Project Documentation
- Software Design –

Stack Overflow Application

Student: Chete Doru
Group: 30433

1. Introduction

This project has as its goal the development of a full stack web application that works much like the famous StackOverflow platform. Users use the platform as a kind of software forum, where questions are asked and answered, questions can be upvoted or downvoted and content is categorized by tags and fields of interest. Users can post questions of their own or answer questions posted by other users. The vote system also encourages users to provide good answers and have meaningful discussions.

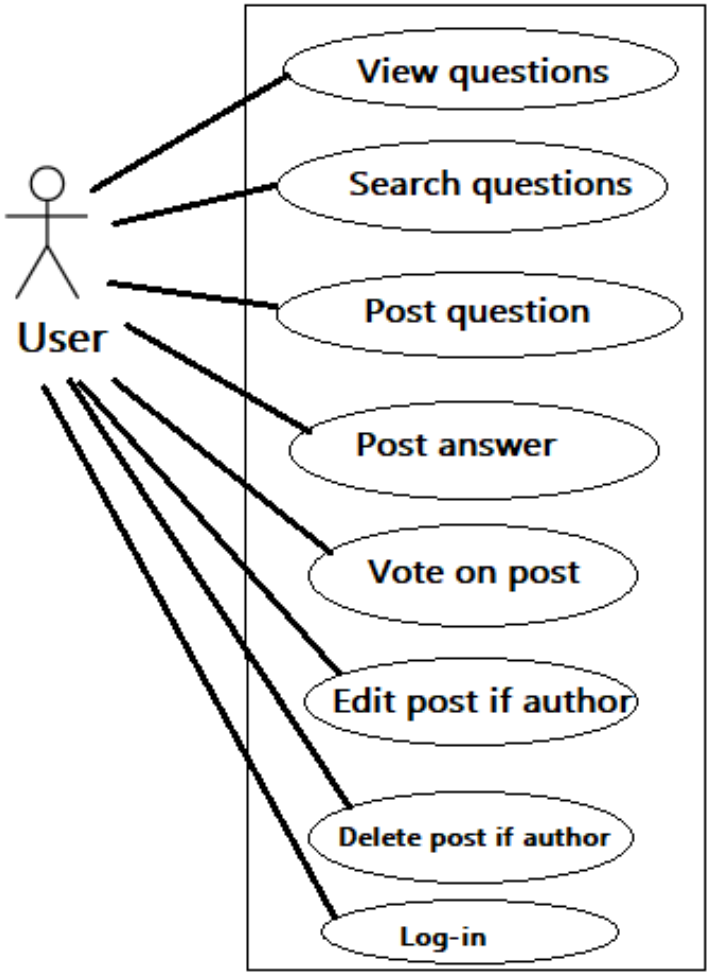
2. Technologies

For this project Java will be used together with the Spring framework for developing the backend part of the application. For developing it, IntelliJ will be used and Apache Maven will be employed for build automation. The database side of the application will be implemented using the MySQL RDMS. The project will rely on the features provided by the popular Spring framework to deliver the application efficiently.

3. Use cases

As with other software project, one must start by thinking of the use cases for the application. A use case is just a description of how users will perform various actions within the application. This outlines the system's behaviour from a user perspective.

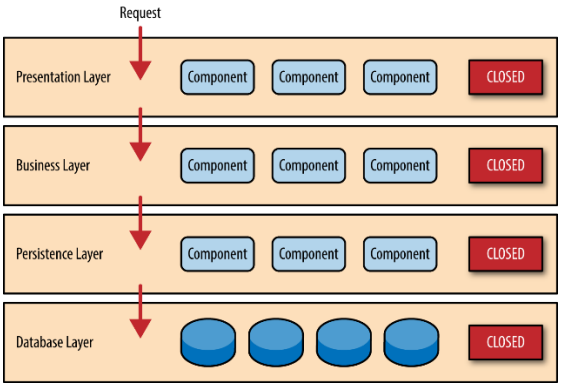
Use case diagram:



As indicated by the use case diagram, a simple user should have a couple of options of interacting with the application. First he must log-in and no actions are possible if he is not logged. Then there are actions concerning the interaction with the forum: he can add a post of his own, view posts and search through them (using filters). Later, after posting a question or answer, a user can come back and edit its text, if he is the author of that post of course. A user may even a delete one of his posts if he so wishes.

4. Architecture

For the purpose of implementing this application, a layered architecture will be used. As a starting point, we can consider the 4 layer architecture:



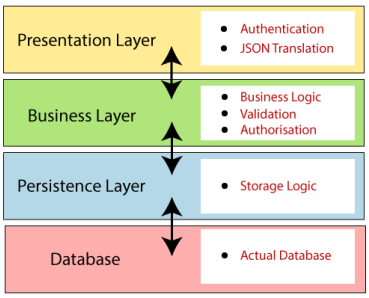
4 Layer architecture. Source: https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch01.html

As indicated in the diagram, in this architecture a request passes through the different layers to interact with the system. A key concept is the fact that each layer is closed, that is, every request moves from layer to layer. A request that comes from the business layer must first "pass" through the persistence layer before getting to the database layer.

In more specific Spring terms, we have a Presentation layer that is made up of the controllers, which interact with requests directly, a Business layer that is made up of services and a Persistence layer made of repositories and the entities.

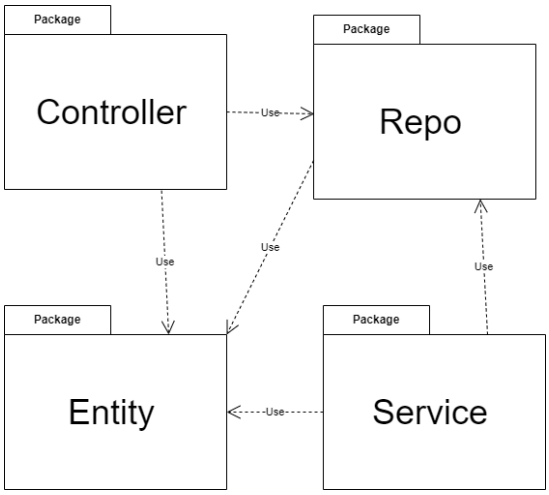
Layered architecture more specific to Spring:

(Source: https://www.javatpoint.com/spring-boot-architecture)



5. Package diagram

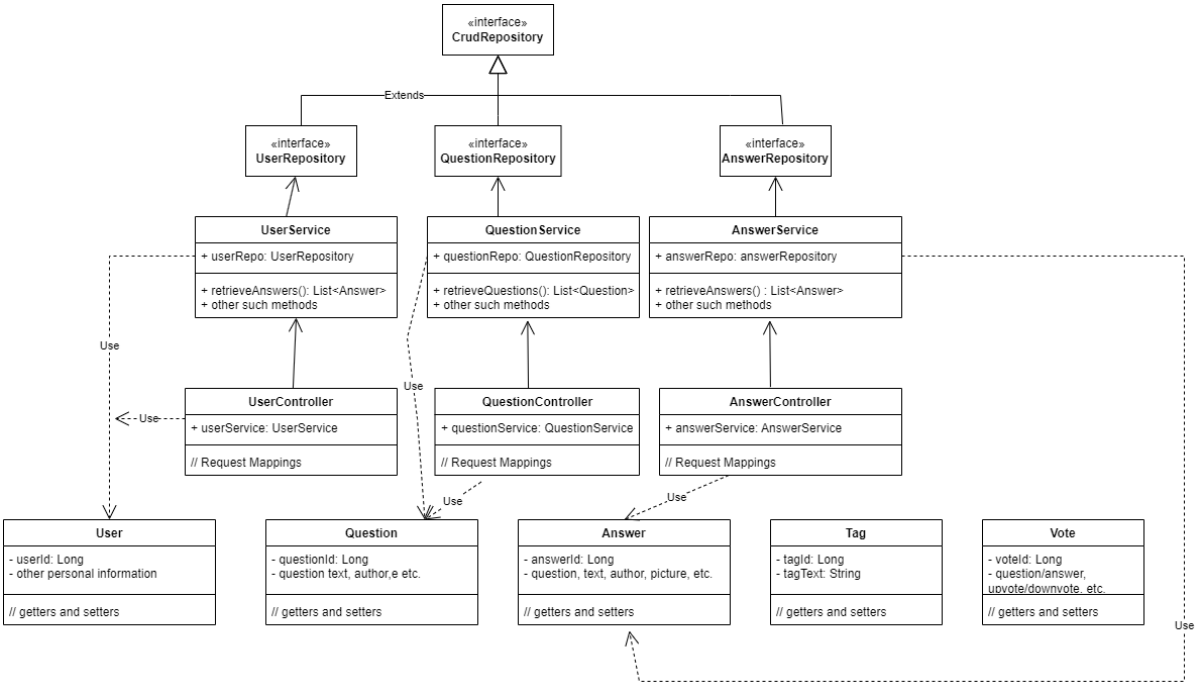
For illustrating the structure of the project, a package diagram shows the arrangement and organization of model elements within the project.



So the packages are structured according to the layered architecture the project is following.

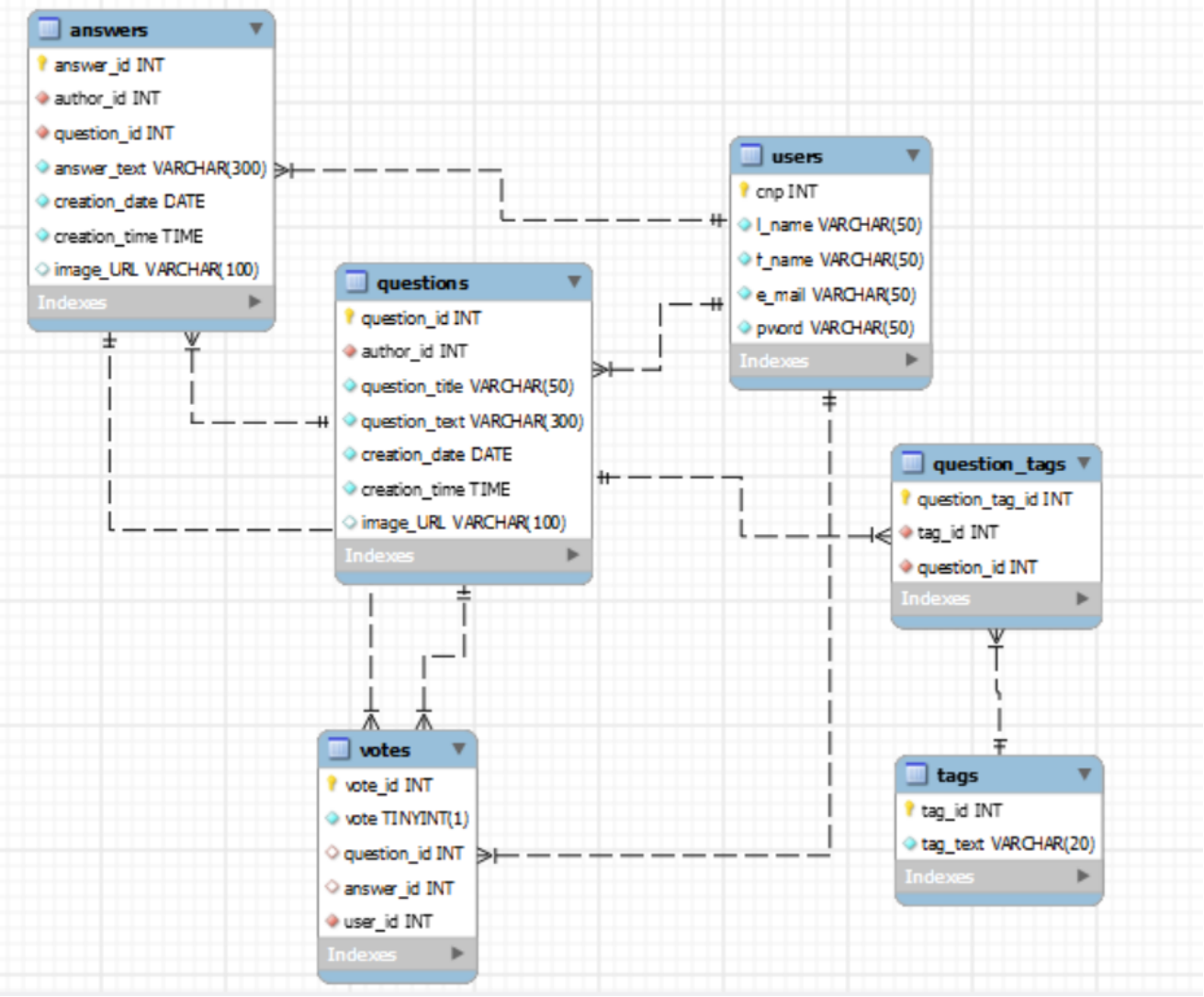
6. Class diagram

For illustrating the classes of the application, a class diagram is used, which is a representation of the relationships and source code dependencies among classes in UML.



7. Database diagram

For representing the structure of the database, we'll make use again of a diagram that illustrates the tables and their relationships.



As can be noted, in the case of tags and questions, we have a many to many relationship which must be split using a join table that is called question_tags.

8. Endpoint requests

The endpoints of the application follow the use cases defined previously. When a user takes an action, there will be a request, which has to be mapped to a method that handles it. For this purpose the app relies on the Spring framework and its annotation mechanism. Some more simple endpoints are the ones for implementing CRUD operation for the users.

1. A request that gets all users. GET request.

```
@GetMapping("/getAllUsers")
public ResponseEntity<List<User>> getAllUsers() {
    return new ResponseEntity<>(userService.retrieveUsers(), HttpStatus.OK);
}
```

Here we take no parameters from the body of the request (there are none) and we return a full list of all users from the system by calling methods from our User Service.

2. A request that creates a new user. POST request.

```
@PostMapping(value = "/createUser", consumes = "application/json", produces =
"application/json")
public ResponseEntity<User> createUser(@RequestBody User newUser) throws
ServerException {
    User user = userService.saveUser(newUser);
    if(user == null) {
        throw new ServerException("create user failed");
    } else {
        return new ResponseEntity<>(user, HttpStatus.CREATED);
    }
}
```

Here we take the whole user as a parameter (in the request body) and if the user is created successfully, a response with the details of the user will be returned.

3. A request that updates a question by its ID. PUT request.

```
public ResponseEntity<Question> updateUser(@RequestBody Question
updatedQuestion, @PathVariable("id") Long id) throws ServerException {
    Optional<Question> oldQuestion =
questionService.retrieveQuestionById(id);
    if(oldQuestion.isPresent()) {
        questionService.saveQuestion(updatedQuestion);
        return new ResponseEntity<>(updatedQuestion, HttpStatus.OK);
    }
    else { throw new ServerException("edit question by id failed"); }
}
```

Here the details of the updated question are sent in the body of the request and a path variable is used to identify the id of the question we are looking for to edit. If the Question Service manages to update it the response will be the details of the edited question and an OK status, otherwise we get a Server Exception.

4. A request for showing all questions sorted by creation date (first questions are first in the list), GET request.

```
@GetMapping("/getAllQuestionsSortedDate")
public ResponseEntity<List<Question>> getQuestionsSortedDate() {
    List<Question> sortedQuestions = questionService.retrieveQuestions();
    Collections.sort(sortedQuestions, new Comparator() {
        public int compare(Object o1, Object o2) {
            Date d1 = ((Question) o1).getDate();
            Date d2 = ((Question) o2).getDate();
            int comp = d2.compareTo(d1);
            if(comp != 0) {
                return comp;
            }
            Time t1 = ((Question) o1).getTime();
            Time t2 = ((Question) o2).getTime();
            return t2.compareTo(t1);
        }
    });
    return new ResponseEntity<>(sortedQuestions, HttpStatus.OK);
}
```

Here we take no parameters in the body of the request, and we simply get all of the questions by calling the Question

Service and ordering them high-to-low first by date and then by time. The response is the sorted list of questions.

5. Get all answers for a specific question (by its ID). GET request.

```
@GetMapping("/getAnswersByQuestionId/{id}")
public ResponseEntity<List<Answer>> getAnswersOfQuestion(@PathVariable("id")
Long id) {
    return new ResponseEntity<>(answerService.retrieveAnswerOfQuestion(id),
HttpStatus.OK);
}
```

Here we use the path variable where we find the ID of the questions whose answers we are looking for. The Answer Service will be called upon to find this. Behind the scenes, the service will filter answers based on the parameter and will return a list. Whether empty or with answers in it, it will respond with this list and an OK status.

Part II: Frontend

1. Architecture

For the purpose of this application, I used the Angular web framework in order to implement the frontend part. Angular is a TypeScript-based, free and open-source web application framework that is a Single Page Application Framework which is used for creating fast web applications.

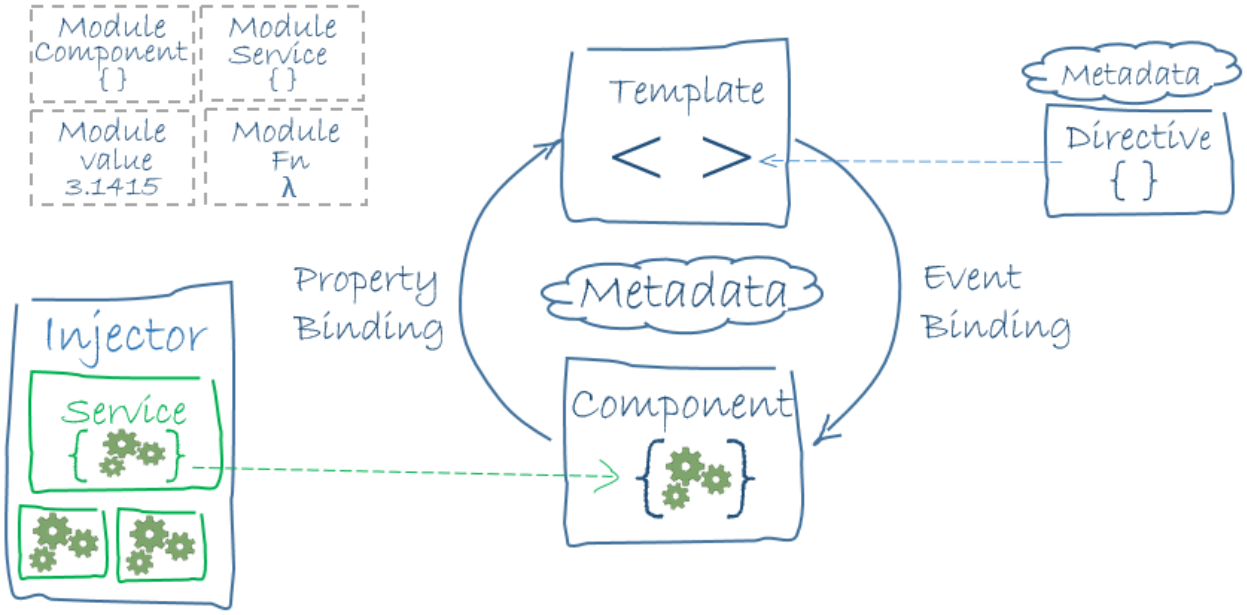


Figure 1 Architecture of an Angular app. From Wikipedia.

Starting from the diagram, the building blocks for such an application are the components, which are organized into NgModules. These modules collect related code into functional sets; an Angular application is therefore defined by a set of NgModules. The Angular components are the ones who define the views and the ones who use the so-called services. Services

provide functionalities not immediately related to views, and are injectable in other components as dependencies. Modules, components and services are classes that use decorators. The decorators mark their type and provide metadata that tells the framework how to use them.

Within our app, we have a main component called the AppComponent. We also have a main module where all of the necessary declarations and imports are specified. Using the built-in functionality of the RouterModule component we can achieve advanced navigation functionalities. Our application is rendering at all times only two components: the NavbarComponent at the top of the page, which is always visible, and the RouterOutlet component. In the backend, we have a number of entities, however for the purposes of the frontend application we can make use of just 3 entites: User, Question and Answer.

The project directory structure is as follows: we have a directory for the pages, which holds components for every type of page we need to render. Some examples are the UserPage Component, where we can view all users on the platform and the UserDetailsPage Component, where we can view the profile of a certain user. Then there are directories for all the 3 entities. We have a class where we define their fields, and in the case of the Question and User we also have a service. These services are extensively used for providing the functionalities required. One example would be injecting the QuestionService in the QuestionPage Component, in order to be able to render the questions dynamically. Using the structural directive “*ngFor” it

is very easy to manipulate the html code within QuestionPage to display our data dynamically. Directives are in fact the ones which provide program logic and binding markup is what connect your application and the DOM. So upon initialization of the QuestionPage Component, it will load all of the users within the component by making calls to the QuestionService, which handles all of that complex work and provides some good decoupling.

The same process is used for displaying all of the users, since these two use cases are quite similar: list of users and list of questions. In the case of the detailed page for one question we must also display all of the answers of a question. This is also done with the help of directives, bindings and the UserService.

2. Routing

Routing is part of the Angular framework as discussed before, and within our application we make use of it on every page. The Angular Router module already provides a service which lets us define paths for navigation. It is modeled after familiar conventions: 1) Enter a URL in the address bar and it goes to corresponding page. 2) Click links on the page and the browser navigates to a new page. 3) Click the browser's back and forward buttons and the browser navigates backward and forward through the history of page you've seen.

We can therefore "associate" a path with a certain component. This is done in our project in the main app.module. Specific attention must be paid to the order of the path

definitions. The path will match with the first component it can. This means in practice that if we set a path such as /user before /user/details we will never be able to match to the second one. The longer the path, the sooner it needs to be put in the list of routes.

We can also make use of data within a path for handling methods. When we want to access the details page of a specific user or question we will go to a path that looks like this: /users/details/1, where the trailing number is actually that user's id. In the TypeScript code of the component, we can actually get this as a parameter, and use it to make calls to the service. This is how the calls to get a specific user can be made to the UserService.

The paths for our application are in order, as follows (from app.module.ts):

```
     const routes: Routes = [ { path: '', component: HomePageComponent
},

     { path: 'users/details/:id', component :
UserDetailsPageComponent},

      { path: 'users/details', component : UserDetailsPageComponent },

      { path: 'users', component: UserPageComponent }, { path:
'questions/details/:id', component: QuestionDetailsPageComponent },

      { path: 'questions/details', component:
QuestionDetailsPageComponent },

      { path: 'questions', component: QuestionPageComponent },

      { path: 'askSomething', component: AskPageComponent },

      { path: 'login', component: LoginPageComponent }, { path:
'register', component: RegistrationPageComponent } ];
```

Code snippet for taking the id of a specific question from the path we are currently on (from QuestionDetailsPage component):

```
getQuestion(): void {
      const id = +this.route.snapshot.paramMap.get('id');
      this.questionService.getQuestion(id).subscribe(question =>
      (this.question = question) );
}
```