**Mesurez l’importance de la sécurité pour les applications web**

* Les attaques malveillantes visant les applications web sont en augmentation.
* Deux des attaques les plus communes sont le piratage informatique des données et les injections SQL.
* Pour les éviter, vous devez sécuriser votre programmation.
* OWASP définit un Top 10 des attaques visant les applications web.

### Découvrez comment fonctionne Spring Security

Lorsque vous ajoutez Spring Security, son premier objectif est de protéger les **requêtes HTTP** traitées sur votre application.

Ces requêtes HTTP adressées à votre application web traversent différents niveaux de protection après l'installation de Spring Security :

1. Un pare-feu HTTP.
2. Un proxy.
3. Des filtres.

#### Layer 1: HTTP Firewall

The **HTTP firewall** is at the front line. A firewall is a wall that allows communication to go out but is very selective about what goes in.

HTTP Firewall Protection

It's like the bouncer at a nightclub, selecting who gets into the club based on specific criteria. In an application, a firewall blocks suspicious things that are coming in - usually, bad requests. It throws away any HTTP request that looks suspicious right off the bat.

#### **Layer 2: Proxy Design Pattern**

Second, comes the proxy design pattern. That's just a nerdy way of saying there's an outside delegating authority that will manage its access to a protected resource, which is your web application.

The Proxy checks and directs traffic

It essentially **classifies the HTTP traffic** and sends them to the appropriate servlet filters in the filter chain.

OK, awesome...but what on earth are servlet filters? Or a filter chain? 😦

Spring Security contains an **engine of servlet filters.** In Java, a **servlet** is a program that runs on your web application. The **filter**part is where Spring Security’s primary function lies.

Spring Security calls each little security configuration (or ruleset) a filter. So, *servlet* filters in Spring Security are just a bunch of small programs that perform a security function on your web application.

#### 3e niveau : les filtres

**Les filtres** s’assurent que toutes les **requêtes HTTP** qui pénètrent dans votre application web sont sécurisées. Chaque filtre fournit une configuration de sécurité que vous pouvez intégrer à votre application web. Cette collection de filtres servlet implantés est appelée **chaîne de filtres de Spring Security**. 😄

Selon le niveau de complexité désiré par le développeur, les filtres peuvent fournir des **configurations de sécurité simples ou avancées** contre les attaques. De plus, le framework Spring permet de personnaliser la configuration des servlets nécessaires à l’application web.

Même si vous ne programmez aucun filtre, votre application web disposera quand même de fonctionnalités autoconfigurées et performantes de Spring Security, simplement grâce à l'ajout de ce dernier !

Disons, par exemple, que vous possédez un filtre permettant de sécuriser vos informations de connexion. Lorsque HTTP formule sa requête depuis le front-end et passe le premier pare-feu, DelegatingFilterProxy s’occupe de classer le trafic HTTP, et de l’envoyer vers les filtres de la chaîne de Spring Security prenant en charge les informations de connexion. Tout est donc bien contrôlé au bon endroit.

En résumé, Spring Security sécurise les requêtes HTTP de votre application web en les faisant passer par trois niveaux :

* Premièrement, le **pare-feu HTTP** bloque les requêtes suspectes.
* Deuxièmement, le **proxy** (DelegatingFilterProxy) prend en charge le reste des requêtes HTTP, et les envoie vers la chaîne de filtres de Spring Security.
* Enfin, les **filtres de la chaîne** s’assurent que ces requêtes HTTP soient conformes à leurs critères de sécurité.

### Créez une app Spring Boot avec Spring Security

ajouter vos **dépendances**, qui représentent une partie essentielle du projet. Ajoutez les dépendances suivantes en les tapant dans la barre de recherche et en cochant la case à côté de l’intitulé :

* Spring Web ;
* Spring Security ;
* OAuth 2 Client.

#### Gérez votre hiérarchie de dépendances sur Spring

Grâce à **Spring Boot**, le **Spring Initializr** a créé une application très astucieuse, dotée des fichiers nécessaires. Il ne vous reste plus qu’à ajouter vos fichiers.

Vous venez de créer une app Spring en utilisant **SpringBoot**, et en y ajoutant trois dépendances sur Spring Initializer : Spring Security, Spring Web et OAuth Client.

Mais pourquoi on a ajouté ces dépendances-là, en fait ?

Nous avons ajouté :

* **Spring Web** pour vous donner une série de fonctionnalités liées au Web, étant donné qu’il s’agit d’une application web.
* **Spring Security** pour vous fournir un niveau de sécurité qui se déploie avant même d'initialiser l’application.
* **OAuth 2.0 Client**, car vous utiliserez son protocole de sécurité spécial sur votre formulaire de protection. Nous n’avons pas encore abordé ce point, mais nous y reviendrons en partie 2 de ce cours.

Vous constatez sûrement un élément intéressant : **OAuth 2.0 Client**est la première dépendance sur la liste, suivie de Spring Security et Spring Web.

* Pourquoi ça ?
* C’est le principe de la **hiérarchie des dépendances**. L’application Spring Boot requiert que certaines dépendances soient injectées et initialisées avant les autres. Dans ce cas, OAuth 2.0 représente la base des règles de sécurité qui seront ajoutées automatiquement, suivies par l’ensemble des règles de Spring Security.
* Si ces dépendances ne sont pas initialisées dans le bon ordre, elles ne fonctionneront pas correctement. Si vous les ajoutez manuellement, assurez-vous qu’elles soient placées dans le bon ordre de haut en bas. Appliquez ce même principe à vos filtres de Spring Security.

## Sécurisez l’accès à une application en utilisant l’authentification et l'autorisation

En sécurité, nous parlons de **contrôle d’accès**. Il s’agit de contrôler l’accès à une application web nécessitant un mécanisme de connexion. Le **contrôle d’accès** se compose de deux étapes. Vous avez deviné ?

Authentification et autorisation ?

C’est ça !

Nous allons nous servir de **Spring Security** pour intégrer un contrôle d’accès à votre app Spring Boot, mais avant ça, plongeons-nous dans les détails de l’authentification et de l’autorisation.

Il existe différentes manières de s'authentifier :

* l’authentification à facteur unique, qui permet de se connecter grâce à un mot de passe ;
* et l’authentification à facteurs multiples, qui requiert la livraison de plusieurs éléments pour prouver votre identité.

Eh bien, sachez que votre navigateur peut également gérer l’authentification de différentes manières :

* l'**authentification par session** ;
* l'**authentification par jeton** (ou ***token***, en anglais).

##### Authentification par session

Dans l’**authentification par session**, les utilisateurs se connectent d’abord grâce à leurs identifiants. Ils sont alors authentifiés et commencent une **session**. **Une session représente la période entre la connexion et la déconnexion de l’utilisateur**. Le serveur sauvegarde les informations de session de l’utilisateur, et transmet une copie à un petit fichier (un cookie), également sauvegardé dans le navigateur de l’utilisateur. Ces informations contiennent généralement les identifiants de l’utilisateur, la durée de la session et un **numéro de session**. Chaque fois que l’utilisateur envoie une requête via l’application web, le serveur consulte le cookie pour s’assurer que les identifiants de la session correspondent à ceux du serveur, et sont encore valides.

En raison de la copie des identifiants de l’utilisateur sur le serveur, et du cookie, l’authentification par session est qualifiée comme étant**à états**(en anglais**stateful)**.

##### Authentification par token

Dans le cadre de**l'authentification par token**, l’utilisateur s'authentifie sur le serveur, et ce dernier transmet les informations de la session vers un petit fichier (un token) pour effectuer la sauvegarde uniquement sur le navigateur ou l’ordinateur de l’utilisateur. Un token a la même fonction qu’un cookie, mais ces deux processus présentent tout de même des différences significatives. Une authentification par session avec cookie sauvegarde les identifiants sur le serveur et sur le navigateur de l’utilisateur. Tandis que lors d’une authentification par token, un **token web JSON (JWT)** dispose déjà des informations nécessaires pour valider l’utilisateur ; ainsi, il n’est pas nécessaire de sauvegarder les informations de la session sur le serveur. En cela, cette authentification est qualifiée comme étant **sans état**(en anglais **Stateless**).

C’est quoi exactement, ce**token web JSON** ?

**JSON**signifie **JavaScript Object Notation**. Le token web JSON est un **objet JavaScript** qui encode et transmet vos informations d’authentification. Il valide l’authentification de l’utilisateur, car il détient des informations codées que seul le serveur d'autorisation comprend. Les procédures sont alors plus performantes et sécurisées qu’avec le cookie ; c’est pourquoi la plupart des applications web Java utilisent **JWT**.

* **L’autorisation** permet de s’assurer que l’utilisateur authentifié se rend uniquement sur les pages qu’il est autorisé à consulter.

## Configurez un contrôle d’accès basé sur les rôles

Où peut-on définir les rôles dans notre application web ?

Dans la chaîne de filtres Spring Security. Allons-y !

### Installez votre chaîne de filtres Spring Security

Spring Security propose une méthode intitulée  loginForm()  pour créer une page de connexion par défaut, vous évitant ainsi de la créer vous-même. Vous aurez un peu de code à écrire lorsque vous mettrez en place votre chaîne de filtres customisée, mais rien d’insurmontable. Ce sera très facile d’ajouter et de supprimer différentes méthodes.

En premier lieu, vous devez créer un fichier pour votre toute nouvelle configuration Spring Security.

Une application créée sous **Eclipse** répond à une **hiérarchie**. Afin de vous assurer que votre méthode  main  soit la **première** à fonctionner, créez des packages “enfants” sous votre package principal. Ainsi, si votre package principal est  com.openclassrooms, vos autres classes seront placées dans les packages se basant sur ce dernier.

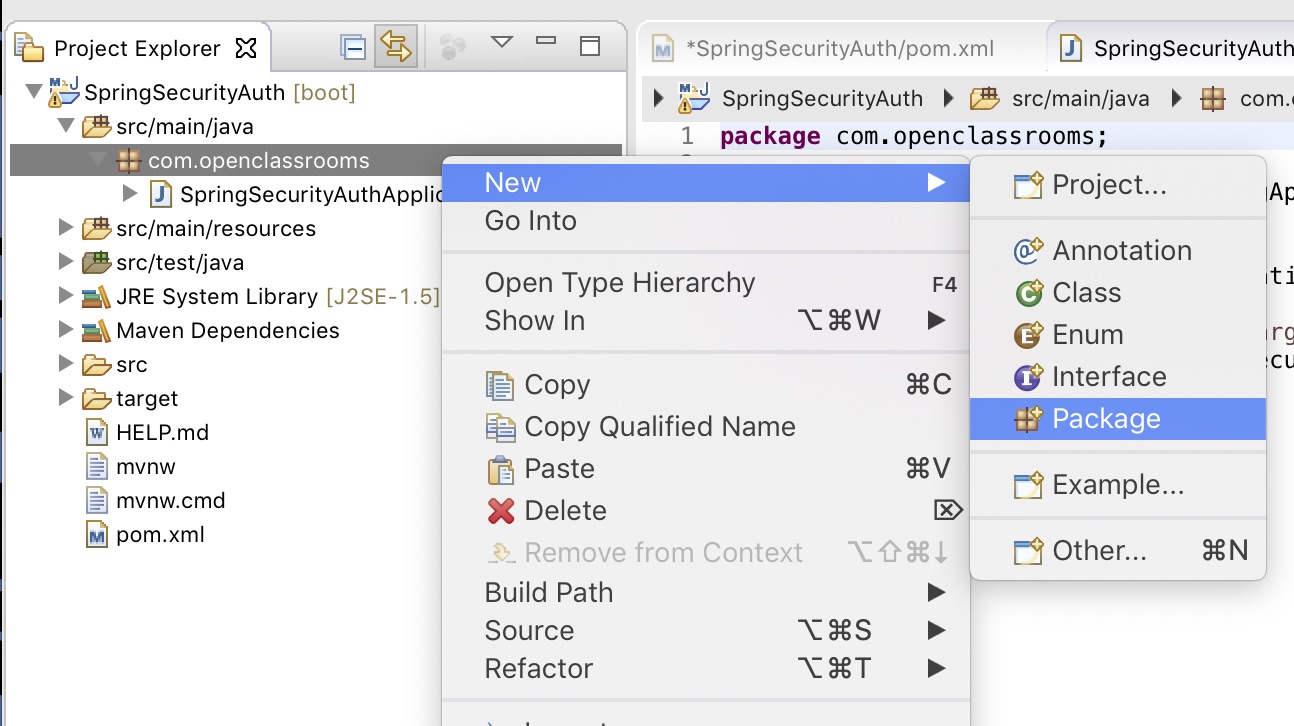
Spring Security propose une méthode intitulée  loginForm()  pour créer une page de connexion par défaut, vous évitant ainsi de la créer vous-même. Vous aurez un peu de code à écrire lorsque vous mettrez en place votre chaîne de filtres customisée, mais rien d’insurmontable. Ce sera très facile d’ajouter et de supprimer différentes méthodes.

En premier lieu, vous devez créer un fichier pour votre toute nouvelle configuration Spring Security.

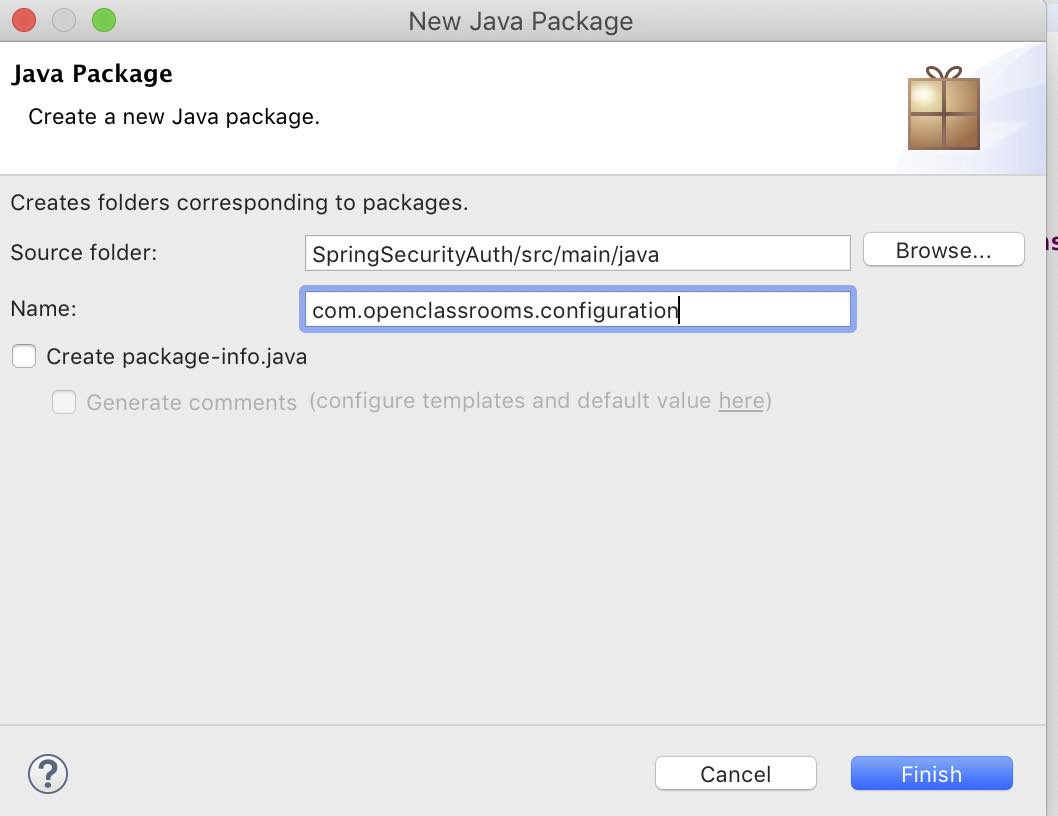
Une application créée sous **Eclipse** répond à une **hiérarchie**. Afin de vous assurer que votre méthode  main  soit la **première** à fonctionner, créez des packages “enfants” sous votre package principal. Ainsi, si votre package principal est  com.openclassrooms, vos autres classes seront placées dans les packages se basant sur ce dernier.

À ce stade, vous disposez de votre classe principale, **SpringSecurityAuthApplication.java**, qui devrait se situer sous le dossier**src/main/java**.

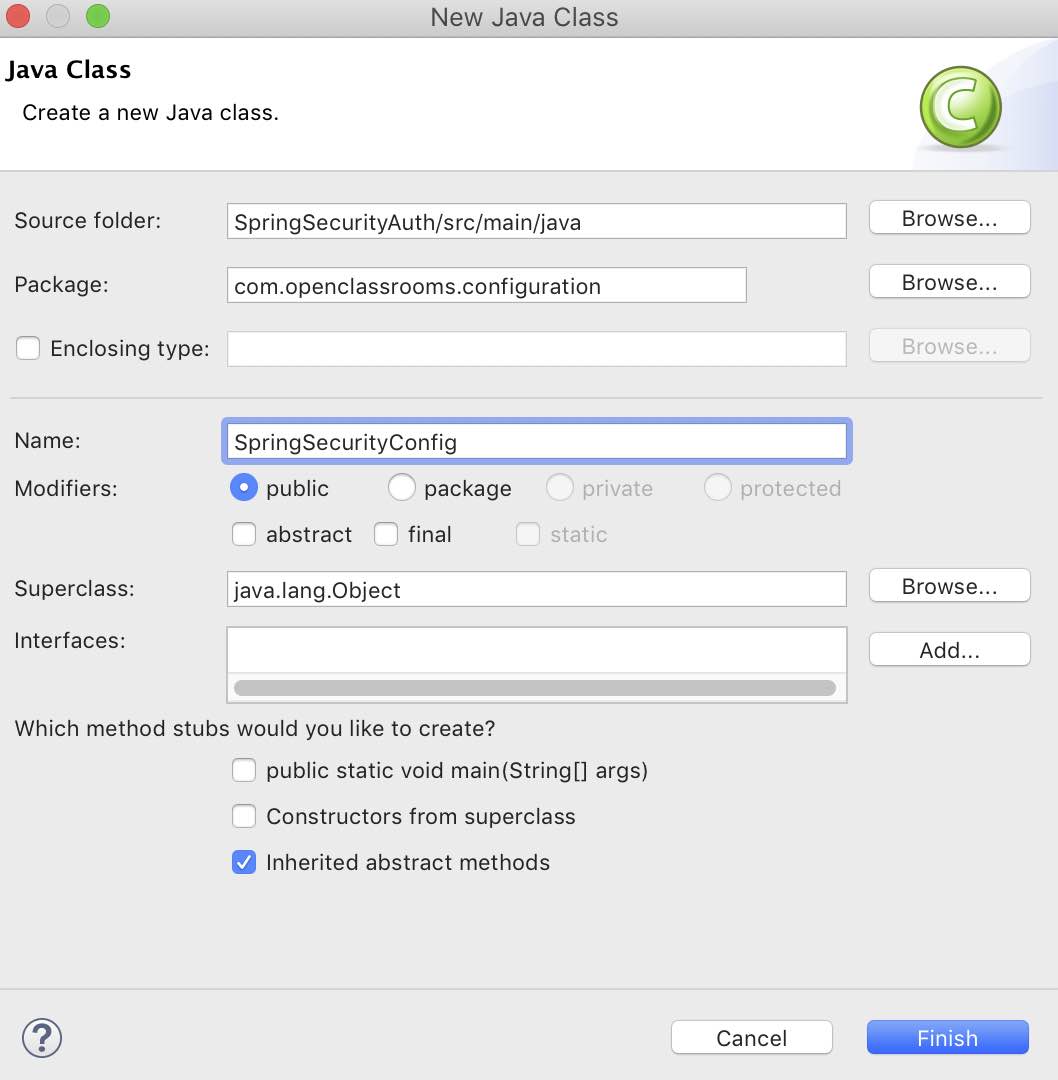
Nous allons ajouter une nouvelle classe pour la configuration du projet ; pour cela, cliquez sur le package  com.openclassrooms  dans vos fichiers Java, et créez un nouveau package.

Créez un nouveau package

Ajoutez la configuration de l’extension au nom du package. Il s'appelle désormais  com.openclassrooms.configuration  .

Créez un package intitulé com.openclassrooms.configuration

Créez une nouvelle classe dans ce package appelée **SpringSecurityConfig**. Faites un clic droit sur  com.openclassrooms.configuration*Package -> New -> Class*:

Créez une classe SpringSecurityConfig

Appelons le nouveau fichier SpringSecurityConfig*.* Il ne vous reste plus qu’à ajouter le nom de classe SpringSecurityConfig sous*Name.*Vous pouvez laisser les informations par défaut, étant donné qu’elles concernent une nouvelle classe publique, et cliquer sur *Finish*.

### Construisez la chaîne de filtres de sécurité

* Ajoutez les annotations **Configuration** et **EnableWebSecurity** pour désigner la **classe SpringSecurityConfig**en tant que configuration de la chaîne de filtres de sécurité pour l’app Spring Boot, et pour la création de la chaîne.
* Ajoutez la méthode  configure()  avec la classe **AuthenticationManagerBuilder** pour gérer la série de règles d’authentification.
* Ajoutez la méthode  configure() qui prend en paramètre un objet  **HTTPSecurity** pour faire passer toutes les requêtes HTTP à travers la chaîne de filtres de sécurité, et configurez le formulaire de connexion par défaut avec la méthode  form Login() .

À présent, regardons en détail chacune de ces étapes.

#### Ajoutez la classe de configuration

Ajoutez @Configuration à votre application web, au-dessus de votre déclaration de classe.

#### Ajoutez WebSecurityConfigurerAdapter

Maintenant, étendez votre classe SpringSecurityConfig avec WebSecurityConfigurerAdapter. Cela définit la classe SpringSecurityConfig en tant que configuration Spring Security.

@Configuration

public class SpringSecurityConfig extends WebSecurityConfigurerAdapter {

}

#### Ajoutez EnableWebSecurity

La dernière étape, c’est l’ajout de l'annotation EnableWebSecurity. Cela permettra de vous assurer que l’application web Spring sache importer votre configuration Spring Security sécurisée.

@Configuration

@EnableWebSecurity

public class SpringSecurityConfig extends WebSecurityConfigurerAdapter {

}

À présent, utilisez la méthode  configure()  de Spring Security pour contenir votre chaîne de filtres de sécurité. C’est une étape délicate, soyez attentif !

##### Add HTTPSecurity

To add some functionality, create a login page with buttons. You can do that in the Spring Security configuration file you are making with **HTTPSecurity.** This class is invoked to run the security filter chain on HTTP requests. By default, it runs security on all of the HTTP requests unless specified in this security filter chain.

This is where you build your security filter chain.

Take a look at this code here. You can start with  configure()  method for all HTTP requests that are the inputs:

@Override

public void configure(HttpSecurity http) throws Exception {

}