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**
  - Dédiée 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

Authentification

- **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/webflux* et s*ecurity* 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 redéfinir les beans Spring impliqués dans la sécurité :**
  - ***SecurityFilterChain*** permet de définir la chaîne de filtres traitant les requêtes HTTP entrantes, chaque élément de la chaîne a une responsabilité fine (extraire un Jeton, stocker l'authentification dans la Session, vérifier les ACLs, …)
  - Définir l'authentification :
    - En utilisant le gestionnaire d'authentification par défaut (*AuthenticationManager*) et en configurant la liste des ***AuthenticationProvider*** à utiliser
    - En définissant directement le bean de type ***AuthenticationManager*** (Exemple instanciation d'un *LdapAuthenticationManager*)
    - En fournissant un classe de type ***UserDetailsService*** utilisé par la gestionnaire d'authentification de Spring
  - ***WebSecurityCustomizer***
    - Ce bean permet d'ignorer la sécurité pour certaines URLs (ressources statiques typiquement)

ProviderManager

- **ProviderManager est 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>
```

Configuration Java – Plusieurs Authentication Provider

```
@Autowired
   CustomAuthenticationProvider customAuthProvider;

   @Bean
   public AuthenticationManager authManager(HttpSecurity http) throws Exception {
      AuthenticationManagerBuilder authenticationManagerBuilder =
        http.getSharedObject(AuthenticationManagerBuilder.class);

      authenticationManagerBuilder.authenticationProvider(customAuthProvider);

      authenticationManagerBuilder.inMemoryAuthentication()
         .withUser("memuser")
         .password(passwordEncoder().encode("pass"))
         .roles("USER");

      return authenticationManagerBuilder.build();
   }
```

AuthenticationProvider

- **Des `AuthenticationProvider` pour toute situation :**
  - **`AuthByAdapterProvider`** : authentification depuis le conteneur
  - **`AnonymousAuthenticationProvider`** : identifie un anonymous
  - **`DaoAuthenticationProvider`** : info dans une base de données
  - **`CasAuthenticationProvider`** : authentification CAS
  - **`OAuth2LoginAuthenticationProvider`** : authentification oAuth2
  - **`JaasAuthenticationProvider`** : authentification JAAS
  - **`LdapAuthenticationProvider`** : authentification LDAP
  - **`RememberMeAuthenticationProvider`** : authentification 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 *AuthentificationProvider*
- **Note : tous les providers ne sont pas inclus dans le starter spring-security**
  - Mais dans des librairies spécifiques
- **Bien sur il n'est pas obligatoire de passer par un *AuthenticationProvider***
  - Mais il faut fournir un authentification manager

DaoAuthenticationProvider

- **Une implémentation basée sur l'interface UerDetailsService**
  - **`AuthentificationManager`** (**`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="jimispassword" authorities="ROLE_USER, ROLE_ADMIN" />
  <user name="bob" password="bobspassword" authorities="ROLE_USER" />
</user-service>
```

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

```
jimi=jimispassword,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:hsql://localhost:9001"/>
  <property name="username" value="sa"/>
  <property name="password" value=""/>
</bean>

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

```
@Bean
public JdbcDaoImpl userDetailsService() {
      JdbcDaoImpl jdbcDaoImpl = new JdbcDaoImpl();
      jdbcDaoImpl.setDataSource(dataSource());
      return jdbcDaoImpl;
}
```

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 (déprécié)*** : 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 : Authenttification LDAP
- **Il permet de positionner facilement un UserDetailService 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 Authentification Provider ?

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

Implémenter son UserDetailService

- **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 l’ 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**

Autres alternatives

- **Injection dans un contrôleur**

```
    @RequestMapping(value = "/username", method = RequestMethod.GET)
    @ResponseBody
    public String currentUserName(Principal principal) {
```

```
    @RequestMapping(value = "/username", method = RequestMethod.GET)
    @ResponseBody
    public String currentUserName(Authentication authentication) {
```

- **A partir de la requête**

```
@RequestMapping(value = "/username", method = RequestMethod.GET)
    @ResponseBody
    public String currentUserNameSimple(HttpServletRequest request) {
        Principal principal = request.getUserPrincipal();
        return principal.getName();
    }
```

A partir des vues

- **A partir d'une page JSP**

```
@RequestMapping(value = "/username", method = RequestMethod.GET)
    @ResponseBody
    public String currentUserNameSimple(HttpServletRequest request) {
        Principal principal = request.getUserPrincipal();
        return principal.getName();
    }
```

- **A partir d'une vue Thymeleaf**
  - Nécessite :
    - org.thymeleaf.extras :thymeleaf-extras-springsecurity5
    - org.thymeleaf:thymeleaf-spring5


```
<html xmlns:th="https://www.thymeleaf.org"
  xmlns:sec="https://www.thymeleaf.org/thymeleaf-extras-springsecurity5">
<body>
    <div sec:authorize="isAuthenticated()">
      Authenticated as <span sec:authentication="name"></span></div>
</body>
</html>
```

Login

- **http.formLogin()**
  - Construit une page de login par formulaire
  - Customisable...

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) {
    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**

Login

- **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) ;
```

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'interfacer 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 servlets ou les WebFilter pour la programmation réactive**
  - 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
- **Les classe *HttpSecurity ou ServerHttpSecurity* proposent des méthodes simplifiant la construction de cette chaîne**
  - En évitant la construction de nombreux beans
  - En évitant de ne pas respecter l'ordre imposé
  - En conséquence déclare 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

- **Les filtres disponibles évoluent avec les versions de Spring**
  - **`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`** , **`BasicAuthenticationFilter`**, **`DigestAuthenticationFilter`**, **`CasAuthenticationFilter`**, **`Saml2WebSsoAuthenticationFilter`**, **`OAuth2LoginAuthenticationFilter.`**..
  - **`SecurityContextHolderAwareRequestFilter`** : injection explicite du **`SecurityContext`**
  - **`RequestAwareCacheFilter`** : Met en cache une requête pour la rejouer plus tard
  - **`AnonymousAuthenticationFilter`** : garantir qu'un Authentication existe (anonyme)
  - **`ExceptionTranslationFilter`** : gère les exceptions de sécurité lors de l'authentification
  - **`AutorizationFilter`**: lève les exceptions si l'accès est interdit
  - **`oAuth2*Filter`** : Gère le protocole oAuth2
  - **`DefaultLoginPageGeneratingFilter`**, **`DefaultLogoutPageGeneratingFilter`** : Filtre générant des écans des login et logout par défaut
  - **`CorsFilter`**, **`CsrfFilter`** : Filtre de protection contre les attaques

Initialisation des filtres web

- **Les appels aux méthode de HttpSecurity / ServerHttpSecurity permettent de modifier ces filtres**
  - Soit rajouter des paramètres (exemple avec le 403)
  - Soit rajouter des filtres nouveaux (authentification par mot de passe)
  - On peut également rajouter manuellement des filtres

Exemple

```
    @Bean
public SecurityWebFilterChain securitygWebFilterChain(ServerHttpSecurity http) {
  return http.authorizeExchange()
         .pathMatchers("/actuator/**").permitAll()
         .pathMatchers("/auth/**").permitAll()}
         .anyExchange().authenticated()
         .and()
         .oauth2Login().csrf().disable().build();
}
```

AuthorizationFilter

- ***AuthorizationFilter* : S'occupe de vérifier les droits d'accès aux URL (autorisations)**
- **Il est configuré via la méthode authorizeHttpRequests**

```
@Bean
SecurityFilterChain web(HttpSecurity http) throws AuthenticationException {
    http
        .authorizeHttpRequests((authorize) -> authorize
            .anyRequest().authenticated();
        )
        // ...

    return http.build();
}
```

ExceptionTranslationFilter

- **ExceptionTranslationFilter : Gestion des pages 403.**
  - Prend la main 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>
```

UsernamePasswordAuthentificationFilter

- **Traite la soumission du formulaire de login**
  - Extrait le login et le mot de passe fourni
  - Appelle l'authentificationManager !

```
<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.

Remember me

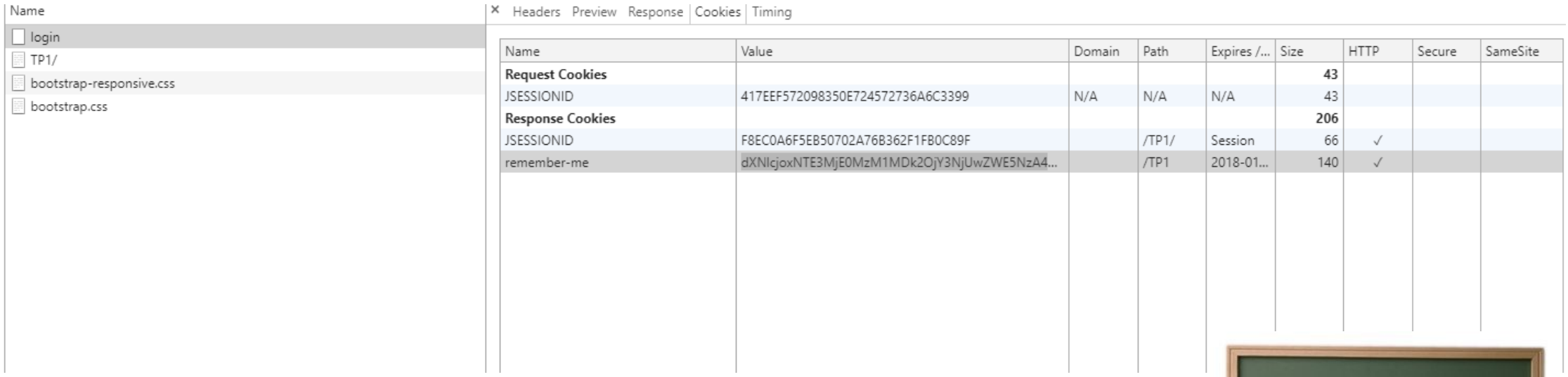
- **Se rappeler de l'identité entre 2 sessions grâce à un cookie**
- **2 méthodes**
  - Hash-Based Token (persistance en mémoire)
  - Persistence Token (persistance en base)
- **Nécessite un bean *UserDetailService***
- **Le token est généré par le serveur et envoyé sous forme de cookie au client**
-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



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

Authentification 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.WebAuthenticationDetai
ls@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 nécessire (défaut)
  - never : SS ne créé 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();
```

Fonctionnalités d'autorisations

Deux types de sécurisation

- **Sécurisation des applications web**
  - Utilisation de filtres (servlet filters/webfilter) 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 à 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`**
  - **`AuthorizationFilter`**: intercepte les requêtes HTTP
  - **`MethodSecurityInterceptor`** : intercepte les appels de méthode
  - **`AspectJMethodSecurityInterceptor`** : idem mais avec AspectJ

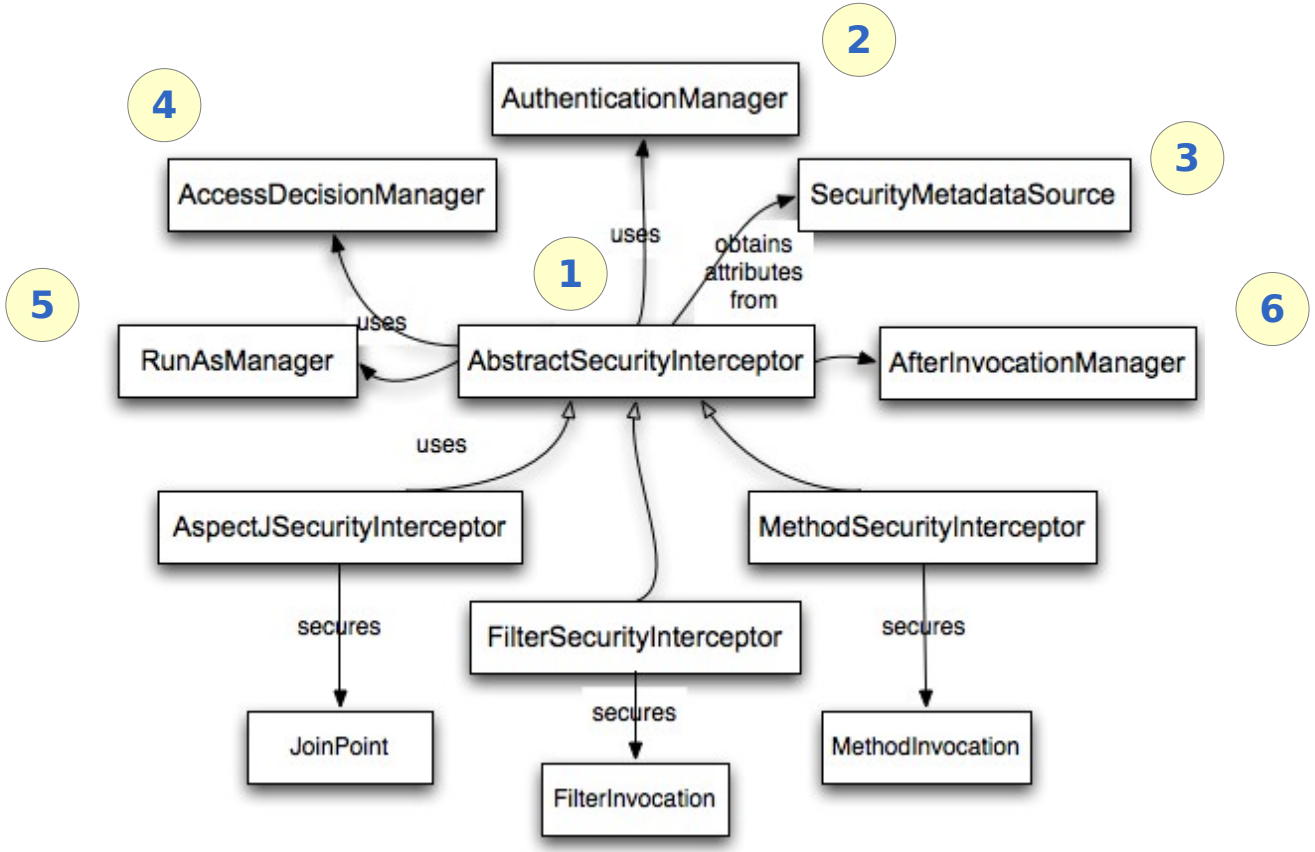


Schéma général : 4 niveaux de manager

- **AuthenticationManager**
  - Responsable de l'identification de l'utilisateur
- **AuthorizationManager**
  - 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)

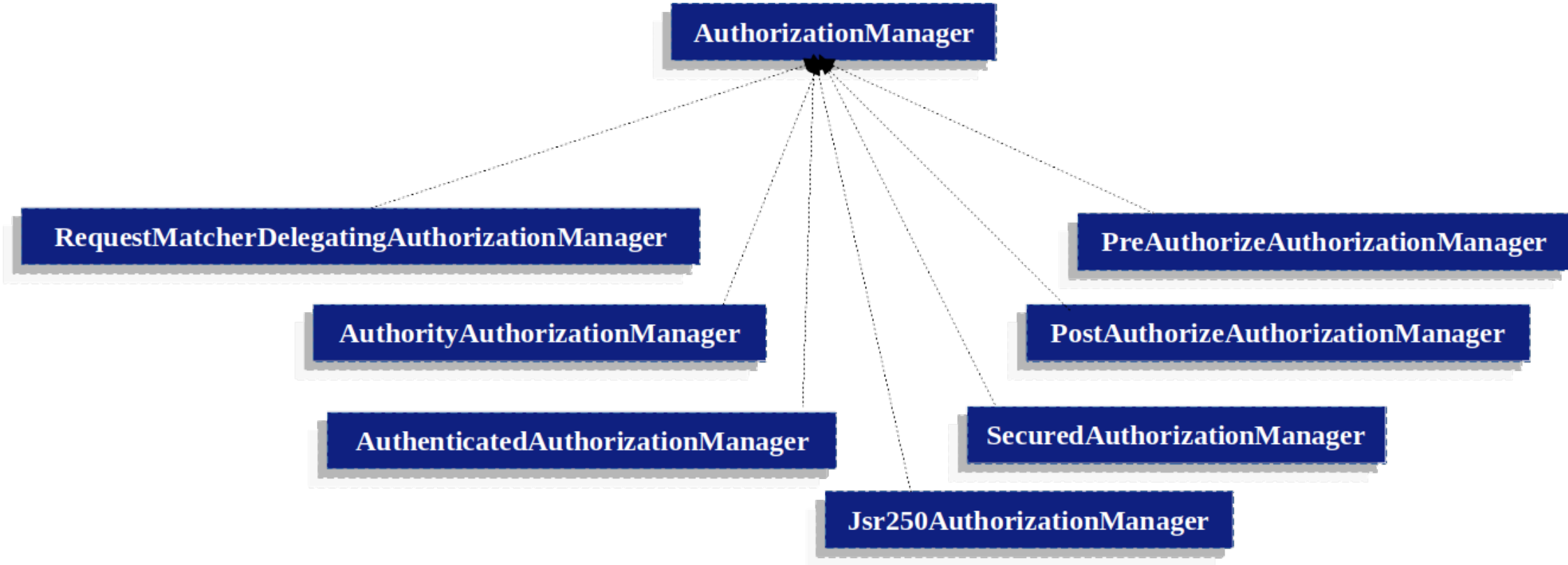
Autorisation

- **Basé sur les GrantedAuthority contenu dans Authentication**
- **AuthorizationManager : décide de l'accès à une ressource via ses 2 méthodes**
  - AuthorizationDecision check(Supplier<Authentication> authentication, Object secureObject);
  - default AuthorizationDecision verify(Supplier<Authentication> authentication, Object secureObject) throws AccessDeniedException { }
- **AuthorizationDecision: interface**
  - isGranted()
- **La méthode verify appel *check* et envoie un AccessDeniedException l'AuthorizationDecision est négative**
- **Spring Security fourni un AuthorizationManager qui collabore avec plusieurs autres AuthorizationManagers .**

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

Délégation


AuthorizationManager
RequestMatcherDelegatingAuthorizationManager
AuthorityAuthorizationManager
AuthenticatedAuthorizationManager
Jsr250AuthorizationManager
SecuredAuthorizationManager
PostAuthorizeAuthorizationManager
PreAuthorizeAuthorizationManager

Autorisation des requêtes Servlet HTTP

- **AuthorizationFilter remplace FilterSecurityInterceptor. Pour rester rétrocompatible, FilterSecurityInterceptor reste la valeur par défaut.**
- **Il est cependant recommandé d'utiliser AuthorizationFilter en utilisant authorizeHttpRequests à la place de authorizeRequests,**

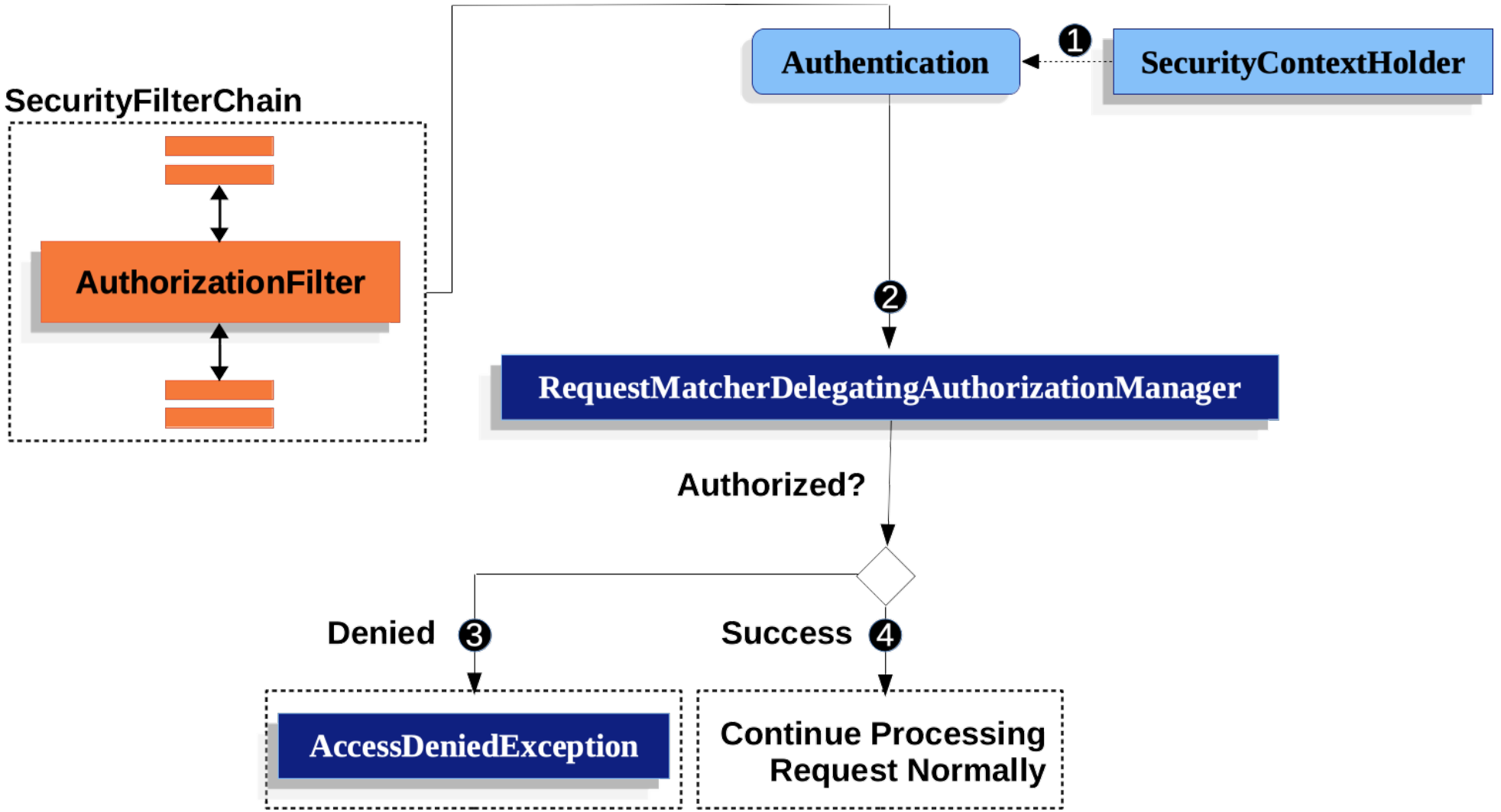
```
@Bean
SecurityFilterChain web(HttpSecurity http) throws AuthenticationException {
    http
        .authorizeHttpRequests((authorize) -> authorize
            .anyRequest().authenticated();
        )
        // ...

    return http.build();
}
```

Améliorations

- **Utilise l'API AuthorizationManager simplifiée au lieu des sources de métadonnées, des attributs de configuration, des gestionnaires de décision et des votants. Cela simplifie la réutilisation et la personnalisation.**

- **Retarde la recherche d'authentification. Au lieu de rechercher l'authentification pour chaque demande, il ne la recherchera que dans les demandes où une décision d'autorisation nécessite une authentification.**

- **Prise en charge de la configuration basée sur le bean.**


SecurityFilterChain
AuthorizationFilter
Authentication
1
SecurityContextHolder
2
RequestMatcherDelegatingAuthorizationManager
Authorized?
Denied
3
Success
4
AccessDeniedException
Continue Processing
Request Normally

Autorisation sur des requêtes

- **Rappel : Faire du plus spécifique au plus général**
  - Exemple de configuration

```
@Bean
SecurityFilterChain web(HttpSecurity http) throws Exception {
     http
          // ...
          .authorizeHttpRequests(authorize -> authorize
               .requestMatchers("/resources/**", "/signup", "/about").permitAll()
               .requestMatchers("/admin/**").hasRole("ADMIN")
               .requestMatchers("/db/**").access(new
WebExpressionAuthorizationManager("hasRole('ADMIN') and hasRole('DBA')"))
               //
.requestMatchers("/db/**").access(AuthorizationManagers.allOf(AuthorityAuthorizationManager.h
asRole("ADMIN"), AuthorityAuthorizationManager.hasRole("DBA")))
               .anyRequest().denyAll()
          );

     return http.build();
}
```

Spring Webflux

```
import static
org.springframework.security.authorization.AuthorityReactiveAuthorizationManager.hasRole;
// ...
@Bean
SecurityWebFilterChain springWebFilterChain(ServerHttpSecurity http) {
     http
          // ...
          .authorizeExchange((authorize) -> authorize                          (1)
               .pathMatchers("/resources/**", "/signup", "/about").permitAll()  (2)
               .pathMatchers("/admin/**").hasRole("ADMIN")                      (3)
               .pathMatchers("/db/**").access((authentication, context) ->      (4)
                    hasRole("ADMIN").check(authentication, context)
                         .filter(decision -> !decision.isGranted())
                         .switchIfEmpty(hasRole("DBA").check(authentication, context))
               )
               .anyExchange().denyAll()                                        (5)
          );
     return http.build();
}
```

RequestMatcher

- **L'interface RequestMatcher est utilisée pour déterminer si une requête correspond à une règle donnée**
- **securityMatcher permet de déterminer si la sécurité est appliqué à une requête donnée**

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

     @Bean
     public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
          http
               .securityMatcher("/api/**")
               .authorizeHttpRequests(authorize -> authorize
                    .requestMatchers("/user/**").hasRole("USER")
                    .requestMatchers("/admin/**").hasRole("ADMIN")
                    .anyRequest().authenticated()
               )
               .formLogin(withDefaults());
          return http.build();
     }
}
```

RequestMatcher

- **Les méthodes securityMatcher() et requestMatcher() décident de l'implémentation à utiliser en fonction de l'application. Cela peut être :**
  - MvcRequestMatcher
  - AntPathRequestMatcher
- **On peut également préciser l'implémentation que l'on désire**

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

     @Bean
     public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
          http
               .securityMatcher(antMatcher("/api/**"))
               .authorizeHttpRequests(authorize -> authorize
                    .requestMatchers(antMatcher("/user/**")).hasRole("USER")
                    .requestMatchers(regexMatcher("/admin/.*")).hasRole("ADMIN")
                    .requestMatchers(new MyCustomRequestMatcher()).hasRole("SUPERVISOR")
                    .anyRequest().authenticated()
               )
               .formLogin(withDefaults());
          return http.build();
     }
}
```

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.

Contrôle d’accès basé sur des expressions

- **Spring Security utilise SpEL pour les ACLs. Les expressions sont évalués par rapport à un objet racine de type SecurityExpressionRoot**
- **SecurityExpressionRoot propose les méthodes :**
  - hasRole(String rôle) : Un rôle est une Authority préfixée par ROLE_
  - hasAnyRole(String… roles)
  - hasAuthority(String authority)
  - hasAnyAuthority(String… authorities)
  - isAnonymous(), isAuthenticated()
  - isRememberMe(), isFullyAuthenticated()
  - ..
- **Et les champs suivants :**
  - principal
  - authentication

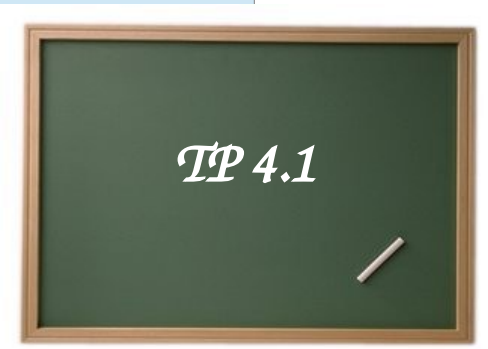
Fonctionnalités des expressions

- **Les expressions SpEL permettent de faire référence aux beans exposés,**

```
// websecurity est un bean contenant une méthode check
http
    .authorizeHttpRequests(authorize -> authorize
        .requestMatchers("/user/**").access(new
WebExpressionAuthorizationManager("@webSecurity.check(authentication,request)"))
        ...
    )
```

- **D'accéder aux variables de chemin de la requête**

```
// websecurity est un bean contenant une méthode checkUserId
http
     .authorizeHttpRequests(authorize -> authorize
          .requestMatchers("/user/{userId}/**")
          .access("@webSecurity.checkUserId(authentication,#userId)")
          ...
     );
```



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**

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

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

@EnableMethodSecurity

- **L’Annotation @EnableMethodSecurity sur une classe @Configuration remplace et améliore @EnableGlobalMethodSecurity de plusieurs façons:**
  - Utilise l'API AuthorizationManager simplifiée .
  - Favorise la configuration directe basée sur les beans, au lieu de nécessiter l'extension de GlobalMethodSecurityConfiguration pour personnaliser les beans
  - Est construit à l'aide de Spring AOP natif, supprimant les abstractions et vous permettant d'utiliser les blocs de construction Spring AOP pour personnaliser
  - Vérifie les annotations en conflit pour garantir une configuration de sécurité sans ambiguïté
  - Conforme à JSR-250
  - Active @PreAuthorize, @PostAuthorize, @PreFilter et @PostFilter par défaut

Alternatives pour la sécurisation

- **Par défaut @EnableMethodSecurity permet d'annoter les méthodes de la couche service via :**
  @PreAuthorize, @PostAuthorize, @PreFilter, et @PostFilter

- **@EnableMethodSecurity(securedEnabled=true) permet d'utiliser l'annotation Spring :**
  @Secured

- **@EnableMethodSecurity(jsr250Enabled = true) permet d'utiliser les annotations JavaEE :**
  @RolesAllowed, @PermitAll, @DenyAll

Spring Reactive

- **L’annotation @EnableReactiveMethodSecurity(useAuthorizationManager=true) sur une classe @Configuration améliore @EnableReactiveMethodSecurity**
  - Utilise l'API AuthorizationManager
  - Prend en charge les types de retour réactifs.
  - Est construit à l'aide de Spring AOP natif, supprimant les abstractions et vous permettant d'utiliser les blocs de construction Spring AOP pour personnaliser
  - Vérifie les annotations en conflit pour garantir une configuration de sécurité sans ambiguïté
  - Conforme à JSR-250

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`**, ...

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

Autorisation sur des objets et méthodes

- **@PreAuthorize(Expr) : Vérifie l'expression avant d'entrer dans la méthode**
- **@PostAuthorize(Expre) : Vérifie l'expression au retour de la méthode qui peut utiliser la valeur retournée**

```
@PostAuthorize
  ("returnObject.username == authentication.principal.nickName")
public CustomUser loadUserDetail(String username) {
```

- **@PreFilter(Expr) : filtre les collections en entrée d'une méthode**

```
@PreFilter
  (value = "filterObject != authentication.principal.username",
  filterTarget = "usernames")
public String joinUsernamesAndRoles(
```

- **@PostFilter(Expr) : filtre les collections en sortie d'une méthode**

```
@PostFilter("filterObject != authentication.principal.username")
public List<String> getAllUsernamesExceptCurrent() {
```

Expression-Based Acces 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

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 généralement à l'aide d'un jeton

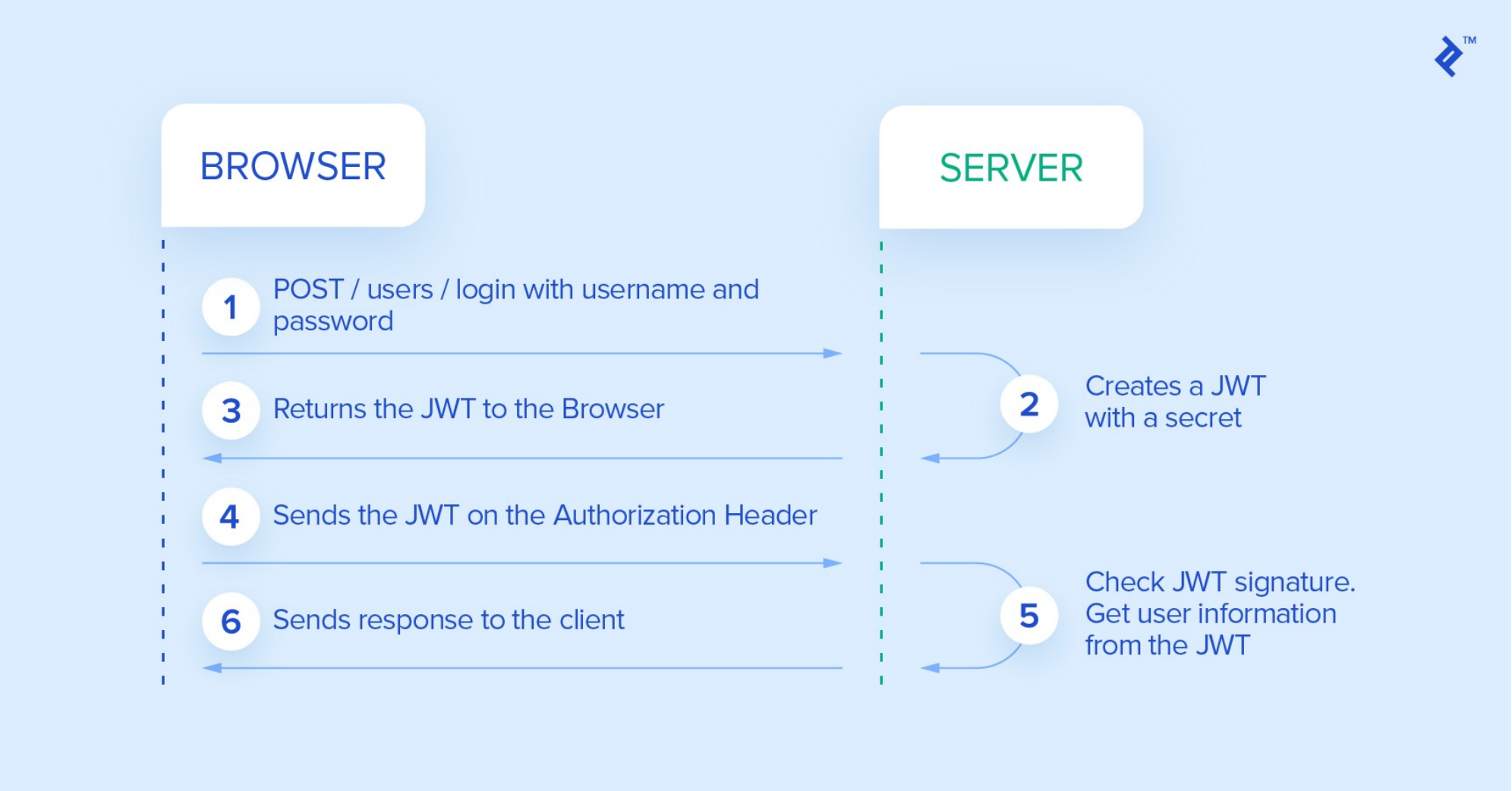
Processus d'authentification d'une appli web back-end

- **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 auhentification 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 crédentiels 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 authentification.
- **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 crédentiels 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

Exemple Authentification 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 proteger 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 ressource 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>
```

Page 403

- **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 léve 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 httpServletRequest = (HttpServletRequest) servletRequest;
        String jwt = resolveToken(httpServletRequest);
        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

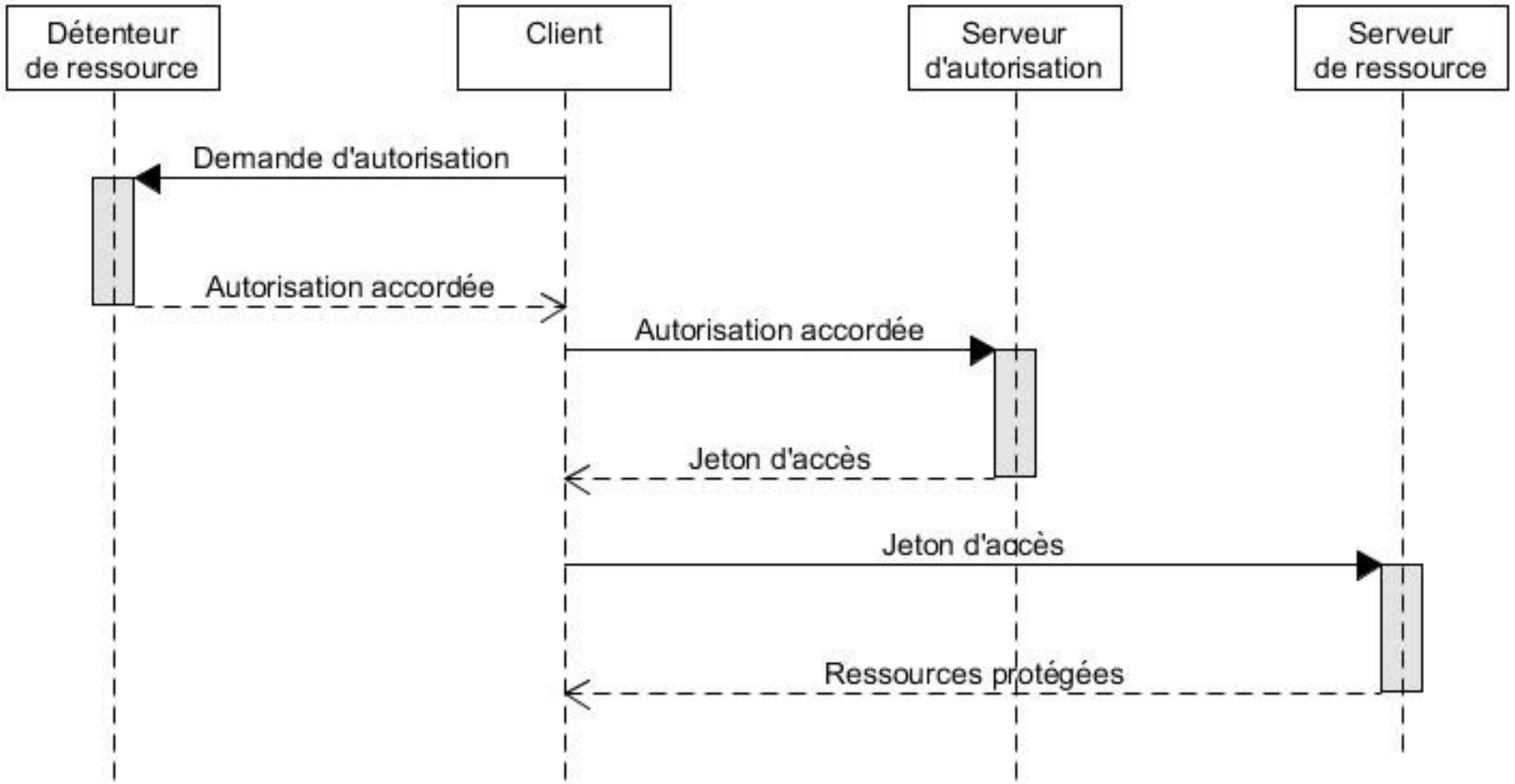
```
    @Autowired
    TokenProvider tokenProvider ; // Générateur et validateur de Jeton


@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);
}
```

oAuth2 / OpenID

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 a 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


Détenteur de ressource
Client
Serveur d'autorisation
Serveur de ressource
Demande d'autorisation
Autorisation accordée
Autorisation accordée
Jeton d'accès
Jeton d'accès
Ressources protégées

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 (entête Authorization) et contiennent des informations sensibles => HTTPS**
- **Il y a plusieurs types de jeyon**
  - Le **jeton d'accès**: Il a une durée de vie limité. Il contient les informations permettant de déduire les ACLs
  - Le ***Refresh Token***: Délivré avec le jeton d'accès. Il est renvoyé au serveur d'autorisation pour renouveler le jeton d'accès lorsque celui-ci a expiré

Contenu du jeton et scope

- **Le contenu du jeton peut être assez varié en fonction de la configuration du serveur d'autorisation.**
- **On y trouve généralement :**
  - Des informations sur le serveur d'autorisation
  - Des informations d'identification de l'utilisateur (login, email, …)
  - Des informations sur le rôle des utilisateur permettant une stratégie d'autorisation RBAC
  - Des informations sur le client oAuth et ses permissions : ses **scopes**

- **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 principalement fournir :**
  - ***Application Name***: Le nom de l'application
  - ***Redirect URLs***: Les URLs permises du client pour recevoir le code d'autorisation et le jeton d'accès
  - ***Grant Types*** : Les types de consentement utilisables par le client
- **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**
    - **Le serveur d'autorisation fournit un code d'autorisation via une URL de redirection**
    - **Le client utilise le code pour obtenir le jeton**
  - ***implicit*** **: Jeton fourni directement. Certains serveurs interdisent de mode**
  - ***password :*** **Le client fournit les crédentiels de l'utilisateur. Pas recommandé**
  - ***client credentials*** **: Le client est l'utilisateur, ses crédentiels suffisent à obtenir un jeton**
  - ***device code*** **: Mode utilisé lorsque il n'y pas de navigateur disponible sur le client**

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`**

Validation du jeton

Lors de la réception du jeton, le serveur de ressource doit valider l'authenticité du jeton et extraire ses informations différentes techniques sont possibles

- Appel REST vers le serveur d'autorisation
- Utilisation de JWT (Json Web Token) et validation via clé privé ou clé publique

Le format JWT est recommandé car il permet d'économiser un aller/retour vers le serveur d'autorisation.

- Outre son format facilement parsable, JWT permet de garantir l'authenticité du jeton (le jeton n'a pas été généré par un site malveillant)
- Il existe 2 types de jetons JWT :
  - Jeton transparent : Les données concernant l'utilisateur sont visibles
  - Jeton opaque : Les données sont cryptées et nécessitent une clé supplémentaire pour les décrypter

JWT et JOSE

JWT est issu d'une famille de spécifications connue sous le nom de *JOSE*

- ***JSON Web Token (JWT, RFC 7519)*** : Le jeton se compose de 2 documents JSON encodés en base64 et séparés par un point : un en-tête et un ensemble de revendications (*claims*)
- ***JSON Web Signature (JWS, RFC 7515)*** : Ajoute une signature numérique de l'en-tête et des revendications
- ***JSON Web Encryption (JWE, RFC 7516)*** : Chiffre les revendications
- ***JSON Web Algorithms (JWA, RFC 7518)*** : Définit les algorithmes cryptographiques qui doivent être utilisés pour JWS et JWE
- ***JSON Web Key (JWK, RFC 7517)*** : Définit un format pour représenter les clés cryptographiques au format JSON

OpenId Connect

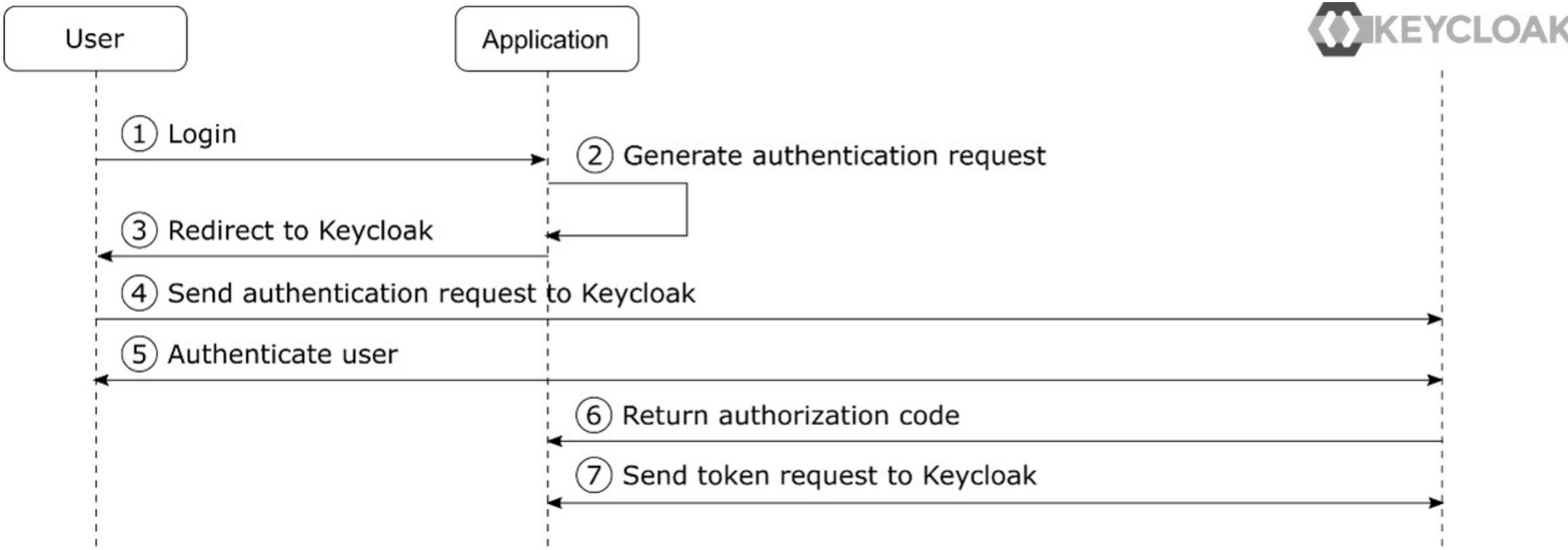
OpenID Connect s'appuie sur OAuth 2.0 pour ajouter une couche d'authentification

Il apporte :

- Social login, (se logger avec son compte Google)
- SSO dans le cadre d'une entreprise
- Les applications clientes n'ont pas accès aux mots de passe des utilisateurs
- Il permet également l'utilisation de mécanismes d'authentification forte comme le OTP (One Time Password) ou WebAuthn

OpenID Connect spécifie clairement le format *JWT* comme format du jeton

Flow OpenID


User
Application
KEYCLOAK
1 Login
2 Generate authentication request
3 Redirect to Keycloak
4 Send authentication request to Keycloak
5 Authenticate user
6 Return authorization code
7 Send token request to Keycloak

Apports de SpringBoot

- **Spring Boot offre 3 starters :**
  - OAuth2 Client : Intégration pour utiliser un login oAuth2 fournit par Google, Github, Facebook, …
  - OAuth2 Resource server : Application pemettant 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

OpenID avec SpringBoot

- **Spring Boot facilite la configuration des providers classiques : Google, Github, Facebook, …**
- **Il permet de s'adapter facilement aux autres solutions : okta, Keycloak**
- **Starter *oauht2-client***

```
@Bean

public SecurityWebFilterChain securityWebFilterChain(ServerHttpSecurity
http) {

    return http.authorizeExchange()

      .anyExchange().authenticated()

      .and().oauth2Login()

      .and().csrf().disable()

      .build();

}
```

Configuration

- **Configuration Google**

```
spring:
  security:
    oauth2:
      client:
        registration:
          google:
            client-id: google-client-id
            client-secret: google-client-secret
```

- **Configuration Keycloak**

```
spring :
  security:
    oauth2:
      client:
        provider:
          keycloak:
            issuer-uri: http://keycloak/realms/<realm-name>/
        registration:
          spring-app:
            provider: keycloak
            client-id: spring-app
            client-secret: 57abb4f6-5130-4c73-9545-6d377dd947cf
            authorization-grant-type: authorization_code
            redirect-uri: "{baseUrl}/login/oauth2/code/keycloak"
            scope :openid
```

Accès à l'utilisateur loggé

```
@GetMapping("/oidc-principal")
public OidcUser getOidcUserPrincipal(
  @AuthenticationPrincipal OidcUser principal) {
    return principal;
}

…

Authentication authentication =
  SecurityContextHolder.getContext().getAuthentication();
if (authentication.getPrincipal() instanceof OidcUser) {
    OidcUser principal = ((OidcUser)
  authentication.getPrincipal());

    // ...
}
```

Serveur de ressources

- **Dépendance : oauth2-resource-server**
- **Configuration : Le serveur de ressources doit vérifier la signature du jeton pour s'assurer que les données n'ont pas été modifiées.**
  - jwk-set-uri contient la clé publique que le serveur peut utiliser pour la vérification
  - issuer-uri pointe vers l'URI de Keycloak. Utilisé pour la découverte de jwk-set-uri

- **Configuration minimale**

```
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: https://idp.example.com/issuer
```

Scope vs Spring Authority

- **Spring ajoute des Authority à l'Authentication en fonction des des scopes présent dans le jeton**
- **Les autorités correspondant aux scopes sont préfixées par "SCOPE_".**
- **Cela peut être adapté via le bean JwtAuthenticationConverter**
  - Positionner les Authorities à partir d'autre Claim
  - Changer le préfixe utilisés

```
@Bean
public JwtAuthenticationConverter jwtAuthenticationConverter() {
    JwtGrantedAuthoritiesConverter grantedAuthoritiesConverter = new
    JwtGrantedAuthoritiesConverter();
    grantedAuthoritiesConverter.setAuthoritiesClaimName("authorities");

    JwtAuthenticationConverter jwtAuthenticationConverter = new JwtAuthenticationConverter();
    jwtAuthenticationConverter.setJwtGrantedAuthoritiesConverter(grantedAuthoritiesConverter);
    return jwtAuthenticationConverter;
}
```

Configuration SecurityFilterChain

```
@Configuration
@EnableWebSecurity
public class MyCustomSecurityConfiguration {
    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http
            .authorizeHttpRequests(authorize -> authorize
                .requestMatchers("/messages/**").hasAuthority("SCOPE_message:read")
                .anyRequest().authenticated()
            )
            .oauth2ResourceServer(oauth2 -> oauth2
                .jwt(jwt -> jwt
                    .jwtAuthenticationConverter(myConverter())
                )
            );
        return http.build();
    }
}
```

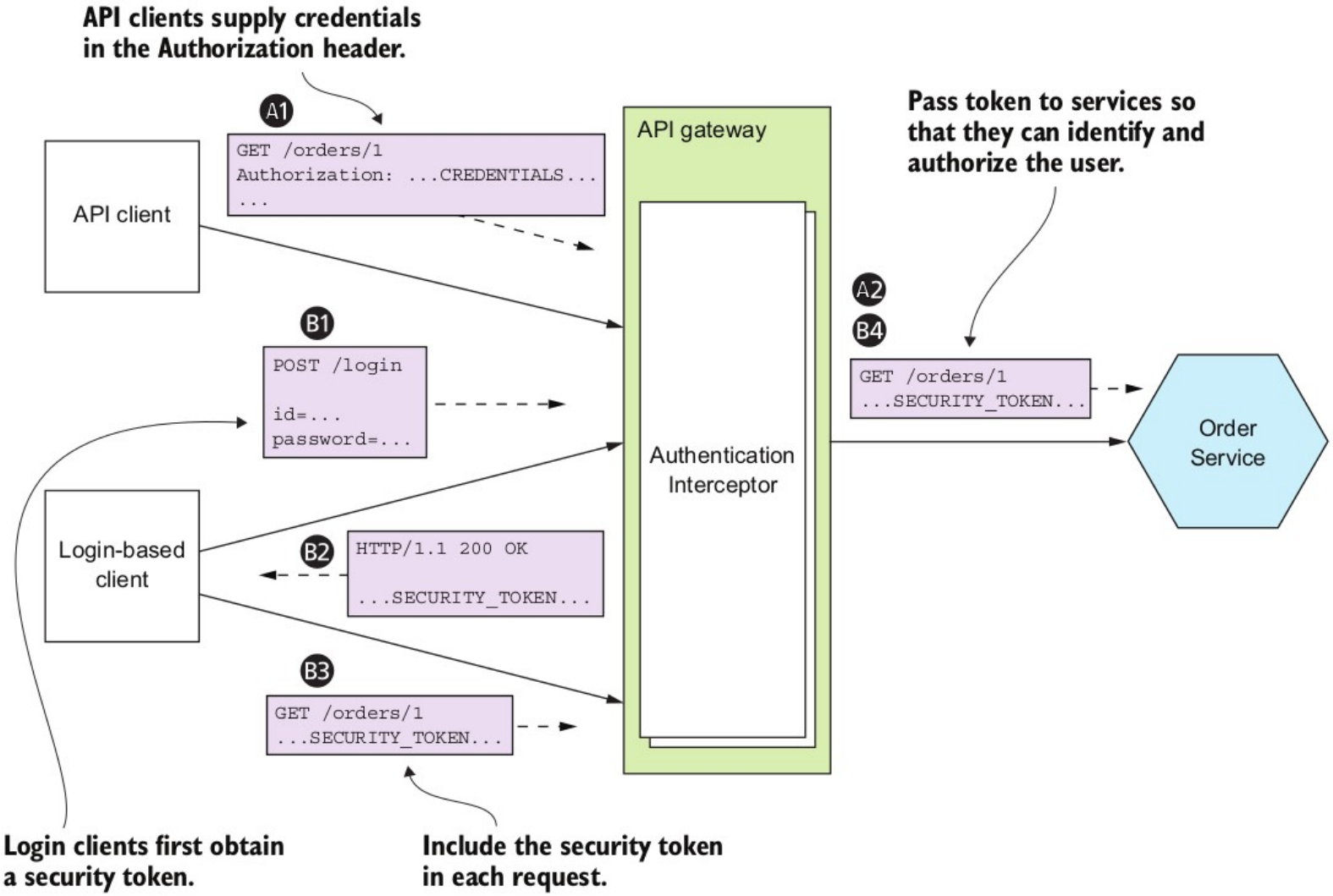
Micro-services

Lorsqu'une application souhaite invoquer une API REST protégée par oAuth2, elle obtient d'abord un jeton d'accès de Keycloak, puis inclut le jeton d'accès dans l'en-tête d'autorisation.

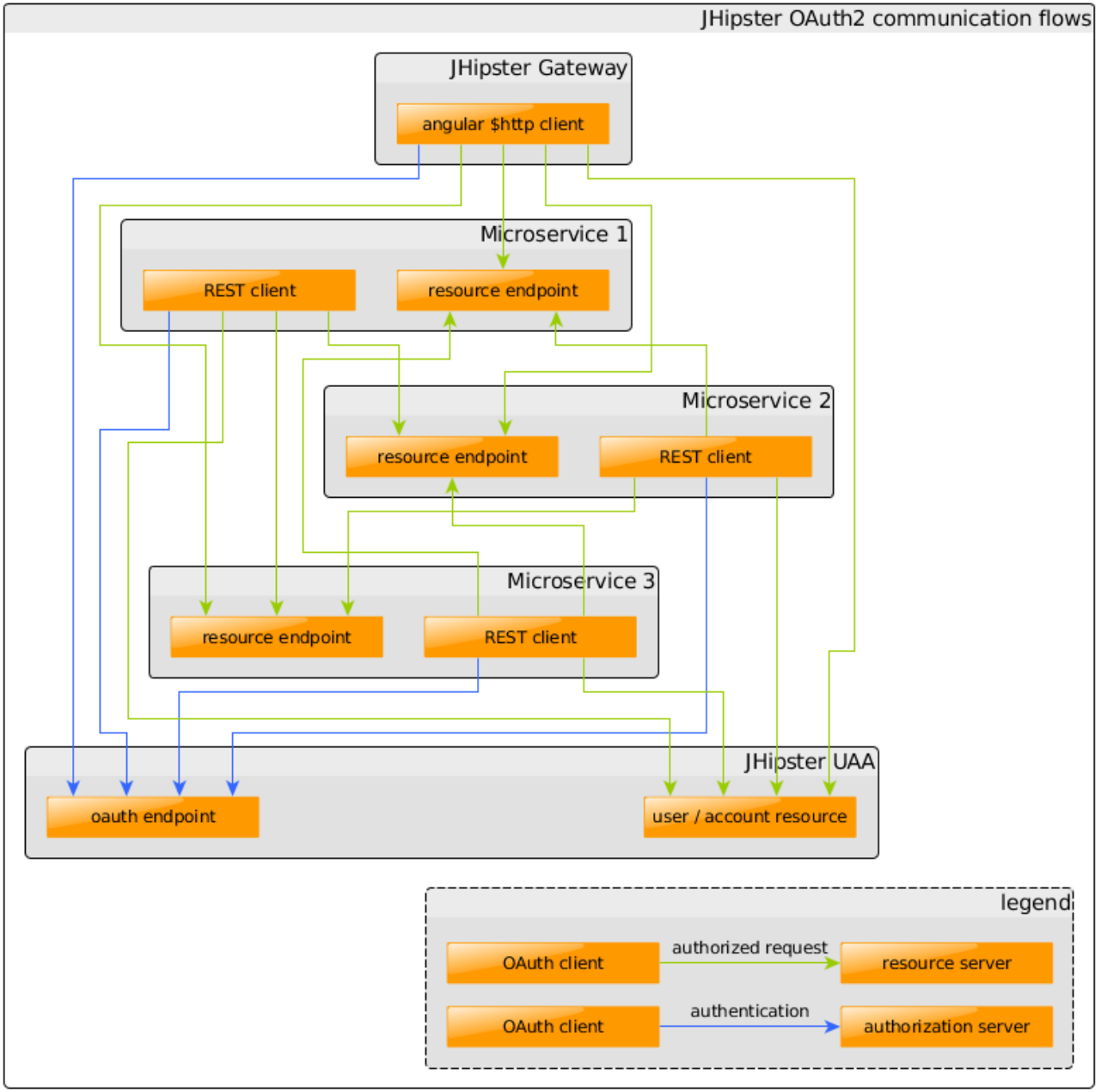
Lors des architecture micro-services, 2 alternatives sont possibles :

- Propagation de jeton, le même jeton est propagé lors des appels inter-services. Tous les micro-services partagent le même contexte de sécurité
- Chaque micro-service utilise son propre jeton (obtenu avec un *Client Credentials grant)*. Chaque appel a alors son propre contexte de sécurité

Access Token Pattern
Relai du jeton / Cas de la Gateway



Jeton par micro-services



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());
    }
}
```

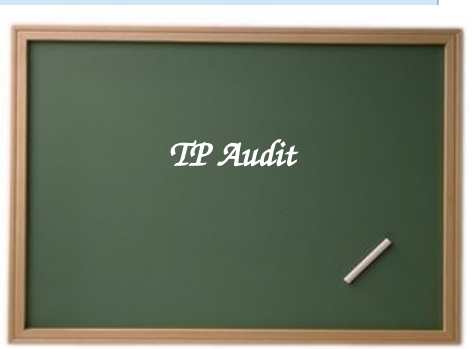
Ajouter des informations d'audit

- **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));
    }
```



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 { }
```

Spring Test MVC

- **Il est possible de tester les droits sur une application Spring MVC**
  - De base Spring-MVC posséde 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 framwork**
  - 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

CRSF 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 securiry

```
<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 dédie : 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**

```
//desactivation des headers par defaut
http.headers().defaultsDisabled();
```

En-têtes de réponse HTTP (cache)

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

```
        //desactivation des headers par defaut
        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);
```

WSS

- **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 authentifications '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/mailing/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 userDetailService**

```
    @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

Authentification 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 sécure
- **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
```

Authentification Digest (2)

```
<bean id="digestFilter" class=

"org.springframework.security.web.authentication.www.DigestAuthentication
Filter">
<property name="userDetailsService" 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>
```

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 Exemple

- **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>
```

Autorisation

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