

# Spring, sécurité et authentification

## GLG 203/Architectures Logicielles Java

Serge Rosmorduc  
serge.rosmorduc@lecnam.net  
Conservatoire National des Arts et Métiers

2019–2020

# Démonstrations

```
https:  
//gitlab.cnam.fr/gitlab/glg203_204_demos/07_spring_security.git
```

Introduisent le fil rouge de cette session : un petit forum.

## Définition

Framework permettant de gérer la connexion à une application et fournissant un certain nombre de protections.

- permet de varier la source d'authentification ;
- gère des sources externes (OAuth) ;
- facilités pour Spring MVC
  - ▶ pour la connexion ;
  - ▶ pour protéger d'attaques courantes.
- gère l'accès à des ressources :
  - ▶ URL ;
  - ▶ méthodes (programmation orientée aspect) ;
  - ▶ objet + méthode (ACL).

# Plan

- quelques éléments ;
- identification (*authentication*) (interne et externe) ;
- configuration de Spring Security ;
- autorisations (*authorization*)
- externalisation du login : OAuth ;
- authentification et REST ; JWT ;  
<https://www.javainuse.com/spring/jwt>
- https ; let's encrypt.

# Éléments de sécurité

# Identification et autorisations

*authentication vs authorization*

## Définition

Identification/**Authentication** : qui êtes vous ?

## Définition

Autorisations/authorization : qu'avez-vous le droit de faire ?

# Identification

- typiquement login/password ;
- interne (bd locale, mémoire, fichier) ;
- externe : serveur LDAP, OpenID, OAuth ...
- système de jeton / connexion persistante ;
- autres : biométrie ...
- plusieurs mécanismes distincts selon les contextes (appli Web classique, appli web REST...)

Risque : se faire voler sa session ou son mot de passe.

# Autorisation

grain plus ou moins fin :

- orientée URL : une *partie* du site est protégée ;
- orientée opération : certaines opérations sont protégées ;
- orientée données : certaines opérations sur certaines données sont protégées ;
- ACL : access control list.

# Quelques attaques

# Injection de SQL

- l'attaquant entre une donnée qui contient du code SQL :

```
login = "a'; drop table user; --"
```

```
sql= "select * from user where login = '" + login "'";
```

- la cible ne gère pas correctement le code en question et le code est exécuté.

## Contre-mesure

- éviter de construire des requêtes par concaténation ;
- utiliser des expressions préparées ;
- plus ou moins automatique avec JPA.
- les langages fortement typés aident un peu : un int est un int...

# Injection de Javascript

- le site affiche du texte entré par les utilisateurs ;
- un utilisateur entre du texte contenant du code javascript ;
- celui-ci est exécuté sur les machines des autres utilisateurs quand le texte entré y est affiché...
- exemple : un utilisateur choisit comme login :

```
foo<script>>window.location='http://pirate.org'</script>
```

- l'administrateur visualise les logins sur le site : il est envoyé sur `pirate.org` ;
- le site `pirate.org` se présente comme le site d'origine, et simule une déconnexion. L'administrateur (novice) retape son mot de passe, qui est maintenant connu de `pirate.org`.

# Injection de JavaScript

- autre attaque possible
  - ▶ viser une page particulière ;
  - ▶ injecter le code javascript qui exécute automatiquement une action de la page ;
- attention, code javascript n'est pas limité à la balise `<script>` !
- injection possible comme valeur d'url (href) comme lien :

```
<a href="javascript:window.location='http://www.cnam.fr'">hello</p>
```

## parades

- ▶ Protection du texte *lors de l'affichage* ;
- ▶ si des liens sont autorisés, définition stricte de ce qui est possible ;

# Cross-Site Request Forgery

- On attaque un site A par injection javascript sur un site tiers B ;
- fonctionne même si A est protégé contre l'injection javascript !!
- principe : l'utilisateur est connecté simultanément sur A et B ;
- le code javascript qui tourne sur chaque site est isolé...
- mais une requête POST qui effectuée en javascript sur la page de B vers le site A enverra les cookies de connexion du site A !!
- le pirate peut donc réaliser des actions sur A s'il a modifié B, et que l'utilisateur est connecté simultanément sur A et B ;
- peut se combiner à du *social engineering* pour obtenir la condition précédente ;
- Attaque très répandue actuellement.

# Cross-Site Request Forgery (CSRF)

## parade

- mettre un champ aléatoire (caché) dans les formulaires POST ;
- lorsqu'on reçoit une requête, vérifier la valeur du champ ;
- **danger : si le script attaquant peut lire la page du formulaire** → *Same-origin policy*

# Session Fixation

- le vol d'id de session est difficile...
- mais si on envoie un lien sur le site cible avec `...?sessionId=1`
- l'utilisateur qui le suivra sera connecté avec ce numéro de session ;
- qui sera plus simple à voler !

## parade

- gérer explicitement le début de session, en forçant un nouveau numéro si nécessaire.
- automatique en Spring.

## Autres Exemples d'attaques

Une vidéo intéressante :

<https://www.youtube.com/watch?v=0dgmeTy7X3I>

- Outre les attaques précédentes, montre en particulier des attaques qui utilisent la tendance des navigateurs à interpréter le HTML incorrect ;
- privilégier des approches *white lists*, et contrôler le texte produit ;
- importance de produire du html correct (jusque dans le contenu des liens) ;

# Prélude : Domain Specific Languages

# Domain Specific Languages

## Définition

langage informatique adapté à un usage particulier

En Java, généralement simulé en combinant :

- des **Fluent Interfaces** (interfaces fluides ou chaînées) ;
- des méthodes statiques.
- avantage : langage expressif et adapté au domaine, facile à lire ;
- inconvénient : détail technique caché, pas forcément facile à suivre.

## Exemple

Cette classe représente une url :

```
public class MyUrl {
    public MyUrl(
        String protocol ,
        String server ,
        int port ,
        List<String> path ,
        LinkedHashMap<String , String> parameters) {
    }
    ...
}
```

- complète ...
- mais constructeur pénible à utiliser ;
- pas facile de savoir quel argument est quoi ;
- ni lesquels peuvent être négligés.

# Avec un Domain-Specific Language

Cas simple :

```
MyUrl url = new URLBuilder()  
            .protocol("http")  
            .server("www.cnam.fr")  
            .build();  
System.out.println(url);
```

## Avec un Domain-Specific Language

Cas complexe :

```
MyUrl url = new URLBuilder()
    . protocol ("https")
    . server ("www.mysearch.com")
    . port (443)
    . addPath ("do")
    . addPath ("search")
    . addParameter ("query", "langage java")
    . addParameter ("lang", "fr")
    . build ();
System.out.println (url);
```

- sens de chaque élément explicite;
- logique de construction plus proche du modèle.

# Implémentation en java

- pattern builder ;
- Idée : utiliser une classe pour chaque étape de la construction.
- voir démo fluentInterface pour le code.

```
public class URLBuilder {
    public URLBuilder protocol(String protocol) {...}
    public ServerPart server(String server) {}
    public class ServerPart {
        public ServerPart port(int port) {}
        public PathPart addPath(String pathPart) {}
        public ParameterPart addParameter(String name, String value) {}
        public MyUrl build() {}
    }
    public class PathPart {
        public PathPart addPath(String pathPart) {}
        public ParameterPart addParameter(String name, String value) {}
        public MyUrl build() {}
    }
    public class ParameterPart {
        public ParameterPart addParameter(String name, String value) {}
        public MyUrl build() {}
    }
}
```

# Implémentation en java

- les méthodes retournent :
  - ▶ l'objet courant si on reste à la même étape (ex : `addPath` dans `PathPart`)
  - ▶ l'objet qui représente l'étape suivante sinon (ex. `addPath` dans `ServerPart`)
  - ▶ on peut travailler sur des données communes.

- plusieurs types :
  - ▶ langage à part entière : COBOL ;
  - ▶ externe au langage principal (ex. SQL depuis Java) ;
  - ▶ codé *dans* le langage principal
    - ★ en utilisant des interfaces fluides : Spring security, streams de java...
    - ★ des méthodes statiques (assertEquals de JUnit)
    - ★ un mélange de tout ça.

# Présentation de l'application d'exemple

## L'application d'exemple

Un petit forum en ligne avec quelques règles de sécurité.

- tout le monde peut voir les messages
- seul un utilisateur connecté peut créer un message
- seul un administrateur peut créer des utilisateurs
- un message ne peut être détruit que par son auteur ou un administrateur

On essaie de montrer le plus possible de caractéristiques de Spring Security :

- mise en place d'un login par JPA ; hachage du mot de passe ;
- utilisation de la protection contre le csrf ;
- protection par URL et par méthode ;

# Mise en place de Spring Security

# Dans Spring boot

Dans le fichier gradle :

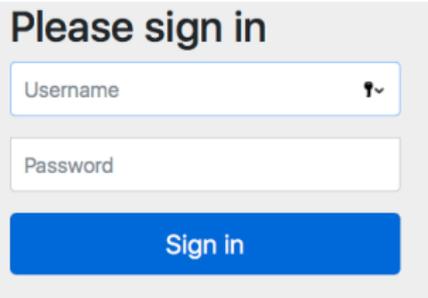
```
dependencies {  
    implementation  
        'org.springframework.boot:spring-boot-starter-security'  
    ...  
    testImplementation  
        'org.springframework.security:spring-security-test'  
}
```

# Spring boot

- met en place l'authentification Web.
- utilisateur par défaut configuré à partir de `application.properties` :

```
spring.security.user.name=user # Default user name.  
spring.security.user.password= # Password for the default user name.  
spring.security.user.roles= # Granted roles for the default user name.
```

- configuration par défaut inutilisable en production.
- page de login :



The image shows a login form with a light gray background. At the top, the text "Please sign in" is displayed in a bold, black font. Below this, there are two input fields: the first is labeled "Username" and has a small eye icon to its right; the second is labeled "Password". At the bottom of the form is a prominent blue button with the text "Sign in" in white.

## Classe de paramétrage

Le gros du paramétrage se fait dans une classe de configuration qui étend `WebSecurityConfigurerAdapter`

```
@Configuration
```

```
public class WebSecurityConfig
    extends WebSecurityConfigurerAdapter {

    // Pour l'identification
    protected void configure(
        AuthenticationManagerBuilder auth)
        throws Exception {...}

    // Pour les ressources
    public void configure(WebSecurity web)
        throws Exception {...}

    // le gros de la configuration est ici
    protected void configure(HttpSecurity http)
        throws Exception {...}
}
```

# Paramétrage de l'identification

# Paramétrage de l'identification

- Se fait dans la méthode `configure(AuthenticationManagerBuilder)`
- si elle n'est pas définie, identification par défaut (un utilisateur, paramétré par `application.properties`)
- le paramètre `auth` sert à mettre en place un *ou plusieurs* systèmes d'identification : des `AuthenticationManager`

# Principes

- chaîne de responsabilité d'AuthenticationManager, qui valident, ignorent, ou rejettent une Authentication.
- la chaîne est gérée par des ProviderManager ;
- ... et configurée par AuthenticationManagerBuilder (qu'on peut injecter).

# Description d'un utilisateur

- interface `UserDetails` ;
- et généralement son implémentation  
`org.springframework.security.core.userdetails.User` ;

## Méthodes de `UserDetails`

`getUsername` le login de l'utilisateur ;

`getPassword` son mot de passe (normalement haché) ;

`getAuthorities` la collection des droits de l'utilisateur  
(`GrantedAuthority`) ;

`isAccountNonExpired...` diverses caractéristiques possibles.

# Les droits GrantedAuthority

- Interface avec une seule méthode :

```
public interface GrantedAuthority
    extends Serializable {
    String getAuthority();
}
```

- renvoie une String qui est le nom du droit ;
- ou null... si le droit est plus complexe à décrire que par une simple String
- typiquement, on utilise SimpleGrantedAuthority ;
- exemple :

```
GrantedAuthority role_admin =
    new SimpleGrantedAuthority("ROLE_ADMIN");
```

# Sémantique des droits

- « grain » variable ;
- une `GrantedAuthority` peut décrire :
  - ▶ un droit sur un type d'objet : `CAN_CREATE_TOPIC` ;
  - ▶ ou un rôle (`ROLE_USER`, `ROLE_EDITOR`, `ROLE_ADMIN`, ...);
  - ▶ convention : commencer les noms de rôles par « `ROLE_` » ;
- elle ne décrit pas un droit sur un objet précis : le droit d'éditer le message d'id 50 ;
- ça, ce serait les ACL (Access Control List).
- par défaut, les droits sont indépendants ;
- il est possible de les hiérarchiser, mais c'est plus de travail.

# InMemoryUserDetailsManager

Généralement en test ou en début de développement :

```
@Configuration
```

```
@EnableWebSecurity
```

```
public class WebSecurityConfig extends
```

```
    WebSecurityConfigurerAdapter {
```

```
    @Override
```

```
    public void configure(
```

```
        AuthenticationManagerBuilder auth) throws Exception {
```

```
    auth.inMemoryAuthentication()
```

```
        .withUser("user").password("user").roles("USER")
```

```
        .and()
```

```
        .withUser("admin").password("admin")
```

```
            .roles("USER", "ADMIN");
```

```
    }
```

```
}
```

Définit directement les utilisateurs et leurs rôles.

# Création d'utilisateur

Ici :

```
auth.inMemoryAuthentication()  
    .withUser("admin")  
    .password("admin")  
    .roles("ADMIN", "USER")
```

Crée l'utilisateur admin, avec le mot de passe non haché « admin », et les deux rôles ADMIN et USER, qui correspondent aux `GrantedAuthorities` `ROLE_ADMIN` et `ROLE_USER`.

# Builder de User

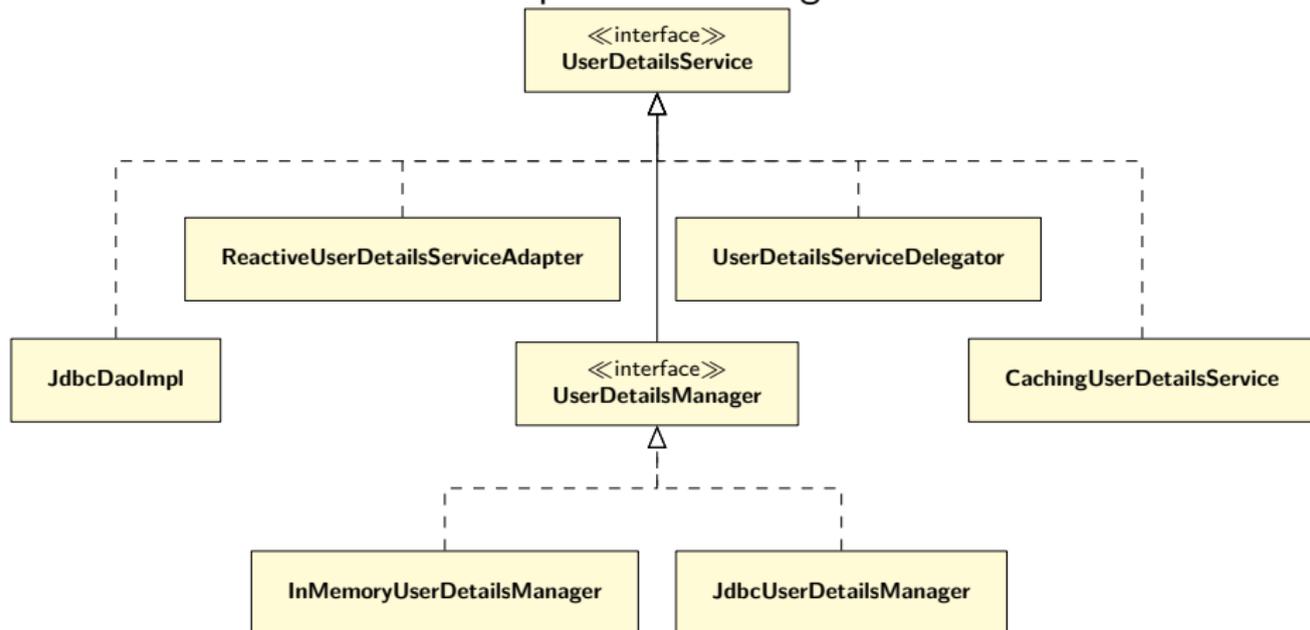
Builder permettant de créer des User en utilisant l'encodeur de mots de passe par défaut :

```
UserBuilder builder = User.withDefaultPasswordEncoder();
User u = builder.username("user")
               .password("password")
               .roles("USER").build();
```

Évite d'oublier de hacher le mot de passe.

# UserDetailsService

Sert à trouver un utilisateur à partir de son login :



(ce schéma ne sert pas à grand chose, mais je me suis embêté à le faire, alors je le laisse là).

# Le codage des mots de passe

- Les mots de passe sont normalement traités pour ne pas être lisible ;
- le système de codage doit être **lent** pour réduire les attaques « brute-forces » ;
- il utilise une part aléatoire, le « sel » pour que le même mot de passe n'ait pas toujours la même forme ;
- conseil actuel : utiliser BCrypt.

## Mise en place de l'encodeur de mots de passe

Définition d'un bean passwordEncoder :

```
@Configuration
public class WebSecurityConfig
    extends WebSecurityConfigurerAdapter {
    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
}
```

... mise en place dans auth :

```
@Override
protected void configure(
    AuthenticationManagerBuilder auth) throws Exception {
    auth.passwordEncoder(passwordEncoder()).... ;
}
```

## Autres AuthenticationManagers

- `DaoAuthenticationProvider` créé par `auth` avec la commande `jdbcAuthentication` ;  
fournit un schéma de BD pour coder les `UserDetails`.
- `LdapAuthenticationProvider`, créé par `auth` avec la commande `ldapAuthentication` ;
- ... beaucoup d'autres (voir exemple OAuth2 dans les démos) ;
- plus général : fourniture de votre propre `UserDetailsService`

# UserDetailsService et UserDetailsManager

Lecture seule, et lecture/écriture...

```
public interface UserDetailsService {  
    /**  
     * Locates the user based on the username.  
     * @param username  
     * @return a fully populated user record (never null)  
     * @throws UsernameNotFoundException if user could not be found or  
     *         has no GrantedAuthority  
     */  
    UserDetails loadUserByUsername(String username)  
        throws UsernameNotFoundException;  
}
```

```
public interface UserDetailsManager extends UserDetailsService {  
    void createUser(UserDetails user);  
    void updateUser(UserDetails user);  
    void deleteUser(String username);  
    void changePassword(String oldPassword, String newPassword);  
    boolean userExists(String username);  
}
```

## Création d'un UserDetailsService JPA (ou pas)

```
@Service
```

```
@Transactional
```

```
public class AuteurService implements UserDetailsService {
```

```
    @Autowired
```

```
    AuteurRepository auteurRepository;
```

```
    @Autowired
```

```
    PasswordEncoder passwordEncoder;
```

```
    @Override
```

```
    public UserDetails loadUserByUsername(String username)  
        throws UsernameNotFoundException {
```

```
        Auteur auteur = auteurRepository.findById(username)
```

```
            .orElseThrow(() -> new UsernameNotFoundException("inconnu"));
```

```
        List<GrantedAuthority> authorities = new ArrayList<>();
```

```
        authorities.add(RolesRepository.USER);
```

```
        if (auteur.isAdmin()) {
```

```
            authorities.add(RolesRepository.ADMIN);
```

```
        }
```

```
        return new User(auteur.getLogin(), auteur.getPassword(), authorities);
```

```
    }
```

```
...
```

## Sauvegarde d'un utilisateur

- Pour le forum, utilisateur = auteur
- On pourrait avoir deux classes distinctes (l'admin est-il un auteur?)
- important : **bien hacher les mots de passe.**

```
@Service
@Transactional
public class AuteurService implements UserDetailsService {
    @Autowired AuteurRepository auteurRepository;
    @Autowired PasswordEncoder passwordEncoder;
    public void sauverAuteur(String login, String pwd, String signature) {
        Auteur auteur = buildAuteur(login, pwd, signature, false);
        auteurRepository.save(auteur);
    }

    /**
     * Crée un auteur (et encode son mot de passe).
     */
    private Auteur buildAuteur(String login, String pwd,
                               String signature, boolean isAdmin) {
        return new Auteur(login, passwordEncoder.encode(pwd),
                          signature, isAdmin);
    }
}
```

# Autorisations

## Ressources communes

Pour des raisons d'efficacité, on court-circuite le système d'autorisation pour celles-ci :

```
@Configuration
public class WebSecurityConfig
    extends WebSecurityConfigurerAdapter {
    @Override
    public void configure(WebSecurity web) throws Exception{
        web.ignoring()
            .antMatchers("/css/**")
            .antMatchers("/picture/**");
    }
}
```

- `web.ignoring()` : désactive les autorisations sur ce qui suit ;
- `antMatcher("/css/**")` : tout ce qui est dans le dossier css.

## Configuration de httpSecurity

```
@Configuration
public class WebSecurityConfig
    extends WebSecurityConfigurerAdapter {
@OVERRIDE
protected void configure(HttpSecurity http)
    throws Exception {
http
    .authorizeRequests()
        .mvcMatchers("/admin/**").hasRole("ADMIN")
        .mvcMatchers("/zoneConnectee").authenticated()
        .anyRequest().permitAll()
    .and()
        .formLogin().permitAll()
    .and()
        .anonymous().permitAll()
    ;
}
```

paramètre l'objet http

# Paramétrage

```
http
    .authorizeRequests()
        .mvcMatchers("/admin/**").hasRole("ADMIN")
        .mvcMatchers("/zoneConnectee").authenticated()
        .anyRequest().permitAll()
    .and()
        .formLogin().permitAll()
    .and()
        .anonymous().permitAll()
;
```

- interface fluide ;
- série de règles séparées par `and()` (qui renvoie le `http`)
- typiquement : une commande sur le `http` commence une règle ;
- éventuellement suivie d'un paramétrage :  
`.formLogin().loginPage("/monLogin")`

## authorizeRequests()

- Commence une *suite ordonnées* de règles de la forme : *Sélecteur + Autorisation*
- si une url correspond à un chemin :
  - ▶ si l'autorisation est vraie, l'action est acceptée
  - ▶ si l'autorisation est fausse, l'action est refusée
- sinon, on continue dans la liste.
- la séquence suivante est donc absurde :

```
http.authorizeRequests()  
    .mvcMatchers("/**").permitAll()  
    .mvcMatchers("/admin/**").hasRole("ADMIN")
```

la première règle reconnaît tous les chemins...

## Quelques sélecteurs

- `MvcMatcher` reconnaît un ou plusieurs chemin vu par Spring MVC : url + méthode (GET, POST... optionnelle) ;
- `antMatcher` similaire, moins puissant, moins sûr ;
- `anyRequest()` reconnaît n'importe quelle requête ;

## Quelques autorisations

`hasRole(X)` vraie si l'utilisateur a le rôle X ;

`hasAnyRole(X1,X2...)` vraie si l'utilisateur a le rôle X1 ou X2 ou ... ;

`permitAll()` toujours vraie ;

`denyAll()` toujours fausse ;

`anonymous()` l'utilisateur est anonyme ;

`authenticated()` l'utilisateur est connecté ;

`hasIPAddress(X)` vraie si l'utilisateur a l'adresse IP X ;

`access(S)` vraie si la formule décrite par S est vraie ;

`not()` inverse l'autorisation suivante ;

## access()

Permet d'écrire des règles d'autorisation plus complexes :

```
.antMatchers("/db/**")  
    .access("hasRole('ADMIN') and hasRole('DBA')")
```

*(doc. spring security, 11.4)*

# Méthodes de HttpServletRequest

- `getRemoteUser()` : `String`
- `getUserPrincipal()` : `Principal`
- `isUserInRole(role : String)` : `boolean`
- `login(login : String, password : String)`
- `logout()`

## Protection des méthodes (et des classes)

À activer avec :

```
@Configuration
```

```
@EnableGlobalMethodSecurity(
```

```
    prePostEnabled = true ,
```

```
    securedEnabled = true ,
```

```
    jsr250Enabled = true
```

```
)
```

```
public class WebSecurityConfig
```

```
    extends WebSecurityConfigurerAdapter {
```

`securedEnabled` autorise l'annotation `@Secured`

`prePostEnabled` autorise les annotations `@PreAuthorize`,  
`@PostAuthorize`, `@PreFilter` et `@PostFilter`;

`jsr250Enabled` ensemble d'annotations non spring.

# Protection des méthodes et des classes

## Important

Ces règles décorent souvent des contrôleurs...  
mais aussi n'importe quel type de composant : les services, les  
Repositories...

## Utilisation de @Secured

- Généralement sur des méthodes de contrôleurs, ou sur les contrôleurs
- Prend comme argument une liste d'autorités :

```
@Controller
@RequestMapping("/message/creer")
@Secured({ "ROLE_USER" })
public class CreerMessageController {
    ...
}
```

ici, seuls les utilisateurs de rôle USER peuvent créer des messages ;

## Utilisation de @PreAuthorize

Intérêt : prend une expression qui peut être assez complexe.

Ici : injection du paramètre `m` dans l'expression.

```
@Service @Transactional
```

```
public class ForumService {
```

```
    @Autowired
```

```
    AuteurRepository auteurRepository;
```

```
    @Autowired
```

```
    MessageRepository messageRepository;
```

```
/**
```

```
 * Détruit un message.
```

```
 */
```

```
@PreAuthorize(
```

```
    "isAuthenticated() and (hasRole('ADMIN') "
```

```
    + "or #m.auteur.login == principal.username)")
```

```
public void deleteMessage(Message m) {
```

```
    messageRepository.deleteById(m.getId());
```

```
}
```

```
}
```

## Accès « manuel » aux informations

On peut injecter `Principal` et `Authentication` comme argument d'un contrôleur.

`Principal` essentiellement utile pour `principal.getName()` ;

`Authentication` permet d'avoir accès au `User` :

```
if (authentication != null) {  
    user = (User) authentication.getPrincipal();  
}
```

cet objet-là permet de récupérer simplement les droits, etc...

## Question ouverte (pour l'auteur de ces lignes)

À partir de quel moment cela devient-il plus compliqué à maintenir que d'écrire ses règles comme des tests dans le code ?

# Facilités pour Thymeleaf

# Mise en place

Ajouter la dépendance

```
implementation 'org.thymeleaf.extras:thymeleaf-extras-springsecurity5'
```

dans `build.gradle`

## annotations supplémentaires

`sec:authorize` affiche de manière conditionnelle un élément :

```
<a sec:authorize="isAuthenticated()"
  th:href="@{/logout}">se déconnecter</a>
```

`sec:authorize-url` affichage conditionnel, si l'URL est autorisée à l'utilisateur :

```
<li sec:authorize-url="/admin/">
  <a th:href="@{/admin/auteur}">
    gérer les auteurs
  </a>
</li>
```

... mais n'utilise pas les règles `@Secured` et `@PreAuthorize` ;  
uniquement la configuration de `HttpSecurity`

`sec:authentication="name"` permet d'insérer le login de l'utilisateur :

```
<span sec:authentication="name"></span>
```

## Variables dans les règles

On peut faire référence à des éléments du modèle dans les règles :

```
<section class="message" th:each="m : ${liste}">
<div
  sec:authorize=
    "${hasRole('ADMIN') or #vars.m.login == #authentication.name}">
      Ce message peut être détruit...
</div>
</section>
```

- L'ensemble des données du modèle est accessible à travers l'objet `#var` ;
- un certain nombre d'autres objets sont prédéfinis, comme `#authentication`

### Solution plus simple

Placer les informations nécessaires dans le modèle (voir démo).

# Tests

# Dépendance

Dans gradle :

```
testImplementation 'org.springframework.security:spring-security-test'
```

Ajoute des annotations pour exécuter tel ou tel code avec les droits de tel ou tel utilisateur.

# Annotations

`@WithAnonymousUser` exécute le test en étant l'utilisateur anonyme ;

`@WithMockUser(roles = "ADMIN", "USER")` exécute comme un utilisateur ayant les rôles ADMIN et USER ;

`@WithMockUser(username = "toto")` ...comme un utilisateur de login « toto » ;

`@WithUserDetails("toto")` ...idem, ou presque...

Différence entre `@WithMockUser` et `@WithUserDetails` :

- le premier crée un « faux » utilisateur avec des données minimales ;
- le second va chercher le « vrai » utilisateur « toto » dans `UserDetailsService`

# Divers

# Page de login

Voir `customLoginLogout`

# Protection contre CSRF

- activée par défaut ;
- rejette automatiquement les requêtes POST si un paramètre `_csrf` avec la « bonne » valeur aléatoire n'est pas incluse.
- les requêtes GET ne sont pas concernées.
- désactivable (pas recommandé!!!) :

```
@Override
```

```
protected void configure(HttpSecurity http)
    throws Exception {
    http
        .csrf().disable() // désactive la protection
        ...
}
```

# CSRF/inclusion du paramètre

Pour inclure les Deux solutions :

- Manuelle :

```
<form method="POST" >
  <input type="hidden" th:name="${_csrf.parameterName}"
                    th:value="${_csrf.token}" />
```

- automatique : il **suffit** d'utiliser l'attribut th:action :

```
<form th:object="${form}" th:action="@{/message/creer}"
      method="POST">
  titre : <input type="text" th:field="*{titre}"/>
  <textarea th:field="*{texte}"></textarea>
  <input type="submit"/>
</form>
```

# Cross-Origin Resource sharing

## Problème de départ

- attaque de A en CSRF depuis un site externe B, par injection JS sur B ;
- protection : les formulaires de A envoient un token caché ;
- problème si le code JS exécuté depuis B peut lire (mode GET) les pages de A...
- solution : le navigateur refuse de charger, dans un script exécuté sur B, des ressources situées sur A ;
- problème : certaines ressources sont légitimes :
  - ▶ web sémantique par exemple ;
  - ▶ API REST séparée du site http qui l'utilise ;
- voir <https://developer.mozilla.org/fr/docs/Web/HTTP/CORS>
- pour Spring : <https://spring.io/guides/gs/rest-service-cors/>

# Https, Spring boot et Let's Encrypt

- Les certificats SSL pour Https sont la plupart du temps payant ;
- Les certificats auto-signés peuvent perturber les utilisateurs ;
- Let's Encrypt propose gratuitement des certificats ;
- ...mais il faut les renouveler.

# Https, Spring boot et Let's Encrypt

- Les certificats SSL pour Https sont la plupart du temps payant ;
- Les certificats auto-signés peuvent perturber les utilisateurs ;
- Let's Encrypt propose gratuitement des certificats ;
- ...mais il faut les renouveler.
- bonne nouvelle : c'est automatisable !

# Bibliographie

Outre les ouvrages généraux sur Spring, la [documentation de Spring Security](#) est très bien faite, et décrit en particulier les attaques possibles.

# Guide des démonstrations

- `fluentInterface` : démonstration des idées derrière les *fluentInterface* ;
- `minimalBootDemo` : projet minimaliste Spring boot avec sécurité ;
- `securityDemo` : un petit forum, fil rouge de cette séance ;
- `customLoginLogout` : login et logout personnalisés ;
- `oauth2` : identification en utilisant `oauth2` (testé avec Google).