FEDDIT

Ugly by choice

Curreri Dario
Di Fina Domenico
Domingo Emanuele
Gristina Antonino Salvatore
Vanspauwen Arnaud

Index

1. Task repartition [3-4]
2. Structure of the database [5-13]
3. Management of authentication system and roles [14-21]
4. Strengths and weaknesses of the project [22, 27]
5. brief demonstration of the web application [**live**]

Task repartition

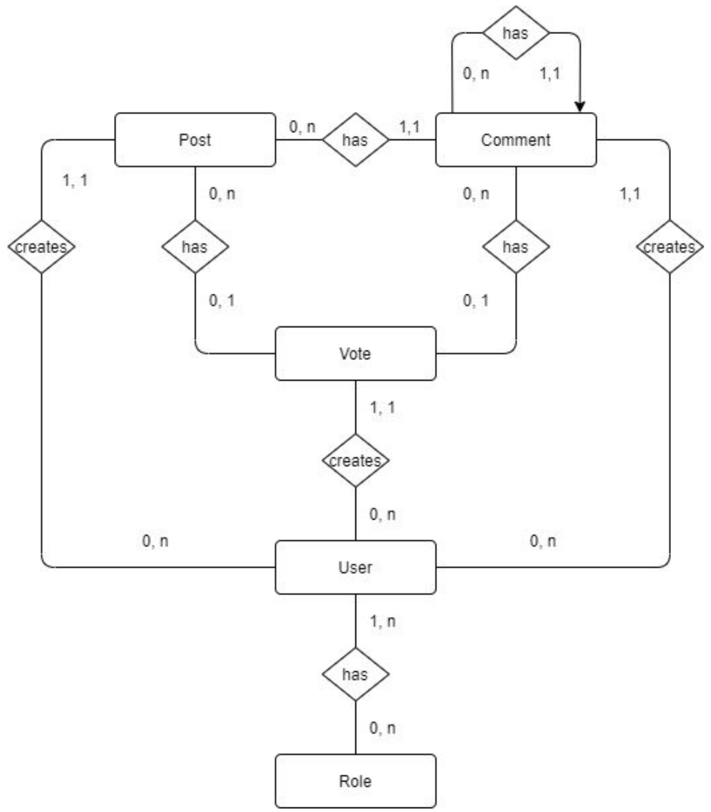
Tasks

making a precise division of roles is complex as we all helped each other in different parts of the project, but we can summarize in:

- Dario Curreri: back-end: database model, log, recursive comments exploration
- Domenico Di Fina: back-end: vote system, post management
- Emanuele Domingo: front-end, user and role management
- Salvatore Gristina: front-end, error handling, documentation
- Arnaud Vanspauwen: general bug fix, testing, documentation

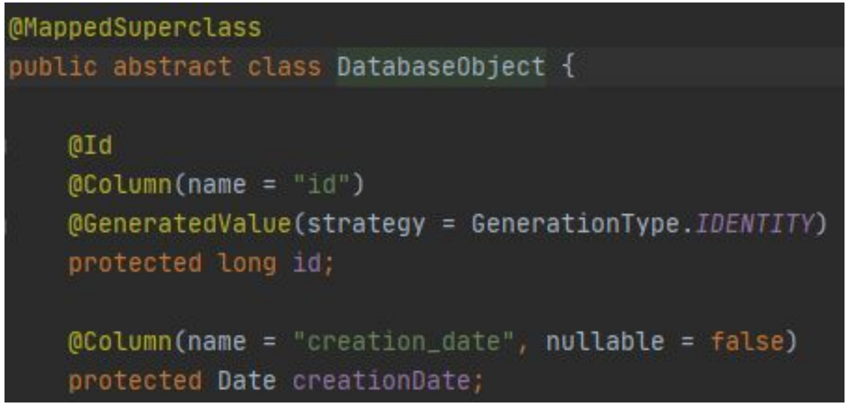
Database structure

ER schema


has
0, n
1,1
Post
0, n
has
1,1
Comment
1, 1
0, n
0, n
1,1
creates
has
has
creates
0, 1
0, 1
Vote
1, 1
creates
0, n
0, n
0, n
User
1, n
has
0, n
Role

DatabaseObject class

All the objects that are stored in the database inherit this abstract class, which contains the attributes *id* and *creationDate*

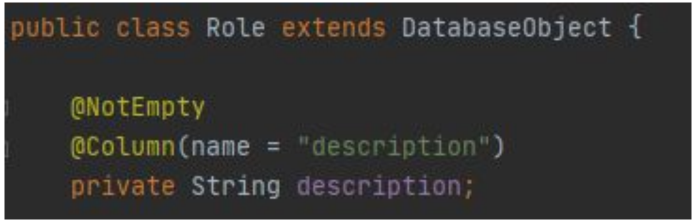


```
@MappedSuperclass
public abstract class DatabaseObject {

    @Id
    @Column(name = "id")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    protected long id;

    @Column(name = "creation_date", nullable = false)
    protected Date creationDate;
```

Role class

This class is used to assign the roles to users.

In our implementation there are just 2 roles available: *ADMIN* and *USER*.



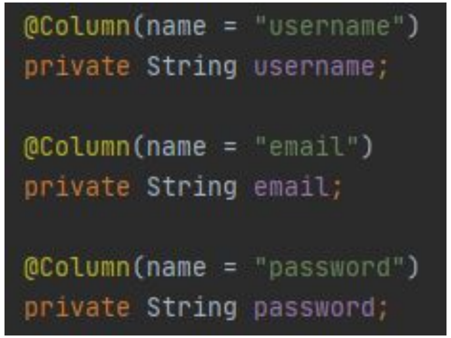
```
public class Role extends DatabaseObject {

    @NotEmpty
    @Column(name = "description")
    private String description;
```

Note: an admin has both roles

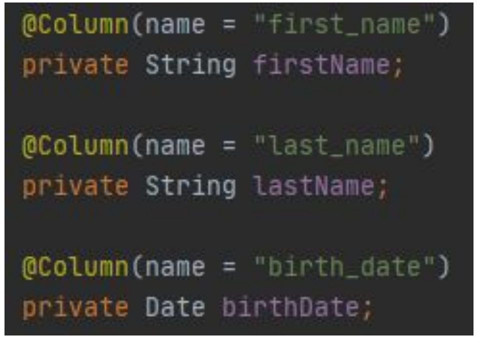
User class

This is the class used to represent an user. Since user data have to be stored in the database, it extends and concretises the *DatabaseObjec*t class.

This class contains all the attributes relative to the user, that is:



```java
@Column(name = "username")
private String username;

@Column(name = "email")
private String email;

@Column(name = "password")
private String password;
```

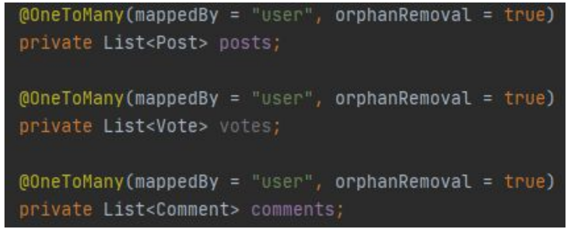


```java
@Column(name = "first_name")
private String firstName;

@Column(name = "last_name")
private String lastName;

@Column(name = "birth_date")
private Date birthDate;
```

...

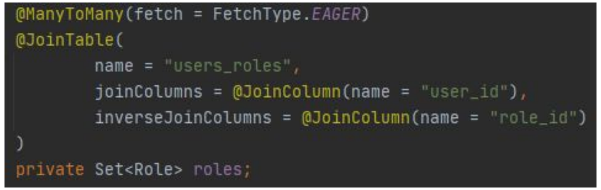
User class

and all the references to the *ForumObject* created by it:



```
@OneToMany(mappedBy = "user", orphanRemoval = true)
private List<Post> posts;

@OneToMany(mappedBy = "user", orphanRemoval = true)
private List<Vote> votes;

@OneToMany(mappedBy = "user", orphanRemoval = true)
private List<Comment> comments;
```

And a set of *Role* (which contains just the role *USER*):

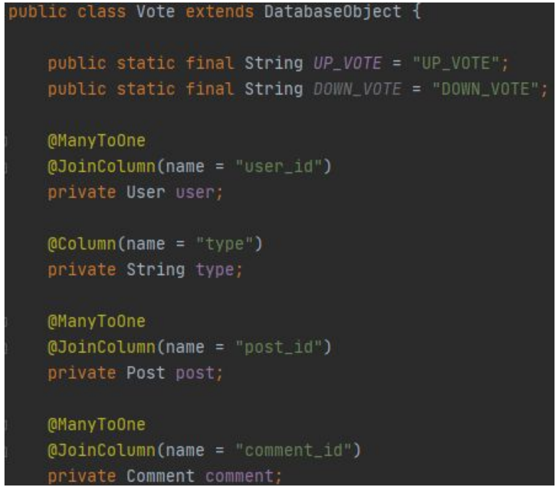


```
@ManyToMany(fetch = FetchType.EAGER)
@JoinTable(
        name = "users_roles",
        joinColumns = @JoinColumn(name = "user_id"),
        inverseJoinColumns = @JoinColumn(name = "role_id")
)
private Set<Role> roles;
```

Vote class

This class, that is a *DatabaseObject*, represents the concept of a Vote, which:

- is assigned by a *User*,
- has a type (*UP* or *DOWN*),
- refers to a *Comment* or to a *Post*



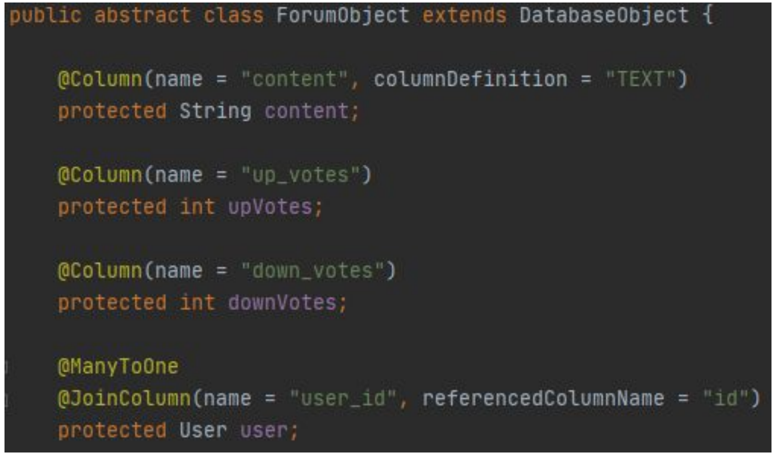
```
public class Vote extends DatabaseObject {

    public static final String UP_VOTE = "UP_VOTE";
    public static final String DOWN_VOTE = "DOWN_VOTE";

    @ManyToOne
    @JoinColumn(name = "user_id")
    private User user;

    @Column(name = "type")
    private String type;

    @ManyToOne
    @JoinColumn(name = "post_id")
    private Post post;

    @ManyToOne
    @JoinColumn(name = "comment_id")
    private Comment comment;
```

ForumObject class

The class abstracts the concept of a forum item. A forum item surely has:

- a content, that is the text
- up votes
- down votes

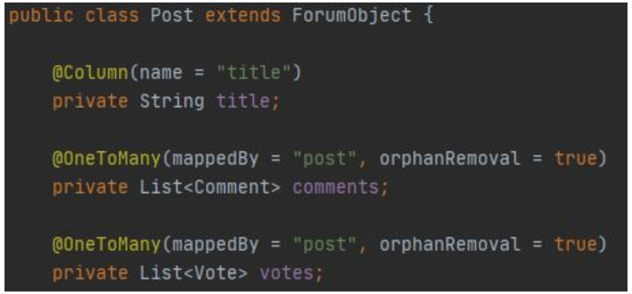
and it is created by a *User*



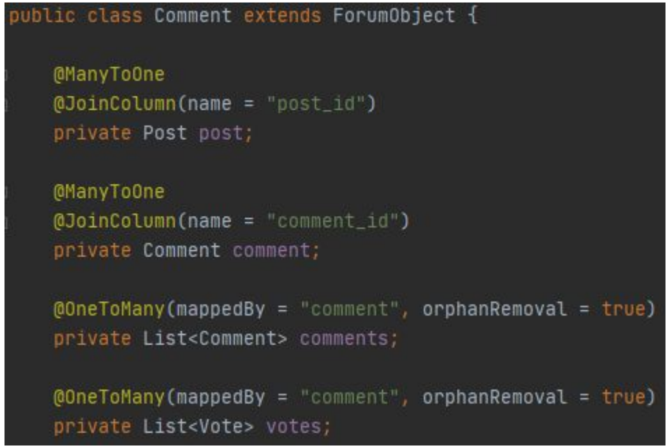
```
public abstract class ForumObject extends DatabaseObject {

    @Column(name = "content", columnDefinition = "TEXT")
    protected String content;

    @Column(name = "up_votes")
    protected int upVotes;

    @Column(name = "down_votes")
    protected int downVotes;

    @ManyToOne
    @JoinColumn(name = "user_id", referencedColumnName = "id")
    protected User user;
```

Post and Comment class

The *Post* and *Comment* classes concretise the concepts of a forum post and forum comment:



```
public class Post extends ForumObject {

    @Column(name = "title")
    private String title;

    @OneToMany(mappedBy = "post", orphanRemoval = true)
    private List<Comment> comments;

    @OneToMany(mappedBy = "post", orphanRemoval = true)
    private List<Vote> votes;
```

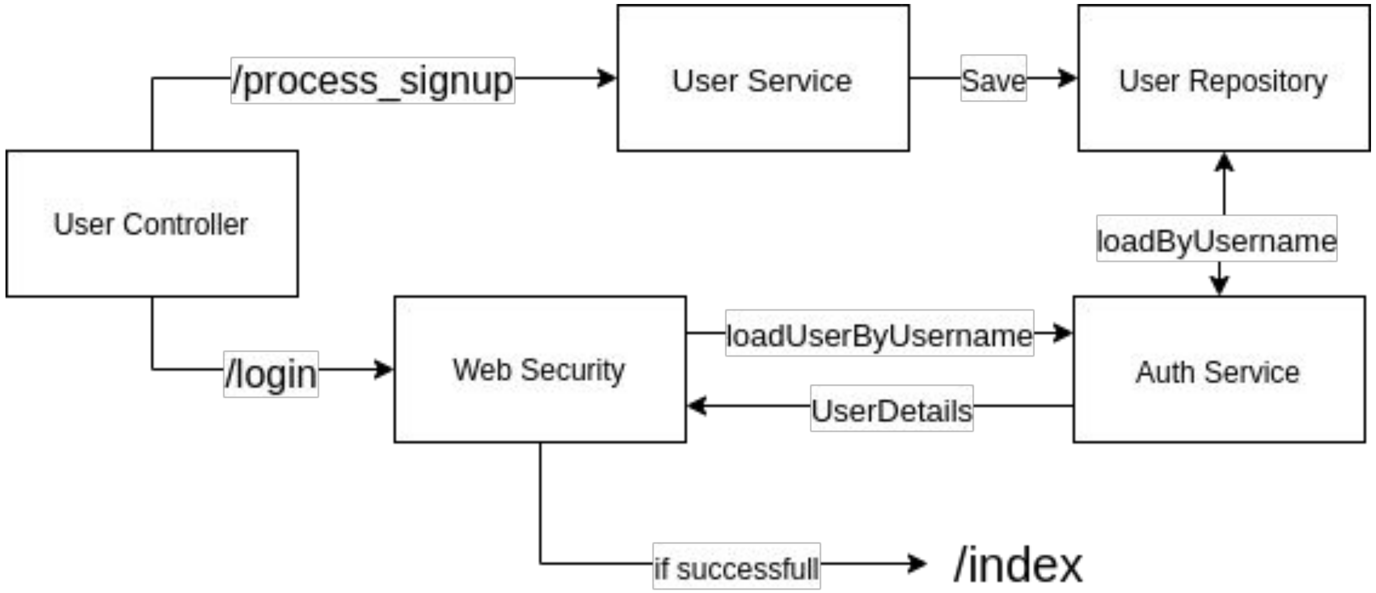


```
public class Comment extends ForumObject {

    @ManyToOne
    @JoinColumn(name = "post_id")
    private Post post;

    @ManyToOne
    @JoinColumn(name = "comment_id")
    private Comment comment;

    @OneToMany(mappedBy = "comment", orphanRemoval = true)
    private List<Comment> comments;

    @OneToMany(mappedBy = "comment", orphanRemoval = true)
    private List<Vote> votes;
```

A comment can refer to a *Post* or to another *Comment* (exclusively)

Authentication & Roles Management

Authentication Workflow

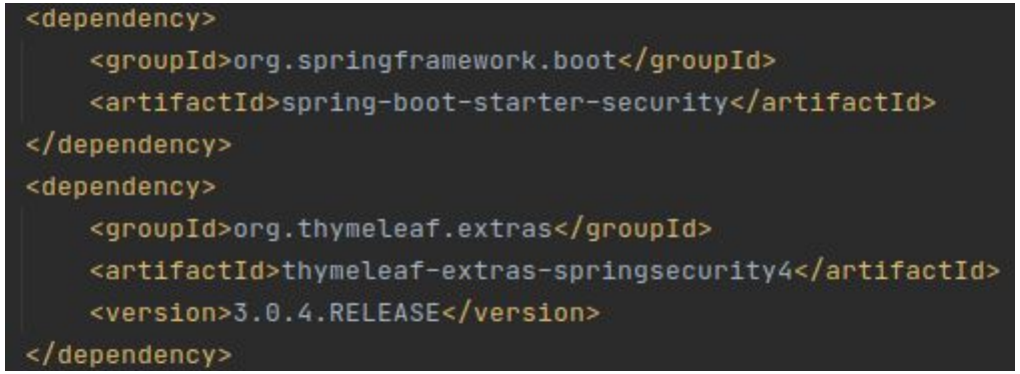


Dependencies

if we want to prevent unauthorized users from viewing some pages we need to add a barrier that forces the visitor to sign in before they can see that pages.

We do that by configuring Spring Security in the application.

First of all, we need to include it into the dependencies



```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
    <groupId>org.thymeleaf.extras</groupId>
    <artifactId>thymeleaf-extras-springsecurity4</artifactId>
    <version>3.0.4.RELEASE</version>
</dependency>
```

Controller

To manage users we need a proper controller. In our application is named ***UserController***.

The role of this controller is to handle the requests to the home page, login page, sign up page etc.

Some operations need more handlers. For example, the sign up has:

- `@GetMapping("/signup") public String showSignUpForm(...)`
- `@PostMapping("/process_signup") public ModelAndView processSignUp(...)`

The first get mapping is for showing the registration form, se second post mapping is to handle the request and process it through the user service

UserDetails

Next, in order to implement authentication feature, we need to create a class of subtype UserDetails (defined by Spring Security) to represent an authentication user.

This step is fundamental because Spring Security will invoke methods in this class during the authentication process.

In our project this class is ***FedditUserDetails***. It's basically a wrapper for the User class with some methods used by spring security.

An important method is *getAuthorities()* that provides the authorities depending on the role of the user.

UserDetailsService

To tell Spring Security how to look up the user information, we need to code a class that implements the UserDetailsService interface. In our case that class is ***AuthService***.

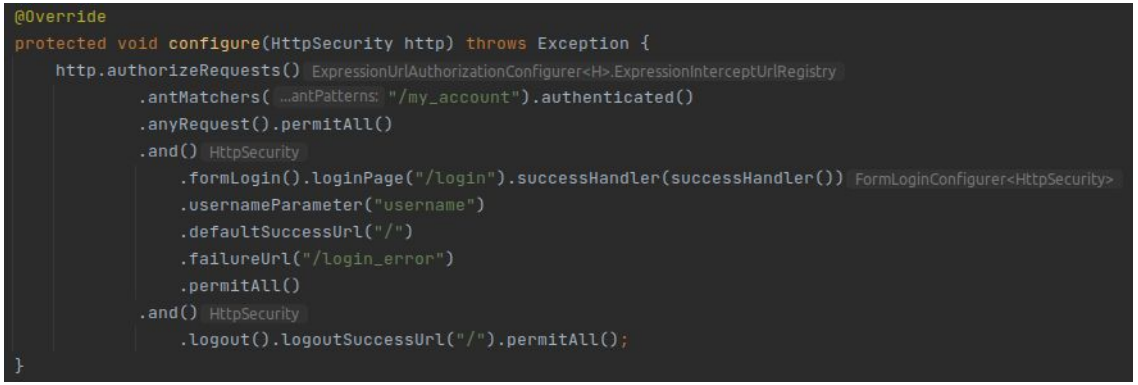
Spring Security will invoke the *loadUserByUsername()* method to authenticate the user, and if successful, a new object of type ***FedditUserDetails*** object is created to represent the authenticated user.

Of course, in the repository, we need to add a method to get an user given the username.

SpringSecurity

Spring Boot automatically secures all HTTP endpoints with “basic” authentication. But we can customize the security settings by configuring a java class annotated with @EnableWebSecurity and it also extends ***WebSecurityConfigurerAdapter***.

This is the main part of the security workflow, by redefining the *configure()* method we can set our configuration. From our ***SecurityConfig*** class:



```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests() ExpressionUrlAuthorizationConfigurer<H>.ExpressionInterceptUrlRegistry
            .antMatchers( ...antPatterns: "/my_account").authenticated()
            .anyRequest().permitAll()
            .and() HttpSecurity
                .formLogin().loginPage("/login").successHandler(successHandler()) FormLoginConfigurer<HttpSecurity>
                .usernameParameter("username")
                .defaultSuccessUrl("/")
                .failureUrl("/login_error")
                .permitAll()
            .and() HttpSecurity
                .logout().logoutSuccessUrl("/").permitAll();
}
```

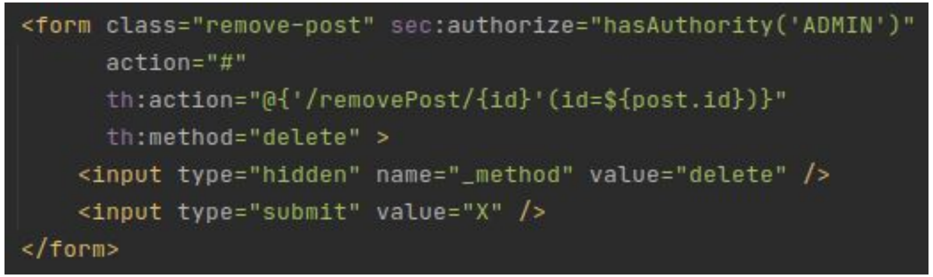
Roles

We defined two roles: USER and ADMIN.

Every user in the database will have at least one role [*slides from 5 to 7*], except for the admin that has two roles (an admin is also an user).

The spring framework knows at runtime the role (and the privilege) of the authenticated user.

We can use the role to dynamically render the front end using thymeleaf. For example:



```
<form class="remove-post" sec:authorize="hasAuthority('ADMIN')"
      action="#"
      th:action="@{'/removePost/{id}'(id=${post.id})}"
      th:method="delete" >
    <input type="hidden" name="_method" value="delete" />
    <input type="submit" value="X" />
</form>
```

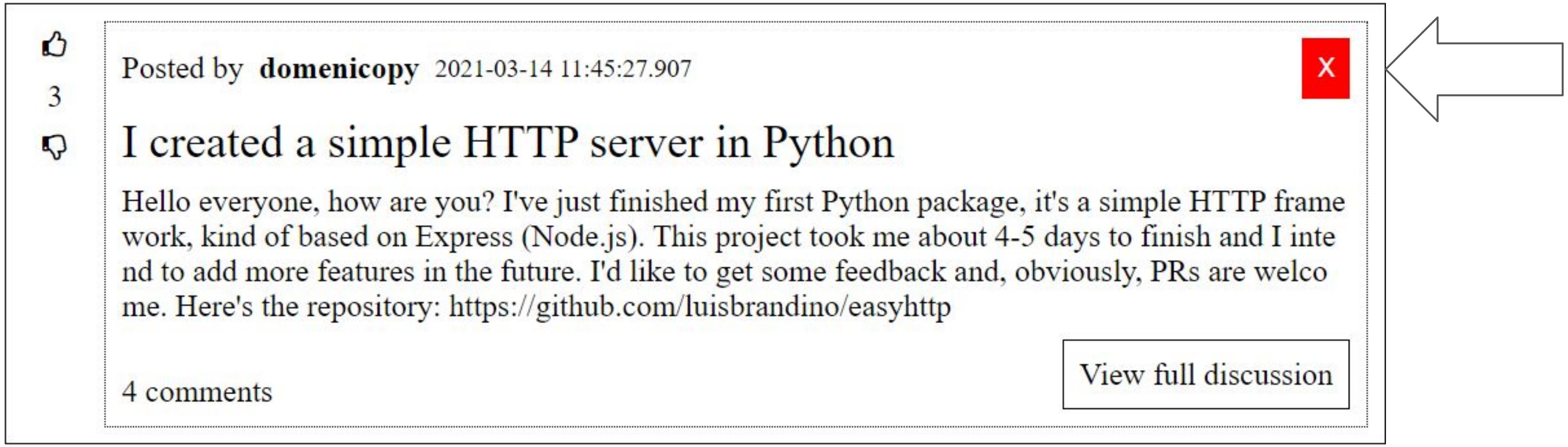
Strengths and weaknesses

Strengths - Roles

One of the strengths of our project is the division of roles.

Our project dynamically checks which user is logged in and consequently adds or remove features.

Here an example:



Posted by **domenicopy** 2021-03-14 11:45:27.907

3

I created a simple HTTP server in Python

Hello everyone, how are you? I've just finished my first Python package, it's a simple HTTP frame work, kind of based on Express (Node.js). This project took me about 4-5 days to finish and I inte nd to add more features in the future. I'd like to get some feedback and, obviously, PRs are welco me. Here's the repository: https://github.com/luisbrandino/easyhttp

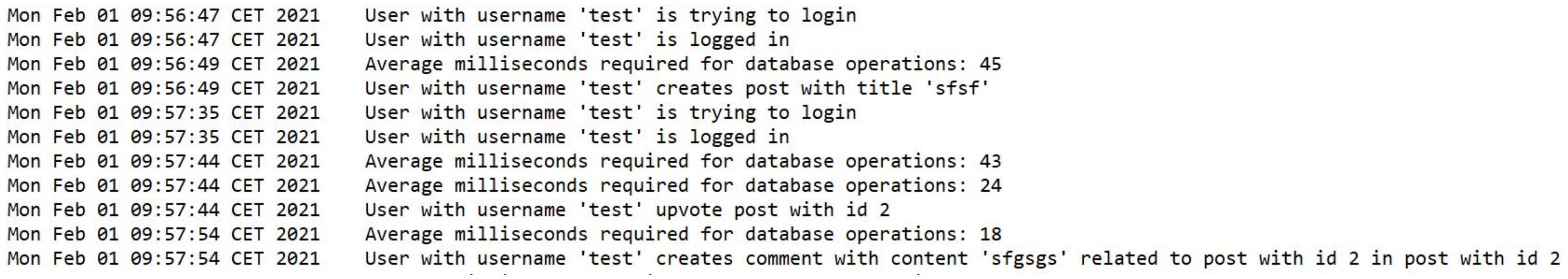
4 comments

Strengths - Log

Another strength of our project is the log system.

In particular, we decided to keep a text log of every action occurs in the system.

The main purpose of this system is to keep track of any comments or posts with offensive content even if these contents have been deleted.



```
Mon Feb 01 09:56:47 CET 2021    User with username 'test' is trying to login
Mon Feb 01 09:56:47 CET 2021    User with username 'test' is logged in
Mon Feb 01 09:56:49 CET 2021    Average milliseconds required for database operations: 45
Mon Feb 01 09:56:49 CET 2021    User with username 'test' creates post with title 'sfsf'
Mon Feb 01 09:57:35 CET 2021    User with username 'test' is trying to login
Mon Feb 01 09:57:35 CET 2021    User with username 'test' is logged in
Mon Feb 01 09:57:44 CET 2021    Average milliseconds required for database operations: 43
Mon Feb 01 09:57:44 CET 2021    Average milliseconds required for database operations: 24
Mon Feb 01 09:57:44 CET 2021    User with username 'test' upvote post with id 2
Mon Feb 01 09:57:54 CET 2021    Average milliseconds required for database operations: 18
Mon Feb 01 09:57:54 CET 2021    User with username 'test' creates comment with content 'sfgsgs' related to post with id 2 in post with id 2
```

Weaknesses - User Interface

One of the weaknesses of our project is the user interface.

It was our choice to design it ugly, also because it wasn’t the goal of this project.

In particular, we decided to keep it simple, avoiding to spend too much time on the front-end development and to focus on the back-end part.

Weaknesses - Spam and Bad Words Filter

Another weakness of our project is the missing spam or bad words filter.

We didn’t think to implement a system that can censor this kind of words.

The simplest solution is to have a dictionary of bad words and a way to substitute them with a symbol (ex. *). Further, and better, solutions of course implies artificial intelligence and text recognition.

With regard to spam we could implement a way to block repeated contents from a user, or we could add some temporary ban after spam identification.