Université Gustave Eiffel

Rapport Projet de JEE
UGERevue

- Menaa Mathis (Scrum master)
- Ramaroson Rakotomihamina Johan (Chef de projet)
- Tellier Quentin (Product owner)
- Regueme Yohann
- Gaudet Clément

Master 2 - Logiciels et ingénierie des données

I. Présentation du projet

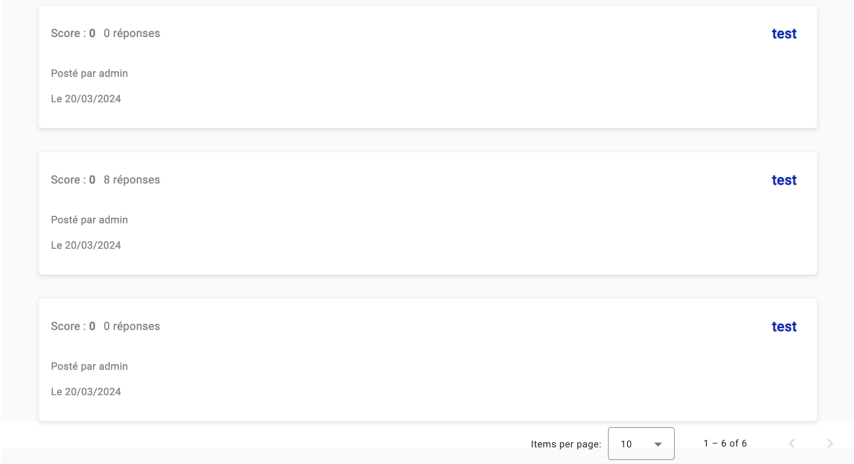
UGERevue est une application permettant à un utilisateur d'écrire des revues de codes sur le code des différents utilisateurs, la soumission de ce code est appelée une "Question". L'idée du site est qu'un utilisateur peut soumettre son code JAVA aux utilisateurs du site pour pouvoir le review, en accédant à un topic visualisant le dit-codes, un titre, une description. Le site permet à ce que les utilisateurs puissent en suivre d'autres, ce qui permet de voir en priorité les questions de ces personnes.
La soumission de code peut s'accompagner d'un fichier de test unitaire associé au dit-code, permettant de montrer aux reviewers les résultats du code sur un jeu de test concret.
Un utilisateur peut "upvote" ou "downvote" une question ou une revue afin de lui donner de la popularité défini par un score.

II. Etat du projet, difficultées et fiertées

Le projet est fonctionnel tel que décrit dans la partie présentation du projet (cf, partie I).

Un des points les plus difficiles a été la mise en place de la partie micro service mélangé avec l'installation de l'environnement Docker pour le sécurisé. Pour le lancement des tests il a fallu build le jar du micro service avec le plugin maven-shade qui permet de packager les librairies dans le jar, avant de trouver cette solution nous sommes passé par maintes et maintes casse-têtes car notre Docker n'avait pas les librairies requises pour compiler les tests. Nous croyions qu'il fallait spécifier un à un les dépendances lors de la création du ClassLoader mais finalement la solution n'était pas acceptable car trop spécifique. Cette partie nous a rendu fiers car nous avons utilisé des technologies dont nous n'avions pas l'habitude.

Nous avons aussi mis en place un composant avec Angular qui permet d'afficher une liste de questions qu'on lui donne avec une pagination, chaque question est affichée sous la forme d'une carte cliquable qui conduit à sa page de détails avec ses principales informations.



La gestion du projet en elle-même sur le long terme fait partie de notre fierté, accompagnée du cours de gestion de projet, qui nous a notamment appris à structurer notre organisation tout en se remettant en question au fur et à mesure d'une manière agile. Tout en étant chacun en entreprise une semaine sur deux. C'est une partie non-négligeable qui nous a permis de rendre un projet non pas parfait mais dont nous sommes fiers devant les défis rencontrés. Nous avons surmonté cette difficulté au fur et à mesure du développement via l'utilisation des différents principes vus au cours de Gestion de projet, notamment à la pratique de méthode agile et d'outils associés. Nous avons donc désigné 3 acteurs parmi les développeurs : Un scrum master (Mathis), un product owner (Quentin) et un chef de projet (Johan).

Le scrum master a permis de cadrer la mise en place des différents sprints et a été fort de propositions pour améliorer l'organisation du déroulé du projet en dehors du plan technique.

Le product owner est le décisionnaire en ce qui concerne l'UI et UX de l'application, ainsi lorsque toute questions intervenait sur la manière dont les écrans devraient être conceptionné c'est à lui que revenait la décision finale, ce qui permettait aux autres membres d'avoir un poid en moins à gérer de ce côté là tout en permettant la prise en compte des propositions de chacun.

Le chef de projet quant à lui agissait après l'intervention des deux acteurs précédents. C'était à lui de traduire les objectifs du sprint en tâches techniques correspondantes sous forme de tickets via l'outil Gitlab Kanban. Le chef de projet est également le décisionnaire au niveau de la fusion des branches, à l'aide de l'auteur du ticket il s'assurera de régler les conflits de branches lors de la fusion des merges requests.

III. Organisation du projet

Quelles tâches ont été faites par chaque membre ?

Johan (22%)

- Gestion de tous les merge requests et conflits git
- Direction des réunions techniques
- Répartition de chaque tâches (cf. onglet “issues” sur le gitlab du projet)
- Configuration initiale du projet
- Gestion de la structure du projet en multi-module
- Mise en relation du client lourd et du serveur backend pour la plupart des features
- Mise en place de certains tests d'intégrations
- Développement d'une bonne partie du front lourd et léger
- Correction et gestion des injections javascript depuis le front léger
- Développement de l'algorithme de recherche des questions à la google
- Peer programing, consulting et refactor sur certaines features

Le pourcentage de travail supplémentaire de Johan par rapport aux autres membres du groupe est pour nous le pourcentage minimum attendu d'un chef de projet. Dans notre cas, la majorité de ses commits proviennent de la gestion des merges, des diverses corrections des conflits entre les travaux des différents membres du groupe. Le plus gros de son travail a été de permettre à ce que l'environnement de travail des membres du groupe soit toujours cohérent au fur et à mesure de l'avancée du projet.

Yohann (19.5%)

- Ajout de certains Services côté Angular
- Mise en place du controller sur les reviews
- Mise en place de tests unitaires et d'intégration
- Mise en place complète du micro service de lancement de tests :
    - ClassLoader
    - Launcher Junit5
    - Partie sécurité (Timeouts etc…)
    - Controller, Service, Exceptions
    - Tests unitaires
    - Conteneur Docker
- Développement d'une petite partie du front Light
- Redirection en cas de déconnexion sur le front lourd
- Aide pour concevoir tous les algorithmes
- Audit sécurité / fonctionnel

Mathis (19.5%)

- Développement de la partie chiffrage des mots de passes utilisateurs
- Controller – Service - model User – DTO user pour enregistrer/logger un utilisateur en BDD
- Composant Angular "Communauté"
- Relier la back et le front pour la partie Follow (en lien avec le composant créé précédemment)
- Travail sur la partie recommandation de recherche "intelligente" d'une revue par rapport à une question
- Implémentation d'une grande partie des tests unitaires
- Configuration du projet à la fin des devs (Script shell de build, variables d'environnements…)
- Implémentation du controller pour les votes
- Participation aux refactors
- Aide sur l'audit du back

Clément (19.5%)

- Mise en place de presque toute la sécurité initiale (sauf chiffrage mot de passe)
- Configuration des sessions
- Configuration du CSRF, notamment fonctionnement avec front Angular
- Configuration de l'autorisation (implémentation des rôles, annotations pour contrôler)
- Faire fonctionner la validation des DTO côté Spring
- Travail sur le modèle de données (mise en place des follows notamment sur toutes les couches: repository, service, controller)
- Plusieurs composants sur le front Angular (affichage d'une question et des revues, coloration syntaxique du code, page de profil, changement de mot de passe)
- Initialisation du front light (Spring MVC): design initial, structuration des templates Thymeleaf (fragments, pages…), splits des contrôleurs entre les côtés REST et MVC
- Configuration prod pour le back et le front Angular
- Divers ajouts et fix de bugs à tous les niveaux

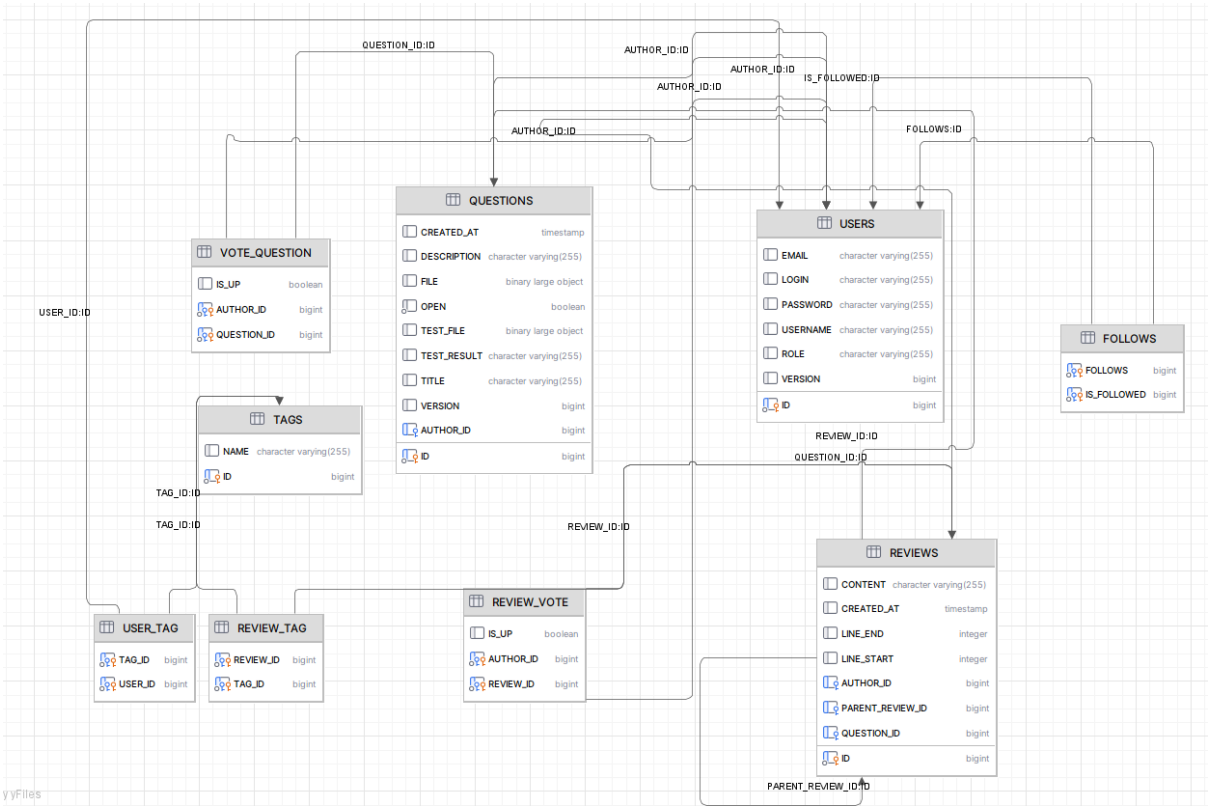
Quentin (19.5%)

- Initialisation du front avec Angular
- Décisionnaire sur les maquettes du site
- Composant angular générique qui prend une liste de questions pour les afficher
- Onglet qui permet de voir les détails d'une review et qui affiche la liste de toutes ses reviews et les boutons pour ajouter, supprimer, …
- Affichage des recommandations de review quand on en crée une nouvelle
- Implémentation des services pour les questions et les reviews
- Création de tests unitaires pour certaines méthodes de ces services
- Initialisation de la base des entités hibernate, les relations et les repository
- Mise en place de l'algorithme pour afficher en priorité les questions des follows, des follows des follows et ainsi de suite
- Mise en place de la gestion de concurrence sur l'ensemble des opérations update (avec les servicesWithFailure)

Tout le monde a aussi touché à toutes les parties du projet sur le tout dernier Rush afin de corriger les bugs existants et ajouter les toutes dernières fonctionnalités.

IV. Mapping et requête en BD

Voici le diagramme de notre base de donnée précis généré par intelliJ :



Pour le mapping voici quelques extraits de code de nos entités :

```
@Table(name = "users")
public final class User {
    …
    @OneToMany(cascade = CascadeType.ALL, mappedBy = "author")
    private List<Question> questions = new ArrayList<>();
    @OneToMany(cascade = CascadeType.ALL, mappedBy = "author")
    private List<Review> reviews = new ArrayList<>();
…
}
```

Dans ce cas, nous avons utilisé cascadeType. ALL, car l'utilisateur est au centre des autres entités, pour les opérations d'update, persistance et suppression, on veut qu'ils se propagent au sein des autres entités.

Par défaut nous avons fait le choix de garder le fetchType de l'ensemble des relations à LAZY afin de garantir qu'on ne récupère jamais l'ensemble des relations pour chaque requête SELECT.

```
@Entity
@Table(name = "reviews")
public class Review {

    @ManyToOne(fetch = FetchType.LAZY)
    private Question question;
    @ManyToOne(fetch = FetchType.LAZY)
    private User author;
    @OneToMany(cascade = CascadeType.REMOVE, mappedBy =
"parentReview")
    private List<Review> reviews = new ArrayList<>();

    @ManyToMany(cascade = CascadeType.DETACH)
    @JoinTable(
        name = "review_tag",
        joinColumns = @JoinColumn(name = "review_id"),
        inverseJoinColumns = @JoinColumn(name = "tag_id")
    )
    private Set<Tag> tagsList = new HashSet<>();

}
```

```
@Table(name = "questions")
public class Question {
    @Lob
    @Column(columnDefinition = "BLOB")
    private byte[] file;

    @Embedded
    private TestKit testKit;

    @ManyToOne(fetch = FetchType.LAZY)
    private User author;
    @OneToMany(cascade = CascadeType.REMOVE, mappedBy = "question")
    private List<Review> reviews = new ArrayList<>();
}
```

Pour l'entité Question, le choix a été fait d'utiliser CascadeType REMOVE pour garantir que quand une question est supprimée, toutes ses reviews le seront toutes également.

Pour précision, nous avons délibérément fait le choix de permettre la suppression de reviews de la base de donnée si l'auteur de la question ou l'admin vient à supprimer la question auquelle elle était reliée.

Concernant les **votes**, nous n'avions pas eu à gérer la concurrence via des optimistic locks ou des pessimistic locks car pour notre implémentation, comme vous pouvez le voir dans notre diagramme, nous avons deux tables dédiées à cette représentation : question_vote et vote_review. Comme représenté ci-dessous :

QuestionVote.java

```java
@Entity(name = "vote_question")
@Table(
        name = "vote_question",
        uniqueConstraints = {@UniqueConstraint(columnNames = {"author_id", "question_id"})}
)
public class QuestionVote {
    @EmbeddedId
    private QuestionVoteId questionVoteId;
    @Column(name = "is_up")
    private boolean isUp;


    public User getAuthorId() {
        return questionVoteId.getAuthor();
    }

    public Question getQuestionId() {
        return questionVoteId.getQuestion();
    }

    public void setQuestionVoteId(QuestionVoteId questionVoteId) {
        this.questionVoteId = questionVoteId;
    }

    public boolean isUp() {
        return isUp;
    }

    public void setUp(boolean up) {
        isUp = up;
    }
    public static QuestionVote upvote(User author, Question question) {
        return vote(author, question, true);
    }
    public static QuestionVote downVote(User author, Question question) {
        return vote(author, question, false);
    }
    private static QuestionVote vote(User author, Question question, boolean isUp){
        var questionVote = new QuestionVote();
        questionVote.setQuestionVoteId(new QuestionVoteId(author, question));
        questionVote.setUp(isUp);
        return questionVote;
    }
}
```

Avec pour ID cette représentation :

QuestionVoteId.java

```java
@Embeddable
public class QuestionVoteId implements Serializable {
    @ManyToOne
    @JoinColumn(name = "author_id")
    private User author;

    @ManyToOne
    @JoinColumn(name = "question_id")
    private Question question;
    // ...
}
```

Ainsi, les votes pour une question est représenté par le nombre de lignes dans la table question_vote concernant l'utilisateur et la question. Il n'y a donc pas d'opération de mise à jour, uniquement des opérations d'ajout et de suppression.

Comment avez-vous factorisé le code entre les commentaires généraux et les commentaires sur le code ?

Pour factoriser cette partie nous avons créer un @Embed entity comme ceci :

CodePart.java

```java
@Embeddable
public class CodePart {
    private int lineStart;
    private int lineEnd;
    // ...
}
```

Qui représente la ligne de début d'un commentaire sur un code et un commentaire général. Si dans l'entité Review le codePart est à null alors on sait que cette review est une review général.

V. Authentification

Gestion authentification avec clients

L'authentification est gérée similairement pour les deux clients.
L'authentification elle-même se fait par envoi d'un simple login/password.

Elle est vérifiée via les mécanismes de base de Spring, en utilisant le AuthentificationManager et un token d'authentification généré avec les informations envoyées par le client.

Pour la persistance de l'authentification (vérification à chaque échange), le système de session de Spring est utilisé, avec donc l'utilisation d'un cookie JSESSIONID, permettant de mémoriser la session en cours, et notamment l'authentification une fois réalisée. Puisqu'en production, les deux clients sont servis par le même serveur, la même session est partagée entre les deux si l'utilisateur alterne entre les deux sur le même navigateur.
En termes de sécurité:

- Le cookie JSESSIONID est changé à chaque authentification pour éviter les failles de sécurité par fixation de session
- La protection CSRF est activée, avec utilisation d'un token XSRF pour éviter les attaques par CSRF

L'utilisation des protections apportées par Spring Security permet de sécuriser au mieux le système de sessions.

Nous avons initialement décidé contre l'implémentation de notre propre système (originalement des jetons JWT), ce qui après documentation extensive, ne revenait qu'à réinventer la roue et sans doute à introduire des failles de sécurité de partout. Le concept de jeton JWT lui-même n'ayant pas vraiment de sens en dehors d'un système d'authentification extérieur (de type OAuth2).

Client lourd

Un endpoint REST (/api/login) est disponible pour permettre au client lourd d'effectuer l'authentification. Login et mot de passe sont envoyés via une payload JSON au controller d'authentification qui renvoie un 200 si valide et stocke cette authentification dans la session actuelle. Cet envoi se fait en clair, ainsi, bien sûr, en production, il faudrait utiliser le protocole HTTPS afin d'éviter un fuitage du mot de passe, ce qui sort du cadre du produit final demandé dans le projet.

Le HttpClient d'Angular est configuré correctement pour renvoyer le cookie JSESSIONID à chaque requête, et est également configuré pour l'utilisation du token XSRF. Il est en effet nécessaire de récupérer le cookie le contenant, et de set un header X-XSRF-TOKEN afin que les requêtes aboutissent. Une configuration était également nécessaire du côté Spring pour permettre le fonctionnement de la protection CSRF avec une application du type Angular. Nous avons suivi pour ça les instructions de la documentation officielle.

Client léger

Un simple formulaire est proposé avec un traitement direct de la réponse POST par Spring avec les identifiants envoyés, et authentification si corrects. Il n'y a pas vraiment de subtilité de ce côté.

Aspect code

Pour gérer l'authentification, les mécanismes de base de Spring Security sont utilisés. Nous les avons paramétrés en définissant nos propres beans là ou nécessaires.
Notamment, nous avons dû implémenter notre propre UserDetailsService afin de se brancher sur notre base de données où sont stockées les informations utilisateurs.

```
@Override
public UserDetails loadUserByUsername(String login) {
  var user = userRepository.findByLogin(login)
      .orElseThrow(() -> HttpException.notFound( msg: "User with login " + login + " does not exists"));
  return new RevevueUserDetail(user.getId(), user.getPassword(), user.getLogin(), user.getRole(), user.getUsername());
}
```

On ne fait que réimplémenter la méthode loadUserByUsername dans celle-ci, qui se contente de faire la recherche dans la base via UserRepository, et renvoie un UserDetails associé. Nous utilisons également notre propre UserDetails, notamment pour gérer les rôles correctement, puisqu'il faut que les bonnes "authorities" soient mises.

Les trois beans importants qui sont définis sont les suivants:

```
@Bean
public PasswordEncoder passwordEncoder() { return new CustomPasswordEncoder(); }

 Clément G.
@Bean
public UserDetailsService userDetailsService(UserRepository userRepository) {
  return new CustomUserDetailsService(userRepository);
}

 Clément G.
@Bean
public AuthenticationManager authenticationManager(UserDetailsService userDetailsService, PasswordEncoder passwordEncoder) {
  var authenticationProvider = new DaoAuthenticationProvider();
  authenticationProvider.setUserDetailsService(userDetailsService);
  authenticationProvider.setPasswordEncoder(passwordEncoder);

  return new ProviderManager(authenticationProvider);
}
```

Nous injectons notre propre PasswordEncoder, UserDetailsService et enfin AuthenticationManager configuré pour utiliser ces deux beans là.

Avec cette configuration du AuthenticationManager, voici le code concret pour authentifier un utilisateur. Il est mis à part dans un LoginManager qui lui même est appelé par le controller REST, et par le controller MVC, permettant une factorisation de ce code et donc un point d'entrée unique pour l'authentification.

```java
public Optional<LoginResponseDTO> login(CredentialsDTO credentialsDTO, HttpServletRequest request, HttpServletResponse response) {
  LOGGER.info( msg: "try to login");
  var token = UsernamePasswordAuthenticationToken.unauthenticated(credentialsDTO.login(), credentialsDTO.password());
  try {
    var authentication = authenticationManager.authenticate(token);
    if (authentication.isAuthenticated()) {
      var securityContext = this.contextHolderStrategy.createEmptyContext();
      securityContext.setAuthentication(authentication);
      contextHolderStrategy.setContext(securityContext);
      this.securityContextRepository.saveContext(securityContext, request, response);
    }
    var currentUser = AuthenticationChecker.checkAuthentication();
    var role = currentUser.getAuthorities().stream().findFirst().orElseThrow();
    return Optional.of(new LoginResponseDTO(currentUser.getUsername(), role.getAuthority(), currentUser.displayName()));
  } catch (AuthenticationException e) {
    return Optional.empty();
  }
}
```

La première étape est de construire un token d'authentification à partir de la méthode **UsernamePasswordAuthenticationToken.unauthenticated(...)**.
On passe ensuite ce token au AuthenticationManager via la méthode authenticate, et celui-ci se charge alors de vérifier si ce token est valide, en utilisant donc notre configuration. Si l'authentification est invalide, une exception est jetée et rattrapée et une réponse négative est envoyée.

La dernière étape de cette méthode est la persistance de cette authentification. Une subtilité avec Spring Security 6 est qu'il est nécessaire de manuellement stocker l'authentification dans le SecurityContext (qui doit être persisté dans la session). Cela était fait automatiquement dans les versions précédentes, mais ce n'est plus le cas.
On crée donc un nouveau SecurityContext dans lequel on enregistre l'authentification (via la méthode setAuthentication), et on le stocke dans la session actuelle, à la fois dans l'immédiat via le ContextHolderStrategy, et pour les appels suivants via le SecurityContextRepository.

L'authentification est alors validée et enregistrée dans la session actuelle, et l'utilisateur peut rester authentifié via sa session tant qu'elle reste valide.

Authentification Web Service

Pour notre micro-service, celui-ci étant enveloppé dans un conteneur Docker, nous n'avons pas prévu dans le code de gérer l'authentification du backend de notre application, cette partie est à configurer au niveau de l'infrastructure Docker en paramétrant des firewalls pour contrôler les IP autorisées à communiquer avec le conteneur.

Une solution que nous aurions dû coder est un filtre Spring Security qui authentifie notre serveur back en renvoyant une erreur HTTP Forbidden si l'adresse entrant ne fait pas partie des adresses autorisées au lancement de l'application, sous la base d'une liste d'IP autorisées au déploiement du JAR du micro-service.

Custom Password Encoder

Pour encoder et vérifier les passwords de nos utilisateurs dans la base de données, nous avons implémenté notre propre encoder de password basé sur l'algorithme de hachage cryptographique SHA-256.
Pour ce faire, la classe dispose d'une méthode qui génère un sel aléatoire lors de l'encodage, le concatène avec le password brut pour ensuite le hacher à l'aide de l'algorithme.
Lors de la vérification, la classe extrait le sel de la chaîne encodé stocké en base de données, concatène celui-ci avec le password à vérifier et le hache avec l'algorithme.
Ensuite, elle compare le password résultant à partir du password récupéré en base de données.

VI. Features

Coloration syntaxique

Nous l'avons géré du côté client, en utilisant une librairie JavaScript appropriée. En l'occurrence, highlight.js
Il a suffi de le configurer correctement, et il a fonctionné sans trop de problème à la fois dans le client léger et lourd.

Recommandation de nos commentaires passées de manière intelligente

Lorsque nous avons abordé cette partie, au vu de la formulation de la fonctionnalité demandée, l'idée de traiter cela à l'aide d'un modèle du traitement naturel du langage (NLP) nous a paru la solution la plus appropriée à cela.
Avoir un modèle permettant de faire une correspondance entre deux questions sur leurs sujets aurait permis de faire remonter les commentaires déjà écrits sur ceux-ci de la manière la plus intelligente et optimale possible.
Pour cela, il aurait fallu entraîner un modèle de machine learning en partant par exemple d'un classifier (les données d'entraînement auraient pu être récupérées par exemple sur StackOverflow).

Après l'étude du sujet, nous nous sommes rendu compte que la grosse charge de travail associée, les priorités actuelles du projet et de notre faible connaissance sur le sujet ne rendait pas facile la tâche voir n'était pas très pertinente.
Nous nous sommes alors rabattus sur 2 options différentes pour traiter ce problème.
Dans un premier temps, l'apparition d'une nouvelle entité : les "Tags".

Chaque fois que l'utilisateur va créer une revue, il lui sera proposé de lui créer/attribuer un ou plusieurs tags pour celle-ci.

Sur la même fenêtre de création d'une revue, il sera possible à l'utilisateur d'utiliser ses tags pour retrouver les revues concernées.

Dans un second temps, nous avons opté pour une autre solution, celle de la comparaison littérale de textes.
Pour faire cela, il existe plusieurs algorithmes plus ou moins performants en fonction de la situation.
Après sélection et études, nous en avons essayé 3 :

- **LevenshteinDistance** : renvoie le nombre de transformation nécessaire pour arriver d'un texte A à un texte B, logiquement, plus le nombre de transformation est élevé plus les deux textes ont de chances d'être différent. Cette méthode dépend de nombreux paramètres, la taille du texte joue beaucoup aussi. Ainsi, lorsque l'on se retrouve avec des longues questions, c'est compliqué d'établir un seuil de transformation à partir duquel on accepte la question ou non.
- **CosineDistance** : donne la similarité de deux vecteurs en déterminant le cosinus de leur angle. Dans ce cas-ci, il faut construire deux vecteurs à partir de nos deux questions, généralement par un calcul de type TF-IDF (calcul d'un score d'importance pour chaque terme, frequence du terme dans un document multiplier par la fréquence d'apparition dans l'ensemble des documents). Le score est donné entre 0 et 1, plus le score est proche de 1 plus la ressemblance est similaire.

- **JaccardDistance** : la distance de Jaccard pour un texte A et un texte B est calculée comme le cardinal de l'intersection des éléments des textes A et B divisé par le cardinal de l'union des éléments des textes A et B. Cela donne une valeur entre 0 et 1, plus le résultat est proche de 0 plus les ensembles se ressemblent.

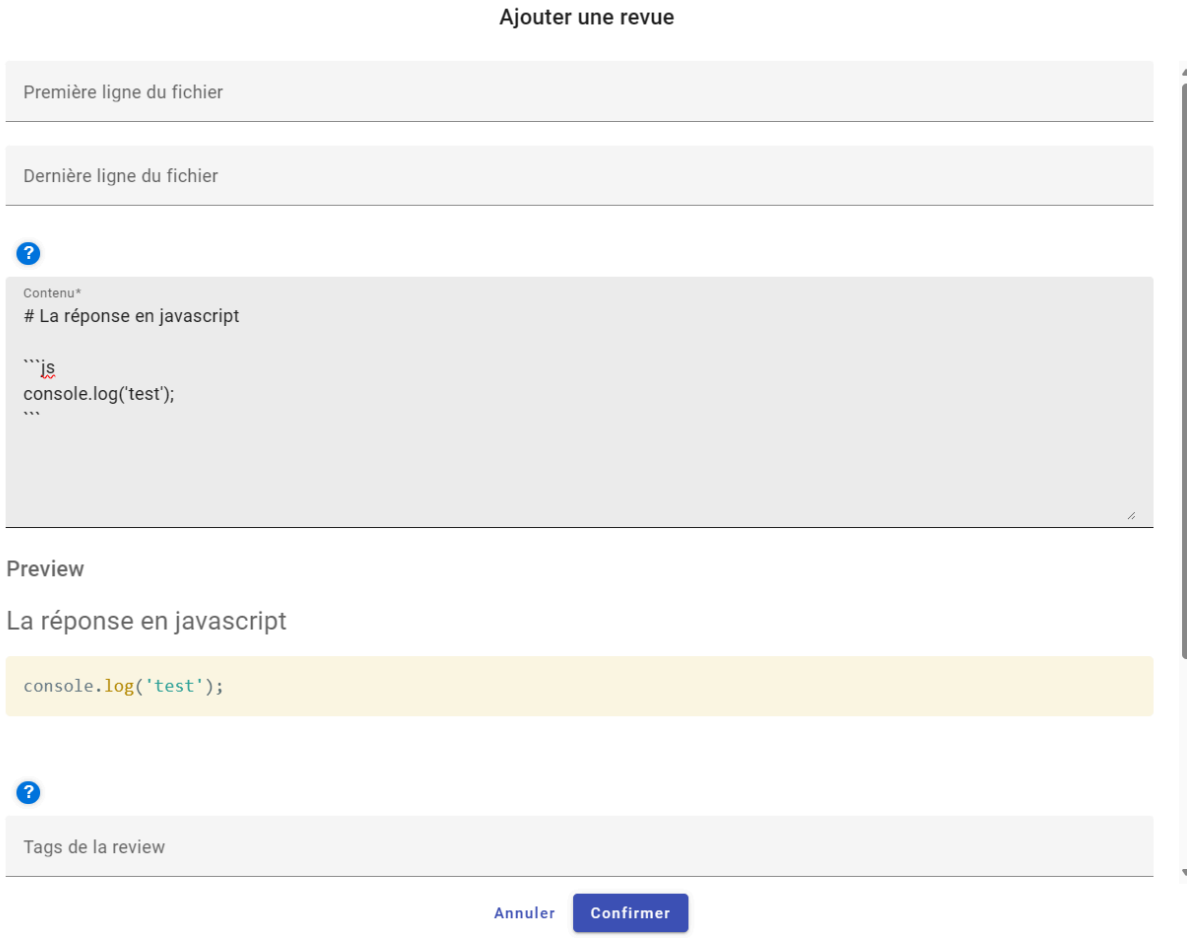
Pour ne pas réinventer la roue, nous avons trouvé que la librairie "org.apache.commons.text.similarity" implémente ces algorithmes.
Par tâtonnement, nous avons établi des seuils d'acceptation sur les scores qui nous étaient remontés.
Actuellement, seule la distance de Jaccard est implémentée en production.
Nous l'utilisons pour comparer la question sur laquelle l'utilisateur veut faire une review, avec les questions auxquelles ce même utilisateur a déjà fait des reviews. Si l'une d'elles passe le seuil d'acceptation alors elle est remontée.

Ces deux features ne sont disponibles que sur le client lourd.

VII. Sécurité

Est-il possible pour un utilisateur de mettre du javascript dans un de ses commentaires ?
Est-ce que ce javascript sera exécuté sur les machines des clients qui vont lire le commentaire ?

Nous avons essayé de contrôler l'injection de code JavaScript dans le front lourd et léger.
Avec Angular, ce qui est pratique, c'est qu'il y a un système d'échappement de code JavaScript intégré qui est le Sanitizer. Il suffit d'utiliser des composants Angular afin qu'il empêche l'injection de code côté client.
Si l'utilisateur souhaite insérer du code javascript (ou tout autre type de code) dans ses revues, alors il pourra le faire de la manière suivante :



Pour ce qui est du front light, Thymeleaf possède une balise th:text qui sérialise les entrées de l'utilisateur. Nous avons donc retiré toutes les balises **th:utext** qui elles ne sérialisent pas les entrées.

Nous avons effectué nos tests avec des CVE XSS qui ont vraiment existé sur des versions récentes de ces deux technologies.

Quels mécanismes avez-vous mis en place pour gérer le fait que le web service qui exécute les tests unitaires doit exécuter du code arbitraire ?

Pour la mise en place des tests unitaires, nous avons décidé de prendre des fichiers d'extension .java pour faciliter l'upload. L'utilisateur n'a pas besoin de compiler ses fichiers, nous le faisons pour lui.

Dans un premier temps les fichiers sont déposés physiquement dans le répertoire courant. Ensuite, toujours pour l'ergonomie, nous supprimons les package du fichier Java dont dépend le test. Dans le fichier de test nous supprimons les import de ce dernier ainsi que le package. Nous possédons maintenant 2 fichiers sans package que nous pouvons compiler.

Après avoir compilé les fichiers, nous utilisons un UrlClassLoader afin de charger le .class de la classe de test. Puis grâce au launcher Junit 5, nous lançons les tests du fichier.

Lorsque l'on exécute du code arbitraire, il y'a 3 principales failles auxquelles on pense directement :

- Le détournement de service autrement dit **Service Hijacking** (Minage de bitcoin, envoie de paquets massif via la machine, utilisation de la machine comme proxy etc.)
- Le dénie de service **Denial of Service - DoS** (Boucle infinie, suppression du système etc.)
- Compromission des données/du système **Data Compromise** (Infos sur la machine, élévation de privilège, buffer overflow etc.)

Pour le détournement de service, bloquer les envoies réseau vers l'extérieur permet d'éviter beaucoup de problèmes. Nous avons donc opté pour cette solution.

Pour ce qui est du déni de services, un timeout sur les tests permet d'éviter de cumuler des processus infinis sur la machine. De plus, un timeout va restreindre les actions de l'attaquant sur d'autres failles, il n'aura pas tout le temps dont il a besoin pour l'exécution.

Finalement, nous arrivons à la partie la plus difficile à corriger. La compromission de données. En effet, à partir du moment où l'on promet à l'utilisateur que son code va être exécuté dans un environnement, il a sa porté un choix énorme de failles. C'est en réfléchissant à cela que nous avons décidé de mettre en place un Conteneur Docker. L'utilisateur est isolé à l'intérieur.

Malheureusement par manque de temps, nous n'avons pas pu explorer le lancement d'un parc de conteneurs avec une queue via kubernetes qui aurait permis de répartir nos utilisateur sur des Docker individuels. Nous avons donc une faille car l'utilisateur peut accéder au code des autres et ainsi le corrompre et en revenir à faire du déni de service.

Qu'est-ce qui empêche un utilisateur de supprimer les commentaires ou reviews d'un autre utilisateur (montrez du code) ?

Pour la partie back:

Dans un premier temps nous vérifions que l'utilisateur qui essaye de supprimer une review est bien connecté avec une annotation @RequireUser qui permet de vérifier que l'utilisateur est connecté dans le controller.

```
@DeleteMapping(Routes.Review.ROOT + "/{reviewId}")
@RequireUser
public ResponseEntity<Void> removeReview(@PathVariable(name = "reviewId") long reviewId) {
  var user = AuthenticationChecker.authentication().orElseThrow();
  LOGGER.info( msg: "perform delete on " + Routes.Review.ROOT);
  reviewService.remove(new ReviewRemoveDTO(user.id(), reviewId));

  return ResponseEntity.ok().build();
}
```

Ensuite comme l'utilisateur est connecté on a accès à son rôle ou directement à la liste de ses reviews, si l'utilisateur a le rôle ADMIN alors il a le droit de supprimer toutes les reviews sinon on vérifie si l'utilisateur qui demande à supprimer une review est bien l'auteur de cette review. Si on est dans un de ces deux cas alors on fait la suite du traitement pour supprimer la review sinon l'utilisateur n'a pas les droits.

```
if (user.getRole() != Role.ADMIN && !userRepository.containsReview(user.getId(), review)) {
    logger.severe( msg: "You are trying to delete a review that is not yours");
    throw HttpException.unauthorized( msg: "Not your review");
}
```

Pour la partie front:

On a un bouton dans le front qui permet de supprimer une review, ce bouton sera seulement visible pour les utilisateurs connectés et si l'utilisateur est connecté il faut qu'il ait le rôle ADMIN ou alors qu'il soit l'auteur de cette review.

Conclusion

Le projet a été très intéressant à réaliser car au-delà du sujet lui-même, il nous a permis d'expérimenter les problématiques d'un réel projet de développement en entreprise, mise en relation avec les notions vues en cours. Certains d'entre nous avons pu comparer les pratiques dans ce projet avec ceux que nous développons en entreprise et nous a donc permis de prendre du recul et surtout de comprendre en détail chaque aspect du projet. En termes de difficulté, le projet semble en adéquation avec le niveau attendu de notre formation, si ce n'est plus dur encore, ce qui reste cohérent.

Ce projet nous a permis de toucher non seulement au dev mais également à de la gestion de projet (méthode agile), mais aussi à du devops (utilisation de docker pour le micro service).
En ce qui concerne nos déceptions, nous sommes conscient qu'à plusieurs échelle notre projet nécessite plus de sécurité ou même de vérifications. Cependant pour un projet aussi conséquent et compte-tenu de notre statut d'apprenti à tous les 5, nous en sommes très fiers.

Il nous a également fait prendre conscience de l'utilité de l'ensemble des cours de cette année, en effet chaque cours ou bien la grande majorité peut y trouver sa place dans un tel projet, une idée intéressante serait de proposer un projet aussi / plus conséquent ralliant le plus de connaissances vu lors de cette année à savoir : JEE, design-patterns mais également No&NewSQL (avec l'utilisation de Redis pour les données de session par exemple, de neo4j pour les algorithmes liés aux followers ou à celles des suggestions de reviews. L'intégration des outils devops pour simuler une intégration continue via des pipelines gitlab peut être aussi très intéressante pour se familiariser avec des outils que nous rencontrerons très certainement après nos études.