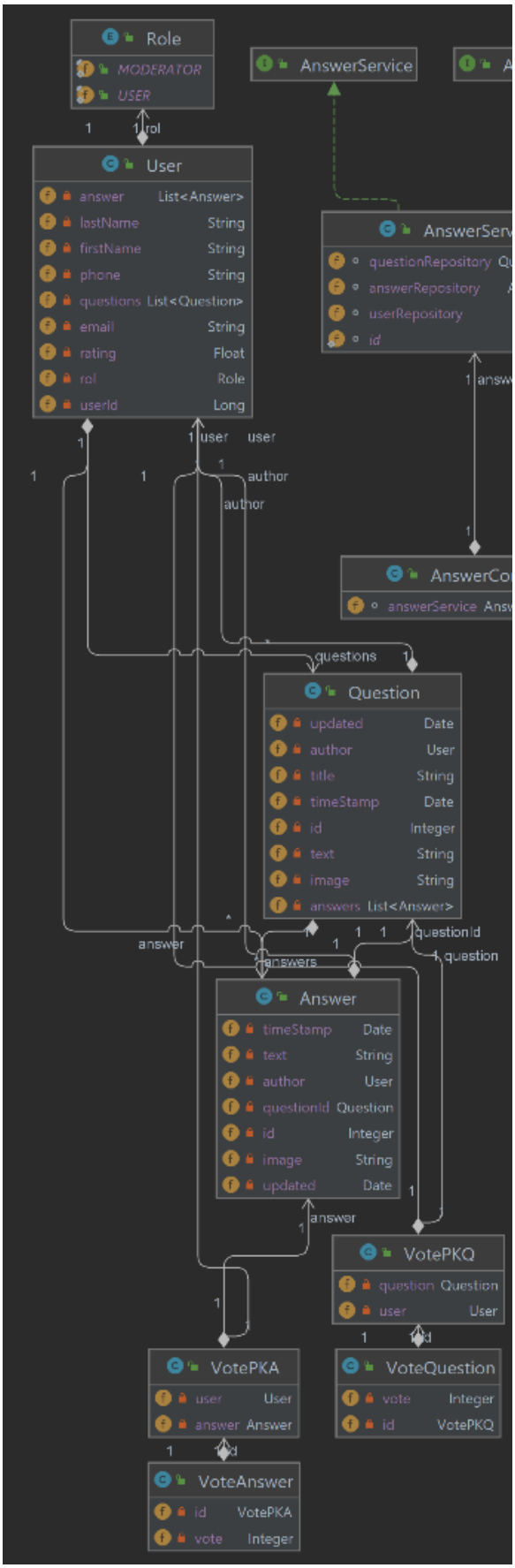
The main package also contains the spring boot application to run.

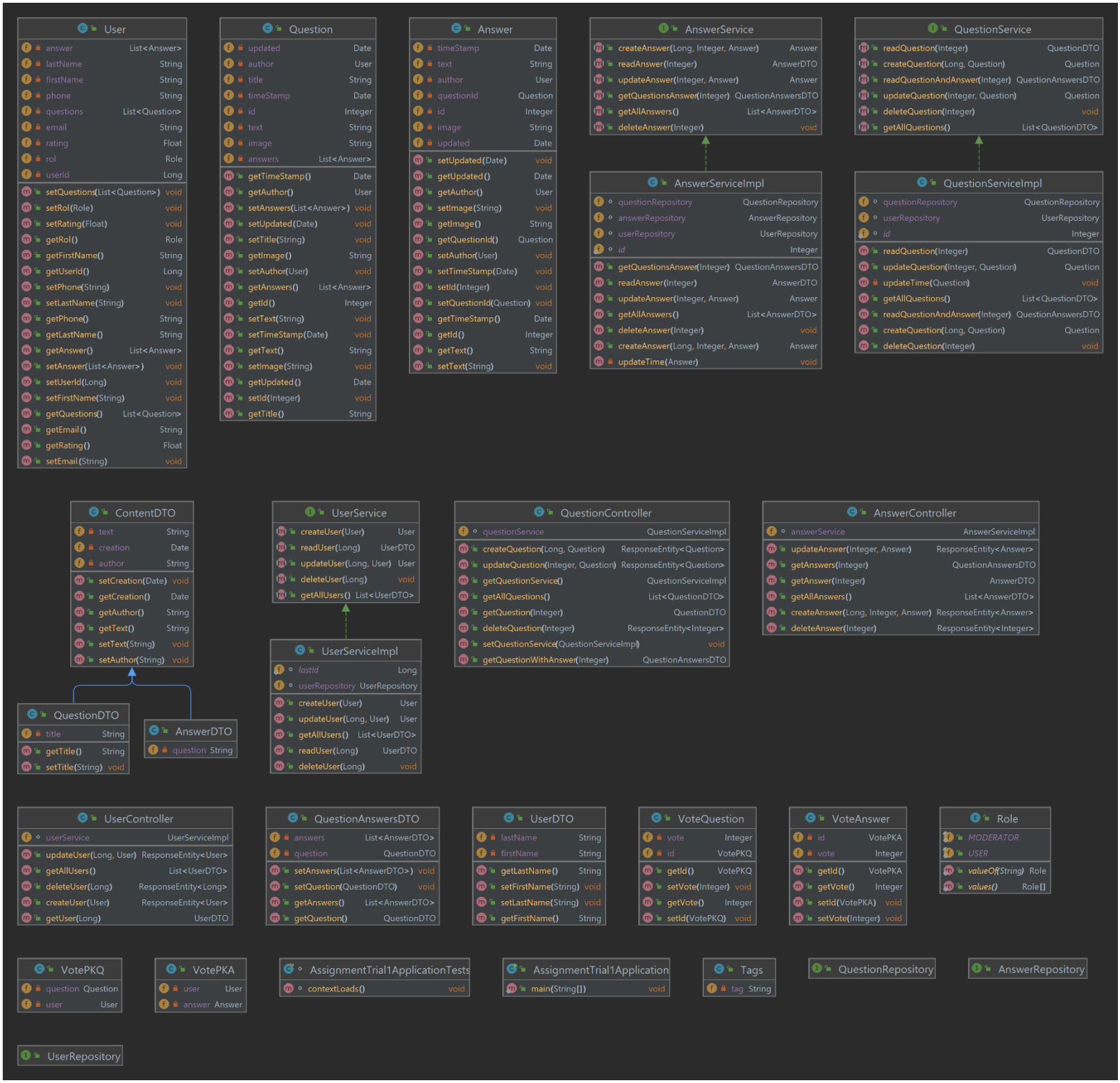
4.2. Class Design

- Class Diagram




AnswerService
AnswerRepository
QuestionRepository
UserRepository
QuestionService
UserService
ContentDTO
text String
creation Date
author String
UserDTO
lastName String
firstName String
AnswerServiceImpl
questionRepository QuestionRepository
answerRepository AnswerRepository
userRepository UserRepository
id Integer
QuestionServiceImpl
questionRepository QuestionRepository
userRepository UserRepository
id Integer
UserServiceImpl
lastId Long
userRepository UserRepository
QuestionDTO
title String
AnswerDTO
question String
QuestionAnswersDTO
answers List<AnswerDTO>
question QuestionDTO
AnswerController
answerService AnswerServiceImpl
QuestionController
questionService QuestionServiceImpl
UserController
userService UserServiceImpl
answerRepository
questionRepository
userRepository
answerService
questionService
userService
question
answers


Role
MODERATOR
USER
AnswerService
User
answer List<Answer>
lastName String
firstName String
phone String
questions List<Question>
email String
rating Float
rol Role
userId Long
AnswerServ
questionRepository
answerRepository
userRepository
id
AnswerCo
answerService
Question
updated Date
author User
title String
timeStamp Date
id Integer
text String
image String
answers List<Answer>
Answer
timeStamp Date
text String
author User
questionId Question
id Integer
image String
updated Date
VotePKQ
question Question
user User
VotePKA
user User
answer Answer
VoteQuestion
vote Integer
id VotePKQ
VoteAnswer
id VotePKA
vote Integer
rol
user
author
questions
answer
answers
questionId
question


User
answer List<Answer>
lastName String
firstName String
phone String
questions List<Question>
email String
rating Float
rol Role
userId Long
setQuestions(List<Question>) void
setRol(Role) void
setRating(Float) void
getRol() Role
getFirstName() String
getUserId() Long
setPhone(String) void
setLastName(String) void
getPhone() String
getLastName() String
getAnswer() List<Answer>
setAnswer(List<Answer>) void
setUserId(Long) void
setFirstName(String) void
getQuestions() List<Question>
getEmail() String
getRating() Float
setEmail(String) void
Question
updated Date
author User
title String
timeStamp Date
id Integer
text String
image String
answers List<Answer>
getTimeStamp() Date
getAuthor() User
setAnswers(List<Answer>) void
setUpdated(Date) void
setTitle(String) void
getImage() String
setAuthor(User) void
getAnswers() List<Answer>
getId() Integer
setText(String) void
setTimeStamp(Date) void
getText() String
setImage(String) void
getUpdated() Date
setId(Integer) void
getTitle() String
Answer
timeStamp Date
text String
author User
questionId Question
id Integer
image String
updated Date
setUpdated(Date) void
getUpdated() Date
getAuthor() User
setImage(String) void
getImage() String
getQuestionId() Question
setAuthor(User) void
setTimeStamp(Date) void
setId(Integer) void
setQuestionId(Question) void
getTimeStamp() Date
getId() Integer
getText() String
setText(String) void
AnswerService
createAnswer(Long, Integer, Answer) Answer
readAnswer(Integer) AnswerDTO
updateAnswer(Integer, Answer) Answer
getQuestionsAnswer(Integer) QuestionAnswersDTO
getAllAnswers() List<AnswerDTO>
deleteAnswer(Integer) void
AnswerServiceImpl
questionRepository QuestionRepository
answerRepository AnswerRepository
userRepository UserRepository
id Integer
getQuestionsAnswer(Integer) QuestionAnswersDTO
readAnswer(Integer) AnswerDTO
updateAnswer(Integer, Answer) Answer
getAllAnswers() List<AnswerDTO>
deleteAnswer(Integer) void
createAnswer(Long, Integer, Answer) Answer
updateTime(Answer) void
QuestionService
readQuestion(Integer) QuestionDTO
createQuestion(Long, Question) Question
readQuestionAndAnswer(Integer) QuestionAnswersDTO
updateQuestion(Integer, Question) Question
deleteQuestion(Integer) void
getAllQuestions() List<QuestionDTO>
QuestionServiceImpl
questionRepository QuestionRepository
userRepository UserRepository
id Integer
readQuestion(Integer) QuestionDTO
updateQuestion(Integer, Question) Question
updateTime(Question) void
getAllQuestions() List<QuestionDTO>
readQuestionAndAnswer(Integer) QuestionAnswersDTO
createQuestion(Long, Question) Question
deleteQuestion(Integer) void
ContentDTO
text String
creation Date
author String
setCreation(Date) void
getCreation() Date
getAuthor() String
getText() String
setText(String) void
setAuthor(String) void
UserService
createUser(User) User
readUser(Long) UserDTO
updateUser(Long, User) User
deleteUser(Long) void
getAllUsers() List<UserDTO>
UserServiceImpl
lastId Long
userRepository UserRepository
createUser(User) User
updateUser(Long, User) User
getAllUsers() List<UserDTO>
readUser(Long) UserDTO
deleteUser(Long) void
QuestionController
questionService QuestionServiceImpl
createQuestion(Long, Question) ResponseEntity<Question>
updateQuestion(Integer, Question) ResponseEntity<Question>
getQuestionService() QuestionServiceImpl
getAllQuestions() List<QuestionDTO>
getQuestion(Integer) QuestionDTO
deleteQuestion(Integer) ResponseEntity<Integer>
setQuestionService(QuestionServiceImpl) void
getQuestionWithAnswer(Integer) QuestionAnswersDTO
AnswerController
answerService AnswerServiceImpl
updateAnswer(Integer, Answer) ResponseEntity<Answer>
getAnswers(Integer) QuestionAnswersDTO
getAnswer(Integer) AnswerDTO
getAllAnswers() List<AnswerDTO>
createAnswer(Long, Integer, Answer) ResponseEntity<Answer>
deleteAnswer(Integer) ResponseEntity<Integer>
QuestionDTO
title String
getTitle() String
setTitle(String) void
AnswerDTO
question String
UserController
userService UserServiceImpl
updateUser(Long, User) ResponseEntity<User>
getAllUsers() List<UserDTO>
deleteUser(Long) ResponseEntity<Long>
createUser(User) ResponseEntity<User>
getUser(Long) UserDTO
QuestionAnswersDTO
answers List<AnswerDTO>
question QuestionDTO
setAnswers(List<AnswerDTO>) void
setQuestion(QuestionDTO) void
getAnswers() List<AnswerDTO>
getQuestion() QuestionDTO
UserDTO
lastName String
firstName String
getLastName() String
setFirstName(String) void
setLastName(String) void
getFirstName() String
VoteQuestion
vote Integer
id VotePKQ
getId() VotePKQ
setVote(Integer) void
getVote() Integer
setId(VotePKQ) void
VoteAnswer
id VotePKA
vote Integer
getId() VotePKA
getVote() Integer
setId(VotePKA) void
setVote(Integer) void
Role
MODERATOR
USER
valueOf(String) Role
values() Role[]
VotePKQ
question Question
user User
VotePKA
user User
answer Answer
AssignmentTrial1ApplicationTests
contextLoads() void
AssignmentTrial1Application
main(String[]) void
Tags
tag String
QuestionRepository
AnswerRepository
UserRepository

- Class Description

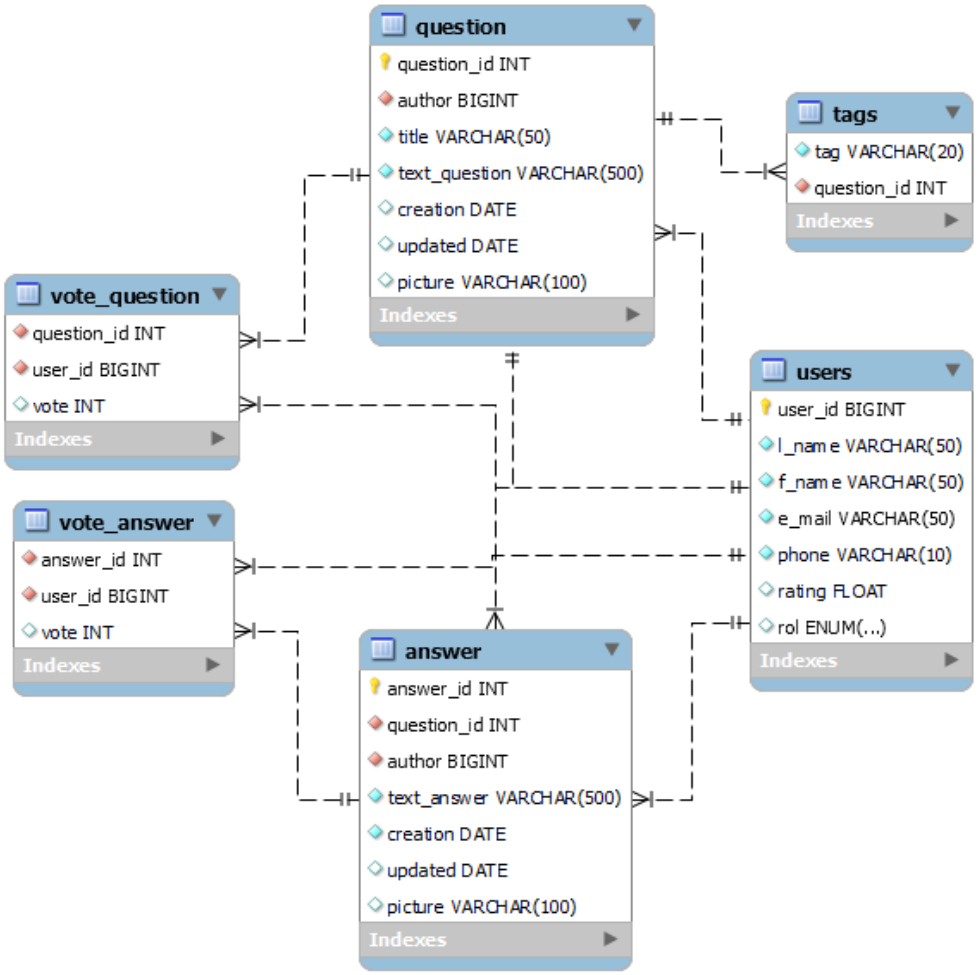
  The classes in the entity package are annotated with the @Entity tag and represent tables in the database, therefore each field that is present in the database is annotated with the @Column tag and the fields that are not present in the database are fields for the bidirectional relationship with another table (represents a foreign key refrenece of the other table) and are annotated with the corresponding relationship.

  The entities which are the owners of the relationship, have the field also as column in the table so the @JoinColumn annotation is added besides the relationship annotation.

  The tags class and tags field in the Question class is the only one to have a ManyToMany relationship annotated and it provides the join columns for join and inverse join.

  The package votes has two classes which represent composite primary keys for the other two classes (their tables) and are annotated with the @Embeddable (in the key class) and @EmbeddedId (in the vote class, key fied) tags.
  The dto package has a corresponding class for each entity that has CRUD operations and defines classes with only some of the fields of the original entities to be printed upon data access/ transfer requests (http requests in our case) and are present in the controllers as output variables for some of the requests. There is a bonus class QuestionAnswersDTO modeled to have a QuestionDTO and a list of AnswersDTO to hold the information wanted for a question that was got with its answers.
  The repositories are just empty interfaces, simply implementing the CrudRepository<> with the arguments type being the type of the corresponding entity and it's id.
  The services each have at least their corresponding entities repository and implement each functionality for the http requests written in the controller class. In the QuestionRepository we also included the @Autowired UserRepository to be able to access/get the user by id for some operations. We did the same thinfg for the AnsweRepository which has all the repositories and uses them for access to each entity.
  There are 3 controllers, all of rest type and one for each CRUD supporting entity. They hold an @Autowired service of the entity type and all the http requests with their appropriate annotations, containing their URIs, accompanied by the @ResponseBody to mark the need of a response in the request. The other annotations will be explained lower, when talking about the endpoints requests. Each method only calls the service and processes the output (as a dto entity or ResponseEntity).

5. Database Design



The tables are as presented above.
The users table only has a primary key and an enumeration as features, but the question and answer table, as well as the vote tables have a foreign key referencing the user_id from users:

foreign key (author) references users(user_id)

The answer table also has a reference to the question table:

foreign key (question_id) references question(question_id)

The vote_question also refers to the same question_id, as does the tags table, whilst the vote_answer refers to the answer_id from the answer table:

foreign key (answer_id) references answer(answer_id)

The tags table does not have a primary key, because it has a many to many relationship between itself and the question table.

The relationships in the diagrams are:

- Users -> Questions : Many -> One
- Users -> Answers : Many -> One
- Users -> Vote (answer/question) : One -> Many
- Question -> Answer : One-> Many
- Question -> Tags : Many -> Many
- Question -> Vote : One -> Many
- Answer -> Vote : One -> Many

6. Endpoint Requests

APIs work using ‘requests’ and ‘responses.’ When an API requests information from a web application or web server, it will receive a response. The place that APIs send requests and where the resource lives, is called an endpoint.

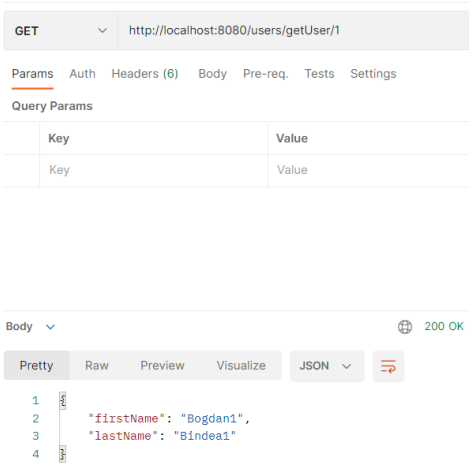
This API requests contain URIs described by the mappings in the controller classes: (examples)


```
no usages   new *
@GetMapping(path="/getAllQuestions")
@ResponseBody
public List<QuestionDTO> getAllQuestions() { return questionService.getAllQuestions(); }
```

- User – get user by id

```
no usages   Monica Utiu
@GetMapping(path="/getUser/{user_id}")
@ResponseBody
public UserDTO getUser(@PathVariable Long user_id) { return userService.readUser(user_id); }
```

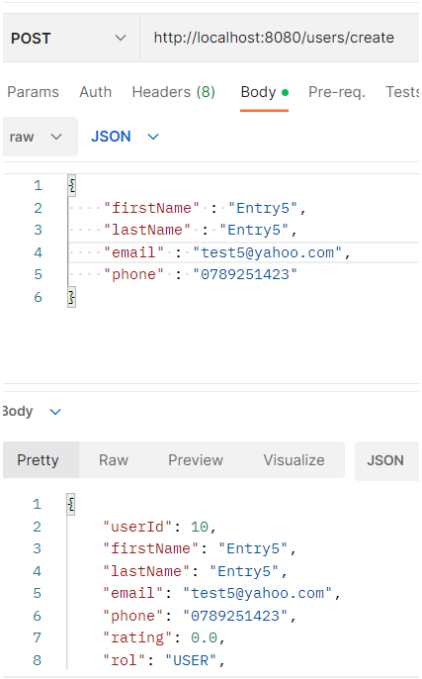
The URI will be of the form “/getUser/{user_id}”, where user_id is a PathVariable given by the GET request. The service and the repository will process further the request and the result, upon success, will be given through the ResponseBody in the UserDTO format back to the place where the request was sent (Postman), which views it as a JSON:



- User – create a new user

```
no usages  Monica Utiu
@PostMapping(path = "create")
@ResponseBody
public ResponseEntity<User> createUser(@RequestBody User newUser) {
    User user = userService.createUser(newUser);
    if (user == null) {
        return new ResponseEntity<>(null,HttpStatus.BAD_REQUEST);
    } else {
        return new ResponseEntity<>(user, HttpStatus.CREATED);
    }
}
```

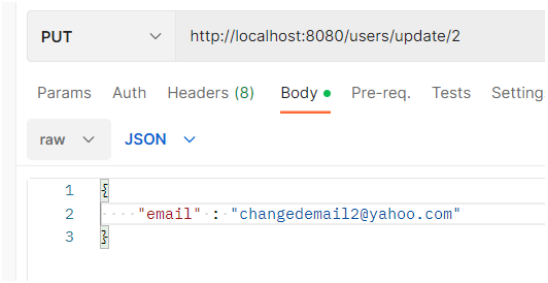
This is a post request type and uses a Request Body to create the new user. This Request Body calls for a json from the request handler which will be parsed to get the User fields for the new user. If the json is wrong, the request will fail and a jakcson type error will appear in the program, whilst postman would display an Internal Server Error. Upon success, Postman would display a json with the User data which was added to the user repository.



- User – update user

```
@PutMapping(path="update/{user_id}")
@ResponseBody
public ResponseEntity<User> updateUser(@PathVariable Long user_id, @RequestBody User userDetails) {
    User user = userService.updateUser(user_id, userDetails);
    if( user!= null)
        return new ResponseEntity<>(user,HttpStatus.OK);
    return new ResponseEntity<>(null, HttpStatus.NO_CONTENT);
}
```

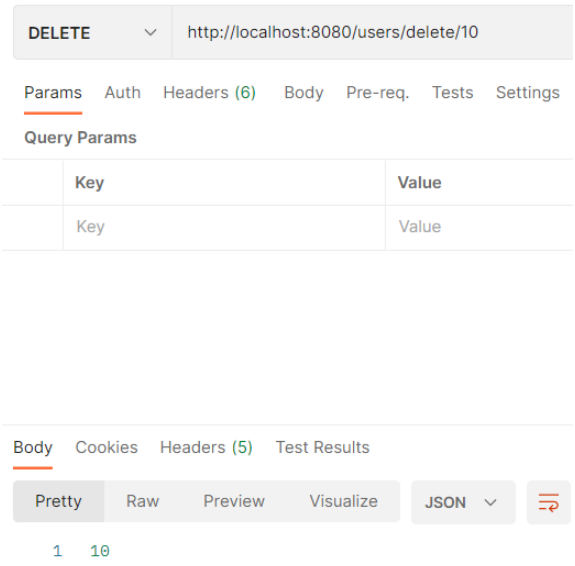
This is a Put Request, which updates the user if it is found and does not save a new user upon giving an unexisting user. It has both a PathVariable and RequestBody described as above. The request gets the User id from the URI and the new details for the update from the RequestBody. A response entity is sent back as a response which displays the user updated data and gets the Http status.



- User – delete user

```
@DeleteMapping(path="delete/{user_id}")
@ResponseBody
public ResponseEntity<Long> deleteUser(@PathVariable Long user_id) {
    userService.deleteUser(user_id);
    return new ResponseEntity<>(user_id, HttpStatus.OK);
}
```

This http request is of delete type. It gets the path variable from the URI and the repository deals with the deletion. If the user id does not exist, no error will appear, and the status of the request is still OK. The ResponseEntity is the Integer of the deleted id.



- Answer – create answer

```
@PostMapping(path = "create/user/{u_id}/question/{q_id}")
@ResponseBody
public ResponseEntity<Answer> createAnswer(@PathVariable Long u_id,@PathVariable Integer q_id,@RequestBody Answer newAnswer) {
    Answer answer = answerService.createAnswer(u_id,q_id,newAnswer);
    if(answer==null)
        return new ResponseEntity<>(null, HttpStatus.BAD_REQUEST);
    return new ResponseEntity<>(answer,HttpStatus.CREATED);
}
```

This http post request has two path variables, one for the user id that answers the question, and a question id for the question responded to. There is also the Request Body for the new answer. For this in the answer service I have all the repositories, for the user, question and answer to be able to find the question and user by id. Besides this, it is processed as all the other requests described.

- Endpoints and their requests:
  - User :
    - Get user by id
    - Get all users
    - Post/create a new user
    - Put/update user
    - Delete user
  - Question:
    - Get question by id
    - Get all question
    - Get question with its answers
    - Post/ create a new question
    - Put/ update question
    - Delete question
  - Answer:
    - Get answer by id
    - Get all answers
    - Get answers for a specific question
    - Post/ create a new answer for a question by an user
    - Put/ update answer
    - Delete answer

Deliverable 2

7. Frontend architecture

The front-end architecture of the application follows the Angular framework's recommended structure. It uses components, services, modules, and routing to create a modular and maintainable codebase.

Components

1. `AppComponent`: The root component of the application.
2. `LoginComponent`: Responsible for handling user login functionality. It takes input from the user, validates the credentials, and logs the user in using the `AuthService`.
3. `HeaderComponent`: Renders the application header. It includes a logo, user profile icon, logout button, and a button to ask a new question.
4. `SearchBarComponent`: Provides a search bar for searching content. It emits an event when the user enters a term.
5. `QuestionComponent`: Displays a single question. It shows information about the question, such as the author, votes, answers, and creation date. It also provides options to edit or delete the question.
6. `QuestionListComponent`: This component represents a list of questions. It retrieves questions from a `QuestionService` and displays them in a sorted order. It also includes a search functionality based on the `term` property.
7. `QuestionPageComponent`: This component represents a single question page. It retrieves a specific question and its answers from the `QuestionService` and displays them. It also includes a form for adding new answers.`AnswerListComponent`: Renders a list of answers for a question.
8. `QuestionCreatePageComponent`: Page for creating a new question.
9. `UserInfoComponent`: Shows user information.
10. `UserPageComponent`: This component represents a user's profile page. It retrieves user information from the `AuthService` and displays it. It also includes a functionality for banning/unbanning users.
11. `AnswerComponent`: Represents a single answer, is responsible for displaying an answer. It handles functionalities like editing and deleting the answer.
12. `AnswerListComponent` : Displays a list of answers. It sorts the answers based on their votes.
13. `AnswerFormComponent`: Form used for creating and updating answers. It takes input from the user and communicates with the `AnswerService` to perform the necessary operations.`QuestionFormComponent`: Form for adding a question.
14. `VoteComponent`: This component allows users to vote on questions or answers. It communicates with the `VoteService` to add votes for the specified question or answer.

Services

1. `AuthGuard`: Protects routes by checking if the user is authenticated.
2. `AuthService`: This service handles user authentication and user-related operations such as login, logout, and retrieving user information. It communicates with the server to perform these operations. It also uses the local storage to store the needed user data and to clear it at logout.
3. `VoteService`: Handles voting functionality.
4. `AnswerService`: This service is responsible for performing CRUD (Create, Read, Update, Delete) operations related to answers. It communicates with the server using HTTP requests.
5. `QuestionService`: This service handles operations related to questions, such as retrieving questions, creating new questions, and updating/deleting existing questions. It also communicates with the server using HTTP requests.
6. `TagService`: This service is responsible for managing tags related to questions. It retrieves tags from the server and provides methods for creating new tags.

Modules

1. `AppModule`: The root module of the application.
2. `AppRoutingModule`: Defines the application routes.

Routing

The application uses the Angular Router to manage navigation and routing between different components. Here are the defined routes:

1. Default Route: Redirects to the `login` path.
2. `/login`: Displays the `LoginComponent` for user authentication.
3. `/gallery`: Shows the `GalleryUserComponent` with a gallery of user profiles.
4. `/account/:id`: Displays the `UserPageComponent` for a specific user profile.
5. `/question`: Shows the `QuestionListComponent` with a list of questions.
6. `/question/:id`: Displays the `QuestionPageComponent` for a specific question and its answers.
7. `/new-question`: Shows the `QuestionCreatePageComponent` for creating a new question.

The `AuthGuard` is used to protect routes that require authentication. If a user is not logged in, they will be redirected to the login page.