Exercice du Jeudi 09 janvier 2020

Pour cet exercice, vous repartirez du précédent exercice, du 13 décembre 2019.
Je vais vous demander beaucoup d'autonomie pour cet exercice. La partie authentification est sans doute assez complexe à appréhender. N'hésitez pas à vous renseigner sur spring security sur internet.
Je serai disponible comme la dernière fois sur Discord et IRC le plus possible, mais étant au travail je ne répondrai pas forcément rapidement. Pour vous connectez simplement sur le serveur IRC où je suis : https://webchat.quakenet.org/ sur le channel #csid. J'ai une préférence pour IRC.

Le code que vous produisez depuis le début de mon cours sera noté à la fin de mon module. Appliquez-vous, respectez au maximum les règles REST et le codes HTTP. N'oubliez pas de bien tester votre code avec Postman. L'export de la collection Postman fait parti de l'exercice.

Lors de notre prochain cours, nous continuerons ce mini projet.

N'hésitez pas à faire les questions 7 et 8 en premier. Elles sont sans doute plus simples à implémenter.

Vous allez implémenter un système d'authentification basé sur une session.
Pour cela, nous allons utiliser **spring-security** ; ajoutez (ou décommentez) les lignes suivantes du fichier *pom.xml* :

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

0. Ajoutez à votre projet les classes **CustomAuthenticationFilter**.java, **WebSecurity**.java et **Security**.java que je vous fourni à la fin du PDF.
La classe CustomAuthenticationFilter vous permet de vous connecter avec votre utilisateur grâce au endpoint ***/login***. Vous recevrez alors un cookie de session. Avec Postman, ce cookie sera normalement intégré à vos requêtes suivantes, ce qui indiquera à Spring avec quel utilisateur vous êtes authentifié.
Pour vous authentifier, envoyez une requête **POST** sur **/login** avec comme body ***{"username" :"xxxx", "password" : "xxxxx"}*** *(une fois l'étape 3 finie).*

Une fois authentifié, lors de vos prochains appels, vous aurez accès à un objet appelé Principal. Cette objet contient les détails de l'utilisateur authentifié. Vous pouvez y accéder à tout moment avec la code suivant : *SecurityContextHolder.getContext().getAuthentication().*
Avec le code suivant, vous pouvez récupérer un User (org.springframework.security.core.userdetails.User) et ainsi accéder ainsi au username utilisé lors de l'authentification :

Authentication authentication = SecurityContextHolder.getContext().getAuthentication();
User user = (User)authentication.getPrincipal();

1. Créer la classe **UserDetailsServiceImpl**.java dont **WebSecurity** a besoin. Cette classe doit implémenter l'interface **UserDetailsService** de Spring Security. Spring se servira de cette classe pour authentifier votre utilisateur sans que vous ayez à redévelopper tout le mécanisme de vérification du password.
Inspirez-vous de https://auth0.com/blog/implementing-jwt-authentication-on-spring-boot/
(Attention à ne pas copier bêtement tout ce que vous trouverez sur le site …).

2. Créer un controller, un service et un repository pour la gestions des utilisateurs.
Créer les DTO, model et entity nécessaires, sans oublier les mappers (prenez exemple sur Employee).
Ici un utilisateur désigne un compte application, qu'il ne faut pas confondre avec un employé.
Un utilisateur (nommez votre classe ApplicationUser ou AppUser pour éviter les conflits de noms avec Spring), est composé d'un ***username*** et d'un ***password***.
Attention ! Le mot de passe doit être enregistrer hashé ! Vous devez utiliser l'objet BcryptPasswordEncoder de la classe Security.java pour hasher le password avant de l'enregistrer !

3. Créer une méthode dans le controller pour qu'un utilisateur puisse s'inscrire. Un simple POST sur */users* devrait faire l'affaire;)
L'utilisateur créé pourra être rattaché à un employé ou non (du coup, il faudra sans doute modifier l'objet AppUser …).
Pour plus de simplicité, on crée un employé, on récupère son ID, puis on crée lors de la création de notre AppUser, on passera l'ID de l'employé auquel on veut le rattacher (les ID de Employee et AppUser peuvent être différents!).

4. Créer un ensemble de méthodes (controller, services, repository …) afin que l'on puisse afficher les informations d'un utilisateur.

5. Créer la classe **UserDetailsServiceImpl**.java dont **WebSecurity** a besoin. Cette classe doit implémenter l'interface **UserDetailsService** de Spring Security. Spring se servira de cette classe pour authentifier votre utilisateur sans que vous ayez à redévelopper tout le mécanisme.
Inspirez-vous de https://auth0.com/blog/implementing-jwt-authentication-on-spring-boot/
(Attention à ne pas copier bêtement tout ce que vous trouverez sur le site …).

6. Tester l'enregistrement d'un utilisateur et la connexion d'un utilisateur. Après la connexion, vous devez avoir un cookie !

7. Revenons aux méthode concernant un employé. Faites en sorte qu'un employé ne puisse consulter que sa fiche employé. Servez-vous du **SecurityContextHolder** cité plus haut pour récupérer l'id du **AppUser** connecté, puis à partir de cet **AppUser** vous récupérerez l'id de l'employé associé et appliquerez le bon traitement en fonction du résultat.
[Profitez-en pour appliquer le même traitement sur votre UserController lors de la lecture d'un AppUser (Trouvez une méthode élégante pour ne pas dupliqué votre code : un indice, on compare toujours des ID de type Long entre eux)]

8. On souhaite maintenant qu'un employé puisse poser des vacances. Pour simplifier le traitement, chaque jour de vacances est une ligne dans le base de données.

Vous pouvez, si vous le souhaitez, envoyez une range de date (01/01/2020-10/01/2020) et faire les insertions ligne par lignes dans votre service OU faire 1 requête HTTP par jour de congé posé. Attention ! Notre employé ne travaille pas les Samedis et Dimanches ! Dans le cas d'un range, vous exclurez les samedis et dimanches avant d'insérer, dans le cas d'une requête REST par jour de congés, vous renverrez le bon code HTTP si le jour envoyé est un samedi ou dimanche.

9. Maintenant que notre employé peut poser des congés, il faut recalculer son salaire en fonction du nombre de jours travaillés (oui, malheureusement, notre employé n'a pas de congés payés).

Bonus Spring-Security :
Il est possible de simplifier la vérification du user connecté et de celui demandé dans la requête grâce à des annotations : si vous avez du temps et encore de l'énergie, suivez le lien suivant :
https://www.baeldung.com/spring-security-create-new-custom-security-expression

CustomAuthenticationFilter.java =>

```
package fr.univparis8.iut.csid.security.filter;

import com.fasterxml.jackson.databind.ObjectMapper;
import fr.univparis8.iut.csid.security.UserCredentials;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;

import javax.servlet.FilterChain;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.ArrayList;


public class CustomAuthenticationFilter extends UsernamePasswordAuthenticationFilter {

    private final AuthenticationManager authenticationManager;
    private final ObjectMapper objectMapper;

    public CustomAuthenticationFilter(AuthenticationManager authenticationManager,
ObjectMapper objectMapper) {
        this.authenticationManager = authenticationManager;
        this.objectMapper = objectMapper;
    }

    @Override
    public Authentication attemptAuthentication(HttpServletRequest request, HttpServletResponse
response) {

        try {
            UserCredentials userCredentials = objectMapper.readValue(request.getInputStream(),
UserCredentials.class);

            return authenticationManager.authenticate(
                    new UsernamePasswordAuthenticationToken(userCredentials.getUsername(),
userCredentials.getPassword(), new ArrayList<>())
            );
        } catch (IOException e) {
            throw new IllegalArgumentException("Invalid authentication object", e);
        }
    }

    @Override
    protected void successfulAuthentication(HttpServletRequest request, HttpServletResponse
response, FilterChain chain, Authentication authResult) throws IOException, ServletException {
        SecurityContextHolder.getContext().setAuthentication(authResult);
```

```
    }
}
```

WebSecurity.java =>

```
package fr.univparis8.iut.csid.security;

import com.fasterxml.jackson.databind.ObjectMapper;
import fr.univparis8.iut.csid.security.filter.CustomAuthenticationFilter;
import
org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuil
der;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import
org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

@EnableWebSecurity
public class WebSecurity extends WebSecurityConfigurerAdapter {

   private UserDetailsServiceImpl userDetailsService;
   private BCryptPasswordEncoder bCryptPasswordEncoder;

   public WebSecurity(UserDetailsServiceImpl userDetailsService, BCryptPasswordEncoder
bCryptPasswordEncoder) {
      this.userDetailsService = userDetailsService;
      this.bCryptPasswordEncoder = bCryptPasswordEncoder;
   }

   @Override
   protected void configure(HttpSecurity http) throws Exception {
      http.cors().and().csrf().disable().authorizeRequests()
            .antMatchers("/users/**").permitAll()
            .antMatchers("/employees/**").permitAll()
            .anyRequest().authenticated()
            .and()
            .addFilter(new CustomAuthenticationFilter(authenticationManager(), new
ObjectMapper()))
            .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.IF_REQUIRED);
   }

   @Override
   public void configure(AuthenticationManagerBuilder auth) throws Exception {
      auth.userDetailsService(userDetailsService).passwordEncoder(bCryptPasswordEncoder);
   }

}
```

```
package fr.univparis8.iut.csid.security;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

@Configuration
public class SecurityConfiguration {

    @Bean
    public BCryptPasswordEncoder bCryptPasswordEncoder() {
        return new BCryptPasswordEncoder();
    }

}
```