ALOM - TP 7 - Security

Table of Contents

1. Présentation et objectifs 1
2. Sécuriser trainer-api 2
   2.1. spring-security 2
   2.2. Configurer un user et un mot de passe 3
   2.3. Votre collection Postman 3
   2.4. Impact sur les tests d'intégration 5
   2.5. Le cas des POST / PUT / DELETE - CSRF & CORS 6
      2.5.1. Exemple d'attaque CSRF 6
      2.5.2. Désactivation du CSRF, et customisation de la configuration 8
3. Impacts sur `game-ui` 8
   3.1. Sécurisation des appels à `trainer-api` 8
      3.1.1. application.properties 8
      3.1.2. Impact sur les HTTP Interfaces ou les `RestTemplate` ! 9
         `RestTemplate` 9
         HTTP Interfaces 11
4. Sécuriser `game-ui` 11
   4.1. Gestion du mot de passe dans `trainer-api` 11
   4.2. Récupération du mot de passe dans `game-ui` 13
   4.3. Configuration de spring-security 13
   4.4. Personnalisation de spring-security 14
      4.4.1. Le test unitaire 14
      4.4.2. L'implémentation 16
   4.5. La page "Mon Profil" 16
      4.5.1. Le @Controller 17
      4.5.2. Le `TrainerService` 17
   4.6. Impacts sur l'IHM avec Mustache 18
      4.6.1. Le `ControllerAdvice` et `ModelAttribute` 18
         Le test unitaire 18
         L'implémentation 19
      4.6.2. Utilisation 20
5. Pour aller plus loin 21

1. Présentation et objectifs

Le but est de continuer le développement de notre architecture "à la microservice".

Nous allons aujourd'hui sécuriser les accès à nos API et à notre application !

Pendant ce TP, nous faisons évoluer les TP précédents !

Nous ne sécuriserons pas l'accès à l'API `pokemon-type`, étant donné que cette API ne présente pas de données sensibles !

2. Sécuriser trainer-api

Nous allons commencer par sécuriser l'API `trainers`.

2.1. spring-security

Configurez `spring-security` dans le `pom.xml` de votre API trainers.

pom.xml

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

Démarrez votre API.

Vous devriez voir des lignes de logs supplémentaire apparaître :

```
INFO --- [main] .s.s.UserDetailsServiceAutoConfiguration :

Using generated security password: 336470fd-a4be-474e-9e1a-84359f8b3808 ①

②
INFO --- [main] o.s.s.web.DefaultSecurityFilterChain    : Creating filter chain: any
request,
[org.springframework.security.web.context.request.async.WebAsyncManagerIntegrationFilt
er@45cf0c15,
org.springframework.security.web.context.SecurityContextPersistenceFilter@becb93a,
org.springframework.security.web.header.HeaderWriterFilter@723b8eff,
org.springframework.security.web.csrf.CsrfFilter@1fec9d33,
org.springframework.security.web.authentication.logout.LogoutFilter@7852ab30,
org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter@5
08b4f70,
org.springframework.security.web.authentication.ui.DefaultLoginPageGeneratingFilter@5e
9f1a4c,
org.springframework.security.web.authentication.ui.DefaultLogoutPageGeneratingFilter@2
f2dc407,
org.springframework.security.web.authentication.www.BasicAuthenticationFilter@67ceaa9,
org.springframework.security.web.savedrequest.RequestCacheAwareFilter@1d1fd2aa,
```

```
org.springframework.security.web.servletapi.SecurityContextHolderAwareRequestFilter@65
a2e14e,
org.springframework.security.web.authentication.AnonymousAuthenticationFilter@c96c497,
org.springframework.security.web.session.SessionManagementFilter@20d65767,
org.springframework.security.web.access.ExceptionTranslationFilter@39840986,
org.springframework.security.web.access.intercept.FilterSecurityInterceptor@42fa5cb]
```

① Le mot de passe généré par défaut !

② On voit également que Spring a décidé de filtrer l'ensemble des requêtes !

2.2. Configurer un user et un mot de passe

Modifiez votre fichier `application.properties` pour changer le mot de passe par défaut.

En effet, ce mot de passe par défaut est différent à chaque redémarrage de notre API. Ce qui n'est pas très pratique pour nos consommateurs !

Vous pouvez générer un mot de passe par défaut en utilisant un UUID (c'est ce que fait Spring).

Si vous êtes sous linux, vous pouvez utiliser la commande `uuidgen`.

Sinon, vous pouvez utiliser un générateur en ligne, par exemple : https://www.uuidgenerator.net/

application.properties

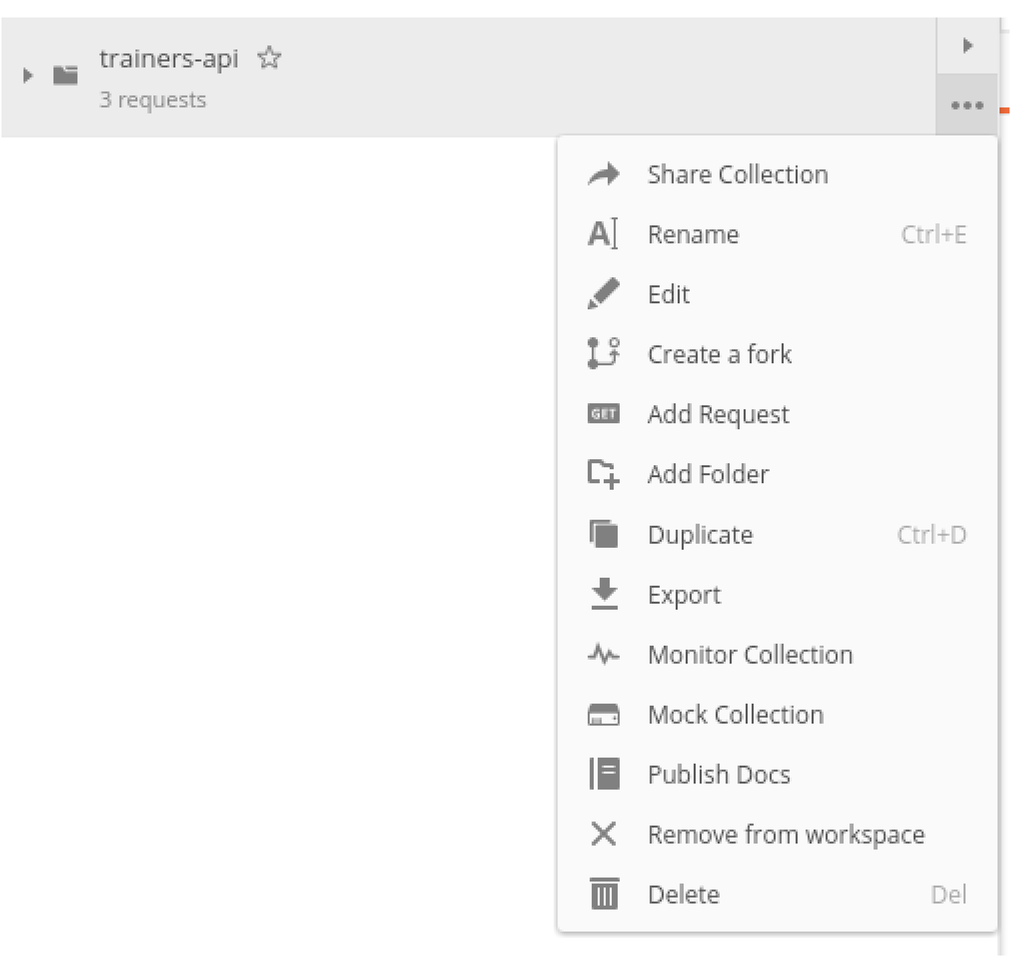
```
spring.security.user.name=user
spring.security.user.password=<votre-uuid>
```

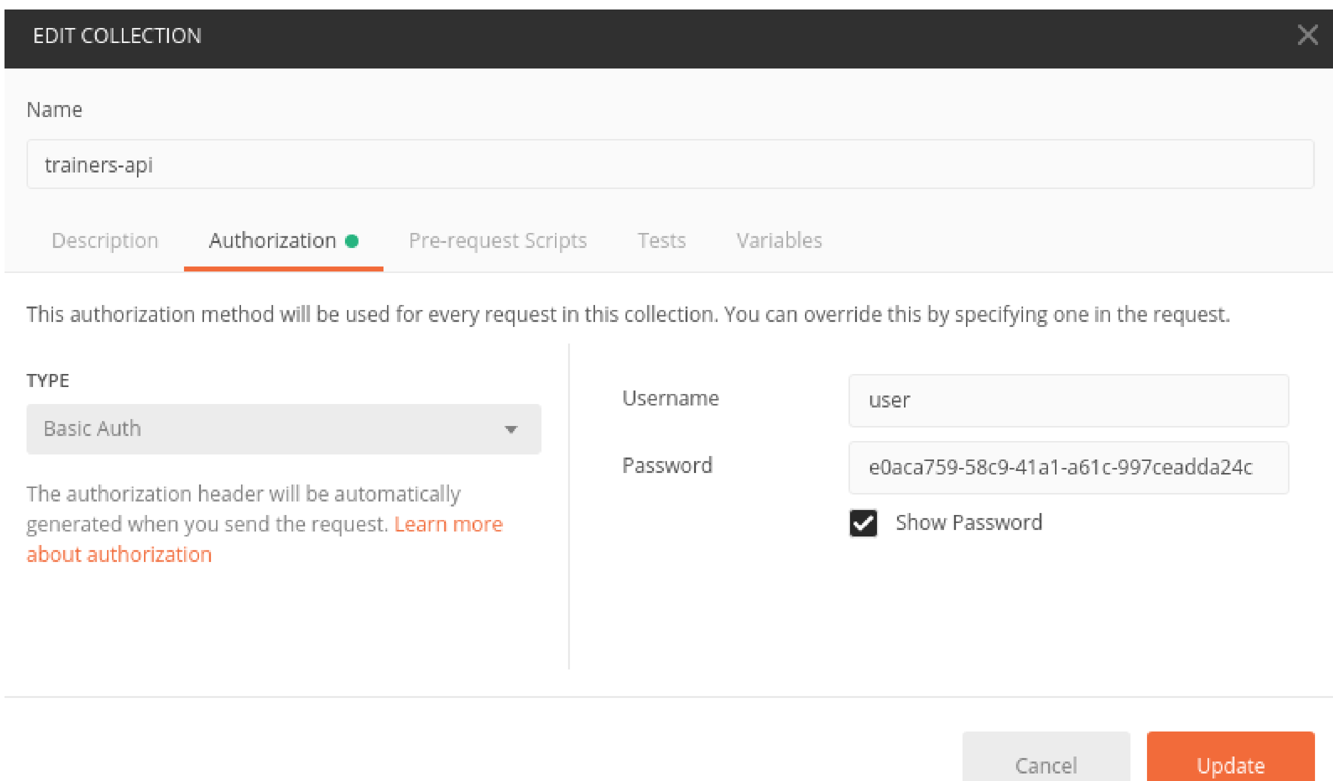
2.3. Votre collection Postman

Vos requêtes Postman doivent maintenant renvoyer des erreurs de ce type :

```
{
    "timestamp": "2019-03-08T09:39:51.720+0000",
    "status": 401,
    "error": "Unauthorized",
    "message": "Unauthorized",
    "path": "/trainers"
}
```

Configurez votre collection Postman pour utiliser l'authentification `Basic`. Pour ce faire, vous pouvez directement ajouter l'authentification au niveau de la collection :





Pour info, vous pouvez aussi constater que spring-security génère une page de login par défaut, si vous allez voir sur l'url de votre api avec un browser classique http://localhost:8081 !

2.4. Impact sur les tests d'intégration

Nos tests d'intégration du TrainerController doivent également être impactés. Ces tests supposaient que l'API n'était pas authentifiée.

Si vous les exécutez, vous devriez voir des logs de ce type :

```
DEBUG XXX --- [main] o.s.web.client.RestTemplate : Response 401 UNAUTHORIZED
DEBUG XXX --- [main] o.s.web.client.RestTemplate : Reading to
[com.miage.alom.tp.trainer_api.bo.Trainer]
```

Le `TestRestTemplate` de spring contient une méthode `withBasicAuth`, qui permet de facilement passer un couple d'identifiants à utiliser sur la requête.

Pour impacter votre test d'intégration, vous devez donc :

- recevoir en injection de dépendance le `user` de votre API
- recevoir en injection de dépendance le `password` de votre API
- passer le `user` et `password` au `TestRestTemplate`

TrainerControllerIntegrationTest.java

```
 1 @SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
 2 class TrainerControllerIntegrationTest {
 3
 4     @LocalServerPort
 5     private int port;
 6
 7     @Autowired
 8     private TestRestTemplate restTemplate;
 9
10     @Autowired
11     private TrainerController controller;
12
13     @Value("") ①
14     private String username;
15
16     ②
17     private String password;
18
19     @Test ③
20     void getTrainers_shouldThrowAnUnauthorized(){
21         var responseEntity = this.restTemplate
22                 .getForEntity("http://localhost:" + port + "/trainers/Ash",
   Trainer.class);
23         assertNotNull(responseEntity);
24         assertEquals(401, responseEntity.getStatusCodeValue());
25     }
26
```

```
27     @Test ④
28     void getTrainer_withNameAsh_shouldReturnAsh() {
29         var ash = this.restTemplate
30                 .withBasicAuth(username, password) ④
31                 .getForObject("http://localhost:" + port + "/trainers/Ash",
   Trainer.class);
32
33         assertNotNull(ash);
34         assertEquals("Ash", ash.getName());
35         assertEquals(1, ash.getTeam().size());
36
37         assertEquals(25, ash.getTeam().get(0).getPokemonType());
38         assertEquals(18, ash.getTeam().get(0).getLevel());
39     }
40
41 }
```

① Injectez votre properties représentant le user ici

② Injectez votre properties de mot de passe ici

③ Ce test permet de valider que l'API est sécurisée

④ Modifiez les autres tests pour ajouter l'authentification

2.5. Le cas des POST / PUT / DELETE - CSRF & CORS

Par défaut, *spring-security* gère une sécurité de type CSRF (Cross-Site-Request-Forgery). Cette mécanique permet de s'assurer qu'une requête qui modifie des données `POST/PUT/DELETE` ne peut pas provenir d'un site tiers.

2.5.1. Exemple d'attaque CSRF

Cette partie n'est qu'informative, pour expliquer comment un pirate pourrait utiliser une API de manière malicieuse. Vous n'avez rien à implémenter ici.

Sur un site web malicieux, un pirate crée un formulaire, par exemple :

www.pirate-moi.fr

```
<form action="https://bank.example.com/transfer" method="post">
<input type="hidden"
    name="amount"
    value="100.00"/>
<input type="hidden"
    name="account"
    value="evilsAccountNumber"/>
<input type="submit"
    value="Win Money!"/>
</form>
```

La requête émise

```
POST /transfer HTTP/1.1
Host: bank.example.com
Content-Type: application/x-www-form-urlencoded

amount=100.00&account=9876
```

Ce petit formulaire affiche un bouton "Win Money!" aux utilisateur, mais en vrai exécute un `POST` sur une banque, en effectuant un virement sur le compte du pirate !

Le service web de la banque n'est pas capable de faire la différence entre une requête émise par son site web, ou par un site web pirate !

Le pirate effectue ensuite une simple attaque de type phishing pour transmettre un lien vers votre page et le tour est joué.

Pour se prémunir de ce genre de cas, 2 parades sont à prévoir :

- CORS : Cross-Origin-Resource-Sharing : Le browser ne transmet la requête au serveur que s'il est dans la même origine. Ici, les requêtes sont émises depuis un site dont l'origine est http://www.pirate-moi.fr. Les browser refusent par défaut ce type de requête (ouf !).
- Synchronizer Token Pattern : Pour s'assurer que le formulaire est bien envoyé par une application qui en a le droit, un token est créé sur les pages du site web. Ce token permet de valider la requête côté serveur. Le but est bien de s'assurer que le pirate ne peut pas disposer de token valide sur son site.

Avec ce token, les requêtes émises doivent donc ressembler à cela :

La requête émise avec le token

```
POST /transfer HTTP/1.1
Host: bank.example.com
Content-Type: application/x-www-form-urlencoded

amount=100.00&account=9876&_csrf=<secure-random>
```

Lorsque nous allons modifier notre IHM, nous devrons intégrer dans nos formulaires la gestion de ce token. Pour l'instant, notre API n'étant consommée que par notre IHM, nous pouvons désactiver cette sécurité.

Ne désactivez cette sécurité uniquement si votre API n'est pas accessible directement !

Attention, ne faites pas ça en entreprise sans la validation d'un responsable sécurité !

En général, les API ne sont jamais consommées en direct, et donc jamais exposées

sur le web. Dans ce cas, il est acceptable de désactiver cette sécurité.

2.5.2. Désactivation du CSRF, et customisation de la configuration

Pour configurer *spring-security*, nous devons implémenter la classe suivante :

SecurityConfig.java

```
1 @Configuration ①
2 public class SecurityConfig {
3
4     @Bean
5     SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
6         http.csrf(csrf -> csrf.disable()); ②
7         http.authorizeHttpRequests(authorize -> {
8             authorize.anyRequest().authenticated(); ③
9           }
10        );
11        http.httpBasic(Customizer.withDefaults()); ④
12        return http.build();
13    }
14 }
```

① Nous créons une classe de configuration dédiée à la configuration de la sécurité

② Nous désactivons la protection CSRF sur notre API

③ Chaque requête doit être authentifiée !

④ On utilise une authentification HTTP Basic

Une fois cette classe implémentée, les tests d'intégration, ainsi que les requêtes Postman `POST/PUT/DELETE` devraient fonctionner !

3. Impacts sur `game-ui`

Maintenant que votre API de Trainers est sécurisée, il faut également reporter la sécurisation dans les services qui la consomment. En particulier sur le `game-ui`.

3.1. Sécurisation des appels à `trainer-api`

3.1.1. application.properties

Commençons par copier le `username`/`password` qui nous permet d'appeler `trainer-api` dans les properties de `game-ui`

application.properties

```
trainer.service.url=http://localhost:8081
trainer.service.username=user
```

```
trainer.service.password=<votre password>
```

3.1.2. Impact sur les HTTP Interfaces ou les `RestTemplate` !

`RestTemplate`

Vous devriez déjà avoir modifié votre code pour ne plus utiliser les `RestTemplate`. Cette section est conservée à but documentaire.

Nous devons également modifier notre usage du `RestTemplate` pour utiliser l'authentification.

Une manière simple et efficace est d'utiliser un `intercepteur`, qui va s'exécuter à chaque requête émise par le `RestTemplate` et ajouter les headers http nécessaire !

Hé ! On pourrait faire pareil pour transmettre la `Locale` de notre utilisateur !

Modifiez votre classe RestConfiguration pour utiliser un intercepteur

Le test unitaire

com.miage.alom.tp.game_ui.config.RestConfigurationTest.java

```
 1 package com.miage.alom.tp.game_ui.config;
 2 
 3 import org.junit.jupiter.api.Test;
 4 import org.springframework.http.client.support.BasicAuthenticationInterceptor;
 5 
 6 import static org.junit.jupiter.api.Assertions.*;
 7 
 8 class RestConfigurationTest {
 9 
10     @Test
11     void restTemplate_shouldExist() {
12         var restTemplate = new RestConfiguration().restTemplate();
13 
14         assertNotNull(restTemplate);
15     }
16 
17     @Test
18     void trainerApiRestTemplate_shouldHaveBasicAuth() {
19         var restTemplate = new RestConfiguration().trainerApiRestTemplate();
20 
21         assertNotNull(restTemplate);
22 
23         var interceptors = restTemplate.getInterceptors();
24         assertNotNull(interceptors);
25         assertEquals(1, interceptors.size());
26 
27         var interceptor = interceptors.get(0);
28         assertNotNull(interceptor);
```

```
29
30          assertEquals(BasicAuthenticationInterceptor.class, interceptor.getClass());
31      }
32 }
```

L’implémentation

Modifiez la classe `RestConfiguration` pour passer les tests unitaires.

RestConfiguration.java

```
 1 @Configuration
 2 public class RestConfiguration {
 3
 4     ①
 5
 6     @Bean
 7     RestTemplate trainerApiRestTemplate(){ ②
 8         // TODO
 9     }
10
11     @Bean
12     RestTemplate restTemplate(){
13         return new RestTemplate();
14     }
15 }
```

① Utilisez l’injection de dépendance pour charger le `user` et `password` de l’API Trainers, avec `@Value`

② Construisez un `RestTemplate` avec un intercepteur `BasicAuthenticationInterceptor`.

Utilisation du bon `RestTemplate`

Maintenant, notre `game-ui` possède deux `RestTemplate`. Un utilisant l’authentification pour `trainer-api`, et l’autre sans, pour `pokemon-type-api`. Il faut indiquer à spring quel `RestTemplate` sélectionner lorsqu’il fait l’injection de dépendances dans le `TrainerServiceImpl`.

Cela se fait à l’aide de l’annotation `@Qualifier`.

Modifiez votre injection de dépendance dans le `TrainerServiceImpl` :

TrainerServiceImpl.java

```
1 @Autowired
2 @Qualifier("trainerApiRestTemplate") ①
3 void setRestTemplate(RestTemplate restTemplate) {
4     this.restTemplate = restTemplate;
5 }
```

① Qualifier prend en paramètre le nom du bean à injecter. Le nom de notre `RestTemplate` est le nom de la méthode qui l’a instancié dans notre `RestConfiguration`

HTTP Interfaces

Pour utiliser une authentification basique sur une *HTTP Interface* Spring, il faut ajouter un *filter* à la construction du `WebCLient` utilisé par l'*HTTP Interface*.

Un exemple est présent dans la documentation de Spring.

Dans la classe qui configure vos *HTTP Interfaces*, recevez en injection de dépendance le `user` et `password` de l’API Trainers, avec `@Value`.

Puis ajoutez un filter `basicAuthentication` à votre `WebClient`.

4. Sécuriser `game-ui`

Nous allons maintenant utiliser une authentification login/mot de passe sur l’ensemble de notre application ! Les login/mot de passe seront ceux de nos dresseurs de pokemon gérés par `trainer-api`.

4.1. Gestion du mot de passe dans `trainer-api`

Nous allons commencer par créer un champ "password" dans la `trainer-api`. Ce champ contiendra le mot de passe du dresseur encrypté avec BCrypt.

BCrypt est un algorithme de hash, comme MD5 ou SHA-1/SHA-256.

Trainer.java

```
1 package com.miage.alom.tp.trainer_api.bo;
2
3 import javax.persistence.*;
4 import java.util.List;
5
6 @Entity
7 public class Trainer {
8
9     [...]
10
11     @Column ①
12     private String password;
13
14     [...]
15
16     ②
17     public String getPassword() {
18     }
19
20     public void setPassword(String password) {
21     }
22 }
```

① On ajoute un nouveau champ `password`

② On n'oublie pas les Getters/Setters

Les mots de passe doivent toujours être chiffrés en base de données. Ne stockez jamais de mots de passe clair.

Nous allons également alimenter nos deux dresseurs iconiques avec des mots de passe par défaut. Pour ce faire, nous modifions la classe principale de notre API :

TrainerApi.java

```java
@Bean
@Autowired
public CommandLineRunner demo(TrainerRepository repository) {
    BCryptPasswordEncoder bCryptPasswordEncoder = new BCryptPasswordEncoder(); ①

    return (args) -> {
        var ash = new Trainer("Ash");
        var pikachu = new Pokemon(25, 18);
        ash.setTeam(List.of(pikachu));
        ash.setPassword(bCryptPasswordEncoder.encode("ash_password")); ②

        var misty = new Trainer("Misty");
        var staryu = new Pokemon(120, 18);
        var starmie = new Pokemon(121, 21);
        misty.setTeam(List.of(staryu, starmie));
        misty.setPassword(bCryptPasswordEncoder.encode("misty_password")); ②

        // save a couple of trainers
        repository.save(ash);
        repository.save(misty);
    };
}
```

① On utilise un BCryptPasswordEncoder, qui est une des classes fournies par spring-security

② On l'utilise pour encrypter les mots de passe de nos dresseurs !

L'algorithme de hashage BCrypt utilise un "sel" de hashage (valeur unique à chaque utilisation), et un "cost" (nombre de boucles d'itérations pour le hashage), ce qui le rend particulièrement robuste (et coûteux à l'exécution).

Cela implique qu'un mot de passe hashé deux fois, aura une valeur de hashage différente (grace au "sel").

Cela nous prémunit des attaques de type "rainbow table/reverse table", qui consiste à calculer de nombreuses valeurs de hashage pour des mots de passe, et donc en ayant accès à un mot de passe hashé, de pouvoir retrouver sa valeur en clair.

Vous devriez voir les mots de passe cryptés lors des appels à votre API !

```
{
    "name": "Ash",
    "team": [
        {
            "id": 1,
            "pokemonType": 25,
            "level": 18
        }
    ],
    "password": "$2a$10$NIDVYQO574l/.8sTdAhEeuc/GW/aKNN5w1eLjg3kr4Oh2u7dFIowC"
}
```

4.2. Récupération du mot de passe dans `game-ui`

Le mot de passe doit également être récupéré dans `game-ui`.

Ajoutez le champ `password` à la classe `Trainer` de votre `game-ui`, ainsi que les getter/setter nécessaires.

4.3. Configuration de spring-security

Commençons par ajouter spring-security au `pom.xml` de `game-ui`.

pom.xml

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

Ouvrez l'url de votre IHM : http://localhost:9000.

Vous devriez tomber sur une page de login !

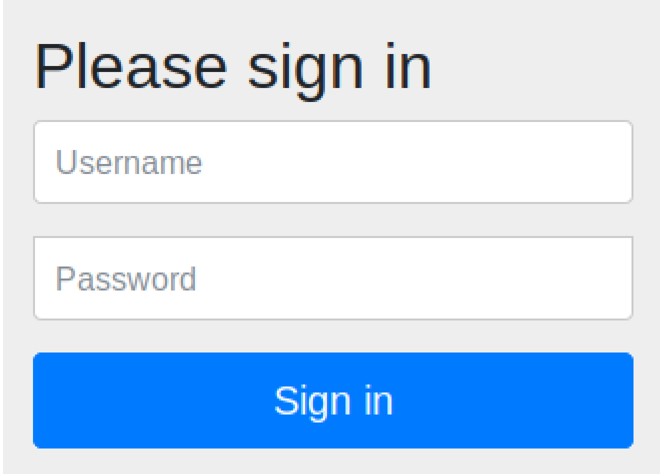


Figure 1. La page de login par défaut de spring-security !

Pour rappel, le user par défaut de spring-security est `user` et le mot de passe par défaut apparaît dans les logs !

4.4. Personnalisation de spring-security

Nous ne voulons pas utiliser un login par défaut, mais bien se logguer avec les comptes de dresseurs de pokémon gérés dans `trainer-api`.

Nous devons donc personnaliser un peu la configuration de spring-security !

4.4.1. Le test unitaire

Implémentez le test unitaire suivant :

SecurityConfigTest.java

```
 1 package fr.univ_lille.alom.game_ui.config;
 2
 3 import fr.univ_lille.alom.game_ui.trainers.Trainer;
 4 import fr.univ_lille.alom.game_ui.trainers.TrainerService;
 5 import org.junit.jupiter.api.Test;
 6 import org.springframework.security.authentication.BadCredentialsException;
 7 import org.springframework.security.core.GrantedAuthority;
 8 import org.springframework.security.core.authority.SimpleGrantedAuthority;
 9 import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
10
11 import static org.junit.jupiter.api.Assertions.*;
12 import static org.mockito.Mockito.*;
13
14 @ExtendWith(MockitoExtension.class)
15 class SecurityConfigTest {
16
17     @InjectMocks
18     private SecurityConfig securityConfig;
19
```

```
20      @Mock
21      private TrainerService trainerService;
22
23      @Test
24      void passwordEncoder_shouldBeBCryptPasswordEncoder() {
25          var passwordEncoder = securityConfig.passwordEncoder();
26          assertNotNull(passwordEncoder);
27          assertEquals(BCryptPasswordEncoder.class, passwordEncoder.getClass());
28      }
29
30      @Test
31      void userDetailsService_shouldUseTrainerService() {
32          var trainer = new Trainer();
33          trainer.setName("Garry");
34          trainer.setPassword("secret");
35          when(trainerService.getTrainer("Garry")).thenReturn(trainer);
36
37          securityConfig.setTrainerService(trainerService);
38
39          var userDetailsService = securityConfig.userDetailsService();
40
41          var garry = userDetailsService.loadUserByUsername("Garry");
42
43          // mock should be called
44          verify(trainerService).getTrainer("Garry");
45
46          assertNotNull(garry);
47          assertEquals("Garry", garry.getUsername());
48          assertEquals("secret", garry.getPassword());
49          assertTrue(garry.getAuthorities().contains(new
SimpleGrantedAuthority("ROLE_USER")));
50      }
51
52      @Test
53      void
userDetailsService_shouldThrowABadCredentialsException_whenUserDoesntExists() {
54          // the mock returns null
55
56          var userDetailsService = securityConfig.userDetailsService();
57
58          var exception = assertThrows(BadCredentialsException.class, () ->
userDetailsService.loadUserByUsername("Garry"));
59          assertEquals("No such user", exception.getMessage());
60
61          // mock should be called
62          verify(trainerService).getTrainer("Garry");
63      }
64
65  }
```

4.4.2. L'implémentation

Implémentez la classe `SecurityConfig` :

SecurityConfig.java

```java
package com.miage.alom.tp.game_ui.config;

①
public class SecurityConfig {

    ②

    ③ ⑤
    PasswordEncoder passwordEncoder(){
    }

    ④ ⑤
    public UserDetailsService userDetailsService() {
    }
}
```

① Cette classe est une `@Configuration`

② Il nous faut probablement un `TrainerService` pour récupérer nos dresseurs

③ Le password encoder est un `BCryptPasswordEncoder`

④ Le `UserDetailsService` doit appeler le `TrainerService` pour récupérer ses objets. On peut faire une classe interne, ou même une lambda !

⑤ Il faut indiquer à Spring de charger ces deux méthodes. Ajoutez l'annotation `@Bean` sur ces méthodes.

Une fois tout cela implémenté, allez faire un tour sur votre IHM http://localhost:9000, vous devriez pouvoir vous connecter avec les noms de dresseurs et leur mot de passe !

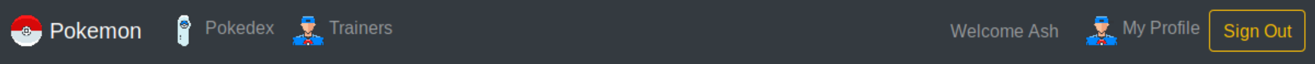
4.5. La page "Mon Profil"

Cette partie est moins guidée. Reportez-vous au cours !

Nous souhaitons créer une page "Mon profil" pour nos dresseurs de Pokemon.

Sur cette page, ils pourraient lister leurs pokemons, et pourquoi pas changer leurs identifiants et mot de passe !

Cette page pourrait être disponible à l'url http://localhost:9000/profile et ressembler à ça :



Ash

Figure 2. La page profil de Sacha

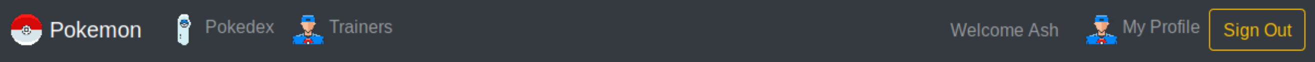
4.5.1. Le @Controller

Développez un controller `ProfileController` ou ajoutez la gestion de l'URL `/profile` dans le `TrainerController`.

Il serait pratique de pouvoir identifier quel est l'utilisateur connecté pour afficher ses informations ! Utilisez le `SecurityContextHolder` pour récupérer le `Principal` connecté, ou récupérez le `Principal` en injection de dépendance (paramètre de méthode de @Controller).

4.5.2. Le `TrainerService`

La méthode `getAllTrainers` pourrait simplement renvoyer les dresseurs différents du dresseur connecté ! La page Trainers ressemblerait donc, pour Sacha à :



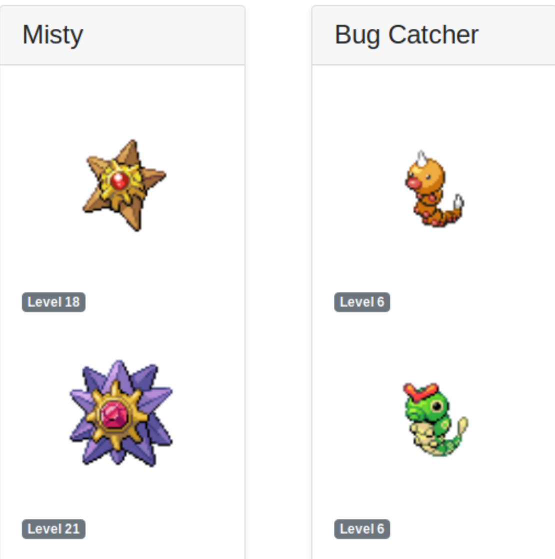


Figure 3. La page Trainers vue par Sacha

4.6. Impacts sur l'IHM avec Mustache

Nous pouvons également utiliser Mustache pour impacter l'IHM de notre application.

4.6.1. Le `ControllerAdvice` et `ModelAttribute`

`ControllerAdvice` est une annotation de Spring, permettant à des méthodes d'être partagées dans l'ensemble des controlleurs. C'est plus propre que de faire de l'héritage :)

L'annotation `@ModelAttribute` permet de déclarer une valeur comme étant systématiquement ajoutée au `Model` ou `ModelAndView` de spring-mvc, sans avoir à le faire manuellement dans une méthode de controller.

Le test unitaire

Implémentez le test unitaire suivant :

fr.univ_lille.alom.game_ui.trainers.ConnectedTrainerControllerAdviceTest.java

```
1 package fr.univ_lille.alom.game_ui.trainers;
2
3 import org.junit.jupiter.api.Test;
4 import org.springframework.security.core.Authentication;
5 import org.springframework.security.core.context.SecurityContextHolder;
6 import org.springframework.security.core.userdetails.User;
7 import org.springframework.web.bind.annotation.ControllerAdvice;
```

```
 8 import org.springframework.web.bind.annotation.ModelAttribute;
 9
10 import static org.junit.jupiter.api.Assertions.*;
11 import static org.mockito.Mockito.mock;
12 import static org.mockito.Mockito.when;
13
14 class ConnectedTrainerControllerAdviceTest {
15
16     @Test
17     void connectedTrainerControllerAdviceTest_shouldBeAControllerAdvice() {
18         assertNotNull(ConnectedTrainerControllerAdvice.class.
   getAnnotation(ControllerAdvice.class));
19     }
20
21     @Test
22     void connectedTrainer_shouldUseModelAttribute() throws NoSuchMethodException {
23         var connectedTrainerMethod = ConnectedTrainerControllerAdvice.class
   .getDeclaredMethod("connectedTrainer");
24         var annotation = connectedTrainerMethod.getAnnotation(ModelAttribute.
   class);
25         assertNotNull(annotation);
26     }
27 }
```

L'implémentation

Implémentez le `ConnectedTrainerControllerAdvice`

SecurityControllerAdvice.java

```
 1 package com.miage.alom.tp.game_ui.controller;
 2
 3 import org.springframework.security.core.context.SecurityContextHolder;
 4 import org.springframework.security.core.userdetails.User;
 5 import org.springframework.web.bind.annotation.ControllerAdvice;
 6 import org.springframework.web.bind.annotation.ModelAttribute;
 7
 8 import java.security.Principal;
 9
10 ①
11 public class ConnectedTrainerControllerAdvice {
12
13     ②
14
15     ③
16     Trainer connectedTrainer(){
17         ④
18     }
19
20 }
```

① Utilisez l'annotation `@ControllerAdvice`

② Vous avez besoin d'un `TrainerService` ici.

③ Cette méthode doit utiliser `@ModelAttribute`

④ Retournez le `Trainer` connecté, en utilisant l'info du `Principal` connecté à l'application

4.6.2. Utilisation

Ajoutez la property suivante dans votre `application.properties`:

application.properties

```
spring.mustache.servlet.expose-request-attributes=true
```

Cette property permet à Mustache de récupérer des attributs de requête dans le `Model` spring. En particulier le token `CSRF` dont nous aurons besoin pour tous les formulaires dans notre application.

Vous pouvez créer une barre de navigation pour votre application, qui affiche le nom de l'utilisateur connecté, ainsi qu'un bouton pour se déconnecter:

navbar.html (ici en bootstrap, utilisez le framework CSS que vous préférez !)

```
1 <nav class="navbar navbar-expand-lg navbar-light bg-light">
2
3     <ul class="navbar-nav mr-auto">
4         <li class="nav-item">
5             <a class="nav-link" href="pokedex">
6                 <img src="/icons/pokedex.png" width="30" height="30" class="d-
  inline-block align-top" alt="">
7                 Pokedex
8             </a>
9         </li>
10        <li class="nav-item">
11            <a class="nav-link" href="trainers">
12                <img src="/icons/player.png" width="30" height="30" class="d-
  inline-block align-top" alt="">
13                Trainers
14            </a>
15        </li>
16    </ul>
17
18    {{#connectedTrainer}}
19    <span class="navbar-text mr-md-3">Welcome {{name}}</span>
20    <ul class="navbar-nav">
21        <li class="nav-item">
22            <a class="nav-link" href="profile">
23                <img src="/icons/player.png" width="30" height="30" class="d-
  inline-block align-top" alt="">
24                My Profile
25            </a>
```

```
26          </li>
27      </ul>
28      <form class="form-inline" action="/logout" method="post">
29          <input type="submit" class="btn btn-outline-warning my-2 my-sm-0" value
   ="Sign Out"/>
30          <input type="hidden" name="{{_csrf.parameterName}}" value=
   "{{_csrf.token}}"/>
31      </form>
32      {{/connectedTrainer}}
33  </nav>
```

5. Pour aller plus loin

- implémentez le changement de mot de passe d'un dresseur de pokemons
- implémentez une page d'inscription au jeu (vous pouvez réutiliser la page 'register' du TP 5 comme point de départ)
- une fois un joueur inscrit, il peut choisir l'un des 3 Pokemons starter (id 1, 4, ou 7) pour constituer son équipe de départ.
- le joueur doit également saisir un mot de passe
- la dernière étape de son inscription consiste à faire un `POST` sur l'API Trainers, pour créer le compte du joueur en base de données.