Développement d'applications modulaires en Java

Chouki Tibermacine

Chouki.Tibermacine@umontpellier.fr

Plan de l'ECUE

1. Système de modules et services Java9+
2. (Rappels sur le) Développement d'applications Web avec Java
3. Modulariser les applications Java avec Spring
4. Bien structurer une application Web avec Spring MVC
5. Auto-configurer une application Web avec Spring Boot
6. Sécuriser une application Web avec Spring Security
7. Gérer des données massives avec Apache Kafka et Spring
8. Tester une application Web Spring
9. Écrire des applications Web (API) réactives avec Spring WebFlux

Plan du cours

1. Intro à la sécurité des app Web avec Spring
2. Sécuriser en utilisant différentes sources de données
3. Personnaliser le processus d'authentification
4. Gérer le processus d'enregistrement d'utilisateurs
5. Sécuriser les méthodes et les éléments du Front
6. Conclusion : synthèse et ajustements supplémentaires

Besoins en sécurité

Authentication

Who am I?

Authorization

What can I do?

Features de *Spring Security*

- *Spring Security* : framework construit au dessus de Spring
- Il s'intègre bien avec un CAS, des tokens OAuth, LDAP, ou des bases de données d'utilisateurs
- Il est basé sur la notion de Filter (sevlets) comme point d'entrée (mécanisme interne au framework)
- Il gère de multiples modèles d'authentification (*in-memory*, *database*, ...)
- Il fournit une liste assez importante de features pour gérer l'authentification & l'autorisation et leur personnalisation (perte de mots de passe, par exemple), ce qui simplifie la sécurisation d'une app Web Java

Dépendances vers *Spring Security*

- Déclarer l'utilisation de Spring Security dans le script de build Gradle :

```
implementation('org.springframework.boot:spring-boot-
    starter-security')
testImplementation('org.springframework.security:spring
    -security-test')
```

- Pour les besoins de l'exemple ci-après, nous allons déclarer l'utilisation d'un compilateur JSP

```
runtimeOnly('org.apache.tomcat.embed:tomcat-embed-
    jasper')
```

Un premier exemple

- Nous allons ajouter une classe de configuration de la sécurité et l'annoter d'une certaine façon pour qu'elle soit prise en compte par le framework (la mettre dans le package racine de votre app)

```
@Configuration
@EnableWebSecurity
public class CovidAlertSecurityConfig extends
    WebSecurityConfigurerAdapter {
  ...
}
```

Définition de la sécurité – *Builder pattern*

- Dans la classe précédente, on redéfinit la méthode `configure` :

```
protected void configure(final HttpSecurity httpSecurity)
    throws Exception {
  httpSecurity
  .authorizeRequests()
  .antMatchers("/login*").permitAll()
  .antMatchers("/static/css/**","/static/js/**",
               "/images/**").permitAll()
  .antMatchers("/index*").permitAll()
  .anyRequest().authenticated()
  .and().formLogin().loginPage("/login")
  .loginProcessingUrl("/doLogin")
  .failureUrl("/login?error=true").permitAll()
  .defaultSuccessUrl("/",true);
}
```

Lire chaque ligne et se documenter sur son rôle

Pas besoin de définir la vue `doLogin`. Les requêtes qui visent cette vue sont traitées par les filtres Spring Security

Un premier exemple -suite-

- Toujours dans la classe précédente, on ajoute les méthodes suivantes :

```
protected void configure(final
  AuthenticationManagerBuilder auth) throws Exception {
  auth.inMemoryAuthentication()
  .withUser("bob").password(passwordEncoder()
  .encode("password_de_bob")).roles("ADMIN");
}

@Bean
public PasswordEncoder passwordEncoder() {
  return new BCryptPasswordEncoder();
}
```

Cela permet de déclarer un premier utilisateur (*bob*) dans l'application, avec un mot de passe encodé avec la fonction de hachage `BCrypt`, et lui donner le rôle de `ADMIN`

Côté contrôleur

Une classe Contrôleur comme solveur des vues

```
@Controller
public class ViewController {
  @GetMapping({"/", "/index"})
  public String home() { return "index"; }

  @GetMapping({"/login"})
  public String login() { return "login"; }

  @GetMapping({"/changeUser"})
  public String changeUser() { return "changeUser"; }

  @GetMapping({"/listUsers"})
  public String listUsers() { return "listUsers"; }
}
```

Dans `application.properties`, ajouter :

```
spring.mvc.view.prefix=/WEB-INF/jsp/
spring.mvc.view.suffix=.jsp
```

Côté Front-end

Dans une page JSP (login.jsp) : (fournie dans le dépôt Git)

```
<div class="container">
 <h1>Login</h1>
 <% if(request.getParameter("error") != null) { %>
  <div class="errblock">Invalid Username and Password</div>
 <% } %>
 <form:form action="/doLogin" method="POST">
  <label for="username">User name: </label>
  <input type="text" id="username" name="username"/>
  <br/>
  <label for="password">Mot de passe</label>
  <input type="password" id="password" name="password"/>
  <br/>
  <input type="submit" value="Login" role="button" class="
    btn btn-lg btn-primary" />
 </form:form>
</div>
```

La placer dans le dossier src/main/webapp/WEB-INF/jsp/

Plan du cours

1. Intro à la sécurité des app Web avec Spring
2. Sécuriser en utilisant différentes sources de données
3. Personnaliser le processus d'authentification
4. Gérer le processus d'enregistrement d'utilisateurs
5. Sécuriser les méthodes et les éléments du Front
6. Conclusion : synthèse et ajustements supplémentaires

Une source de données *in-memory*, LDAP, ...

- *In-memory*, utilisé dans l'exemple précédent :

```
auth.inMemoryAuthentication()
    .withUser("bob").password(passwordEncoder()
    .encode("password_de_bob")).roles("ADMIN");
```

- Il est possible également d'utiliser LDAP. Pour cela, il faudra :
  - ajouter de nouvelles dépendances Spring Security LDAP
  - configurer les paramètres de connexion au service LDAP (dans *application.properties*)

Une source de données *in-memory*, LDAP, ...

- Utiliser LDAP : -suite-
  - personnaliser la configuration en indiquant :

```
auth.ldapAuthentication()
    .userDnPatterns("uid={0},ou=people")
    .groupSearchBase("ou=groups")
    .contextSource()
    .url("ldap://...:8389/dc=iwa,dc=fr")
    .and()
    .passwordCompare()
    .passwordEncoder(passwordEncoder())
    .passwordAttribute("userPassword")
```

- Nous n'allons pas tester cette source de données d'authentification

Une source de données externe

- Une base de données qui contient la table des utilisateurs
- Dans la classe de configuration de la sécurité :

```
@Configuration
@EnableWebSecurity
public class CovidAlertSecurityConfig extends
    WebSecurityConfigurerAdapter {
 @Autowired
 DataSource dataSource;
 // ... configure() method with security definition
 @Override
 protected void configure(final
  AuthenticationManagerBuilder auth) throws Exception {
  auth.jdbcAuthentication().passwordEncoder(passwordEncoder
    ()).dataSource(dataSource)
  .withUser("admin").password(passwordEncoder().encode("
    adminadmin")).disabled(false).roles("USER","ADMIN");
 } }
```

Config de Postgres identique au cours précédent

Côté Postgres

- Mettre à jour la table users (la table doit s'appeler ainsi) en ajoutant :

```
username varchar(50) NOT NULL PRIMARY KEY,
password varchar(100) NOT NULL,
enabled boolean NOT NULL DEFAULT false
```

  et en supprimant les contraintes NOT NULL pour les autres champs. Par exemple, dans le Shell `psql` :

```
alter table users alter column email drop not null;
```

- Ajouter la table suivante :

```
CREATE TABLE authorities(
  authority_id serial primary key,
  username varchar(50) NOT NULL
           REFERENCES users (username),
  authority varchar(50) NOT NULL DEFAULT 'ROLE_USER'
);
```

Tester cela

- Modifier la base de données Postgres comme expliqué ci-haut
- Insérer quelques utilisateurs dans les deux tables :
  - Pour obtenir l'encodage en `BCrypt` d'un mot de passe, utiliser la classe de test de votre projet Gradle : déclarer dans cette classe un attribut auto-injecté (`@Autowired`) de type `PasswordEncoder` et, dans la méthode de test (`contextLoads()`), faire un `print` du résultat retourné par la méthode encode invoquée sur ce bean (auto-injecté)
  - Faire un *Run* de cette classe (clique bouton droit, puis Run)
  - Autre option : outils en ligne, comme `https://bcrypt-generator.com/`
- Changer la configuration `in-memory` par une config. JDBC

Tester cela -suite

- Modifier la classe Entity `User` pour refléter les changements faits dans la base de données (champs username, password et enabled)
- Ajouter une nouvelle classe Entity Authority qui correspond à la table `authorities`
- Démarrer Postgres si ce n'est pas déjà fait, puis démarrer l'app
- Pour suivre l'évolution du système d'authentification, avec ses différents filtres mis en place, les exceptions qui sont levés et capturés pour déclencher l'authentification, ajouter la ligne suivante dans `application.properties` :

```
logging.level.org.springframework.security=DEBUG
```

Les logs sont visibles dans la console du serveur (sur votre IDE)

Plan du cours

1. Intro à la sécurité des app Web avec Spring
2. Sécuriser en utilisant différentes sources de données
3. Personnaliser le processus d'authentification
4. Gérer le processus d'enregistrement d'utilisateurs
5. Sécuriser les méthodes et les éléments du Front
6. Conclusion : synthèse et ajustements supplémentaires

Remember Me

- On aimerait que l'application se rappelle d'un utilisateur déjà authentifié
- Nous allons d'abord ajouter une case à cocher dans le formulaire d'authentification que l'utilisateur peut cocher :

```
<label>Remember me:
  <input type="checkbox" name="remember-me">
</label>
```

- Ensuite, nous personnalisons la méthode

`configure(final HttpSecurity httpSecurity):`

```
...
.defaultSuccessUrl("/",true)
.and()
.rememberMe()
.key("cleSuperSecrete")
.tokenRepository(tokenRepository())
```

Remember Me -suite-

- Nous allons ajouter la méthode `tokenRepository()` invoquée avant :

```
@Bean
public PersistentTokenRepository tokenRepository() {
  JdbcTokenRepositoryImpl token = new
    JdbcTokenRepositoryImpl();
  token.setDataSource(dataSource);
  return token;
}
```

Remember Me - côté Postgres

- Ajouter la table suivante dans laquelle les tokens sont générés :

```
CREATE TABLE persistent_logins(
 username varchar(50) NOT NULL
          REFERENCES users (username),
 series varchar(64) PRIMARY KEY,
 token varchar(64) NOT NULL,
 last_used timestamp NOT NULL
)
```

- Tester cette fonctionnalité
- Un nouvel enregistrement est ajouté dans cette table si on coche la case “*Remember Me*”
- Même au redémarrage du serveur, l’utilisateur reste authentifié

Se déconnecter (*logout*)

- Personnaliser la méthode configure, en ajoutant :

```
...
.and()
.logout()
.logoutSuccessUrl("/login?logout=true")
.logoutRequestMatcher(
   new AntPathRequestMatcher("/doLogout","GET")
)
.invalidateHttpSession(true)
.deleteCookies("JSESSIONID")
.permitAll();
```

- Ici, les requêtes vers la vue `doLogout` sont gérées par les filtres Spring Security (pas besoin donc de la définir)

Côté Front-end

- Dans la vue de login :

```
<% if(request.getParameter("logout") != null) { %>
  <div class="alert alert-success" role="alert">
    Logout was successful
  </div>
<% } %>
```

- N'importe où dans les vues de l'application, ajouter la possibilité de faire un *logout*

```
<a href="/doLogout">Logout</a>
```

Plan du cours

1. Intro à la sécurité des app Web avec Spring
2. Sécuriser en utilisant différentes sources de données
3. Personnaliser le processus d'authentification
4. Gérer le processus d'enregistrement d'utilisateurs
5. Sécuriser les méthodes et les éléments du Front
6. Conclusion : synthèse et ajustements supplémentaires

Le processus d'enregistrement

- Le processus d'enregistrement d'un nouvel utilisateur est le suivant :
  1. L'utilisateur remplit le formulaire d'inscription et valide
  2. L'application envoie un email de confirmation, avec un lien
  3. L'utilisateur confirme la création de son compte, en cliquant sur le lien
- Ajouter au formulaire d'enregistrement des utilisateurs de votre application, un champ de mot de passe et un deuxième champ de confirmation du mot de passe (les attributs "`name`" des champs les plus importants pour la suite du cours doivent avoir comme valeurs : username et password)
- Pour l'envoi d'emails, utiliser un compte personnel (pas Gmail, car depuis l'été 2022, l'option d'utilisation de ce compte par les applications ("Less Secure Apps") dans les paramètres de config Gmail a été désactivée)

Côté Backend

Prévoir les méthodes nécessaires dans le contrôleur (solveur de vues) :

1. `register` pour l'accès à la vue qui fournit le formulaire : Récupérer le fichier register.jsp sur le dépôt Git
2. `doRegister` pour le traitement des données du formulaire (voir ci-après)

Côté Backend -suite-

- La seconde méthode du contrôleur :

```
@PostMapping("doRegister")
public String register(@Valid @ModelAttribute("user")
        User user, BindingResult result) {
 // check for errors ...
 // verify that username does not exist:
 if(userRepository.existsUserByUsername(user.getUsername()))
     {
  return "register.jsp?user=true";
 }
 else { // encrypt password:
  user.setPassword(passwordEncoder.encode(user.getPassword()
    ));
  // save user object:
  userRepository.saveAndFlush(user);
  // create/save an Authority obj ...
  return "login";
 }
}
```

Côté Backend -suite-

- Dans la méthode précédente, nous avons utilisé le bean `userRepository`. Celui-ci doit être déclaré comme attribut auto-injecté dans la classe :

```
@Autowired
private UserRepository userRepository;
```

  De même pour `authorityRepository` (dont l'interface doit être créée en même temps que la classe `Entity Authority`)

- Nous avons également utilisé le bean `passwordEncoder`. Celui-ci doit être déclaré de la même façon :

```
@Autowired
private PasswordEncoder passwordEncoder;
```

Côté Backend -suite-

- Nous avons également utilisé une méthode `existsUserByUsername`, qui n'existe pas encore dans le bean `userRepository`. Il faudra l'ajouter à l'interface `UserRepository` définie dans le cours précédent (interface *JPA Repository*) :

```
boolean existsUserByUsername(String username);
```

  Spring fournira l'implem de cette méthode en cherchant parmi les entities de type `User` un objet qui a le `username` passé en argument (respecter le nommage comme ci-dessus)

- La méthode ajoutée ci-dessus complète l'interface, qui contient par défaut une méthode `existsById()` qui prend en argument un Id de type long, ce qui n'est pas approprié pour l'exemple

Côté Front-end

- Nous allons prévoir dans la vue, qui présente la formulaire, un bloc pour l'affichage du message d'erreur si jamais un utilisateur existe avec le même nom :

```
<% if(request.getParameter("user") != null) { %>
<div class="errorblock">A user with the same username
    already exists</div>
<% } %>
```

Envoi de mail

- L'envoi de mail doit se faire, de préférence, en mode asynchrone
- Nous allons voir comment le faire avec des événements
- D'abord ajouter la dépendance suivante :

```
implementation('org.springframework.boot:spring-boot-
    starter-mail')
```

- Ensuite indiquer la config du serveur mail (SMTP) dans le fichier `application.properties`

```
spring.mail.host=smtp.<votre-fournisseur>.com
spring.mail.port=587
spring.mail.username=<username>@<votre-fournisseur>.com
spring.mail.password=<votre-mot-de-passe>
spring.mail.properties.mail.smtp.auth=true
spring.mail.properties.mail.smtp.starttls.enable=true
```

Définition de l'événement

- Créer une classe dans un sous-package `util`, qui s'appelle `OnCreateUserEvent`
- Cette classe doit étendre `org.springframework.context.ApplicationEvent`:

```
public class OnCreateUserEvent extends ApplicationEvent {
 private String appUrl; private User user;
 public OnCreateUserEvent(String appUrl, User user) {
  super(user);
  this.appUrl = appUrl;
  this.user = user;
 }
 public String getAppUrl() {
  return appUrl;
 }
 public User getUser() {
  return user;
 }
}
```

Mise à jour du contrôleur

Pour signaler l'événement, nous allons modifier le contrôleur qui gère le POST :

- Ajouter d'abord un attribut auto-injecté :

```
@Autowired
private ApplicationEventPublisher eventPublisher;
```

- Ensuite, nous allons déclencher l'événement :

```
@PostMapping("doRegister")
public String register(@Valid @ModelAttribute("user")
        User user, BindingResult result) {
  ...
  // Dans le else
  eventPublisher.publishEvent(new OnCreateUserEvent("/"
    ,user));
}
```

Définition de l'écouteur d'événement

- Créer une classe dans le package `util`

```
@Component
public class UserListener implements
    ApplicationListener<OnCreateUserEvent> {
 @Override
 public void onApplicationEvent(OnCreateUserEvent event
    ) {
  this.confirmCreateUser(event);
 }
}
```

Définition de l'écouteur d'événement -suite-

- Ajouter dans la classe précédente :

```
// Ceci peut etre fait dans application.properties
private String serverUrl = "http://localhost:8080";
@Autowired
private JavaMailSender mailSender;
private void confirmCreateUser(OnCreateUserEvent event) {
 // get the user
 User user = event.getUser();
 // create verification token
 String token = UUID.randomUUID().toString();
 VerificationToken verifToken = new VerificationToken();
 // VerificationToken est une classe qu'on va definir apres
 verifToken.setToken(token);
 verifToken.setUsername(user.getUsername());
 verifToken.setExpiryDate(verifToken.calculateExpiryDate(
     VerificationToken.EXPIRATION));
 verificationTokenRepository.saveAndFlush(verifToken);
 // ...
}
```

Définition de l'écouteur d'événement -suite-

- Toujours dans la classe précédente :

```
...
private void confirmCreateUser(OnCreateUserEvent event) {
 // get email properties
 String recipientAddesss = user.getEmail();
 String subject = "User Account Confirmation";
 String confirmationUrl = event.getAppUrl()+"userConfirm?
    token="+token;
 String message = "Please confirm:";
 // send email
 SimpleMailMessage email = new SimpleMailMessage();
 email.setTo(recipientAddesss);
 email.setSubject(subject);
 email.setText(message+"\r\n"+serverUrl+confirmationUrl);
 mailSender.send(email);
}
```

Définition de l'écouteur d'événement -suite-

- Créer une classe dans le package `models` pour gérer les tokens

```java
@Entity(name="verif_tokens")
public class VerificationToken {
  public static final int EXPIRATION = 60 * 24; // nb min
  @Id
  private String token;
  private String username;
  private Date expiryDate;
  public Date calculateExpiryDate(int expiryTimeInMinutes) {
    Calendar cal = Calendar.getInstance();
    cal.add(Calendar.MINUTE,expiryTimeInMinutes);
    return cal.getTime();
  }
  // + getters and setters
}
```

Ajouter une table avec les 3 champs et l'interface JPA repository VerificationTokenRepository

Confirmation de la création du compte utilisateur

- Pour confirmer la création du compte utilisateur, on a besoin juste d'afficher :

```
<div class="alert alert-success" role="alert">
  Your user account has been created
</div>
```

- Mais il faudrait également vérifier un certain nombre de choses avant
- D'abord, nous allons mettre à jour le contrôleur pour prendre en charge la requête correspondant à l'URL envoyé à l'utilisateur
- Le code qui va être défini dans le contrôleur peut être déplacé dans une méthode définie dans une classe `@Service` pour que le contrôleur soit déchargé de cette "logique métier"

Prendre en charge la requête

- Le contrôleur de la requête :

```
@GetMapping({"/userConfirm"})
public String confirmUser(@RequestParam("token") String
    token) {
 VerificationToken verifToken = verificationTokenRepository.
    getOne(token);
 if(verifToken != null) {
  Date date = Calendar.getInstance().getTime();
  if(date.before(verifToken.getExpiryDate())) {
   verificationTokenRepository.delete(verifToken);
   User user = userRepository.findByUsername(verifToken.
    getUsername());
   user.setEnabled(true);
   userRepository.saveAndFlush(user);
   return "login.jsp?confirm=true";
  }
```

Prendre en charge la requête -suite-

- Suite de la méthode du contrôleur :

```
    else {
      verificationTokenRepository.delete(verifToken);
      return "register.jsp?expired=true";
    }
  }
  else { return "register.jsp?confirm=false"; }
}
```

Plus haut, nous avons utilisé une nouvelle méthode `findByUsername()`. Il faut ajouter la signature suivante à l'interface UserRepository (Spring fournit l'implem qu'il faut) :

```
User findByUsername(String username);
```

Prendre en charge la requête -suite-

- Prévoir les messages d'erreur dans `register.jsp`. Exemple :

```
<% if(request.getParameter("expired") != null) { %>
 <div class="errorblock">The registration has expired.
    Please re-sign-up</div>
<% } %>
```

- Prévoir le message de succès dans la vue `login.jsp` :

```
<% if(request.getParameter("confirm") != null) { %>
 <div class="alert alert-success" role="alert">User
    creation is confirmed. Please log in</div>
<% } %>
```

- Prévoir d'ajouter la nouvelle route (`/userConfirm`) dans la méthode `configure` de définition de la sécurité pour autoriser l'accès sans authentification à cette vue :

```
.antMatchers("/userConfirm").permitAll()
```

Mise en place de la fonctionnalité “Oubli du mot de passe”

- Ajouter un lien dans la page de login : *Forgot Password*
- Ajouter un formulaire pour saisir le `username` et l’`email`
- Ajouter le contrôleur nécessaire qui permet d’envoyer un email (*Reset Password*) avec un lien (incluant un token qui expire, comme précédemment) pour la mise à jour du mot de passe
- Mettre en place la mise à jour du mot de passe (vue + contrôleur)

Plan du cours

1. Intro à la sécurité des app Web avec Spring
2. Sécuriser en utilisant différentes sources de données
3. Personnaliser le processus d'authentification
4. Gérer le processus d'enregistrement d'utilisateurs
5. Sécuriser les méthodes et les éléments du Front
6. Conclusion : synthèse et ajustements supplémentaires

Sécuriser les méthodes

- D'abord, ajouter une annotation sur la classe de config de la sécurité :

```
@EnableGlobalMethodSecurity(
   prePostEnabled=true,
   securedEnabled=true,
   jsr250Enabled=true
)
```

  Ceci permet d'activer les annotations prePost pour gérer plus finement l'avant et l'après exécution d'une méthode, les annotations Spring (deuxième true) et les annotations standards Java

- Ces annotations sont désactivées, par défaut

Sécuriser les méthodes -suite-

- Ensuite, on peut annoter une méthode en particulier, en indiquant le rôle de l'utilisateur qui a le droit d'exécuter la méthode (après s'être authentifié) :

```
@PostMapping("/listUsers")
@Secured("ROLE_ADMIN")
public String listUsers() {
   ...
}
```

  `@Secured` est une annotation Spring activée par `securedEnabled`

- Si on teste cela, lorsqu'un utilisateur, avec le rôle `USER` seulement, clique sur le lien pour lister les utilisateurs, il reçoit une réponse HTTP 403 (*Forbidden*)

Sécuriser les éléments du Front

- Pour cacher par exemple le lien "List Users" à l'utilisateur s'il n'est pas ADMIN
- Cela se fait simplement avec des balises spéciales de Spring Security qui vont entourer les éléments du Front à sécuriser
- D'abord, ajouter la dépendance au script de build :

```
implementation('org.springframework.security:spring-
    security-taglibs')
```

- Ensuite, dans une vue (`index.jsp`, par exemple) nous allons importer la taglib :

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

Sécuriser les éléments du Front

- Afficher un élément si un utilisateur est authentifié ou pas :

```
<sec:authorize access="! isAuthenticated()">
 <a href="/login">Log in</a><br/>
 <a href="/register">Create a User Account</a><br/>
</sec:authorize>
<sec:authorize access="isAuthenticated()">
 <a href="/changeUser">Change User Account</a><br/>
 <a href="/listUsers">List User Accounts</a>
</sec:authorize>
```

- Afficher le nom de l'utilisateur authentifié :

```
<sec:authentication property="principal.username"/>
```

- Dans les méthodes des contrôleurs, nous avons la possibilité d'ajouter un dernier paramètre de type `Authentication`. Cet objet (injecté) fournit une méthode `getPrincipal()` qui retourne un objet comportant le username, password, rôles, ...

Plan du cours

1. Intro à la sécurité des app Web avec Spring
2. Sécuriser en utilisant différentes sources de données
3. Personnaliser le processus d'authentification
4. Gérer le processus d'enregistrement d'utilisateurs
5. Sécuriser les méthodes et les éléments du Front
6. Conclusion : synthèse et ajustements supplémentaires

Wrap-up

- Spring Security protège votre application des attaques classiques comme CSRF et XSS, grâce à : i) un HTTP Firewall, ii) des filtres de contrôle implicites activés par défaut avant le servlet dispatcher, et iii) des tokens supplémentaires
- En inspectant les headers des réponses HTTP de votre app :

```
Cache-Control: no-cache; ...
X-Content-Type-Options: nosniff
X-XSS-Protection: 1; mode=block
```

- Ces filtres sont désactivables dans `configure(HttpSecurity http):.headers().defaultsDisabled()` ou `.csrf().disable()`, mais ce n'est pas ce qui est recommandé

Wrap-up -suite- : derniers ajustements

- Pour activer HTTPS dans votre app Spring Boot :

  `https://www.tutorialspoint.com/spring_boot/spring_boot_enabling_https.htm`

- Le tuto ci-dessus explique comment générer et utiliser un certificat auto-signé. Le navigateur vous demandera d'accepter l'exception de sécurité
- Pour générer un certificat fourni par une autorité de certification reconnue par les navigateurs, compléter votre app :

  `https://dzone.com/articles/spring-boot-secured-by-lets-encrypt`

- Pour rediriger toutes les requêtes HTTP, qui arrivent à votre app, vers HTTPS :

  `https://jonaspfeifer.de/redirect-http-https-spring-boot/`

“Peut mieux faire” en termes de sécurité

- Configurer HSTS (*HTTP Strict Transport Security*) pour éviter des attaques de type *Man-in-the-middle* :
  `https://docs.spring.io/spring-security/site/docs/5.4.1/reference/html5/#headers-hsts`
  Ce header est activé par défaut dans Spring uniquement en réponse à des requêtes HTTPS (le faire donc explicitement pour les premières requêtes HTTP qui arrivent au serveur, avant la redirection vers HTTPS) :
  `https://docs.spring.io/spring-security/site/docs/5.4.1/reference/html5/#servlet-headers-hsts`
- Mettre en place les handlers des exceptions liées à l’authentification et à l’autorisation, pour ne pas montrer aux utilisateurs de l’app qu’il y a Spring et une base de données dans le Back-end (ne pas laisser les handlers par défaut, qui sont parfois trop verbeux – utiles en dev mais pas en prod)

“Peut mieux faire” en termes de sécurité (*out of scope* de ce cours)

- Crypter les mots de passe en utilisant une bibliothèque dédiée, comme jasypt :

  `https://www.baeldung.com/spring-boot-jasypt`
- Fréquemment modifier les mots de passe et auditer : “par qui”, “quand” et “pourquoi” un mot de passe a été utilisé ?
  - avec Spring Vault :

    `https://spring.io/projects/spring-vault`

Autre solution pour gérer la sécurité dans Spring

- Utiliser un serveur de gestion d'accès, Keycloak, qui s'intègre facilement avec Spring Boot
- Lire ce tuto : `https://www.baeldung.com/spring-boot-keycloak`
- Keycloak avec Docker : `https://hub.docker.com/r/jboss/keycloak/`

Références biblio

- Site Web de Spring Security : `https://spring.io/projects/spring-security`
- Doc actuelle, qui explique le fonctionnement de Spring Security : `https://docs.spring.io/spring-security/site/docs/5.4.1/reference/html5/`
- Tutoriels sur Pluralsight et Baeldung