

Sommaire

Plan

- **Introduction**
- **Généralités**
- **Sécuriser une application Spring**
 - Authentification
 - Implémenter un AuthenticationProvider
 - Login, logout
 - SecurityContext
- **Le mécanisme de WebFilter**
 - FilterChainProxy
 - Les principaux filtres
 - Authentification Basic et Digest
 - Remember-me
 - Authentification anonyme
 - Gestion des sessions
- **Fonctionnalités d'autorisations**
 - Requêtes
 - Objets et méthodes
 - Authentification par rôle, par type d'authentification
 - Hiérarchie des rôles
- **Intégration dans une application Web**
 - Appli Web Backend
 - API Rest et JWT
 - OAuth2

Plan

- **Tests**
 - Spring security test
 - Test MVC
- **Configuration avancée des requêtes HTTP**
 - CSRF
 - XSS
 - Iframe
 - Cache

Introduction

Les besoins

- **Gérer des utilisateurs**

- Utilisateurs, mots de passe, droits, informations sur l'utilisateur, ...

- **Sécuriser des URL**

- Empêcher l'accès à certaines URL en fonction du type d'utilisateur

- **Sécuriser des services**

- Empêcher l'accès à certains services, d'activer certaines opérations, ...

- **Sécuriser des objets du domaine**

- Sécuriser certaines instances d'objet métier
 - Empêcher d'accéder aux données d'un autre utilisateur
 - Alors qu'on accède aux siennes

Les apis Java

- **JAAS**

- Dédicée à la gestion fine des droits d'exécutions du code
- Vise surtout la sécurisation

- **Spécification Jakarta / Java EE**

- Essentiellement basée sur la sécurisation des URLs des applis Web
- Peu portable : la spécification s'arrête très tôt, chaque serveur a ses spécificités

- **Intérêts de Spring Security**

- Fournit une solution complète de sécurité
- Gestion de l'authentification
- Gestion des autorisations
 - Au niveau des requêtes web
 - Au niveau des invocations de méthodes
- Portable (Juste une JVM)

Vocabulaire

- **Authentification**

- Vérifier qu'un utilisateur est bien celui qu'il prétend être
- Généralement basé sur la notion d'identifiant et de mot de passe

- **Autorisation**

- Vérifier que l'utilisateur authentifié a bien le droit d'exécuter une action
- Un utilisateur a généralement plusieurs autorisations gérées par groupes

- **Subject et Principal : deux objets issus des spécifications Java**

- **Subject** : l'utilisateur vu par l'application
- **Principal** : une représentation de cet utilisateur
 - login, adresse mail, matricule, ... (un **Subject** peut disposer de plusieurs **Principal**)

- **Ressource et permissions**

- Ressource : une entité (URL, objet, ...) protégée
- Permission : le droit d'accéder à cette ressource

Généralités

XML Vs Java Configuration Vs Spring Boot

- **Depuis Spring 3 on peut déclarer sa configuration**

- En XML (méthode historique)
- En Java

- **Avantages d'une configuration en XML**

- Compatible Legacy
- La configuration et le code ne sont pas mélangés
- Plus puissant

- **Avantage des configurations en annotations**

- La configuration est compilée (moins d'erreur)
- Plus simple, mécanisme d'auto-configuration
- Plus en avant dans la communauté Spring

- **Spring Boot :**

- aucun fichier XML
- Facilite le démarrage de projet via l'autoconfiguration,
- Beaucoup de beans sont instanciés *derrière* notre dos

Authentication

- **La première chose à mettre en place**
 - Identifier l'utilisateur et garantir qu'il est bien celui qu'il prétend
- **Se fait à l'aide de deux éléments**
 - « principal » (généralement un username)
 - « credentials » (généralement un mot de passe)
- **Interface AuthenticationManager**
 - Définit la méthode **authenticate**
 - Prend un **Authentication** en paramètre
 - Retourne un **Authentication** renseigné en sortie
 - Ou bien lève une exception **AuthenticationException**

```
public interface AuthenticationManager {  
    public Authentication authenticate(Authentication a) throws AuthenticationException;  
}
```

Authentication

- **L'objet qui représente le principal qui utilise l'application**
 - Il donnera accès aux informations nécessaires
 - Il est accessible via le **SessionContext** (voir ci-après)

```
public interface Authentication extends Principal, Serializable {  
    Collection<? extends GrantedAuthority> getAuthorities();  
    Object getCredentials();  
    Object getDetails();  
    Object getPrincipal();  
    boolean isAuthenticated();  
    void setAuthenticated(boolean isAuthenticated)  
        throws IllegalArgumentException;  
}
```

GrantedAuthority

- Une « autorité » donné à un **Principal**
 - Typiquement un **rôle** tel que **ROLE_ADMINISTRATOR**
 - L'**Authentication** donne la liste des **GrantedAuthority**
 - Chargée par le **UserDetailsService**
- **Constitue la base des autorisations transverses du système**
- **L'implémentation la plus utilisée est *SimpleGrantedAuthority***
 - Qui est juste une chaîne de caractères
 - Donc une liste de *GrantedAuthorities* est une liste de String



Sécuriser une application Spring

Contexte Spring Boot

- **Dans le contexte d'utilisation de Spring Boot, si les starters *web* et *security* sont dans le classpath, par défaut on a:**
 - Toutes les URLs de l'application web par l'authentification formulaire
 - Un gestionnaire d'authentification mémoire est configuré pour permettre l'identification d'un unique utilisateur : user avec un mot de passe aléatoire s'affichant sur la console
- **Les propriétés peuvent être changées via *application.properties* et le préfixe *spring.security*.**
 - *spring.security.user.name= myUser*
 - *spring.security.user.password=secret*

Contexte Spring Boot

- **D'autres fonctionnalités sont automatiquement obtenues :**

- Les chemins pour les ressources statiques standard sont ignorées (`/css/**`, `/js/**`, `/images/**`, `/webjars/**` et `**/favicon.ico`).
- Les événements liés à la sécurité sont publiés via le bean *ApplicationEventPublisher* (Voir *DefaultAuthenticationEventPublisher*)

Voir : <https://www.baeldung.com/spring-events>

- Des fonctionnalités communes de bas niveau (HSTS, XSS, CSRF, caching)



Personnalisation de la configuration par défaut

- **La personnalisation consiste à définir une classe de configuration implémentant *WebSecurityConfigurer* permettant de :**
 - Définir l'AuthenticationManager :
 - En définissant directement un bean de type AuthenticationManager
 - En implémentant une méthode de WebSecurityConfigurer donnant de accès au builder : AuthenticationManagerBuilder
 - Spécifier les autorisations
 - Implémenter la méthode *protected void configure(HttpSecurity http)*

ProviderManager

- **L'implémentation par défaut de AuthenticationManager**

- Permet de déléguer l'authentification auprès de plusieurs sources
 - Interface **AuthenticationProvider**
- Testé l'un après l'autre jusqu'à ce qu'un retourne un **Authentication** complet
 - Si aucun une exception **ProviderNotFoundException** est levée
- Permet de gérer plusieurs mécanismes d'identification pour une application

```
<bean id="authenticationManager"
      class="org.springframework.security.authentication.ProviderManager">
  <constructor-arg>
    <list>
      <ref local="daoAuthenticationProvider"/>
      <ref local="anonymousAuthenticationProvider"/>
      <ref local="ldapAuthenticationProvider"/>
    </list>
  </constructor-arg>
</bean>
```

AuthenticationProvider

- **Des AuthenticationProvider pour toute situation :**
 - **AuthByAdapterProvider** : authentication depuis le conteneur
 - **AnonymousAuthenticationProvider** : identifie un anonymous
 - **DaoAuthenticationProvider** : info dans une base de données
 - **CasAuthenticationProvider** : authentication CAS
 - **OAuth2LoginAuthenticationProvider** : authentication OAuth2
 - **JaasAuthenticationProvider** : authentication JAAS
 - **LdapAuthenticationProvider** : authentication LDAP
 - **RememberMeAuthenticationProvider** : authentication auto
 - **RemoteAuthenticationProvider** : auth. avec un service distant
 - **X509AuthenticationProvider** : auth. avec un certificat X.509
 - ...

AuthenticationProvider

- **C'est le principe de Spring Security**
- **Pouvoir se connecter sur tout un panel d'IDP (identity Provider)**
 - Ainsi que pouvoir faire une authentification en local
 - Ou implémenter son propre AuthenticationProvider
- **Note : tous les providers ne sont pas inclus dans le package Security.jar**
 - Mais dans des librairies spécifiques
- **Bien sur il n'est pas obligatoire de passer par un AuthenticationProvider**
 - Mais il faut réécrire un authentication manager

DaoAuthenticationProvider

- Une implémentation basée sur l'interface **UserDetailsService**
 - **AuthenticationManager** (**ProviderManager**) appelle **authenticate()**
 - Sur le **DaoAuthenticationManager**
 - Celui-ci accède au **UserDetailsService**
 - Pour aller chercher les informations (user, password) dans une base de données par exemple
 - Il compare le **principal** et le **credentials** proposés
 - Si ça correspond, retourne un **Authentication** entièrement renseigné
 - Sinon, lève une **AuthenticationException**
 - Remarque : le **passwordEncoder** est obligatoire (encodage du mot de passe)

```
<bean id="authenticationProvider"  
  class="org.springframework.security.authentication.dao.DaoAuthenticationProvider">  
  <property name="userDetailsService" ref="userDetailsService"/>  
  <property name="passwordEncoder" ref="passwordEncoder"/>  
</bean>
```

UserDetailsService

- **L'interface du service d'accès aux informations de l'utilisateur**
 - Spring n'impose pas un objet particulier (interface **UserDetails**)
 - Cela permet de stocker son propre objet avec tous les détails souhaités
 - Exemple : adresse mail, numéro de téléphone, matricule, ...
- **Spring fournit des implémentations du UserDetailsService**
 - **JdbcDaoImpl, LdapUsersDetailsService et InMemoryDaoImpl**

```
public interface UserDetailsService {  
    UserDetails loadUserByUsername(String username)  
        throws UsernameNotFoundException;  
}
```

```
public interface UserDetails extends Serializable {  
    Collection<? extends GrantedAuthority> getAuthorities();  
    String getPassword();  
    String getUsername();  
    boolean isAccountNonExpired();  
    boolean isAccountNonLocked();  
    boolean isCredentialsNonExpired();  
    boolean isEnabled();  
}
```

Gestion en mémoire

- **Pour les applications simples ou les prototypes**

- Garde une map des utilisateurs et leurs droits
- Facile à construire avec le namespace security
- Soit directement, soit en chargeant un fichier properties
 - username=password,grantedAuthority[,grantedAuthority][,enabled|disabled]

```
<user-service id="userDetailsService">  
  <user name="jimi" password="jimispasword" authorities="ROLE_USER, ROLE_ADMIN" />  
  <user name="bob" password="bobspassword" authorities="ROLE_USER" />  
</user-service>
```

```
<user-service id="userDetailsService" properties="users.properties"/>
```

```
jimi=jimispasword,ROLE_USER,ROLE_ADMIN,enabled  
bob=bobspassword,ROLE_USER,enabled
```

JdbcDaoImpl

- Récupère les informations depuis une base de données

```
<bean id="dataSource"
      class="org.springframework.jdbc.datasource.DriverManagerDataSource">
  <property name="driverClassName" value="org.hsqldb.jdbcDriver"/>
  <property name="url" value="jdbc:hsqldb:hsqldb://localhost:9001"/>
  <property name="username" value="sa"/>
  <property name="password" value=""/>
</bean>

<bean id="userService"
      class="org.springframework.security.core.userdetails.jdbc.JdbcDaoImpl">
  <property name="dataSource" ref="dataSource"/>
</bean>
```

JdbcDaoImpl

- **Deux requêtes par défaut (impose la structure)**

```
SELECT username, password, enabled FROM users WHERE username = ?  
SELECT username, authority FROM authorities WHERE username = ?
```

- **Possibilité de spécifier ses propres requêtes**

```
<bean id="authenticationDao"  
  class="org.springframework.security.userdetails.jdbc.JdbcDaoImpl">  
  <property name="dataSource" ref bean="dataSource" />  
  <property name="usersByUsernameQuery">  
    <value>requête spécifique</value>  
  </property>  
  <property name="authoritiesByUsernameQuery">  
    <value>requête spécifique</value>  
  </property>  
</bean>
```


Encryptage du mot de passe

- **Pour plus de sécurité, ne jamais garder un mot de passe en clair**
 - L'encrypter avec un **PasswordEncoder**
 - En réalité on ne l'encrypte pas, on le « hash » (décryptage impossible)
 - On l'injecte dans le **DaoAuthenticationProvider**
- **Implémentations proposées**
 - BcryptPasswordEncoder (recommandé)
 - NoOpPasswordEncoder (n'encode pas, pour les tests ou cas spéciaux, Déprécié)
 - StandardPasswordEncoder : combine plusieurs choses

Encryptage avec clé

- **Problème avec des mots de passes faibles**
 - On peut hasher une liste de mots (dictionnaire)
 - Et retrouver le mot de passe d'origine en comparant avec chaque valeur
- **Bcrypt fabrique une clé automatiquement à chaque mot de passe**
 - => Recommandation Spring : utiliser Bcrypt
- **Attention : il faut 1 seconde pour crypter un mot de passe (bcrypt)**
 - A refaire à chaque demande de credentials

{noop}

- Si les mots de passes sont stockés en clair,
 - il faut les préfixer par **{noop}** afin que Spring Security n'utilise pas d'encodeur
 - Naturellement, cela n'est pas recommandé

```
public UserDetails loadUserByUsername(String login) throws UsernameNotFoundException {
    Member member = memberRepository.findByEmail(login);
    if ( member == null )
        throw new UsernameNotFoundException("Invalides login/mot de passe");
    Set<GrantedAuthority> grantedAuthorities = new HashSet<>();

    return new User(member.getEmail(), "{noop}" + member.getPassword(), grantedAuthorities);
}
```

AuthenticationManagerBuilder

- **Il propose des méthodes permettant de facilement construire des *AuthenticationManager***
 - `inMemoryAuthentication()` : Authentification mémoire
 - `jdbcAuthentication()` : Authentification JDBC
 - `ldapAuthentication` : Authentification LDAP
- **Il permet de positionner facilement un `UserDetailsService` personnalisé :**
 - `userDetailsService(T userDetailsService)`
- **Ajouter un fournisseur d'authentification personnalisé :**
 - `authenticationProvider(AuthenticationProvider authenticationProvider)`

Contexte SpringBoot

- Dans un contexte SpringBoot, la méthode recommandée pour configurer l'authentification est
 - de fournir une classe de configuration implémentant *WebSecurityConfigurer*
 - Puis implémenter la méthode `configure(AuthenticationManagerBuilder auth)`

```
@Configuration
public class InMemorySecurityConfiguration extends WebSecurityConfigurerAdapter
{

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {

        auth.inMemoryAuthentication().withUser("user").password("{noop}password").
            roles("USER")
            .and().withUser("admin").password("{noop}password").
            roles("USER", "ADMIN");
    }
}
```

Exemple LDAP

- **Utilisation d'un LDAP pour l'authentification**
 - En utilisant l'extension Spring-security-ldap et le Builder

```
@Autowired
public void configureGlobal(
    AuthenticationManagerBuilder auth) throws Exception {
    auth
        .ldapAuthentication()
            .userDnPatterns("uid={0},ou=people")
            .groupSearchBase("ou=groups");
}
```

Gestion d'un cache

- **Interface UserCache**

- Pour éviter de trop solliciter la base pour les mêmes informations

- **On configure le DaoAuthenticationProvider**

- En injectant dans sa propriété **userCache**
 - Typique des applications stateless

- **Implémentations proposées :**

- **NullUserCache** : pas de cache (implémentation par défaut)
 - **EhCacheBasedUserCache** : basé sur EHCache
 - **SpringCacheBasedUserCache** : basé sur un Cache de Spring

- **Attention, cela peut introduire d'autres types de problèmes**

- Exemple : le mot de passe soumis est modifié après une authentification réussie, et l'application étant stateless, on doit s'authentifier à chaque requête

Implémenter son propre Authentication Provider ?

- **Pourquoi ?**

- Connexion à un Identity Provider d'entreprise par exemple
- Probablement très complexe

Implémenter son UserDetailsService

- **A quoi ça sert ?**

- A compléter le DaoAuthenticationProvider

- **Pourquoi ?**

- Ne plus se baser sur le schéma Spring
- S'interfacer directement avec son SI
- Rajouter des fonctionnalités (Groupes de droits par exemple)

- **Comment ?**

- Réécrire un nouveau service implémentant

```
UserDetails loadUserByUsername(String username) throws UsernameNotFoundException;
```

- Et utiliser sa propre table user et rights
 - Avec ses paramètres spécifiques (email, tel etc...)
 - Avoir un système de droit plus complexe (groupe de droits)



SecurityContextHolder

- **Stocke les informations de sécurité**
 - Dont le principal
- **Par défaut stocke ses informations dans une variable `THREAD_LOCAL`. Dans un contexte web :**
 - Initialisé à la réception de la requête
 - Détruit après l'envoi de la réponse
 - Et donc accessible par toutes les classes traversés lors du traitement de la requête WEB
- **On peut modifier la durée de stockage des informations**
 - `SecurityContextHolder.MODE_GLOBAL` (application lourde)
 - `SecurityContextHolder.MODE_INHERITABLETHREADLOCAL` (application créant des threads)

Authentication/Principal

- **Récupération de authentification :**

```
Authentication auth =  
SecurityContextHolder.getContext().getAuthentication();
```

- **Récupération du principal :**

```
Object principal = auth.getPrincipal();  
  
if (principal instanceof UserDetails) {  
    System.out.println("UserName " + ((UserDetails)  
principal).getUsername());  
    System.out.println("Password " + ((UserDetails)  
principal).getPassword());  
    System.out.println("Name " + ((UserDetails) principal).getName());  
} else {  
    String username = principal.toString();  
}
```

- **On accède aussi aux Authorities de l'utilisateur**



TP 2.3

Login

- **http.formLogin()**

- Construit une page de login par défaut
- Customisable...

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.formLogin()
        .loginPage("/login.html")
        .loginProcessingUrl("/perform_login")
        .defaultSuccessUrl("/homepage.html", true)
        .failureUrl("/login.html?error=true")
}
```

- **Retourne à la page demandée en cas de succès**
- **Penser à rendre cette page accessible à tout le monde**
 - `formLogin().permitAll()`

- **Paramètres :**

- Always-use-default-target
- Authentication-details-source-ref
- Authentication-failure-handler-ref
- Authentication-failure-url
- Authentication-success-handler-ref
- **Default-target-url**
- **Login-page**
- **Login-processing-url**
- **Password-parameter/username-parameter**
- Authentication-success-forward-url
- Authentication-failure-forward-url

Logout

- **HTTP.logout()**

- Construit une servlet de logout (/logout)
- Customisable :
 - delete-cookies
 - Invalidate-session
 - Logout-success-url
 - Logout-url
 - Success-handler-ref

```
http.logout()    // Comportement du logout
.logoutUrl("/my/logout")
.logoutSuccessUrl("/my/index")
.invalidateHttpSession(true)
.addLogoutHandler(logoutHandler)
.deleteCookies(cookieNamesToClear) ;
```



TP 2.4

SecurityContextHolder

- **L'objet fondamental dans la gestion de la sécurité**
 - Il détient un **SecurityContext** qui détient les informations de sécurité
 - Basé par défaut sur un **ThreadLocal**
 - Mécanisme permettant de stocker des informations liées au thread courant
 - Désactivable si l'application utilise de manière spéciale les thread
- **Exemple pour accéder au username**

```
Object principal = SecurityContextHolder.getContext().getAuthentication().getPrincipal();  
  
if (principal instanceof UserDetails) {  
    String username = ((UserDetails)principal).getUsername();  
} else {  
    String username = principal.toString();  
}
```

Autres providers

- **Spring Security est fait pour s'interface avec des plateformes externe (IDP) de gestion d'identité.**
 - spring-security-cas.jar
 - spring-security-openid.jar
 - spring-security-oauth2-client.jar
 - spring-security-ldap.jar
 - Etc.
- **Pour s'interfacer avec ces plate-formes, suivre la documentation.**
 - La documentation de Spring security contient un repository git contenant des exemples (en gradle)

Le mécanisme de web filters

Sécurité des applications web

- **La chaîne de sécurité est basée sur les filtres de servlets**
 - Technologie complètement standard
 - Spring conserve une chaîne de filtres internes ou chacun à sa responsabilité
 - Les filtres sont ajoutés/supprimés par configuration en fonction des besoins
 - Mais il est extrêmement important de respecter l'ordre logique d'enchaînement
- **L'injection n'est pas possible dans le filtre**
 - Le **DelegatingFilterProxy** délègue à un bean (portant le même nom)
 - Recherché dans le contexte Spring
- **La namespace security simplifie la construction de cette chaîne**
 - En évitant la construction de nombreux beans
 - En évitant de ne pas respecter l'ordre imposé
 - Déclare par défaut le bean "**springSecurityFilterChain**"

La classe FilterChainProxy

- **Deux dépendances minimales**
 - org.springframework.security.**spring-security-web**
 - org.springframework.security.**spring-security-config**
- **Modules complémentaires selon les choix d'implémentations**
 - spring-security-ldap, spring-security-cas, spring-security-openid, ...
- **Configuration du filtre de sécurité (dans le web.xml) ou automatiquement via SpringBoot**
 - Prend en charge toutes les URLs (« /* »)

```
<filter>
<filter-name>springSecurityFilterChain</filter-name>
<filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
</filter>

<filter-mapping>
<filter-name>springSecurityFilterChain</filter-name>
<url-pattern>/*</url-pattern>
</filter-mapping>
```

Principaux filtres

- **Cet ordre illustre les fonctionnalités supportées**

- **ChannelProcessingFilter** : rediriger sur un autre protocole (https)
- **SecurityContextPersistenceFilter** : gérer le **SecurityContext**, stocké dans la session par défaut
- **ConcurrentSessionFilter** : gère les sessions multiples, peut invalider la session
- Le mécanisme d'authentification : **UsernamePasswordAuthenticationFilter** , ...
- **SecurityContextHolderAwareRequestFilter** : injection explicite du **SecurityContext**
- **JaasApiIntegrationFilter** : un jeton **JaasAuthenticationToken** comme **Subject**
- **RememberMeAuthenticationFilter** : gérer une reconnexion avec un cookie
- **AnonymousAuthenticationFilter** : garantir qu'un Authentication existe (anonyme)
- **ExceptionTranslationFilter** : gère les exceptions de sécurité lors de l'authentification
- **FilterSecurityInterceptor** : lève les exceptions si l'accès est interdit
- **OAuth2*Filter** : Gère le protocole OAuth2

Initialisation des filtres web

- **Les modifications de la configuration Security vont modifier ces filtres**
 - Soit rajouter des paramètres (exemple avec le 403)
 - Soit rajouter des filtres nouveaux (authentification par mot de passe)
 - On peut rajouter manuellement des filtres
- **Dans un contexte SpringBoot et d'un WebSecurityConfigurerAdapter, c'est la méthode**
`protected void configure(HttpSecurity http) throws Exception {`

FilteringSecurityInterceptor

- ***FilteringSecurityInterceptor*** : S'occupe de vérifier les droits d'accès aux URL (autorisations)

```
<bean id="filterInvocationInterceptor"
class="org.springframework.security.web.access.intercept.FilterSecurityInterceptor">
  <property name="authenticationManager" ref="authenticationManager"/>
  <property name="accessDecisionManager" ref="accessDecisionManager"/>
  <property name="runAsManager" ref="runAsManager"/>
  <property name="securityMetadataSource">
    <security:filter-security-metadata-source request-matcher="regex">
      <security:intercept-url pattern="\A/secure/super/.*\Z"
access="ROLE_WE_DONT_HAVE"/>
      <security:intercept-url pattern="\A/secure/.*\"
access="ROLE_SUPERVISOR,ROLE_TELLER"/>
    </security:filter-security-metadata-source>
  </property>
</bean>
```

ExceptionTranslationFilter

- **ExceptionTranslationFilter : Gestion des pages 403.**

- Lors d'une exception `AccessDeniedException` ou `AuthenticationException`, le filtre génère la réponse HTTP adéquate

```
<bean id="exceptionTranslationFilter"
class="org.springframework.security.web.access.ExceptionTranslationFilter">
  <property name="authenticationEntryPoint" ref="authenticationEntryPoint"/>
  <property name="accessDeniedHandler" ref="accessDeniedHandler"/>
</bean>

<bean id="authenticationEntryPoint"
class="org.springframework.security.web.authentication.LoginUrlAuthenticationEntryPoi
nt">
  <property name="loginFormUrl" value="/login.jsp"/>
</bean>

<bean id="accessDeniedHandler"
  class="org.springframework.security.web.access.AccessDeniedHandlerImpl">
  <property name="errorPage" value="/accessDenied.htm"/>
</bean>
```

UsernamePasswordAuthenticationFilter

- Appelle l'authenticationManager !

```
<bean id="authenticationFilter" class=
"org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter">
<property name="authenticationManager" ref="authenticationManager"/>
</bean>
```


Debug

- **Debug de spring security**

```
@EnableWebSecurity(debug = true)
```

- **Activation des logs depuis logback.xml**

- Très verbeux mais utile

```
<logger name="org.springframework.security" level=" debug " />
```

- **Permet de suivre le passage des différents filtres**

- **A ne pas activer en production**

- De base Spring masque les passwords, mais quand même faire attention.

Debug (2)

```
/secure/ServletTest at position 1 of 12 in additional filter chain; firing Filter:
'WebAsyncManagerIntegrationFilter'
/secure/ServletTest at position 2 of 12 in additional filter chain; firing Filter:
'SecurityContextPersistenceFilter'
Obtained a valid SecurityContext from SPRING_SECURITY_CONTEXT:
'org.springframework.security.core.context.SecurityContextImpl@442be9fb: Authentication:
org.springframework.security.authentication.UsernamePasswordAuthenticationToken@442be9fb:
Principal: org.springframework.security.core.userdetails.User@36ebcb: Username: user;
Password: [PROTECTED]; Enabled: true; AccountNonExpired: true; credentialsNonExpired: true;
AccountNonLocked: true; Granted Authorities: ROLE_USER; Credentials: [PROTECTED];
Authenticated: true; Details:
org.springframework.security.web.authentication.WebAuthenticationDetails@0: RemoteIpAddress:
0:0:0:0:0:0:0:1; SessionId: D3D679FD8EC85F06E133B9B7D6A6C231; Granted Authorities:
ROLE_USER'
/secure/ServletTest at position 3 of 12 in additional filter chain; firing Filter:
'HeaderWriterFilter'
Not injecting HSTS header since it did not match the requestMatcher
org.springframework.security.web.header.writers.HstsHeaderWriter$SecureRequestMatcher@467297
51
/secure/ServletTest at position 4 of 12 in additional filter chain; firing Filter:
'CsrfFilter'
/secure/ServletTest at position 5 of 12 in additional filter chain; firing Filter:
'LogoutFilter'
Request 'GET /secure/ServletTest' doesn't match 'POST /logout'
/secure/ServletTest at position 6 of 12 in additional filter chain; firing Filter:
'UsernamePasswordAuthenticationFilter'
Request 'GET /secure/ServletTest' doesn't match 'POST /login'
```

Debug (3)

```
/secure/ServletTest at position 7 of 12 in additional filter chain; firing Filter:
'RequestCacheAwareFilter'
/secure/ServletTest at position 8 of 12 in additional filter chain; firing Filter:
'SecurityContextHolderAwareRequestFilter'
/secure/ServletTest at position 9 of 12 in additional filter chain; firing Filter:
'AnonymousAuthenticationFilter'
SecurityContextHolder not populated with anonymous token, as it already contained:
'org.springframework.security.authentication.UsernamePasswordAuthenticationToken@442be
9fb: Principal: org.springframework.security.core.userdetails.User@36ebcb: Username:
user; Password: [PROTECTED]; Enabled: true; AccountNonExpired: true;
credentialsNonExpired: true; AccountNonLocked: true; Granted Authorities: ROLE_USER;
Credentials: [PROTECTED]; Authenticated: true; Details:
org.springframework.security.web.authentication.WebAuthenticationDetails@0:
RemoteIpAddress: 0:0:0:0:0:0:0:1; SessionId: D3D679FD8EC85F06E133B9B7D6A6C231; Granted
Authorities: ROLE_USER'
/secure/ServletTest at position 10 of 12 in additional filter chain; firing Filter:
'SessionManagementFilter'
/secure/ServletTest at position 11 of 12 in additional filter chain; firing Filter:
'ExceptionTranslationFilter'
/secure/ServletTest at position 12 of 12 in additional filter chain; firing Filter:
'FilterSecurityInterceptor'
Checking match of request : '/secure/ServletTest'; against '/login.jsp'
Checking match of request : '/secure/ServletTest'; against '/forbidden.jsp'
Checking match of request : '/secure/ServletTest'; against '/secure/user.jsp'
```

Debug (4)

```
Checking match of request : '/secure/ServletTest'; against '/secure/admin.jsp'
Secure object: FilterInvocation: URL: /secure/ServletTest; Attributes: [authenticated]
Previously Authenticated:
org.springframework.security.authentication.UsernamePasswordAuthenticationToken@442be9fb: Principal:
Username: user; Password: [PROTECTED]; Enabled: true; AccountNonExpired: true;
credentialsNonExpired: true; AccountNonLocked: true; Granted Authorities: ROLE_USER;
Credentials: [PROTECTED]; Authenticated: true; Details:
org.springframework.security.web.authentication.WebAuthenticationDetails@0:
RemoteIpAddress: 0:0:0:0:0:0:0:1; SessionId: D3D679FD8EC85F06E133B9B7D6A6C231; Granted
Authorities: ROLE_USER
returned: 1
Authorization successful
RunAsManager did not change Authentication object
Chain processed normally
SecurityContextHolder now cleared, as request processing completed
```



TP 3.1

Remember me

- **Se rappeler de l'identité entre 2 sessions grâce à un cookie**
- **2 méthodes**
 - Hash-Based Token (persistance en mémoire)
 - Persistence Token (persistance en base)
- **Nécessite un bean *UserDetailsService***
- **Le token est généré par le serveur et envoyé sous forme de cookie au client**
- **Contenu du cookie**

```
base64(username + ":" + expirationTime + ":" +  
md5Hex(username + ":" + expirationTime + ":" + password + ":" + key))  
username: As identifiable to the UserDetailsService  
password: That matches the one in the retrieved UserDetails  
expirationTime: The date and time when the remember-me token expires, expressed in  
milliseconds  
key: A private key to prevent modification of the remember-me token
```

Remember me : activation

```
http.rememberMe();
```

```
DEBUG TokenBasedRememberMeServices - Added remember-me cookie for user 'user',  
expiry: 'Mon Jan 29 09:24:05 CET 2018'
```

Name	Headers	Preview	Response	Cookies	Timing
login					
TP1/					
bootstrap-responsive.css					
bootstrap.css					

Name	Value	Domain	Path	Expires / ...	Size	HTTP	Secure	SameSite
Request Cookies								
JSESSIONID	417EEF572098350E724572736A6C3399	N/A	N/A	N/A	43			
Response Cookies								
JSESSIONID	F8EC0A6F5EB50702A76B362F1FB0C89F		/TP1/	Session	66	✓		
remember-me	dXNlcjoxNTE3MjE0MzM1MDk2OjY3NjUwZWE5NzA4...		/TP1	2018-01...	140	✓		

TP 3.2

Remember me : TokenBasedRememberMeServices

- **Possibilité de sauvegarder en base le token**

- Pour vérifier qu'il vient bien de notre serveur ? En cas de reboot de l'application ?

```
<bean id="rememberMeFilter" class=
"org.springframework.security.web.authentication.rememberme.RememberMeAut
henticationFilter">
<property name="rememberMeServices" ref="rememberMeServices"/>
<property name="authenticationManager" ref="theAuthenticationManager" />
</bean>
```

```
<bean id="rememberMeServices" class=
"org.springframework.security.web.authentication.rememberme.TokenBasedRem
emberMeServices">
<property name="userDetailsService" ref="myUserDetailsService"/>
<property name="key" value="springRocks"/>
</bean>
```

```
<bean id="rememberMeAuthenticationProvider" class=
"org.springframework.security.authentication.RememberMeAuthenticationProv
ider">
<property name="key" value="springRocks"/>
</bean>
```

Remember me : TokenBasedRememberMeServices

- **Par défaut nécessite une table**

```
create table persistent_logins (username varchar(64) not null,  
                                series varchar(64) primary key,  
                                token varchar(64) not null,  
                                last_used timestamp not null)
```


Authentification Anonyme

- **Filtre** : `AnonymousAuthenticationFilter`
- **Est équivalent à une absence d'authentification**
- **Un utilisateur non logué à quand même des informations dans le `securityContextHolder`**
- **C'est le `AnonymousAuthenticationFilter` qui s'occupe d'ajouter ces informations**
- **Concrètement, l'anonymous ne possède qu'un seul rôle :**
 - `ROLE_ANONYMOUS`
 - On peut s'en servir sur les fonctionnalités d'autorisation

Authentication Anonyme

```
/login.jsp?login_error= at position 10 of 13 in additional filter  
chain; firing Filter: 'AnonymousAuthenticationFilter'  
Populated SecurityContextHolder with anonymous token:  
'org.springframework.security.authentication.AnonymousAuthenticationTo  
ken@da604f00: Principal: anonymousUser; Credentials: [PROTECTED];  
Authenticated: true; Details:  
org.springframework.security.web.authentication.WebAuthenticationDetail  
s@b364: RemoteIpAddress: 0:0:0:0:0:0:0:1; SessionId:  
069DA67CD4E2DED89BF2C87A7E1F3594; Granted Authorities: ROLE_ANONYMOUS'
```

Sessions

- **Filtre SessionManagementFilter permet de contrôler quand les sessions sont créées et comment Spring Security interagit avec**
- **4 comportements sont possibles :**
 - Always : La session est tjrs créée si il y en a pas
 - IfRequired : Seulement si nécessaire (défaut)
 - never : SS ne crée jamais de session mais l'utilise
 - stateless : Aucune session créée ni utilisée

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.sessionManagement()
        .sessionCreationPolicy(SessionCreationPolicy.IF_REQUIRED)
}
```

Comportement par défaut

- **Regarde le contenu du SecurityContextHolder**
 - Si il existe un auth (non anonyme) ne fait rien
 - Sinon regarde si la session du client est toujours valide
 - Si oui, alors il renseigne le context de Sécurité
 - Si non, il ne fait rien

Sessions Concurrentes

- Les utilisateurs aiment bien
- Les administrateurs moins (partage de login)
- Les développeurs non plus (possible incohérences)
- Il est possible de limiter le nombre de session pour un utilisateur
 - Au login (interdiction de se loguer tant qu'on a déjà une session ouverte)
 - Au login (et on invalide la session déjà existante)
- Pour cela il faut préalablement activer la notification des sessions à Spring (fichier web.xml)

```
<listener>
<listener-class>
    org.springframework.security.web.session.HttpSessionEventPublisher
</listener-class>
</listener>
```

- Avec Spring Boot :

```
@Bean
public HttpSessionEventPublisher httpSessionEventPublisher() {
    return new HttpSessionEventPublisher();
}
```

Sessions Concurrentes (2)

- **Il faut ensuite un endroit ou stocker les sessions**

- Bean SessionRegistry à fournir

```
@Bean
public SessionRegistry sessionRegistry() {
    return new SessionRegistryImpl();
}
```

- **Et enfin il faut dire à Security un nombre max de session**

- -1 par défaut (illimité)

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.sessionManagement().maximumSessions(2)
}
```

Information sur les utilisateurs connectés

- **Les sessions sont stockées dans un bean SessionRegistry**
 - On peut consulter ce bean pour récupérer les sessions d'un utilisateur ...
 - ... et invalider sa session par exemple (via son sessionId)

```
//recuperer ses sessions
List<SessionInformation> sessions = sessionRegistry.getAllSessions(auth, false);
...
//recuperation des informations d'une session
SessionInformation sessionInformation = sessionRegistry.getSessionInformation(sessionId);
...
//tuer une session
sessionInformation.expireNow();
```



TP 3.3

Fonctionnalités d'autorisations

Deux types de sécurisation

- **Sécurisation des applications web**

- Utilisation de filtres (servlet filters) pour intercepter les requêtes, traiter l'authentification et gérer la sécurité

- **Sécurisation au niveau des invocations de méthodes**

- S'appuie sur Spring AOP
- Applique des aspects vérifiant que l'utilisateur a les droits suffisants pour invoquer la méthode

- **Dans tous les cas**

- La gestion de la sécurité s'appuie d'abord sur une **interception**

Intercepteurs de sécurité

- **Classe abstraite AbstractSecurityInterceptor**
 - **FilterSecurityInterceptor** : intercepte les requêtes HTTP
 - **MethodSecurityInterceptor** : intercepte les appels de méthode
 - **AspectJMethodSecurityInterceptor** : idem mais avec AspectJ

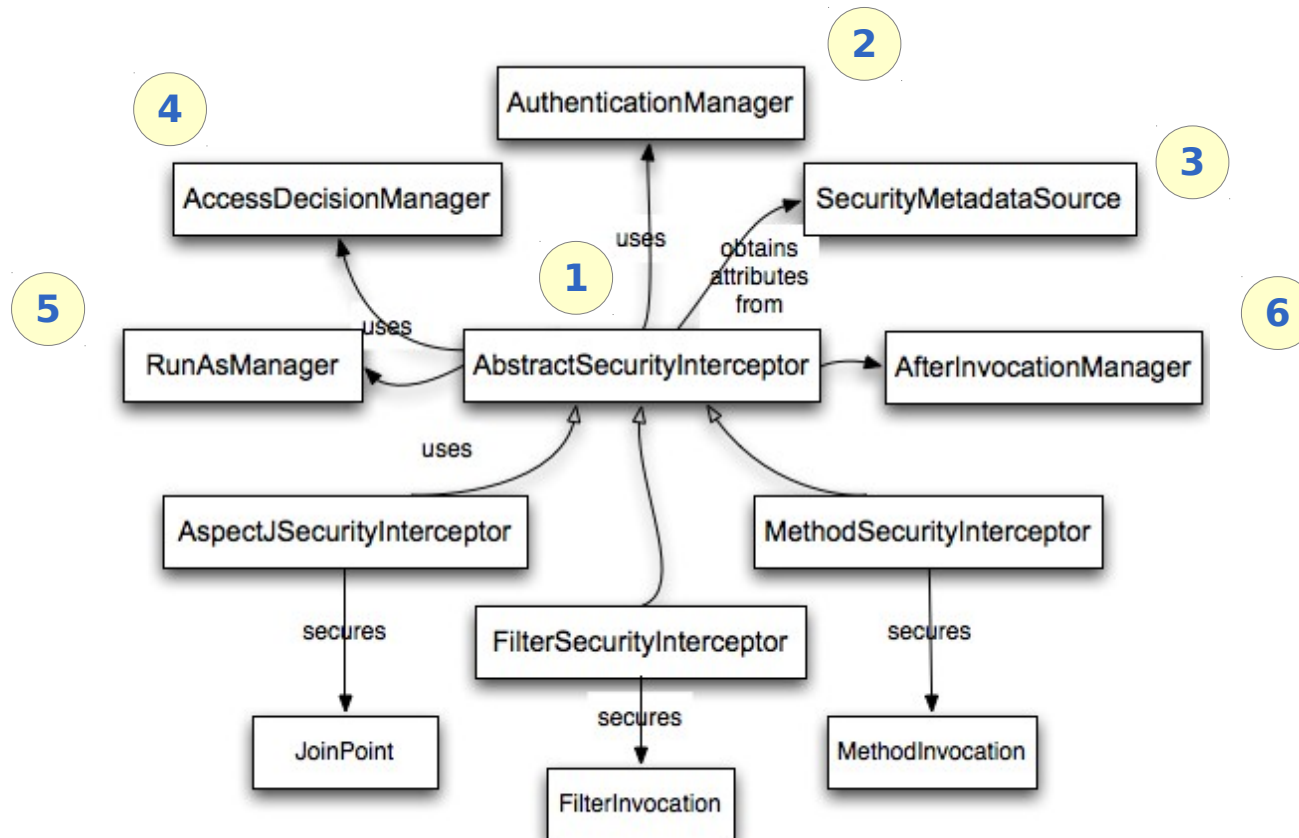


Schéma général : 4 niveaux de manager

- **AuthenticationManager**

- Responsable de l'identification de l'utilisateur

- **AccessDecisionManager**

- Il vérifie l'autorisation d'accès à la ressource sécurisée
 - Pour cela il considère les informations d'authentification ainsi que les attributs de sécurité associés à cette ressource

- **RunAsManager**

- Une étape optionnelle supplémentaire permettant d'attribuer une authentification avec des droits supplémentaires pour accéder à des éléments internes
 - Nécessaire pour certains types d'applications

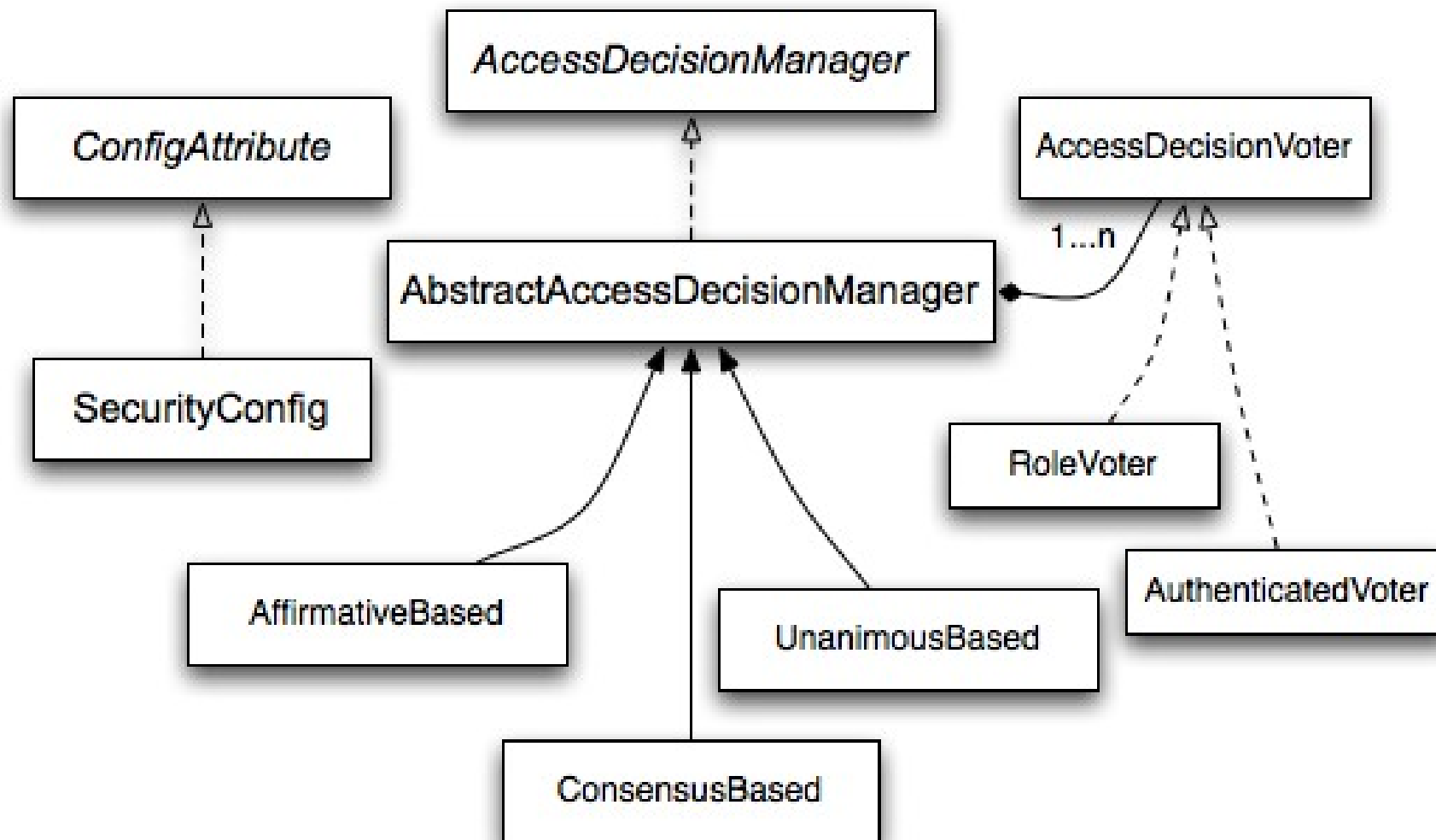
- **AfterInvocationManager**

- Un niveau supplémentaire pour vérifier les droits d'accès aux données affichées ou retournées par le service (rare)

Autorisation

- **Basé sur les GrantedAuthority d'un Authentication**
- **AccessDecisionManager : décide de l'accès à une ressource**
 - En se basant sur une série de RoleVoter
- **RoleVoter : une instance qui donne un avis sur l'accès**
 - ACCES_ABSTAIN
 - ACCES_DENIED
 - ACCES_GRANTED
- **L'AccessDecisionManager liste les résultats et prend une décision**
- **Plusieurs stratégies (ou type d'implémentation):**
 - AffirmativeBased (ok si 1 voter ok)
 - ConsensusBased (ok si majorité ok)
 - UnanimousBased (ok si tous ok)

Authorisation (2)



Contrôle des accès

- **En configurant le AccessDecisionManager**

```
public interface AccessDecisionManager {  
  
    void decide(Authentication authentication, Object object,  
                Collection<ConfigAttribute> configAttributes) throws AccessDeniedException,  
                InsufficientAuthenticationException;  
  
    boolean supports(ConfigAttribute attribute);  
  
    boolean supports(Class<?> clazz);  
}
```

- **Les méthodes « supports »**

- Considèrent le type de la ressource et ses attributs de configurations pour décider si le AccessDecisionManager est apte à décider

- **La méthode « decide »**

- Réalise la décision (lève une exception ou pas)

Décision

- **Le AccessDecisionManager ne décide pas seul**
 - Il prend ses décisions auprès d'un ou plusieurs **AccessDecisionVoter**
 - Un votant peut s'abstenir, voter pour ou contre l'accès
 - En utilisant les **GrantedAuthority** portées par le **Authentication**
- **Plusieurs AccessDecisionManager sont proposés :**
 - **AffirmativeBased** : laisse l'accès si au moins un votant vote l'accès
 - **ConsensusBased** : nécessite une majorité de votes positifs
 - **UnanimousBased** : aucun vote négatif

```
<!-- pour la sécurité des méthode -->
<global-method-security access-decision-manager-ref="myAccessDecisionManagerBean">
...
</global-method-security>

<!-- pour la sécurité web -->
<http access-decision-manager-ref="myAccessDecisionManagerBean">
...
</http>
```

Abstention

- **L'abstention a lieu lorsque la ressource ne réclame aucune autorisation préfixée ROLE_**
 - Peut être contrôlée auprès du **AccessDecisionManager**

```
<bean id="accessDecisionManager"  
  class="org.springframework.security.vote.UnanimousBased">  
  <property name="decisionVoters">  
    <list>  
      <ref bean="roleVoter"/>  
    </list>  
  </property>  
  <property name="allowIfAllAbstain" value="true" />  
</bean>
```


AccessDecisionVoter

- **AuthenticatedVoter**

- Permet de différencier entre **anonymous**, pleinement **authentifié** et authentifié automatiquement par le **remember-me**
- L'attribut IS_AUTHENTICATED_ANONYMOUSLY est traité par lui

- **CustomVoter**

- Permet d'implémenter sa propre stratégie de vote

- **RoleHierarchyVoter**

- Permet de gérer des rôles hiérarchiques
- Exemple : ROLE_ADMIN ⇒ ROLE_STAFF ⇒ ROLE_USER ⇒ ROLE_GUEST

L'interface AccessDecisionVoter

- Elle définit 3 constantes et 3 méthodes

```
int ACCESS_GRANTED = 1;  
int ACCESS_ABSTAIN = 0;  
int ACCESS_DENIED = -1;  
boolean supports(ConfigAttribute attribute);  
boolean supports(Class clazz);  
int vote(Authentication authentication, Object object, ConfigAttributeDefinition config);
```

- Même principe que pour AccessDecisionManager

- Mais ici on ne fait que voter en renvoyant une des constantes

- Une implémentation proposée est RoleVoter

- Elle se base sur les attributs de configurations de la ressource
 - (ceux préfixés par ROLE_) avec les autorisations attribuées à l'utilisateur
 - ACCESS_GRANTED est accordé lorsque les rôles coïncident
 - Remarque : le préfixe ROLE_ peut être modifié

Les élections !

- Possibilité de contrôler la sécurité d'une classe ou d'une méthode

```
DEBUG MethodSecurityInterceptor - Secure object: ReflectiveMethodInvocation: public
void com.oxiane.security.service.CountryService.deleteCountry(java.lang.String);
target is of class [com.oxiane.security.service.CountryService]; Attributes:
[[authorize: 'hasRole('Admin')', filter: 'null', filterTarget: 'null']]
DEBUG MethodSecurityInterceptor - Previously Authenticated:
org.springframework.security.authentication.UsernamePasswordAuthenticationToken@fec4
b7ad: Principal: org.springframework.security.core.userdetails.User@586034f:
Username: admin; Password: [PROTECTED]; Enabled: true; AccountNonExpired: true;
credentialsNonExpired: true; AccountNonLocked: true; Granted Authorities:
ROLE_ADMIN,ROLE_USER; Credentials: [PROTECTED]; Authenticated: true; Details:
org.springframework.security.web.authentication.WebAuthenticationDetails@380f4:
RemoteIpAddress: 0:0:0:0:0:0:0:1; SessionId: 8060A916778D3FB71BE6952389293AC0;
Granted Authorities: ROLE_ADMIN, ROLE_USER
DEBUG AffirmativeBased - Voter:
org.springframework.security.access.prepost.PreInvocationAuthorizationAdviceVoter@31
bcb993, returned: -1
DEBUG AffirmativeBased - Voter:
org.springframework.security.access.vote.RoleVoter@3a25baa2, returned: 0
DEBUG AffirmativeBased - Voter:
org.springframework.security.access.vote.AuthenticatedVoter@3eeb80fc, returned: 0
DEBUG ExceptionTranslationFilter - Access is denied (user is not anonymous);
delegating to AccessDeniedHandler
```

Autorisation sur des requêtes

- **HTTPSecurity s'occupe de sécuriser les requêtes http**
 - Pour les servlets, les .jsp
 - Pour les Controller mvc
 - Pour les Controller REST ...
- **Rappel : Faire du plus spécifique au plus général**
 - Exemple de configuration

```
http.authorizeRequests()  
    .antMatchers("/resources/**", "/login", "/logout",  
"/index.jsp", "/login.jsp", "/forbidden.jsp").permitAll()  
    .antMatchers("/secure/user.jsp").access("hasRole('ADMIN')  
or hasRole('USER')")  
    .antMatchers("/secure/  
admin.jsp").access("hasRole('ADMIN')")  
    .anyRequest().authenticated() ;
```

Autorisation sur des requêtes

- **AntMatcher((Methode,) ...expr)**
 - Peut prendre plusieurs patterns, et des verbes HTTP (GET,POST,DELETE)
- **AnyRequest()**
- **MvcMatcher(...expr)**
 - Même règles que SpringMVC
- **RegexMatcher(expr)**
- **On peut créer ses propres matchers**



Mécanisme de sécurisation

- **Penser à organiser le site en fonction des principaux rôles**
- **Un répertoire par grand rôle**
 - /secure
 - /admin
 - /monitoring
 - Etc.
- **Des sous répertoires pour des sous rôles**
 - /secure/configuration
 - /secure/payment
 - Etc.

Autorisation sur des objets et méthodes

- Possibilité de contrôler la sécurité d'une classe ou d'une méthode (Couche service)
- Nécessite d'activer les annotations

```
@EnableGlobalMethodSecurity(securedEnabled = true)
public class MethodSecurityConfig {
    // ...
}
// OU pour les annotations standard JavaEE
@EnableGlobalMethodSecurity(jsr250Enabled = true)
public class MethodSecurityConfig {
    // ...
}
```

- Il suffit alors d'annoter les méthodes d'une implémentation ou d'une interface
 - **@Secured** (Spring)
 - **@Role** (JavaEE)

Configuration Spring Boot

- **La configuration SpringBoot s'effectue via une classe de type GlobalMethodSecurityConfiguration**

```
@Configuration
@EnableGlobalMethodSecurity(
    prePostEnabled = true,
    securedEnabled = true,
    jsr250Enabled = false)
public class MethodSecurityConfig extends GlobalMethodSecurityConfiguration {

}
```


Sécurisation de méthode

- **@Secured (historique Spring)**

```
@Secured("IS_AUTHENTICATED_ANONYMOUSLY")
public Account[] findAccounts();
@Secured("ROLE_TELLER")
public Account post(Account account, double amount);
```

- **JEE (JSR-250)**

- @RolesAllowed, @PermitAll, @DenyAll, ...

```
<global-method-security jsr250-annotations="enabled" />
```

```
@PermitAll
public Account[] findAccounts();
@RolesAllowed("TELLER")
public Account post(Account account, double amount);
```

Autorisation sur des objets et méthodes

- **Spring permet d'être encore plus fin que de simple contraintes par rôle**
 - en utilisant des annotations spécifiques
 - Combiné avec une syntaxe basée sur des expressions
- **Activation :**

```
@EnableGlobalMethodSecurity(prePostEnabled = true)  
public class MethodSecurityConfig {  
    // ...  
}
```

- **4 Annotations**
 - @PreAuthorize(Expr)
 - @PostAuthorize(Expr) (Utilise le retour de la méthode pour ses tests)
 - @PreFilter(Expr) : filtre les collections en entrée d'une méthode
 - @PostFilter(Expr) : filtre les collections en sortie d'une méthode

Expression-Based Access Control

- `hasRole([role])`
- `hasAnyRole([role1,role2])`
- `hasAuthority([authority])`
- `hasAnyAuthority([authority1,authority2])`
- `principal`
- `authentication`
- `permitAll`
- `denyAll`
- `isAnonymous()`
- `isRememberMe()`
- `isAuthenticated()`
- `isFullyAuthenticated()`
- `hasPermission(Object target, Object permission)`
- `hasPermission(Object targetId, String targetType, Object permission)`



Sécurisation de méthode

- **Sécurisation globale par Pointcut**

- Extrêmement puissant, permet de sécuriser toute une application rapidement

```
<global-method-security>
  <protect-pointcut expression="execution(* com.mycompany.*Service.*(..))"
    access="ROLE_USER"/>
</global-method-security>
```

- **Sécurisation spécifique d'un bean (ou plutôt classe de bean)**

```
<bean:bean id="target" class="com.mycompany.myapp.MyBean">
  <intercept-methods>
    <protect method="set*" access="ROLE_ADMIN" />
    <protect method="get*" access="ROLE_ADMIN,ROLE_USER" />
    <protect method="doSomething" access="ROLE_USER" />
  </intercept-methods>
</bean:bean>
```

Sécurisation des objets de domaine (ACL)

- **Contrôle par programmation possible**
 - Accès aux informations via le **SecurityContextHolder**
 - Supports des méthodes l'API servlet standard de **HttpServletRequest**
 - `getRemoteUser()`
 - `getUserPrincipal()`
 - `isUserInRole(String)`
- **Mais pas suffisant pour ne laisser l'accès qu'à certaines données**
 - Nécessite une gestion Access Control List (ACL)
 - Enregistre pour chaque objet de domaine les détails de qui peut travailler ou non avec cet objet
- **Spring security fournit**
 - Un moyen de récupérer/modifier efficacement toutes les entrées ACL d'un objet
 - Un moyen efficace de s'assurer que le principal a les droits sur l'objet
 - Avant d'invoquer ses méthodes
 - Après avoir invoqué ses méthodes

Hiérarchie des rôles

- *RoleHierarchyVoter* est un voter avec le concept de hiérarchie de rôle
- Exemple de configuration :

```
<bean id="roleVoter"  
class="org.springframework.security.access.vote.RoleHierarchyVoter">  
    <constructor-arg ref="roleHierarchy" />  
</bean>  
<bean id="roleHierarchy"  
class="org.springframework.security.access.hierarchicalroles.RoleHierarchyImpl"  
>  
    <property name="hierarchy">  
        <value>  
            ROLE_ADMIN > ROLE_STAFF  
            ROLE_STAFF > ROLE_USER  
            ROLE_USER > ROLE_GUEST  
        </value>  
    </property>  
</bean>
```

Granted Authority Versus Role

- **Deux moyens de donner des droits**
- **GrantedAuthority est plutôt destiné pour un privilège individuel**
 - `hasAuthority('READ_AUTHORITY')`
- **Rôle destiner à un découpage macro d'une application**
 - `hasRole("ADMIN")`
- **Mais cela reste un découpage sémantique**
 - Voir :
<http://www.baeldung.com/spring-security-granted-authority-vs-role>

Intégration dans les applications web

Introduction

Application Web et API REST

- **Les application web (stateful) et les APIs REST (stateless) n'ont pas la même stratégie pour la gestion de la sécurité.**
 - Dans une application stateful, les informations liées à l'authentification sont stockées dans la session utilisateur (cookie).
 - Dans une application stateless, les droits de l'utilisateur sont transmis à chaque requête

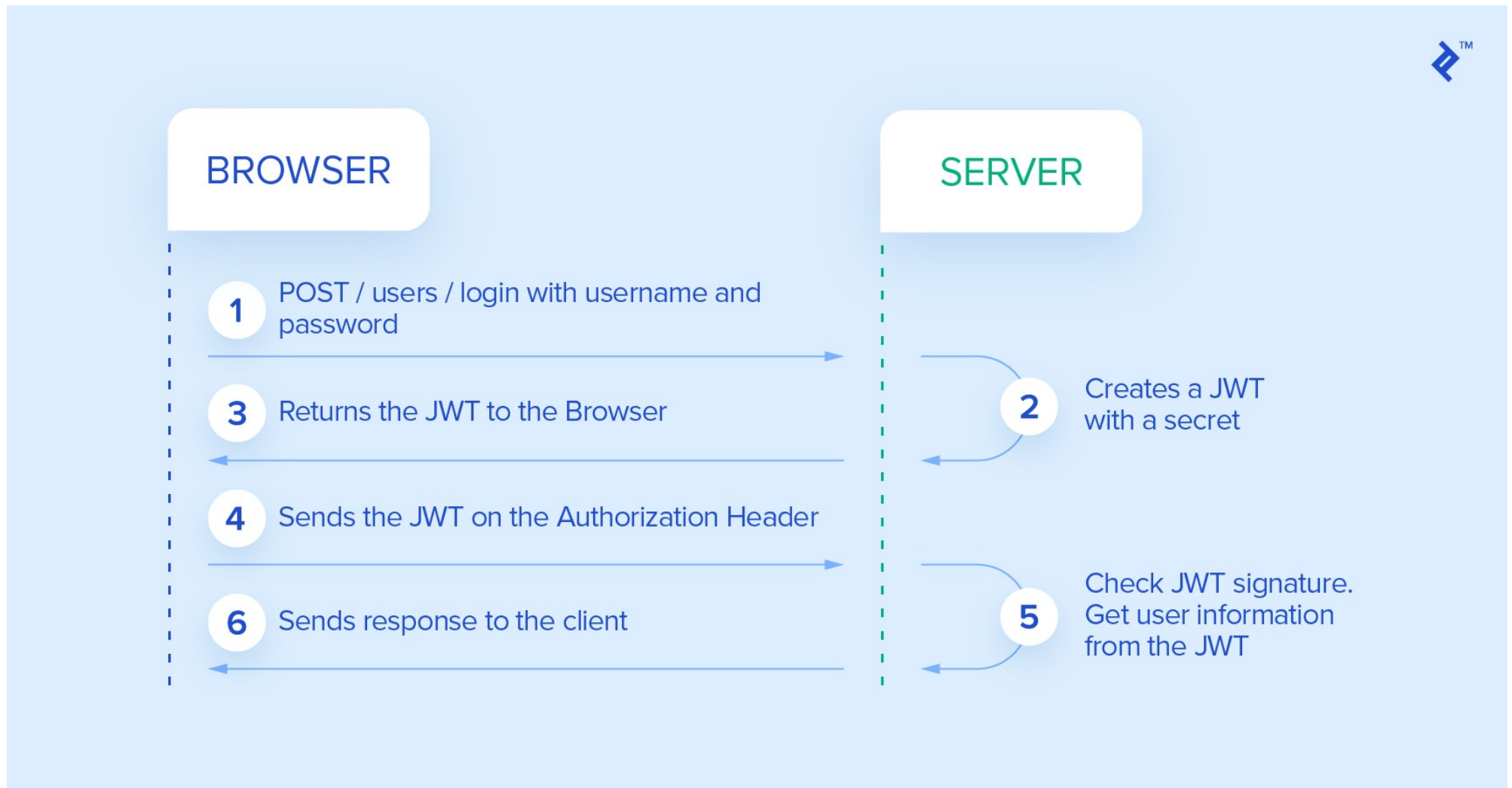
Processus d'authentification d'une appli web back-end

- **Le client demande une ressource protégée.**
- **Le serveur renvoie une réponse indiquant que l'on doit s'authentifier :**
 - En redirigeant vers une page de login
 - En fournissant les entêtes pour une authentication basique du navigateur .
- **Le navigateur renvoie une réponse au serveur :**
 - Soit le POST de la page de login
 - Soit les entêtes HTTP d'authentification.
- **Le serveur décide si les crédits sont valides :**
 - si oui. L'authentification est stockée dans la session, la requête originelle est réessayée, si les droits sont suffisants la page est retournée sinon un code 403
 - Si non, le serveur redemande une authentication.
- **L'objet *Authentication* contenant l'utilisateur et ses rôles est présent dans la session.**
 - Il est récupérable à tout moment par
`SecurityContextHolder.getContext().getAuthentication()`

Processus d'authentification appli REST

- **Le client demande une ressource protégée.**
- **Le serveur renvoie une réponse indiquant que l'on doit s'authentifier en envoyant une réponse 403.**
- **Le navigateur propose un formulaire de login puis envoie le formulaire sur un serveur d'authentification**
- **Le serveur d'authentification décide si les crédeniels sont valides :**
 - si oui. Il génère un token avec un délai de validité
 - Si non, le serveur redemande une authentification .
- **Le client récupère le jeton et l'associe à toutes les requêtes vers l'API**
- **Le serveur de ressources décrypte le jeton et déduit les droits de l'utilisateur.**
 - Il autorise ou interdit l'accès à la ressource

Authentication REST



Intégration dans les applications web

Appli web backend

Intégration dans l'API Servlet

- **Spring security est interrogeable depuis l'API Servlet (v2.5+)**
 - Et dans les JSP ou vue Thymeleaf
- **HttpServletRequest.getRemoteUser()**
 - == SecurityContextHolder.getContext().getAuthentication().getName()
- **HttpServletRequest.getUserPrincipal()**

```
Authentication auth = httpServletRequest.getUserPrincipal();  
MyCustomUserDetails userDetails = (MyCustomUserDetails) auth.getPrincipal();  
String firstName = userDetails.getFirstName();  
String lastName = userDetails.getLastName();
```

- **HttpServletRequest.isUserInRole(String)**

```
boolean isAdmin = httpServletRequest.isUserInRole("ADMIN");
```

Intégration dans l'API Servlet (2)

- **HttpServletRequest.authenticate(HttpServletRequest, HttpServletResponse)**
 - True si le user est authentifié
- **HttpServletRequest.login(String, String)**
 - Authentification du user
- **HttpServletRequest.logout()**
- **AsyncContext.start(Runnable)**
 - Permet la propagation de l'authentification dans le Thread
- **HttpServletRequest#changeSessionId()**
 - Pour se protéger d'une faille dans l'API Servlet 3.1

Localisation des messages

- **Pour localiser les messages d'erreur de Spring Security, il faut définir un bean `messageSource`**
- **Le bean peut charger les messages à partir d'un `resource bundle`**
 - `classpath:org/springframework/security/messages` contient les messages dans les différentes langues
- **La locale est défini via l'entête `Accept-Language` et on peut positionner une locale par défaut**

```
@Bean
public MessageSource messageSource() {
    Locale.setDefault(Locale.ENGLISH);
    ReloadableResourceBundleMessageSource messageSource = new
ReloadableResourceBundleMessageSource();
    messageSource.addBasenames("classpath:org/springframework/security/messages");
    return messageSource;
}
```


Sécurité et Thymeleaf

- **Une dépendance supplémentaire :**

- org.thymeleaf.extras : thymeleaf-extras-springsecurity5

- **Le dialecte Spring Security permet**

- d'afficher de manière conditionnelle du contenu en fonction des rôles d'utilisateur, des autorisations ou d'autres expressions de sécurité.
- d'avoir accès à Spring Authentication

```
<div sec:authorize="hasRole('USER')">Text visible to user.</div>
<div sec:authorize="hasRole('ADMIN')">Text visible to admin.</div>
<div sec:authorize="isAuthenticated()">
    Text visible only to authenticated users.
</div>
Authenticated username:
<div sec:authentication="name"></div>
Authenticated user roles:
<div sec:authentication="principal.authorities"></div>
```



TP 5.1

- **Possibilité d'ajouter une page d'erreur de droits**

```
<access-denied-handler error-page="/errors/403" />
```

```
http.exceptionHandling().accessDeniedPage("/forbidden.jsp");
```

- **Cela permet d'avoir une page plus jolie que le 403 par défaut du serveur d'application**

WebFlux Security

- L'API WebFlux à ses propres paramètres de sécurité

@EnableWebFluxSecurity

```
public class HelloWebfluxSecurityConfig {

    @Bean
    public MapReactiveUserDetailsService userDetailsRepository() {
        UserDetails user = User.withDefaultPasswordEncoder()
            .username("user")
            .password("user")
            .roles("USER")
            .build();
        return new MapReactiveUserDetailsService(user);
    }

    @Bean
    public SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {
        http
            .authorizeExchange()
                .anyExchange().authenticated()
                .and()
            .httpBasic().and()
            .formLogin();
        return http.build();
    }
}
```

Intégration dans les applications web

Api REST et JWT

JWT

- ***JSON Web Token (JWT)* est un standard ouvert défini dans la RFC 75191.**
 - Il permet l'échange sécurisé de jetons (tokens) entre plusieurs parties.
 - La sécurité consiste en la vérification de l'intégrité des données à l'aide d'une signature numérique. (HMAC ou RSA).
- **Dans le cadre d'une application REST SpringBoot, le jeton contient les informations d'authentification d'un user :**
 - Subject + GrantedAuthorities
- **Différentes implémentations existent en Java, dont *io.jsonwebtoken***

Mise en place avec Spring Security

- **La mise en place avec Spring Security dans le cadre d'une API REST stateless nécessite plusieurs étapes :**
 - Fournir un point d'accès permettant l'authentification et la génération d'un Jeton au format JWT
 - A configurer la chaîne de filtre, afin :
 - d'exclure la session de la sécurité
 - d'introduire un filtre traitant le jeton JWT
 - Implémenter le filtre qui extrait le jeton, le valide et si succès positionne un objet Authentication ou lève une exception
 - Mettre à disposition un utilitaire capable de générer un jeton, de le décoder et de le valider

Configuration filtre

```
@Autowired
TokenProvider tokenProvider ; // Générateur et validateur de Jeton

@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        .csrf().disable() // Jeton csrf n'est plus nécessaire
        .and() // Rien dans la session HTTP
        .sessionManagement()
        .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
        .and()
        .authorizeRequests() // ACLs
        .antMatchers("/api/authenticate").permitAll() // Point d'accès pour la génération
        .anyRequest().authenticated()
        .and()
        .addFilterBefore(new JWTFilter(tokenProvider),
                        UsernamePasswordAuthenticationFilter.class); // Configuration filtre
}
```

Implémentation du filtre

```
public class JWTFilter extends GenericFilterBean {
    private TokenProvider tokenProvider; // Codage/Décodage du Token
    public JWTFilter(TokenProvider tokenProvider) {this.tokenProvider = tokenProvider;    }

    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain
filterChain)
        throws IOException, ServletException {
        HttpServletRequest httpRequest = (HttpServletRequest) servletRequest;
        String jwt = resolveToken(httpRequest);
        if (StringUtils.hasText(jwt) && this.tokenProvider.validateToken(jwt)) {
            Authentication authentication = this.tokenProvider.getAuthentication(jwt);
            SecurityContextHolder.getContext().setAuthentication(authentication);
        }
        filterChain.doFilter(servletRequest, servletResponse);
    }

    private String resolveToken(HttpServletRequest request){
        String bearerToken = request.getHeader(JWTConfigurer.AUTHORIZATION_HEADER);
        if (StringUtils.hasText(bearerToken) && bearerToken.startsWith("Bearer ")) {
            return bearerToken.substring(7, bearerToken.length());
        }
        return null;
    }
}
```


Classe utilitaire

```
public String createToken(Authentication authentication, Boolean rememberMe) {
    String authorities = authentication.getAuthorities().stream().map(GrantedAuthority::getAuthority)
        .collect(Collectors.joining(","));

    long now = (new Date()).getTime();
    Date validity = new Date(now + this.tokenValidityInMilliseconds);

    return Jwts.builder()
        .setSubject(authentication.getName())
        .claim(AUTHORITIES_KEY, authorities)
        .signWith(SignatureAlgorithm.HS512, secretKey)
        .setExpiration(validity)
        .compact();
}

public Authentication getAuthentication(String token) {
    Claims claims = Jwts.parser()
        .setSigningKey(secretKey)
        .parseClaimsJws(token)
        .getBody();

    Collection<? extends GrantedAuthority> authorities =
        Arrays.stream(claims.get(AUTHORITIES_KEY).toString().split(","))
            .map(SimpleGrantedAuthority::new)
            .collect(Collectors.toList());

    User principal = new User(claims.getSubject(), "", authorities);

    return new UsernamePasswordAuthenticationToken(principal, token, authorities);
}
```

Point d'accès pour l'authentification

```
@Autowired
TokenProvider tokenProvider ; // Générateur et validateur de Jeton

@RequestMapping("/authenticate")
public ResponseEntity<JWTToken> authorize(@RequestParam String login, @RequestParam String password) {

    UsernamePasswordAuthenticationToken authenticationToken =
        new UsernamePasswordAuthenticationToken(login, password);

    Authentication authentication = this.authenticationManager.authenticate(authenticationToken);
    SecurityContextHolder.getContext().setAuthentication(authentication);

    String jwt = tokenProvider.createToken(authentication, true);
    HttpHeaders httpHeaders = new HttpHeaders();
    httpHeaders.add(JWTConfigurer.AUTHORIZATION_HEADER, "Bearer " + jwt);
    return new ResponseEntity<>(new JWTToken(jwt), httpHeaders, HttpStatus.OK);
}
```



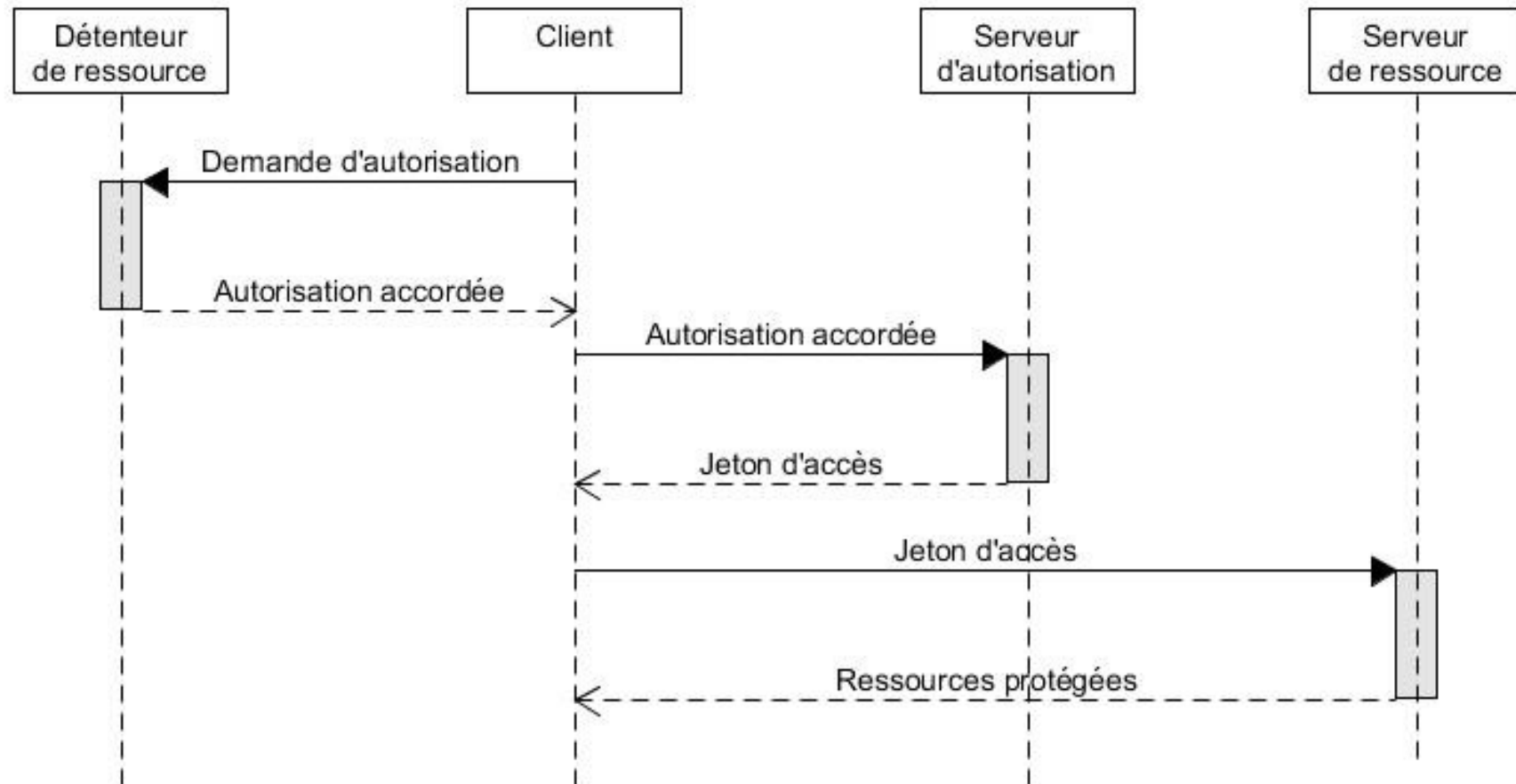
TP 5.2

Intégration dans les applications web oAuth2

Rappels OAuth2 – rôles du protocole

- **Le Client** est l'application qui essaie d'accéder au compte utilisateur. Elle a besoin d'obtenir une permission de l'utilisateur pour le faire.
- **Le serveur de ressources** est l'API utilisée pour accéder aux ressources protégées
- **Le serveur d'autorisation** est le serveur qui autorise un client à accéder aux ressources en lui fournissant un jeton. Il peut demander l'approbation de l'utilisateur
- **L'utilisateur** est la personne qui donne accès à certaines parties de son compte
- **Rq** : Un participant du protocole peut jouer plusieurs rôles

Séquence



Scénario

- Pré-enregistrer le client auprès du service d'autorisation (=> client ID et un secret)
- Obtenir l'autorisation de l'utilisateur.
(4 types de grant)
- Vérifier la réponse du service *oAuth* (state)
- Obtention du token (date d'expiration)
- Appel de l'API pour obtenir les informations voulues en utilisant le token

Jetons

- Les Tokens sont des chaînes de caractères aléatoire générées par le serveur d'autorisation
- Les jetons sont ensuite présents dans les requêtes HTTP et contiennent des informations sensibles => HTTPS
- Il y a 2 types de token
 - Le **jeton d'accès**: Il a une durée de vie limité.
 - Le **Refresh Token**: Délivré avec le jeton d'accès. Il est renvoyer au serveur d'autorisation pour renouveler le jeton d'accès lorsqu'il a expiré

Scope

- Le **scope** est un paramètre utilisé pour limiter les droits d'accès d'un client
- Le serveur d'autorisation définit les *scopes* disponibles
- Le client peut préciser le *scope* qu'il veut utiliser lors de l'accès au serveur d'autorisation

Enregistrement du client

- Le protocole ne définit pas comment l'enregistrement du client doit se faire mais définit les paramètres d'échange.
- Le client doit fournir :
 - ***Application Name***: Le nom de l'application
 - ***Redirect URLs***: Les URLs du client pour recevoir le code d'autorisation et le jeton d'accès
 - ***Grant Types*** : Les types d'autorisations utilisables par le client
 - ***Javascript Origin*** (optionnel): Le host autorisé à accéder aux ressources via *XMLHttpRequest*
- Le serveur répond avec :
 - ***Client Id***:
 - ***Client Secret***: Clé devant rester confidentielle

OAuth2 Grant Type

- Différents moyens afin que l'utilisateur donne son accord : les grant types
 - **authorization code** :
 - L'utilisateur est dirigé vers le serveur d'autorisation
 - L'utilisateur consent sur le serveur d'autorisation
 - Il est redirigé vers le client avec un code d'autorisation
 - Le client utilise le code pour obtenir le jeton
 - **implicit** : Jeton fourni directement. Certains serveurs interdisent de mode
 - **password** : Le client fournit les créidentiels de l'utilisateur
 - **client credentials** : Le client est l'utilisateur
 - **device code** :

Usage du jeton

- Le jeton est passé à travers 2 moyens :
 - Les paramètres HTTP. (Les jetons apparaissent dans les traces du serveur)
 - ***L'entête d'Authorization***
- GET /profile HTTP/1.1
- Host: api.example.com
- **Authorization: Bearer MzJmNDc3M2VjMmQzN**

Apport de SpringBoot

- **Spring Boot 2.x a revu son support pour OAuth2. Il offre 3 starters :**
 - OAuth2 Client : Intégration pour utiliser un login OAuth2 fournit par Google, Github, Facebook, ...
 - OAuth2 Resource server : Application permettant de définir des ACLs par rapport aux scopes client et aux rôles contenu dans des jetons OAuth
 - Okta : Pour travailler avec le fournisseur OAuth Okta

Exemple se logger avec Google

- Ajouter le starter **spring-boot-starter-oauth2-client**
- Déclarer un identifiant **oAuth** sur la **Google console** avec l'**URL de redirection de SpringBoot**
 - <http://localhost:8080/login/oauth2/code/google>
 - Récupérer le **clientId** et le **clientSecret**
- Configurer les propriétés
 - `spring.security.oauth2.client.registration.google.client-id`
 - `spring.security.oauth2.client.registration.google.client-secret`
- => **auto-configuration de OAuth2ClientAutoConfiguration**
- Configurer la sécurité avec **oAuthLogin()**

```
http.authorizeRequests().anyRequest().authenticated().and().oauth2Login();
```

Auditing

Auditing et actuator

- **Actuator permet un point d'accès /auditevents si**
 - Le starter actuator est dans le classpath
 - Un Bean de type *EventRepository* est présent dans le contexte

```
@Bean
public InMemoryAuditEventRepository repository(){
    return new InMemoryAuditEventRepository();
}
```

- */actuator/auditevents*

```
{
  "timestamp": "2020-06-05T06:06:43.894Z",
  "principal": "anonymousUser",
  "type": "AUTHORIZATION_FAILURE",
  "data": {
    "details": {
      "remoteAddress": "127.0.0.1",
      "sessionId": null
    },
    "type": "org.springframework.security.access.AccessDeniedException",
    "message": "Accès refusé"
  }
}
```

EventListener

- On peut également implémenter son propre bean à l'écoute des événements de Log
 - Il suffit d'annoter une méthode prenant un seul argument de type *AuditApplicationEvent* avec *@EventListener*

```
@Component
public class LoginAttemptsLogger {

    @EventListener
    public void auditEventHappened(
        AuditApplicationEvent auditApplicationEvent) {

        AuditEvent auditEvent = auditApplicationEvent.getAuditEvent();
        System.out.println("Principal " + auditEvent.getPrincipal()
            + " - " + auditEvent.getType());

        WebAuthenticationDetails details =
            (WebAuthenticationDetails) auditEvent.getData().get("details");
        System.out.println("Remote IP address: "
            + details.getRemoteAddress());
        System.out.println("Session Id: " + details.getSessionId());
    }
}
```


Ajouter des informations d'audit

- On peut également influencer sur les informations d'audit produites.
- Il faut fournir alors son propre bean de type *AuthorizationAuditListener*
-

```
@Component
public class ExposeAttemptedPathAuthorizationAuditListener
    extends AbstractAuthorizationAuditListener {

    public static final String AUTHORIZATION_FAILURE
        = "AUTHORIZATION_FAILURE";

    @Override
    public void onApplicationEvent(AbstractAuthorizationEvent event) {
        if (event instanceof AuthorizationFailureEvent) {
            onAuthorizationFailureEvent((AuthorizationFailureEvent) event);
        }
    }
}
```

Exemple d'implémentation

```
private void onAuthorizationFailureEvent(
    AuthorizationFailureEvent event) {
    Map<String, Object> data = new HashMap<>();
    data.put(
        "type", event.getAccessDeniedException().getClass().getName());
    data.put("message", event.getAccessDeniedException().getMessage());
    data.put(
        "requestUrl", ((FilterInvocation)event.getSource()).getRequestUrl() );

    if (event.getAuthentication().getDetails() != null) {
        data.put("details",
            event.getAuthentication().getDetails());
    }
    publish(new AuditEvent(event.getAuthentication().getName(),
        AUTHORIZATION_FAILURE, data));
}
```



TP Audit

Tester la sécurité

Spring Test

- **Rappel**

- `@RunWith(SpringJUnit4ClassRunner.class)`
 - Permet de lancer un test via l'API Spring Test
- `@ContextConfiguration("/spring/application-config.xml")`
 - Va chercher le contexte de test
 -

- **JUnit5**

- `@ExtendWith(SpringExtension.class)`

- **SpringBoot**

- Tests auto-configurés : `@WebMvcTest`, `@DataJpaTest`, `@JsonText`
- Mocks : `MockMvc`, `@MockBean`

Spring Security context Test

- **Rajoute des annotations de test**
- **@WithMockUser(user)**
 - @WithMockUser(username="admin",roles={"USER","ADMIN"})
- **@WithAnonymousUser**
- **On peut tester si oui ou non un user à accès a une ressource**


```
@Test(expected = AccessDeniedException.class)
@WithAnonymousUser
public void anonymous() throws Exception {
    countryService.deleteCountry("Suede");
}
```

Spring Security context Test (2)

- **Possibilité de créer des annotations customs**

- Et réutilisable
- En exemple un administrateur

```
@Retention(RetentionPolicy.RUNTIME)  
@WithMockUser(value="rob",roles="ADMIN")  
public @interface WithMockAdmin { }
```



TP 6.1

Spring Test MVC

- **Il est possible de tester les droits sur une application Spring MVC**
 - De base Spring-MVC possède une API de test
 - @WebAppConfiguration pour initialiser un test MVC
 - MockMvc permet de simuler des requêtes HTTP
 - Et tester les résultats (contenu, code HTTP, model etc.)
- **Il existe une surcouche sécurité à ce framework**
 - Permettant de rajouter un contexte de sécurité (UserDetail)
 - Permettant de rajouter des paramètres de test (CSRF)

Spring Webflux Test

```
@RunWith(SpringRunner.class)
@ContextConfiguration(classes = HelloWebfluxMethodApplication.class)
public class HelloWorldMessageServiceTests {
    @Autowired
    HelloWorldMessageService messages;

    @Test
    public void messagesWhenNotAuthenticatedThenDenied() {
        StepVerifier.create(this.messages.findMessage())
            .expectError(AccessDeniedException.class)
            .verify();
    }

    @Test
    @WithMockUser
    public void messagesWhenUserThenDenied() {
        StepVerifier.create(this.messages.findMessage())
            .expectError(AccessDeniedException.class)
            .verify();
    }

    @Test
    @WithMockUser(roles = "ADMIN")
    public void messagesWhenAdminThenOk() {
        StepVerifier.create(this.messages.findMessage())
            .expectNext("Hello World!")
            .verifyComplete();
    }
}
```


Spring Test MVC (2)

- **Exemple :**

```
@Test
public void testAccesProtectedUrl() throws Exception {
    mvc.perform(post("/url").with(csrf()).with(user("toto")).andDo(print()))
        .andExpect(status().isOk());
}
```

- **Notes:**

- Nécessite l'API servlet 3.0
- Permet aussi de tester le login, logout, les accès anonymes
- L'URL de retour
- La View MVC retournée
- Compatible avec les annotations @MockUser



Configuration avancée des requêtes HTTP

Protection contre les attaques CSRF

- **CSRF ?**

- Cross-Site Request Forgery

- **Comment ?**

- En faisait exécuter une requête HTTP dont on n'a pas les droits à un utilisateur qui à les droits (admin)
- « Hey, tu peux aller voir sur le lien <http://appli/dropAllTables> STP ?»
- Si l'admin est loggué, alors l'action sera réalisée.

- **Solution :**

- Le _CSRF Token
- Un token généré au login
- Tout les formulaires doivent contenir ce token
- Le token n'est connu que par une personne

CSRF protection

- **Depuis Spring 5 activé par défaut**

- Désactivable
 - `http.csrf().disable();`
- Généré automatiquement si la page de login est générée par Spring
- Sinon à rajouter manuellement dans la JSP

```
<input type="hidden" name="<c:out value="$  
{_csrf.parameterName}" />" value="<c:out  
value="$ {_csrf.token}" />" />
```

- Ou en utilisant le tag `jsp security`

```
<sec:csrfInput />
```

CRSF protection (2)

▼ Form Data view source view URL encoded

username: user
password: password
submit: Valider
_csrf: 1926e4c1-d605-4915-a65a-96885179775a

CORS

- **CORS ?**
 - Cross-Origin Ressource Sharing
- **Pourquoi faire ?**
 - Réceptionner des requêtes venant de l'extérieur ?
- **Comment :**
 - En rajoutant un filtre dédié : CorsFilter

CORS (2)

- **Exemple**

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.cors();
}

@Bean
CorsConfigurationSource corsConfigurationSource() {
    CorsConfiguration configuration = new CorsConfiguration();
    configuration.setAllowedOrigins(Arrays.asList("https://example.com"));
    configuration.setAllowedMethods(Arrays.asList("GET", "POST"));
    UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
    source.registerCorsConfiguration("/**", configuration);
    return source;
}
```

En-têtes de réponse HTTP

- **Par défaut Spring ajoute les headers suivants :**
 - Cache-Control:no-cache, no-store, max-age=0, must-revalidate
 - Expires:0
 - Pragma:no-cache
 - X-Content-Type-Options:nosniff
 - X-Frame-Options:DENY
 - X-XSS-Protection:1; mode=block
- **On peut**
 - Rajouter des headers
 - Désactiver les headers par défaut
 - Modifier certains headers

En-têtes de réponse HTTP

- **Désactivation de la configuration par défaut**

```
//désactivation des headers par défaut  
http.headers().defaultsDisabled();
```

En-têtes de réponse HTTP (cache)

- **De base pas de cache pour les ressources sécurisées**
 - On peut le désactiver/activer manuellement

```
//désactivation des headers par défaut  
http.headers().defaultsDisabled();
```

- **Spring MVC permet de supprimer le cache sur les ressources**

```
@EnableWebMvc  
public class MvcConf implements WebMvcConfigurer {  
  
    @Override  
    public void addResourceHandlers(ResourceHandlerRegistry registry) {  
        registry  
            .addResourceHandler("/resources/**")  
            .addResourceLocations("/resources/")  
            .setCachePeriod(31556926);  
    }  
}
```

En-têtes de réponse HTTP (Frames)

- **De base interdites**

- Et activable

```
http.headers().frameOptions().disable();  
http.headers().frameOptions().deny();  
http.headers().frameOptions().sameOrigin();
```

En-têtes de réponse HTTP (XSS-Protection)

- **XSS**
 - Cross-Site Scripting
- **De base présent**
 - Et modifiable

```
http.headers().xssProtection().disable();  
http.headers().xssProtection().xssProtectionEnabled(true);  
http.headers().xssProtection().block(true);
```

- **Web Services Security**
- **Protocole de communications qui permet d'appliquer de la sécurité aux services web**
- **WS-Security répond à trois problématiques principales :**
 - Comment signer les messages SOAP pour en assurer l'intégrité (éviter la transformation par un tiers) et la non-répudiation.
 - Comment chiffrer les messages SOAP pour en assurer la confidentialité.
 - Comment attacher des jetons de sécurité pour garantir l'identité de l'émetteur.

Conclusion

Rappel à l'ordre

- **Ça ne sert à rien de faire de la sécurité sans HTTPS !**
 - Sinon les identifiants passent en clairs sur le réseau
- **Pensez à activer l'HTTPS sur les serveurs de production**
 - Obligatoire si l'application est accessible depuis l'extérieur
- **Les authentications 'chiffrés' ne sont pas sécurisés**
 - Par exemple Digest fait des MD5

Spring Security

- **Un framework (pas simple) mais complet**
- **Complexe à mettre en place**
 - Mais facile à maintenir
- **Séparation claire et simple entre le fonctionnel et la sécurité**
- **100 % Spring**
- **S'intègre avec la majorité des providers de sécurité**
- **Hautement customisable**
- **Open source (et gratuit)**

Pour aller plus loin

- **Spring Boot !**

- Starter de projet web Spring
- Customisable (Security, MVC, data, batch etc.)
- Permet de construire une application Spring en quelques minutes
- Mais semble magique au début...

- **Jhipster**

- Générateur de code d'application basé sur Spring boot
- Avec des modules en plus
 - Angular 5 (avec génération de front) et bientôt React
 - Génération de Model/Dao/Service
 - Authentification/création de compte/mailling/administration généré
 - Compatible cloud (Docker, microservice, loadbalancing et serveur de configuration)

Annexes Dépréciées

Mise en place sans SpringBoot

Implémentation sur une application Web

- **Comment j'installe Spring Security moi ?**
- **Deux dépendances minimales**
 - `org.springframework.security.spring-security-web`
 - `org.springframework.security.spring-security-config`
- **Modules complémentaires selon les choix d'implémentations**
 - `spring-security-ldap`, `spring-security-cas`, `spring-security-openid`, ...
- **Si j'utilise Spring-boot, toutes les dépendances sont automatiquement raménées avec le starter :**
 - `org.springframework.boot:spring-boot-starter-security`

Mise en place (XML)

- **Configuration du filtre de sécurité (dans le web.xml)**
 - Prend en charge toutes les URLs (« /* »)

```
<filter>
<filter-name>springSecurityFilterChain</filter-name>
<filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
</filter>

<filter-mapping>
<filter-name>springSecurityFilterChain</filter-name>
<url-pattern>/*</url-pattern>
</filter-mapping>
```

Mise en place (Java Config)

- En ayant activé la configuration par défaut via annotation (ou via les dépendances Spring Boot)

```
@EnableWebSecurity(debug = true)
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {
}
```

- Dans les deux cas, on initialise une `springSecurityFilterChain`
- On profite de l'auto-configuration
 - Toutes les URLs sont protégées
 - L'authentification par formulaire est activée, un formulaire par défaut est généré
 - L'utilisateur peut se délogger
 - Des filtres protègent contre les attaques classiques
 - Les entêtes HTTP de sécurité sont intégrées

Mise en place (Java Config)

- Il faut maintenant définir un bean userDetailsService

```
@Bean
public UserDetailsService userDetailsService() {
    UserDetails user =
User.withDefaultPasswordEncoder().username("user").password("password").roles(
"USER")
        .build();
    UserDetails user2 =
User.withDefaultPasswordEncoder().username("admin").password("password")
        .roles("USER", "ADMIN").build();
    return new InMemoryUserDetailsManager(user, user2);
}
```

- Et définir les ressources sécurisées

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests().anyRequest().authenticated();
}
```

Security : Par défaut

- **Par défaut HTTP génère**

- Une protection des pages web
- Protection CSRF
- Des headers
 - HTTP Strict Transport Security for secure requests
 - X-Content-Type-Options integration
 - Cache Control (can be overridden later by your application to allow caching of your static resources)
 - X-XSS-Protection integration
 - X-Frame-Options integration to help prevent Clickjacking
- Des Servlets
 - `HttpServletRequest#getRemoteUser()`
 - `HttpServletRequest.html#getUserPrincipal()`
 - `HttpServletRequest.html#isUserInRole(java.lang.String)`
 - `HttpServletRequest.html#login(java.lang.String, java.lang.String)`
 - `HttpServletRequest.html#logout()`

Authentication Digest

Authentication Digest

- **Amélioration du http basic**
 - Avec cryptage du mot de passe dans le réseau
 - Mise en place pour éviter les mots de passes en clairs
 - Mais n'est plus considéré comme sécurisé
- **Nécessité de ne pas crypter son mot de passe en base**
- **Le serveur envoie un 'nonce' de la forme**

```
base64(expirationTime + ":" + md5Hex(expirationTime + ":" + key))
```

expirationTime: The date and time when the nonce expires, expressed in milliseconds

key: A private key to prevent modification of the nonce token

Authentication Digest (2)

```
<bean id="digestFilter" class=
"org.springframework.security.web.authentication.www.DigestAuthentication
Filter">
<property name="userService" ref="jdbcDaoImpl"/>
<property name="authenticationEntryPoint" ref="digestEntryPoint"/>
<property name="userCache" ref="userCache"/>
</bean>

<bean id="digestEntryPoint" class=
"org.springframework.security.web.authentication.www.DigestAuthentication
EntryPoint">
<property name="realmName" value="Contacts Realm via Digest
Authentication"/>
<property name="key" value="acegi"/>
<property name="nonceValiditySeconds" value="10"/>
</bean>
```



TP 3.2

Sécurité et JSP

Taglib Security

- **Rajoute des tags sécurité dans les .jsp**

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

- **Authorize : activation selon les rôles**

```
<sec:authorize access="hasRole('supervisor')">
```

This content will only be visible to users who have the "supervisor" authority in their list of `<tt>GrantedAuthority</tt>`s.

```
</sec:authorize>
```

```
<sec:authorize url="/admin">
```

This content will only be visible to users who are authorized to send requests to the "/admin" URL.

```
</sec:authorize>
```

Taglib Security (3)

- **D'autres tags mineurs**

- Authentication (bientôt deprecated)
- Accesscontrollist
- CsrfInput , csrfMetaTags (CSRF)

Taglib Example

- **Afficher les rôles d'un utilisateur**

```
<sec:authentication property="principal.authorities"
    var="authorities" />
<c:forEach items="${authorities}" var="authority" varStatus="vs">
    <p>${authority.authority}</p>
</c:forEach>
```