Spring Security

Description

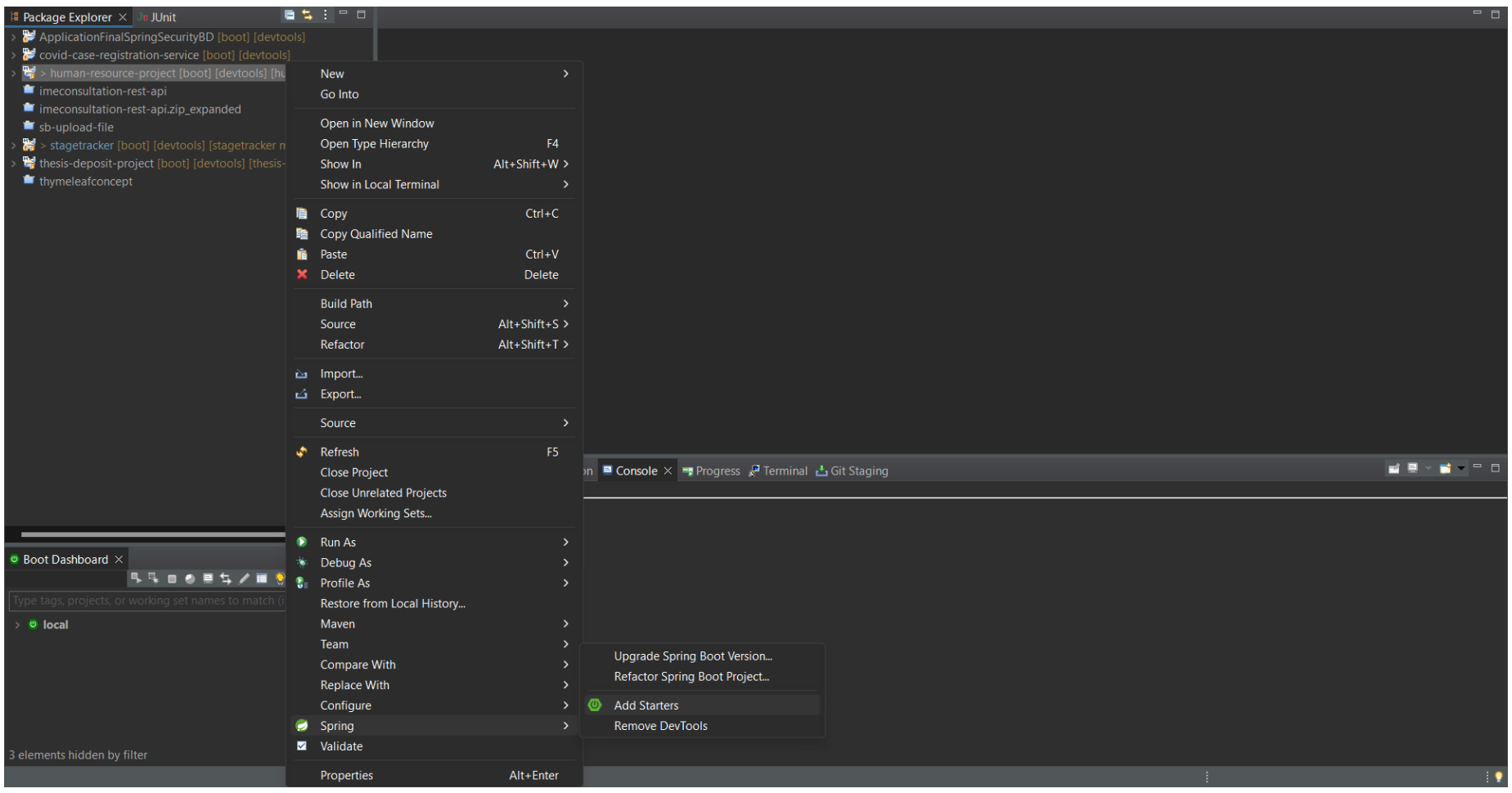
Spring Security est un module de Spring qui permet de gérer la sécurité des applications Java. Il repose sur un ensemble de filtres permettant de filtrer les requêtes entrantes et sortantes. Il peut être intégré avec n'importe quel module de Spring. Spring Security repose sur deux concepts clés :

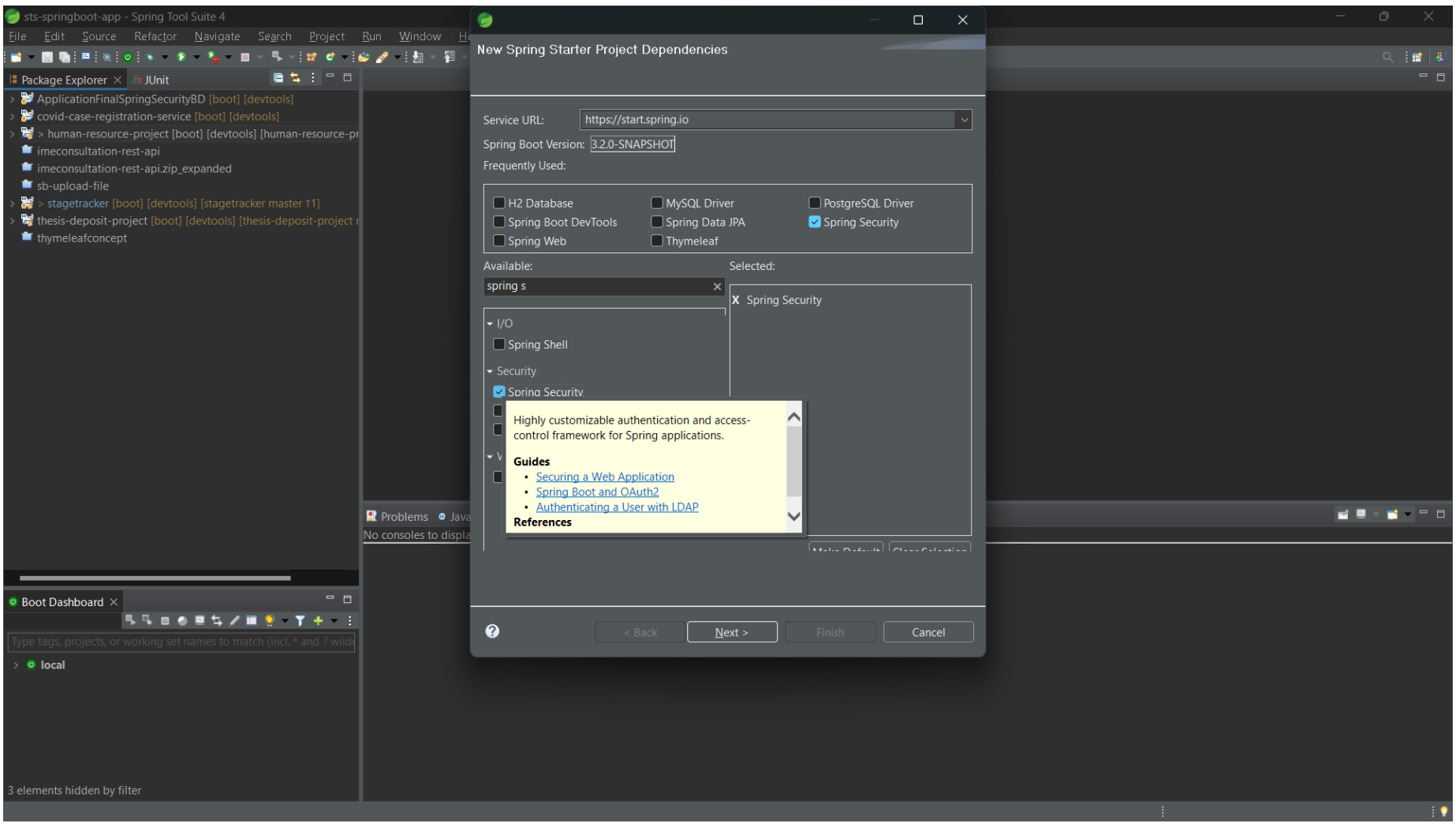
- **Autorisation** : La ressource consultée requiert des permissions en fonction des rôles.
- **Authentification** : Il s'agit de vérifier l'identité de la personne souhaitant accéder à la ressource.

Configuration de Spring Security

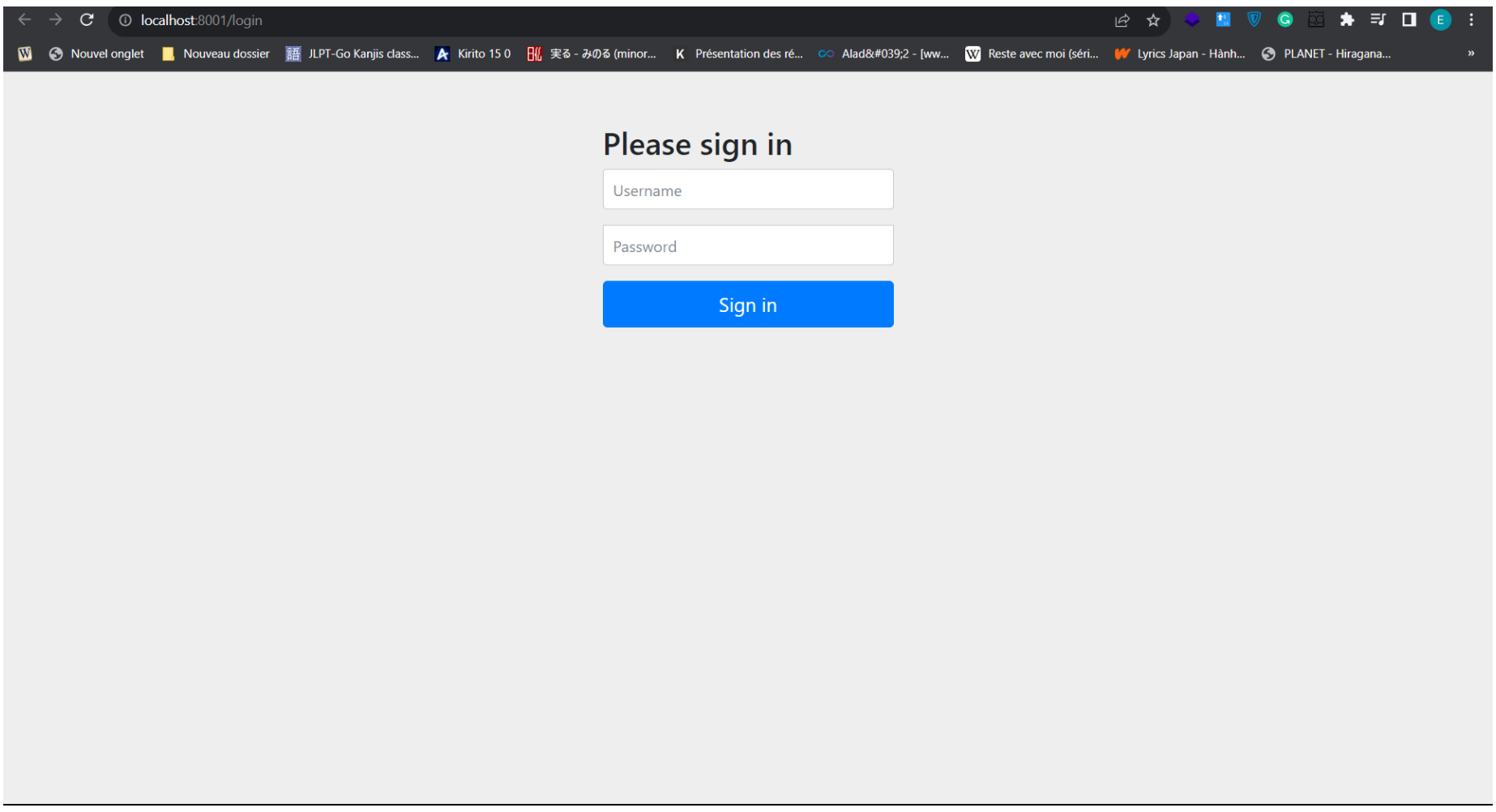
Pour utiliser Spring Security, vous devez ajouter la dépendance directement à votre projet. Actuellement, nous en sommes à la version 6.1.2. Il est essentiel de noter que, comme Spring évolue, la version de Spring Security est cruciale pour éviter les problèmes de dépréciation.

Pour ce faire, accédez au Starter de votre projet, recherchez "Spring Security" et ajoutez-le à votre projet. N'hésitez pas à inclure toutes les informations pertinentes que vous trouvez utiles pour votre configuration.





Une fois ajouté à votre projet, Spring Security renforce considérablement la sécurité de l'application, ce qui signifie que, par défaut, aucune ressource n'est accessible sans autorisation.



Types d'Authentification

Spring Security propose trois types d'authentification :

1. **Authentification en mémoire (InMemory)** : Les informations d'authentification sont stockées en mémoire dans la configuration. Par défaut, Spring Security fournit des informations d'authentification dans la console pour permettre la connexion. Le nom d'utilisateur par défaut est **user**, et le mot de passe généré est affiché dans la console. Spring Security utilise un algorithme de hachage très avancé, *SHA256*, pour le mot de passe, qui est régénéré à chaque redémarrage du projet.
2. **Authentification en base de données** : Dans ce cas, les informations d'authentification sont stockées dans une base de données. Les utilisateurs et les rôles sont gérés via des requêtes SQL, offrant ainsi une flexibilité accrue.
3. **Authentification par console** : Spring Security propose par défaut des informations d'authentification en affichant le nom d'utilisateur et le mot de passe dans la console pour faciliter la connexion.

Vous pouvez choisir le type d'authentification qui convient le mieux à votre application en fonction de vos besoins de sécurité et de gestion des utilisateurs.

```
2023-10-11T14:43:56.534+02:00  WARN 14776 --- [  restartedMain] .s.s.UserDetailsServiceAutoConfiguratic

Using generated security password: d6c98a31-277b-4830-bd7d-ad2b12559c73

This generated password is for development use only. Your security configuration must be updated before
```

Les informations de base de connexion peuvent être modifiées dans le fichier `application.properties`.

```
spring.security.user.name=USERNAME
spring.security.user.password=PASSWORD
```

Configurer la chaine de sécurité (Autorisation)

Pour configurer la chaîne de sécurité dans Spring Security, vous devez créer une classe annotée avec `@EnableWebSecurity` et `@Configuration`. Ensuite, utilisez la méthode `filterChain` pour gérer les autorisations. Cette méthode retourne un objet `SecurityFilterChain` qui contient la chaîne de filtres à utiliser par Spring. N'oubliez pas d'ajouter l'annotation `@Bean` pour charger le contexte de Spring, ce qui vous permettra d'obtenir directement l'objet renvoyé par la méthode.

```
package com.app.config

@Configuration
@EnableWebSecurity
class SecurityConfig {

        @Bean
        public SecurityFilterChain filterChain(HttpSecurity http)
        throws Exception {
                return http.build();
        }
}
```

Les méthodes que l'on peut appliquer aux filtres de Spring Security sont nombreuses. Parmi elles, on trouve :

- `authorizeHttpRequests()`, qui permet d'autoriser les requêtes HTTP en définissant des règles d'autorisation.
- `requestMatchers()`, qui permet de faire correspondre les routes et d'imposer des restrictions sur celles-ci. Vous pouvez les enchaîner avec d'autres méthodes pour des règles plus complexes.
- `anyRequest()`, pour définir des règles s'appliquant à toutes les requêtes.
- `authenticated()`, pour exiger une authentification pour accéder à une ressource.
- `permitAll()`, pour permettre l'accès à toutes les requêtes sans restriction.
- `and()`, qui permet de définir une nouvelle règle en chaînant des conditions.
- `formLogin()`, pour configurer l'authentification via un formulaire.
- `loginPage()`, qui vous permet de spécifier le chemin vers la page d'authentification personnalisée si vous en avez une.
- `defaultSuccessUrl()`, pour définir l'URL vers laquelle l'utilisateur sera redirigé après une authentification réussie.
- `logout()`, pour configurer la déconnexion.
- `hasAuthority()`, pour vérifier si un utilisateur a une autorité particulière.
- `hasRole("ROLE")` : Cette méthode permet de vérifier si l'utilisateur a un rôle spécifique. Par exemple, `hasRole("ADMIN")` vérifiera si l'utilisateur a le rôle "ADMIN".
- `hasAnyRole("ROLE1", "ROLE2", ...)` : Cette méthode permet de vérifier si l'utilisateur a au moins l'un des rôles spécifiés. Par exemple, `hasAnyRole("USER", "ADMIN")` vérifiera si l'utilisateur a soit le rôle "USER" soit le rôle "ADMIN"

Voici un exemple de configuration de filtres pour une application web qui doit réaliser les actions suivantes :

- Autoriser l'accès aux fichiers statiques tels que les fichiers CSS, les images et les scripts JavaScript.
- Autoriser l'accès aux pages d'accueil, d'inscription et de connexion pour tous les utilisateurs.

- Autoriser les utilisateurs ayant les rôles ADMIN et USER à effectuer des actions telles que l'ajout, la modification et la suppression de produits.
- Autoriser les utilisateurs ayant le rôle USER à consulter la liste des produits.
- Configurer une page d'authentification personnalisée pour améliorer l'expérience utilisateur.

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.provisioning.InMemoryUserDetailsManager;
import org.springframework.security.web.SecurityFilterChain;

@Configuration
@EnabledWebConfig
public class SecurityConfig {

    @Bean
    public SecurityFilterChain
    defaultSecurityFilterChain(HttpSecurity http) throws Exception {
        http
            .authorizeRequests(authorizeRequests ->
                authorizeRequests
                // Autoriser l'accès aux fichiers statiques
                        .antMatchers("/css/**", "/images/**","/js/**").permitAll()
                    // Autoriser l'accès aux pages d'accueil, d'inscription et de connexion
                    .antMatchers("/", "/register", "/login").permitAll()
                    .antMatchers("/addProduct", "/modifyProduct",
"/deleteProduct").hasAnyRole("ADMIN", "USER") // Autoriser les actions d'ajout, de modification et de
suppression de produits pour les utilisateurs avec les rôles ADMIN et USER
                    .antMatchers("/viewProducts").hasRole("USER") // Autoriser la consultation des
produits pour les utilisateurs avec le rôle USER
                    .anyRequest().authenticated()
            )
            .formLogin(withDefaults()) // Configuration de l'authentification par formulaire
                    .loginPage("/custom-login") // Configuration d'une page d'authentification
personnalisée
                    .defaultSuccessUrl("/dashboard") // Redirection après une authentification réussie
            .and()
                    .logout()
                            .logoutUrl("/custom-logout"); // Configuration de la déconnexion
personnalisée
        return http.build();
    }
}
```

Authentification basé sur la base donnée

Parmi les méthodes d'authentification proposées par Spring Security, nous avons l'authentification en base de données. Pour l'implémenter, Spring Security fournit l'interface **UserDetailsService**, qui propose une méthode **loadUserByUsername** renvoyant un objet **UserDetails** permettant d'accéder à l'objet **User** de Spring Security, représentant un utilisateur. Cette méthode est automatiquement appelée par Spring Security lors de l'authentification.

L'objet User de Spring Security se trouve dans le package `org.springframework.security.core.userdetails`. Il est essentiel de préciser ce package si votre projet comporte des objets User portant le même nom de classe.

L'objet User est constitué des propriétés suivantes :

- `username` : String
- `password` : String
- `Collections<GrantedAuthority>` : Un tableau contenant les rôles de l'utilisateur.

Dans l'implémentation, il est nécessaire de créer une classe de service qui implémente cette interface et définit la logique pour la méthode `loadUserByUsername`.

Il est également important de mettre à jour la configuration de la sécurité en ajoutant le hachage des mots de passe à l'aide de la méthode `passwordEncoder`.

Exemple d'implémentation possible

```
@Service
public class CustomUserDetailsService implements UserDetailsService {

	@Autowired
	private DBUserRepository dbUserRepository;

	@Override
	public UserDetails loadUserByUsername(String username)
	throws  UsernameNotFoundException
	{
		DBUser user = dbUserRepository.findByUsername(username);
		return new User(user.getUsername(),
		user.getPassword(),
		getGrantedAuthorities(user.getRole()));
	}

	private List<GrantedAuthority> getGrantedAuthorities(String role) {
		List<GrantedAuthority> authorities = new ArrayList<GrantedAuthority>();
		authorities.add(new SimpleGrantedAuthority("ROLE_" + role));
		return authorities;
	}
}
```

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.web.SecurityFilterChain;

@Configuration
@EnabledWebConfig
public class SecurityConfig {

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

SecurityContext

Spring Security fournit un `SecurityContext` qui est un conteneur pour les informations de sécurité dans une application Spring. Le contexte de sécurité contient des informations telles que l'utilisateur actuellement authentifié, ses rôles, ses autorisations, etc. Il est stocké dans un thread local ou une session HTTP et est accessible via la classe `SecurityContextHolder`.

Le contexte de sécurité permet de gérer et d'accéder aux informations de sécurité tout au long du cycle de vie d'une demande HTTP. Il assure la gestion de l'authentification, de l'autorisation et de la gestion des utilisateurs. Il permet également de stocker des détails sur l'utilisateur, comme son nom d'utilisateur, ses rôles, ses autorisations, etc., de manière sécurisée.

Le securityContext contient l'interface `Authentication`

L'interface `Authentication` de Spring Security remplit deux principaux rôles au sein de Spring Security :

1. **Fournir des informations d'authentification à l'AuthenticationManager** : L'interface `Authentication` est utilisée pour transmettre les informations d'authentification (comme les identifiants fournis par l'utilisateur pour s'authentifier, par exemple un nom d'utilisateur et un mot de passe) à l'`AuthenticationManager`. Lorsqu'elle est utilisée à cette fin, la méthode `isAuthenticated()` renvoie généralement `false` car l'authentification n'est pas encore effectuée à ce stade.
2. **Représenter l'utilisateur actuellement authentifié** : Une fois que l'utilisateur est authentifié avec succès, l'objet `Authentication` représente cet utilisateur authentifié. Vous pouvez obtenir l'objet `Authentication` actuel à partir du contexte de sécurité (SecurityContext).

L'objet `Authentication` contient les éléments suivants :

- `principal` : Il s'agit de l'identité de l'utilisateur. En général, lorsqu'on s'authentifie avec un nom d'utilisateur et un mot de passe, le `principal` est souvent une instance de `UserDetails`, qui contient des informations sur l'utilisateur, telles que le nom d'utilisateur, le mot de passe haché, les rôles, etc.
- `credentials` : Il s'agit généralement du mot de passe fourni par l'utilisateur lors de l'authentification. Dans de nombreux cas, ce champ est effacé après une authentification réussie pour éviter toute fuite.
- `authorities` : Il s'agit des instances de `GrantedAuthority`, qui représentent des autorisations de haut niveau accordées à l'utilisateur. Deux exemples courants d'autorisations sont les rôles (rôles) et les scopes.

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

// Principal
Principal = auth.getAuthentication().getPrincipal()

// Roles
Collection<GrantedAuthority> autorities = auth.getAuthorities()
```

On peut injecter directement le **@AuthenticationPrincipal** pour récupérer l'utilisateur connecté.