ALOM - TP 7 - Security

Table of Contents

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 !

1.1. Pré-requis

En pré-requis à ce TP, il faut :

- Avoir terminé la partie 8 du TP Persistence
- Avoir terminé la partie 8.3 du TP GUI (pour la partie 3 de ce TP)
- Avoir terminé la partie 3.1 du TP Interoperability (pour la partie 3.1.2 de ce TP)

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>
```

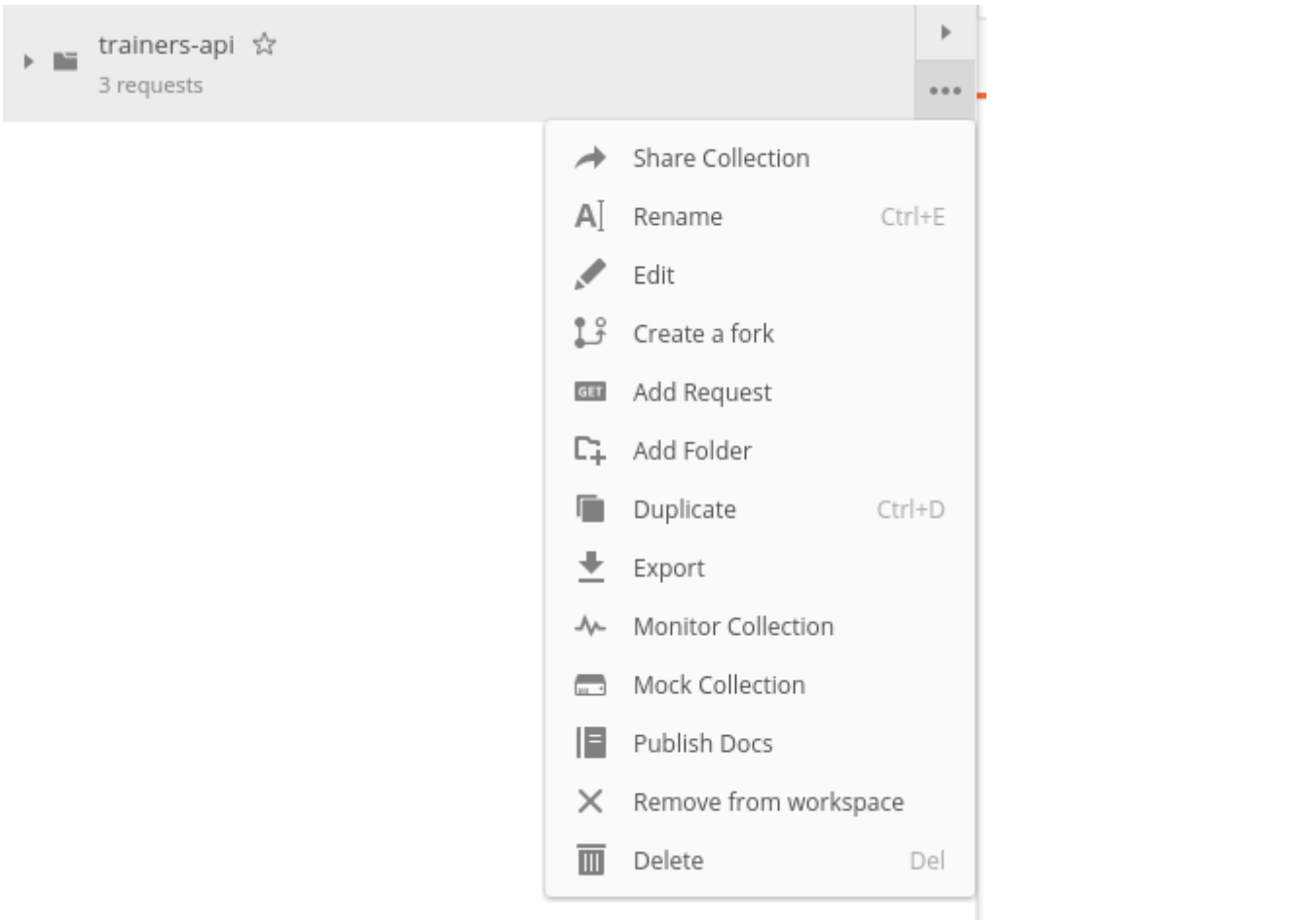
2.3. Votre collection Postman

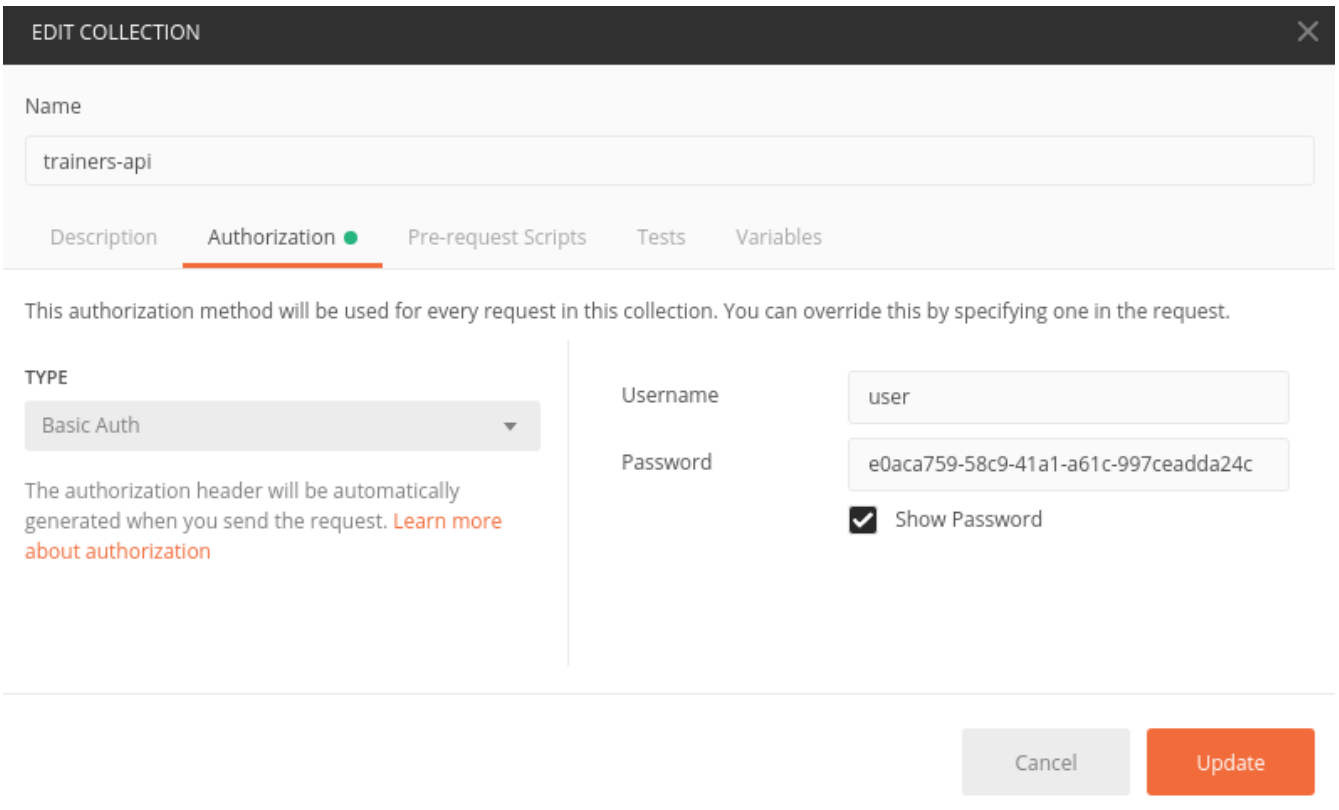
Vos requêtes Postman doivent maintenant renvoyer des erreurs de ce type :

```
{
    "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);
        assertNotNull(responseEntity);
        assertEquals(401, responseEntity.getStatusCodeValue());
    }

    @Test ④
    void getTrainer_withNameAsh_shouldReturnAsh() {
        var ash = this.restTemplate
                .withBasicAuth(username, password) ④
                .getForObject("http://localhost:" + port + "/trainers/Ash",
Trainer.class);

        assertNotNull(ash);
        assertEquals("Ash", ash.getName());
        assertEquals(1, ash.getTeam().size());

        assertEquals(25, ash.getTeam().get(0).getPokemonType());
        assertEquals(18, ash.getTeam().get(0).getLevel());
    }

}
```

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. Désactivation du CSRF, et customisation de la configuration

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

SecurityConfig.java

```
@Configuration ①
public class SecurityConfig {

    @Bean
    SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http.csrf(csrf -> csrf.disable()); ②
        http.authorizeHttpRequests(authorize -> {
            authorize.anyRequest().authenticated(); ③
          }
        );
```

```
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` si vous avez terminé la partie 3 du TP Interoperability. Si ce n'est pas le cas, faites cette partie. Si vous avez déjà terminé la partie 3 du TP précédent, vous devriez pouvoir passer à la partie suivante directement : HTTP Interfaces.

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

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

import org.junit.jupiter.api.Test;
import org.springframework.http.client.support.BasicAuthenticationInterceptor;

import static org.junit.jupiter.api.Assertions.*;

class RestConfigurationTest {

    @Test
    void restTemplate_shouldExist() {
        var restTemplate = new RestConfiguration().restTemplate();

        assertNotNull(restTemplate);
    }

    @Test
    void trainerApiRestTemplate_shouldHaveBasicAuth() {
        var restTemplate = new RestConfiguration().trainerApiRestTemplate();

        assertNotNull(restTemplate);

        var interceptors = restTemplate.getInterceptors();
        assertNotNull(interceptors);
        assertEquals(1, interceptors.size());

        var interceptor = interceptors.get(0);
        assertNotNull(interceptor);

        assertEquals(BasicAuthenticationInterceptor.class, interceptor.getClass());
    }
}
```

L'implémentation

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

RestConfiguration.java

```java
@Configuration
public class RestConfiguration {

    ①

    @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 *defaultHeader* à la construction du `RestCLient` 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`, et utilisez ces valeurs pour générer un header Http avec la méthode `HttpHeaders.encodeBasicAuth()`. Injectez votre header dans le RestClient avec la méthode `requestInitializer` du `RestClient.Builder`.

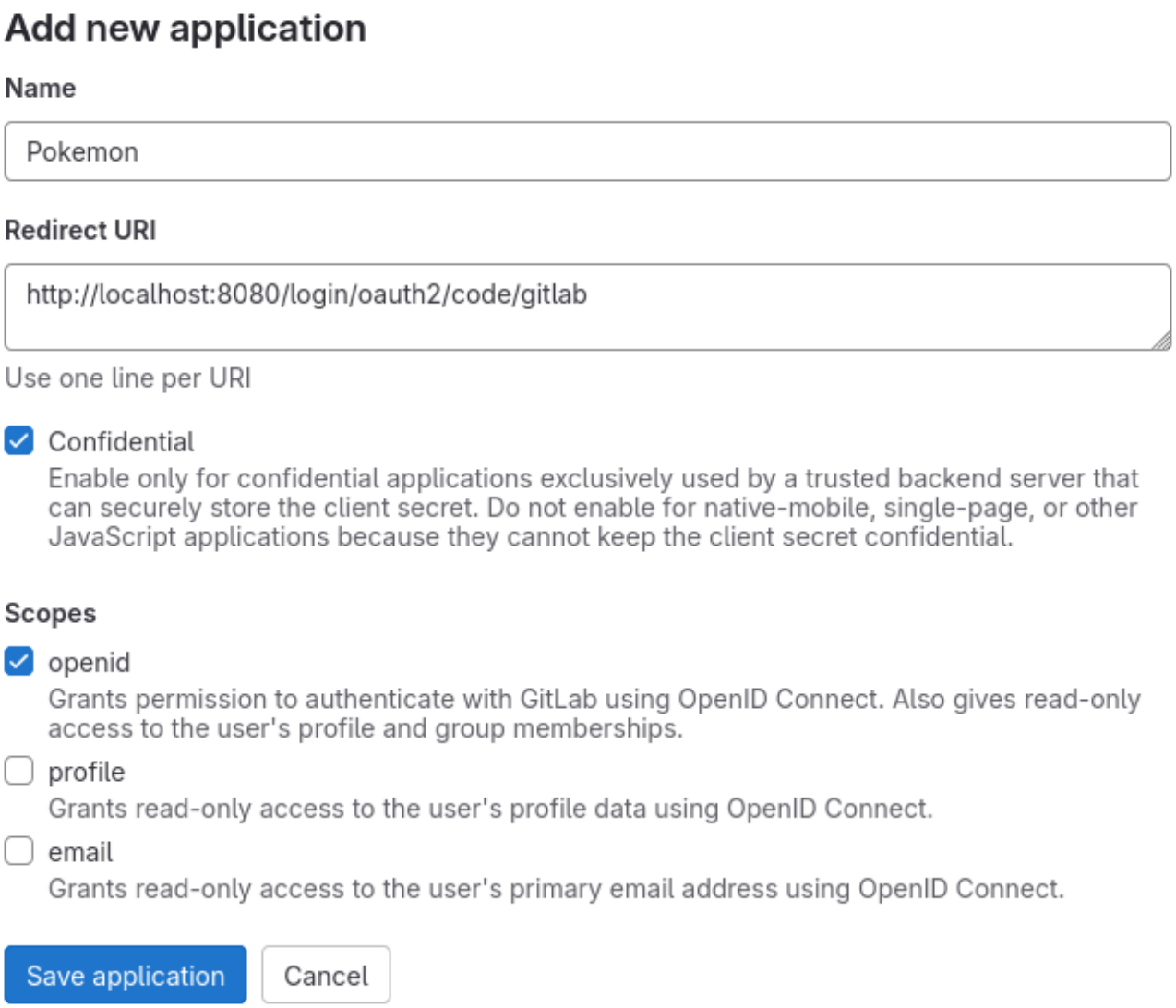
4. Sécuriser `game-ui` avec un accès OpenID Connect

OpenID Connect, abrégé en OIDC, est un protocole d'authentification moderne, permettant de déléguer l'authentification à un fournisseur d'identités externe. C'est à travers ce protocole qu'on peut implémenter l'authentification avec un compte Google, GitHub, Microsoft, etc.

Nous allons maintenant utiliser une authentification OIDC sur notre application, à travers le GitLab de l'université !

4.1. Créer une "application" dans GitLab

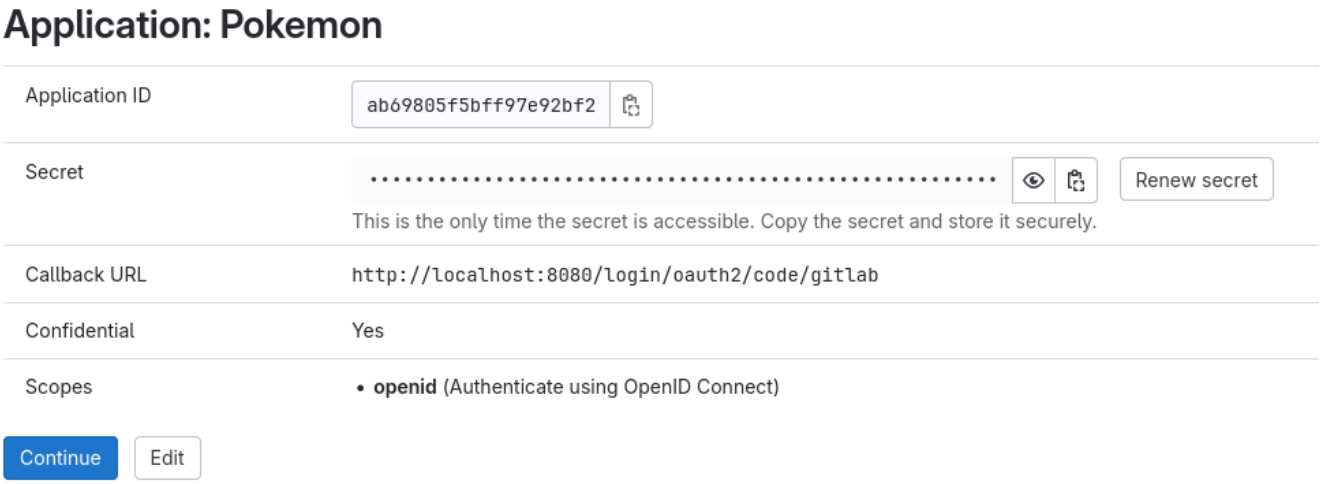
Rendez-vous sur la page de votre profil GitLab, dans l'onglet *Applications* : https://gitlab.univ-lille.fr/-/user_settings/applications, et créez une nouvelle application :



Cochez bien le scope `openid`, et utilisez la redirect URI suivante : `http://localhost:8080/login/oauth2/code/gitlab`.

Pensez à adapter le port de la redirect URI si besoin (8081 ?).

Prenez note de l'*Application ID* et du *Secret* qui vous sont donnés.



4.2. Configuration de spring-security

Commençons par ajouter `spring-security` et `spring-boot-starter-oauth2-client` au `pom.xml` de `game-ui`.

pom.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 !

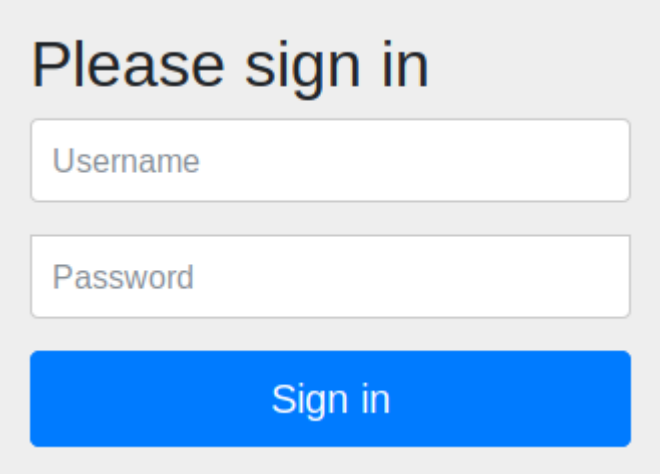


Figure 1. La page de login par défaut de spring-security !

Pour rappel, le user par défaut de spring-security est `user` et le mot de passe par

défaut apparaît dans les logs !

4.3. Endpoint *whoami*

Ajoutez un `RestController` dans *game-ui*, exposant le `Principal` connecté à l'application :

```
@GetMapping("/api/whoami")
Object whoami(Authentication authentication){
    return authentication.getPrincipal();
}
```

Connectez-vous avec les credentials par défaut de Spring Security, et dirigez vous sur ce endpoint pour observer votre user.

4.4. Personnalisation de spring-security

Nous ne voulons pas utiliser un login par défaut, mais bien se loguer avec les comptes GitLab.

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

Ajoutez la dépendance à `spring-boot-starter-oauth2-client` dans votre `pom.xml` :

pom.xml

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-client</artifactId>
</dependency>
```

La configuration de Spring Boot pour OIDC passe principalement par le positionnement de properties.

Insérez les properties suivantes dans l'application.properties de game-ui, en alimentaire le client Id et client secret avec les *Application ID* et *Secret* fournis par GitLab :

application.properties

```
spring.security.oauth2.client.registration.gitlab.client-id=
spring.security.oauth2.client.registration.gitlab.client-secret=
spring.security.oauth2.client.registration.gitlab.scope=openid

spring.security.oauth2.client.provider.gitlab.issuer-uri=https://gitlab.univ-lille.fr
```

Les properties possibles sont détaillées dans la doc de Spring Security

Pour activer l'utilisation de OAuth2 / OIDC, il faut personnaliser la configuration de Spring Boot, pour y enregistrer un `SecurityFilterChain` utilisant l'authentification OIDC, avec la méthode

`oauth2Login()`.

Reportez-vous à cet exemple de la documentation de Spring Security :

OAuth2LoginSecurityConfig.java

```
@Configuration
@EnableWebSecurity
public class OAuth2LoginSecurityConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http
            .authorizeHttpRequests(authorize -> authorize
                .anyRequest().authenticated()
            )
            .oauth2Login(withDefaults());
        return http.build();
    }
}
```

4.5. La page "Mon Profil" et la création du Trainer !

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

Lors du premier login d'un nouvel utilisateur, il faut lui créer un objet "Trainer" dans notre API.

Il est possible de détecter la connexion en customisant l'appel à `oauth2Login` :

filename.java

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http
        .authorizeHttpRequests(authorize -> {
            authorize.anyRequest().authenticated();
        })
        .oauth2Login(customize -> {
            customize.successHandler((request, response, authentication) -> {
                // create a Trainer here if it does not exists
                System.out.println(authentication.getName() + " is connected !");
            });
        });
    return http.build();
}
```

Implémentez l'appel à l'API Trainer, qui vérifie si un Trainer existe déjà pour l'utilisateur authentifié. Si aucun Trainer n'existe, faites un appel `POST` à l'API Trainer pour en créer un nouveau !

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

Sur cette page, ils pourraient lister leurs Pokemons.

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

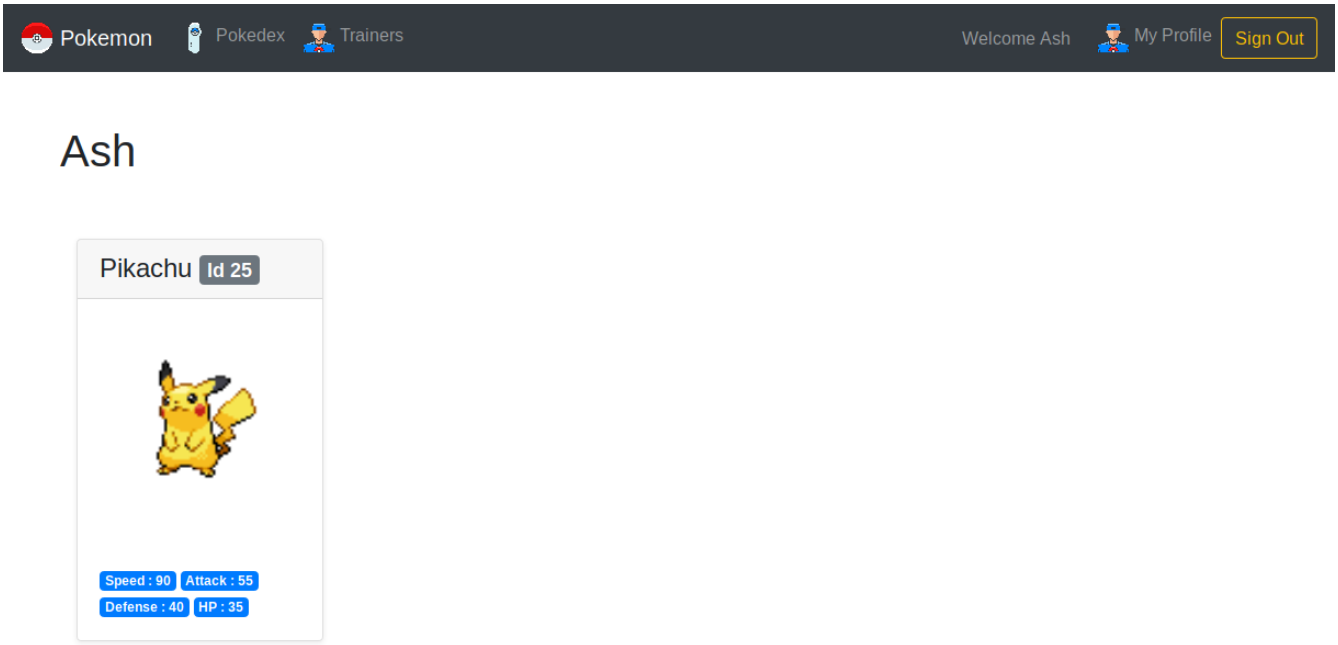


Figure 2. La page profil de Sacha

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).

Lorsque Spring Security est configuré pour utiliser l'authentification OIDC, le `Principal` est de type `OidcUser`. Avec ce type, vous pourrez accéder aux attributs de l'utilisation (nom, email, etc.).

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 :

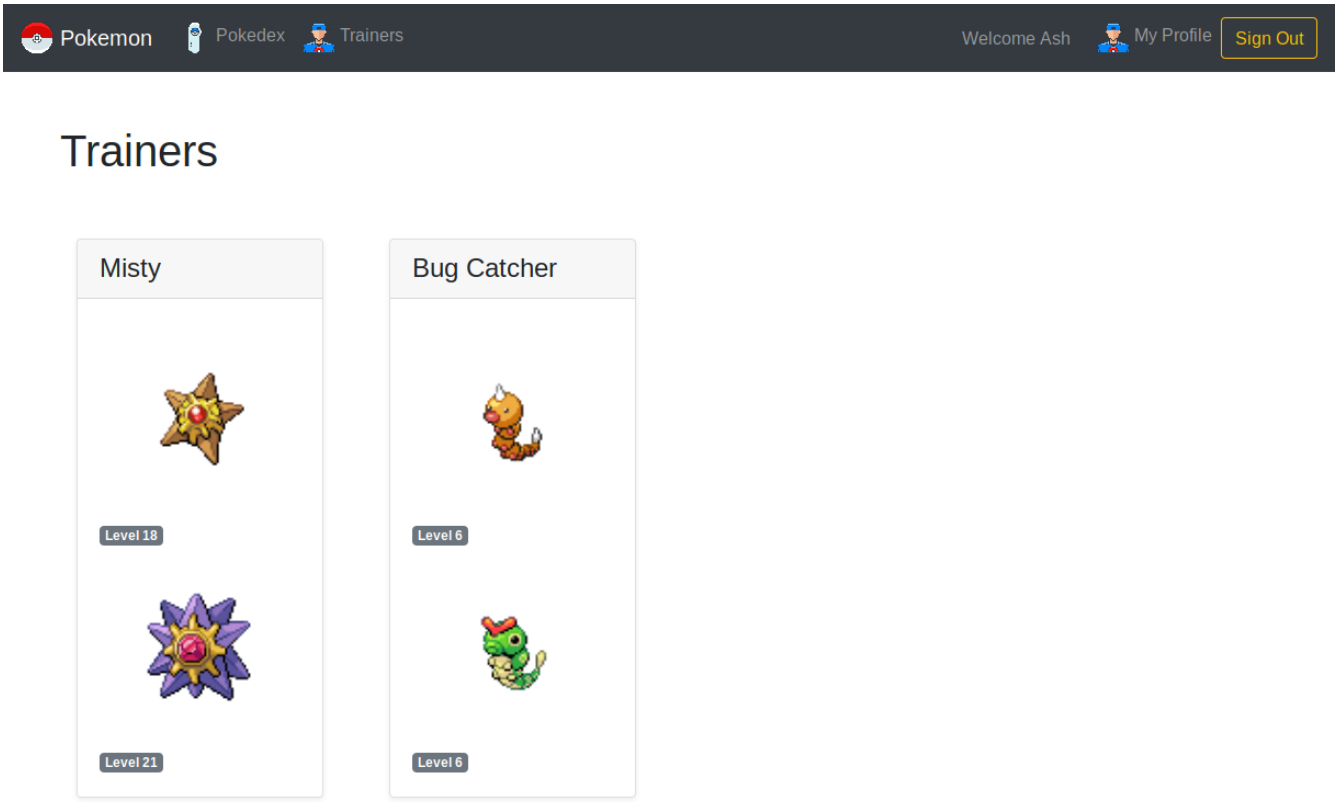


Figure 3. La page Trainers vue par Sacha

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;
```

```java
import org.springframework.web.bind.annotation.ModelAttribute;

import static org.junit.jupiter.api.Assertions.*;
import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.when;

class ConnectedTrainerControllerAdviceTest {

    @Test
    void connectedTrainerControllerAdviceTest_shouldBeAControllerAdvice() {
        assertNotNull(ConnectedTrainerControllerAdvice.class.getAnnotation
(ControllerAdvice.class));
    }

    @Test
    void connectedTrainer_shouldUseModelAttribute() throws NoSuchMethodException {
        var connectedTrainerMethod = ConnectedTrainerControllerAdvice.class
.getDeclaredMethod("connectedTrainer");
        var annotation = connectedTrainerMethod.getAnnotation(ModelAttribute.
class);
        assertNotNull(annotation);
    }
}
```

L'implémentation

Implémentez le `ConnectedTrainerControllerAdvice`

SecurityControllerAdvice.java

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

import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.core.userdetails.User;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ModelAttribute;

import java.security.Principal;

①
public class ConnectedTrainerControllerAdvice {

    ②

    ③
    Trainer connectedTrainer(){
        ④
    }

}
```

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 !)

```
<nav class="navbar navbar-expand-lg navbar-light bg-light">

    <ul class="navbar-nav mr-auto">
        <li class="nav-item">
            <a class="nav-link" href="pokedex">
                <img src="/icons/pokedex.png" width="30" height="30" class="d-
inline-block align-top" alt="">
                Pokedex
            </a>
        </li>
        <li class="nav-item">
            <a class="nav-link" href="trainers">
                <img src="/icons/player.png" width="30" height="30" class="d-
inline-block align-top" alt="">
                Trainers
            </a>
        </li>
    </ul>

    {{#connectedTrainer}}
    <span class="navbar-text mr-md-3">Welcome {{name}}</span>
    <ul class="navbar-nav">
        <li class="nav-item">
            <a class="nav-link" href="profile">
                <img src="/icons/player.png" width="30" height="30" class="d-
inline-block align-top" alt="">
                My Profile
            </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 un flow d'inscription au jeu (vous pouvez réutiliser la page 'register' du TP 5 comme point de départ) :
  - dans le `successHandler` du customizer du `.oauth2Login`, il est possible de faire des `response.sendRedirect` pour rediriger l'utilisateur sur une page précise après son login
- une fois un joueur connecté, il peut choisir l'un des 3 Pokemons starter (id 1, 4, ou 7) pour constituer son équipe de départ, s'il ne possède pas encore de Pokémon dans son équipe
- la dernière étape de son inscription consiste à faire un `POST` sur l'API Trainers, pour modifier le Trainer du joueur en base de données.