TP Spring n°7
Spring Security

On va, dans ce TP, utiliser la couche de sécurité de SpringBoot afin d'authentifier un client. Pour l'instant, on ne visualise que nos produits en base. On souhaiterai maintenant ajouter un produit dans notre panier…

Mais ajouter un produit à notre panier, cela revient à créer une ligne de commande.
Mais pour ajouter une ligne de commande, on a besoin d'une commande.
Mais pour créer une commande, il faut l'associer à un client.
Donc le client doit être authentifié pour ajouter un produit à son panier.

Nous verrons le problème sous cette angle, bien qu'aujourd'hui les sites d'e-commerce ne fonctionnent pas exactement de cette manière (utilisation de cookies et authentification seulement nécessaire pour passer la commande).
Spring Security implémente la sécurité avec des filtres qui s'intercalent entre l'utilisateur et la servlets.

1° - Ajout de la dépendance Spring Security

Pour mettre en place la sécurité, il suffit d'ajouter la dépendance dans le pom.xml :

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

Si vous relancez l'application, elle est maintenant "sécurisée".
Une authentification est obligatoire pour accéder au site.

Comme vous n'avez encore rien défini, le login c'est "user" et le mot de passe est généré et affiché dans la console :

```
Using generated security password: 72**********************dssqd21
```

A la place de ce mot de passe généré, vous pouvez définir un login/password dans le fichier de properties de l'application :

```
spring.security.user.name = moi
spring.security.user.password = lol
```

Pour se déconnecter, l'URL en /logout est configurée pour que cela marche.

2° - Configuration

1° - Nous allons définir un nouveau package *security* dans lequel nous allons créer une classe de configuration. Nommez cette classe *WebSecurityConfig*, elle est annotée de @Configuration, @EnableWebSecurity et elle étend de *WebSecurityConfigurerAdapter.*

2° - Redéfinissez la méthode configure qui prend en paramètre WebSecurity afin de préciser que les ressources (images, css…) ne sont pas sécurisées.

```
web.ignoring().antMatchers("/resources/**");
```

3° - Déclarez dans la classe une constante de type tableau de String et initialisez-le avec les routes qui sont exclues de la sécurité, les routes pour lesquelles un client n'a pas besoin d'être authentifié.

Dans notre cas nous devons avoir :

- / : pour la page home.
- /products : pour la page des produits.
- /products/** : pour la page d'un produit.

4° - Redéfinissez votre seconde méthode de configuration (configure qui prend en paramètre un HttpSecurity) avec le code suivant :

```
http.authorizeRequests()
     .antMatchers(PUBLIC_MATCHERS).permitAll().anyRequest().authenticated()
     .and()
     .formLogin()
      // .loginPage("/login")
      // .usernameParameter("email")
      // .defaultSuccessUrl("/tasks")
     .permitAll()
     .and()
     .logout().logoutSuccessUrl("/").permitAll();
```

Cela permet de configurer les pages publiques et les autres qui nécessitent une authentification.

3° - Authentification d'un client

Nous allons maintenant authentifier un client depuis les informations existantes en base de données.

1° - On va définir l'entité Client en implémentant l'interface UserDetails, ajoutez les méthodes nécessaires (ces méthodes ne seront pas utilisées, faites un return null pour les objets et un return true pour les booleans). Cela permettra de stocker simplement les informations utilisateur qui sont ensuite encapsulées dans des objets d'authentification.

2° - Dans la couche métier de votre client (ClientServiceImpl) implémentez l'interface *UserDetailsService*. Ajoutez la méthode à implémenter.
Dans cette méthode, cherchez l'utilisateur qui a cet *username* (utilisez la méthode du repository), s'il existe retournez, le sinon levez une exception de type *UsernameNotFoundException.*

3° - Dans votre classe Application, ajoutez la méthode suivante permettant de créer un bean de type PasswordEncoder qui en l'occurrence sera initialisé par un NoOpPasswordEncoder car votre mot de passe est en clair en base…

```
@Bean
public PasswordEncoder passwordEncoder() {
	return NoOpPasswordEncoder.getInstance();
}
```

4° - Retournez dans votre classe de configuration. Ajoutez les deux attributs suivants :

```
private final UserDetailsService userDetailsService;

private final PasswordEncoder passwordEncoder;
```

Utilisez l'annotation Autowired sur :

- userDetailsService : pour câbler l'objet au bean *ClientServiceImpl* (qui implémente cette interface).
- passwordEncoder : pour câbler l'objet au bean créé précedemment précisant qu'il n'y a pas d'encodage pour le password.

5° - Ajoutez la méthode configure qui prend en paramètre un *AuthenticationManagerBuilder.*
Puis ajoutez le code suivant permettant de faire l'authentification de l'utilisateur (le client) selon l'encodage du password.

```
auth.userDetailsService(userDetailsService).passwordEncoder(passwordEncoder);
```

Testez votre application avec un utilisateur en base.

4° - Client authentifié

1° - Vous pouvez récupérer, dans un controller, votre client authentifié grâce à la ligne suivante :

```
Client client = (Client)
SecurityContextHolder.getContext().getAuthentication().getPrincipal();
```

2° - Egalement dans le jsp :

```
<security:authorize access="isAuthenticated()">
    Bonjour <security:authentication property="principal.username" />
</security:authorize>
```

En utilisant le taglib suivant :

```
<%@ taglib uri="http://www.springframework.org/security/tags" prefix="security" %>
```