

Zéro Tolerance

Spring Security avec JWT, OAUTH2

SPRING SECURITY

Les questions classiques ?

SECURITY

Comment puis-je implémenter la sécurité de mes applications web/mobiles afin qu'il n'y ait pas de failles de sécurité dans mon application ?

PASSWORDS

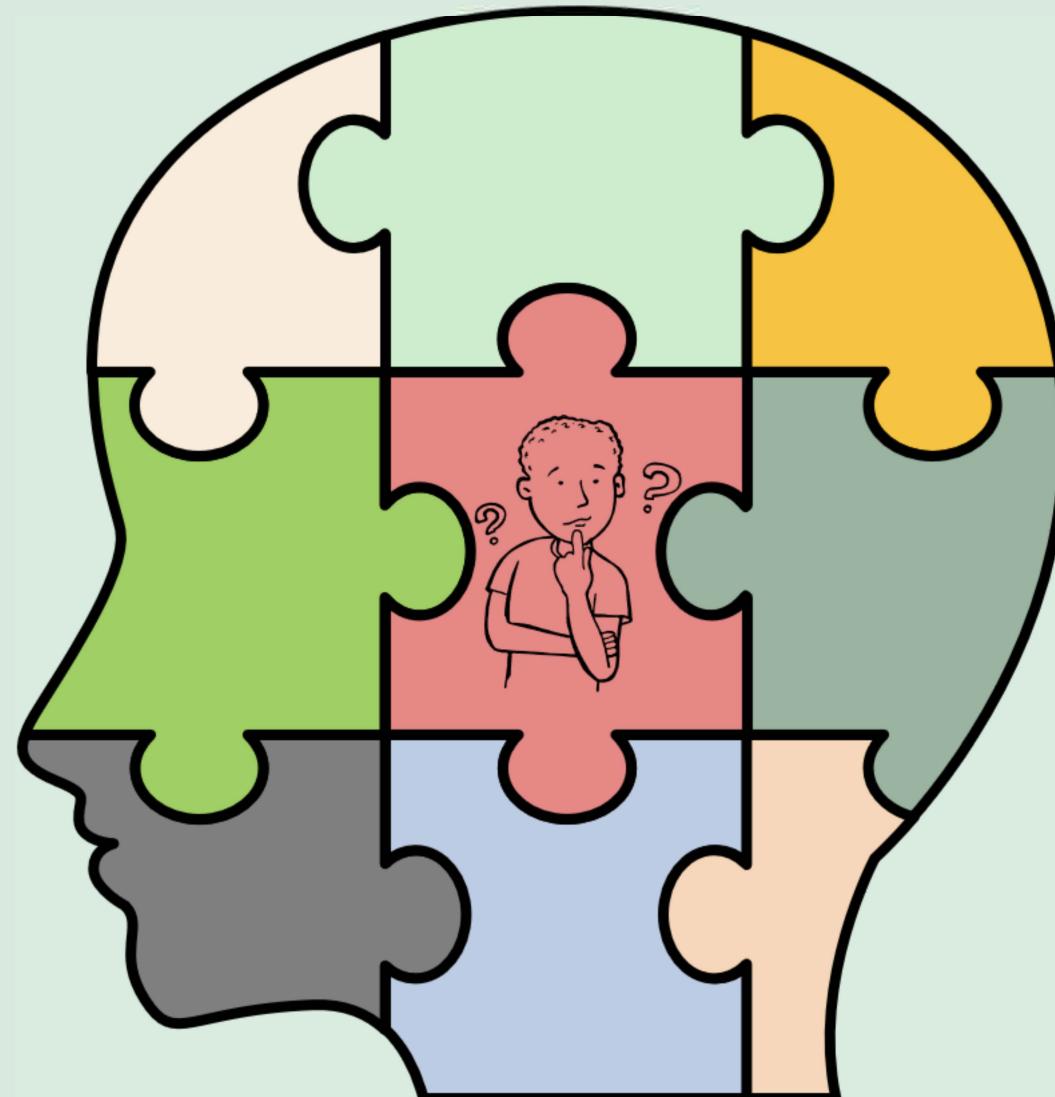
Comment stocker les mots de passe, les valider, les encoder, les décoder à l'aide d'algorithmes de chiffrement standard de l'industrie ?

USERS & ROLES

Comment maintenir la sécurité au niveau des utilisateurs en fonction de leurs rôles et des droits qui leur sont associés ?

MULTIPLE LOGINS

Comment puis-je implémenter un mécanisme où l'utilisateur se connectera uniquement et commencera à utiliser mon application ?



SECURISATION FINE

Comment puis-je mettre en œuvre la sécurité à chaque niveau de mon application à l'aide de règles d'autorisation ?

CSRF & CORS

Qu'est-ce que les attaques CSRF et les restrictions CORS. Comment les surmonter ?

JWT & OAUTH2

Qu'est-ce que JWT et OAUTH2. Comment puis-je protéger mon application Web en les utilisant ?

PRÉVENIR LES ATTAQUES

Comment prévenir les attaques de sécurité comme la force brute, le vol de données, la fixation de session

AGENDA DU COURS



Bienvenue
dans le monde
de Spring
Security

Sécuriser une
application Web
avec Spring
Security

Interfaces
importantes, classes,
annotations de
Spring Security

Configuration de
l'authentification
et de l'autorisation
pour une
application Web

AGENDA DU COURS

Implémentation d'un accès basé sur les rôles à l'aide de RÔLES, AUTORITES

Différentes stratégies fournies par Spring Security en matière de mots de passe

Sécurité au niveau de la méthode à l'aide de Spring Security

Comment gérer les attaques les plus courantes comme CORS, CSRF avec Spring Security

AGENDA DU COURS

Plongée en profondeur sur JWT et son rôle dans l'authentification et l'autorisation

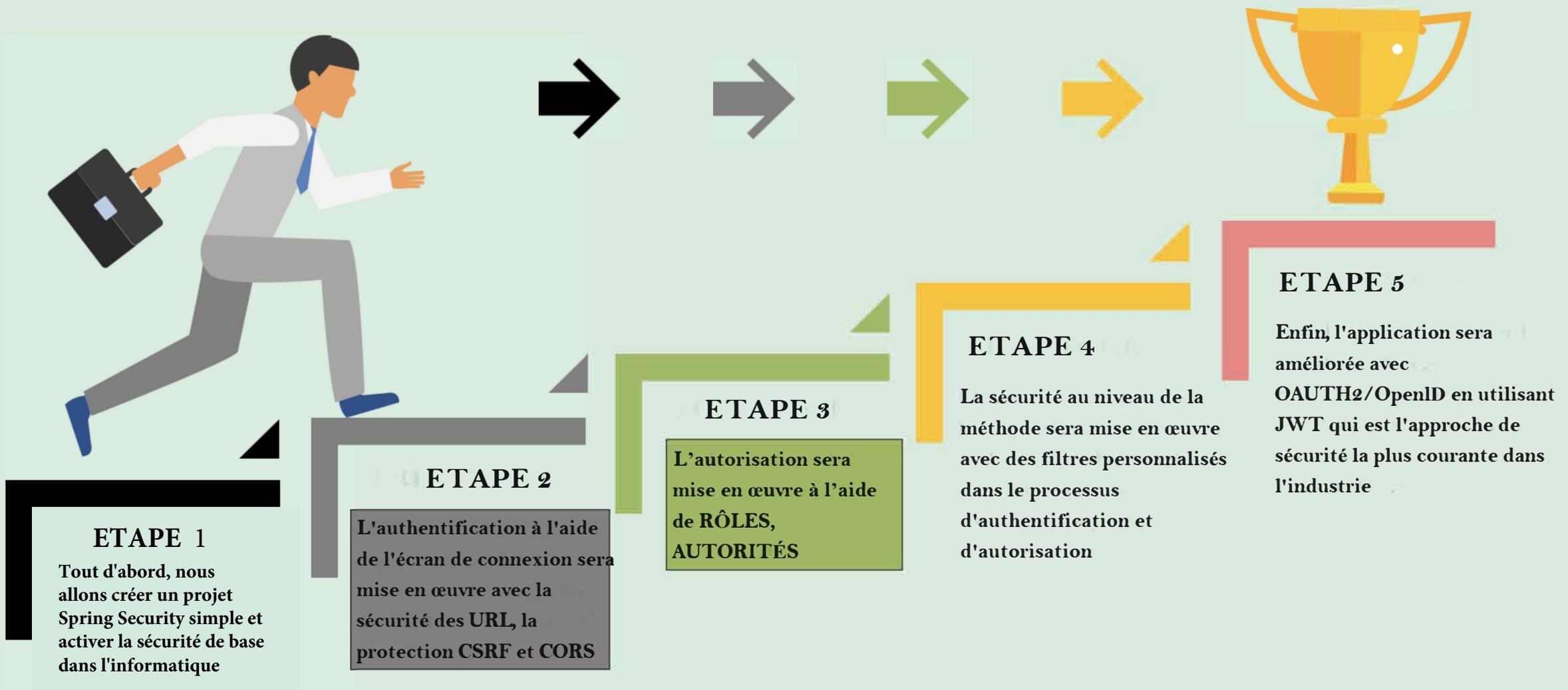
Plongée en profondeur sur OAUTH2, OpenID et sécurisation d'une application Web utilisant le même

Explorer les serveurs d'autorisation disponibles comme Keycloak

Sujets importants de la sécurité comme le hachage, les jetons et bien d'autres

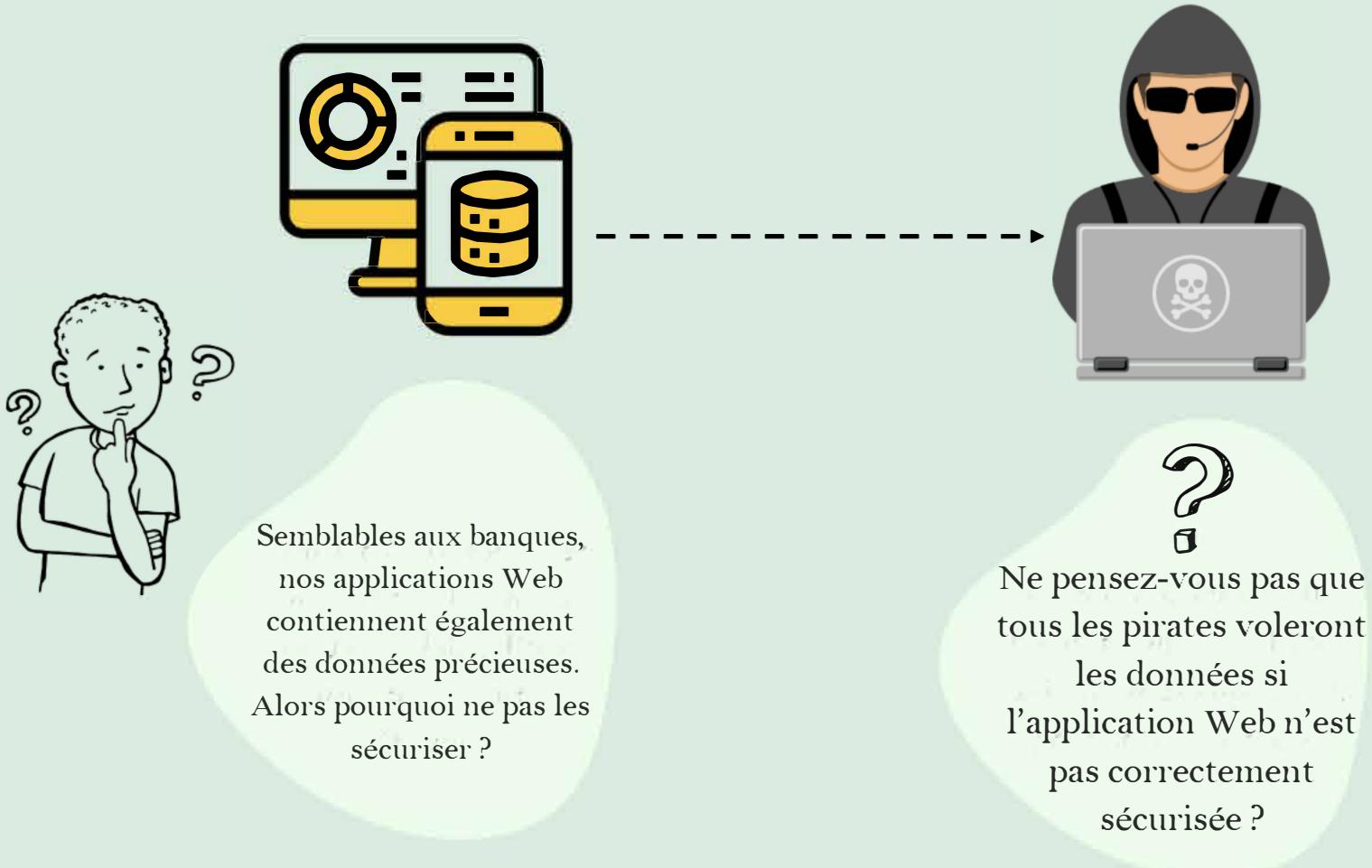
SPRING SECURITY

FEUILLE DE ROUTE DU PROJET



INTRODUCTION À LA SÉCURITÉ

QUOI & POURQUOI



INTRODUCTION À LA SÉCURITÉ

QUOI & POURQUOI



WHAT IS SECURITY?

La sécurité consiste à protéger vos données et votre logique métier dans vos applications Web.



DIFFERENT TYPES OF SECURITY

La sécurité d'une application Web sera implémentée de différentes manières, comme l'utilisation de pare-feu, HTTPS, SSL, authentification, autorisation, etc.



SECURITY IS AN NON FUN REQ

La sécurité est très importante, tout comme l'évolutivité, les performances et la disponibilité. Aucun client ne me demandera spécifiquement que j'ai besoin de sécurité.



WHY SECURITY IMPORTANT?

La sécurité ne signifie pas seulement perdre des données ou de l'argent, mais également la marque et la confiance que vous accordez à vos utilisateurs ont construit au fil des années.



SECURITY FROM DEV PHASE

La sécurité doit être prise en compte dès phase de développement elle-même avec la logique métier.



AVOIDING MOST COMMON ATTACKS

En utilisant la sécurité, nous devons également éviter les attaques de sécurité les plus courantes telles que CSRF, Broken Authentication à l'intérieur de notre application.

POURQUOI SPRING SECURITY ?



La sécurité des applications n'est ni amusante ni difficile à mettre en œuvre avec notre code/framework personnalisé.



Spring Security construit par l'équipe de Spring est bonne en matière de sécurité en considérant tous les scénarios de sécurité. En utilisant Spring Security, nous pouvons sécuriser les applications Web avec des configurations minimales. Il n'est donc pas nécessaire de réinventer la roue ici.



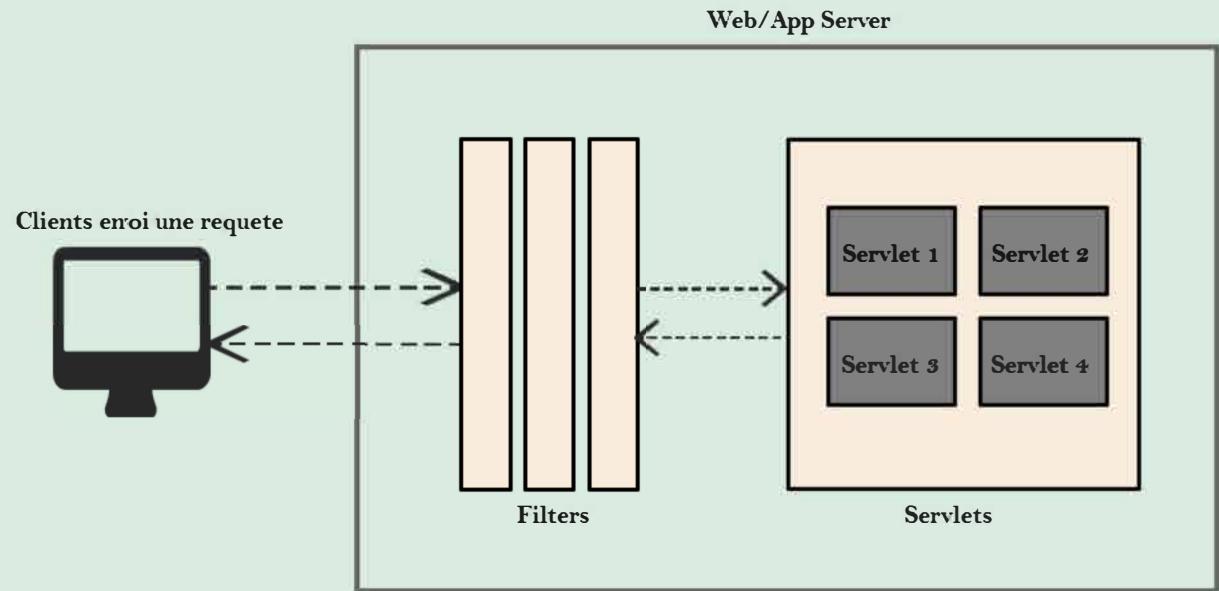
Spring Security gère les vulnérabilités de sécurité courantes telles que CSRF, CORS, etc. Pour toutes les vulnérabilités de sécurité identifiées, le cadre sera immédiatement corrigé car il est utilisé par de nombreuses organisations.

En utilisant Spring Security, nous pouvons sécuriser nos pages/chemins d API, appliquer des rôles, la sécurité au niveau de la méthode, etc. avec des configurations minimales facilement.



Spring Security prend en charge diverses normes de sécurité pour implémenter l'authentification, comme l'utilisation de l'authentification par nom d'utilisateur/mot de passe, les jetons JWT, OAuth2, OpenID, etc.

SERVLETS & FILTRES



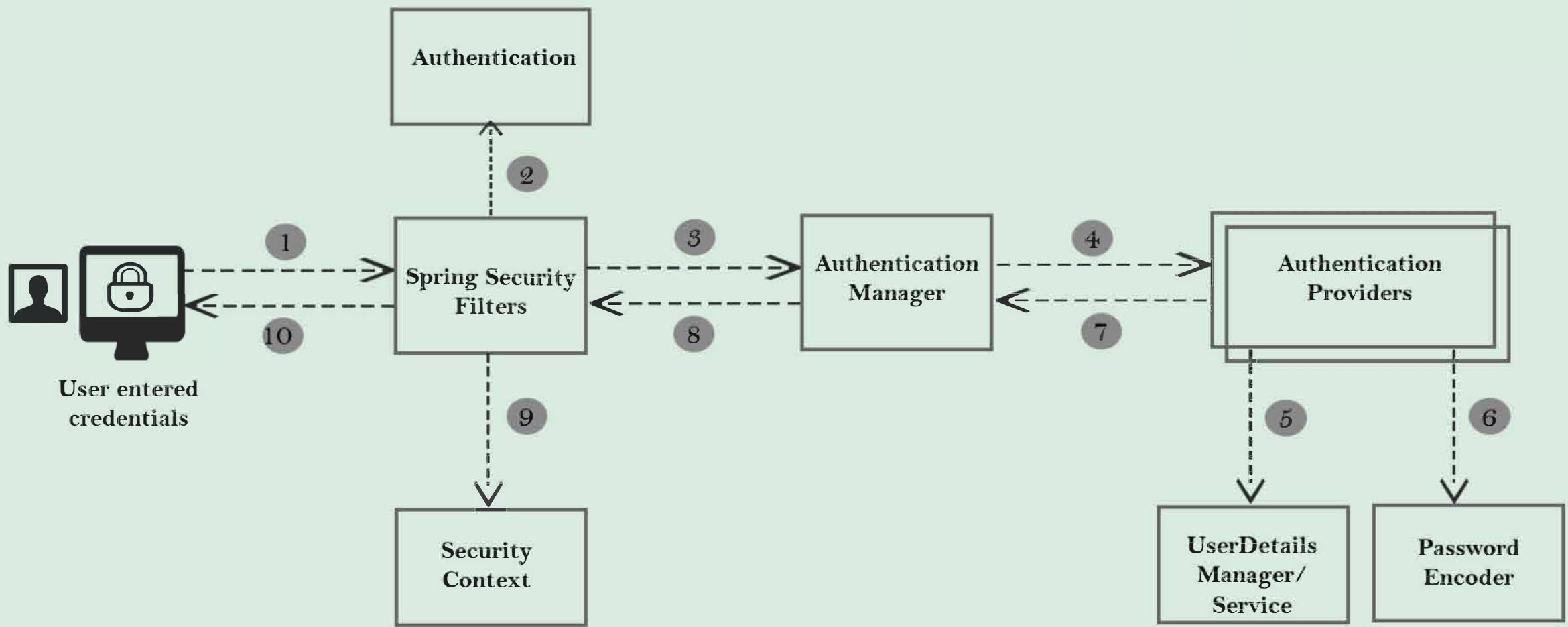
★ Scénario typique dans une application Web

Dans les applications Web Java, Servlet Container (Web Server) se charge de traduire les messages HTTP pour que le code Java les comprenne. L'un des conteneurs de servlet les plus utilisés est Apache Tomcat. Servlet Container convertit les messages HTTP en ServletRequest et les transmet à la méthode `Servlet` en tant que paramètre. De même, `ServletResponse` renvoie en tant que sortie au conteneur de servlet à partir de `Servlet`. Ainsi, tout ce que nous écrivons dans les applications Web Java est piloté par des servlets

★ Rôle des filtres

Les filtres à l'intérieur des applications Web Java peuvent être utilisés pour intercepter chaque requête/réponse et effectuer un travail préalable avant notre logique métier. Ainsi, en utilisant les mêmes filtres, Spring Security applique la sécurité en fonction de nos configurations à l'intérieur d'une application Web.

SPRING SECURITY INTERNAL FLOW



SPRING SECURITY FLUX INTERNE

★ Spring Security Filters

Une série de filtres Spring Security interceptent chaque demande et travaillent ensemble pour identifier si l'authentification est requise ou non. Si l'authentification est requise, naviguez en conséquence sur la page de connexion de l'utilisateur ou utilisez les détails existants stockés lors de l'authentification initiale..

★ Authentication

Des filtres comme UsernamePasswordAuthenticationFilter extrairont nom d'utilisateur/mot de passe de la requête HTTP et prépareront le type d'authentification abject. Parce que l'authentification est la norme de base pour le stockage des détails des utilisateurs authentifiés dans le cadre de sécurité Spring.

★ AuthenticationManager

Une fois la demande reçue du filtre, il délègue la validation des détails de l'utilisateur aux fournisseurs d'authentification disponibles. Puisqu'il peut y avoir plusieurs fournisseurs dans une application, il incombe à AuthenticationManager de gérer tous les fournisseurs d'authentification disponibles.

★ AuthenticationProvider

AuthenticationProviders possède toute la logique de base de la validation des détails de l'utilisateur pour l'authentification.

★ UserDetailsManager/UserDetailsService

UserDetailsManager/UserDetailsService aide à récupérer, créer, mettre à jour, supprimer les détails de l'utilisateur des systèmes de base de données/de stockage

★ PasswordEncoder

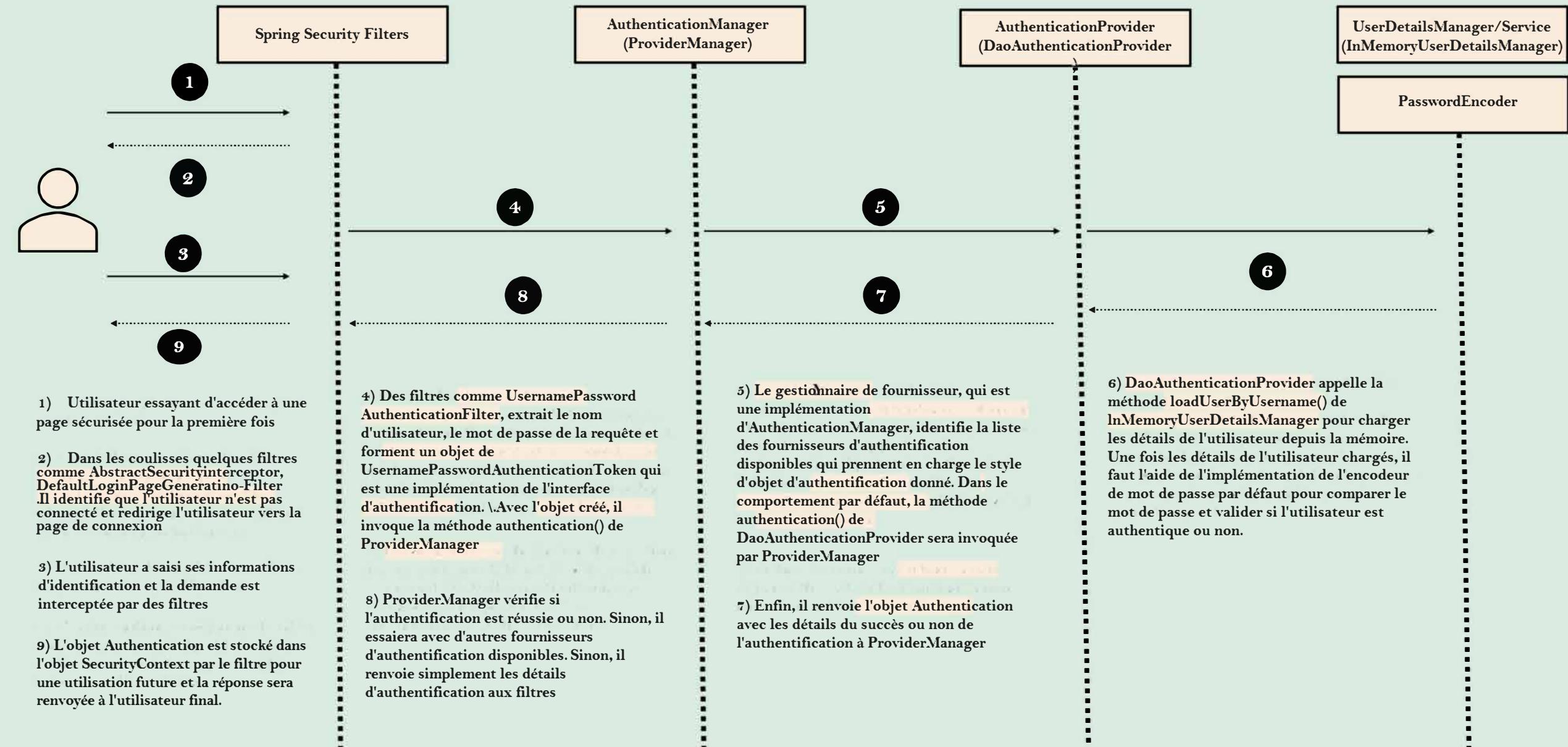
Interface de service qui aide à encoder et hacher les mots de passe. Sinon, nous devrons peut-être vivre avec des mots de passe en texte brut.

★ SecurityContext

Une fois la requête authentifiée, l'authentification sera généralement stockée dans un thread-localSecurityContext géré par le SecurityContextHolder. Cela aide lors des requêtes à venir du même utilisateur.

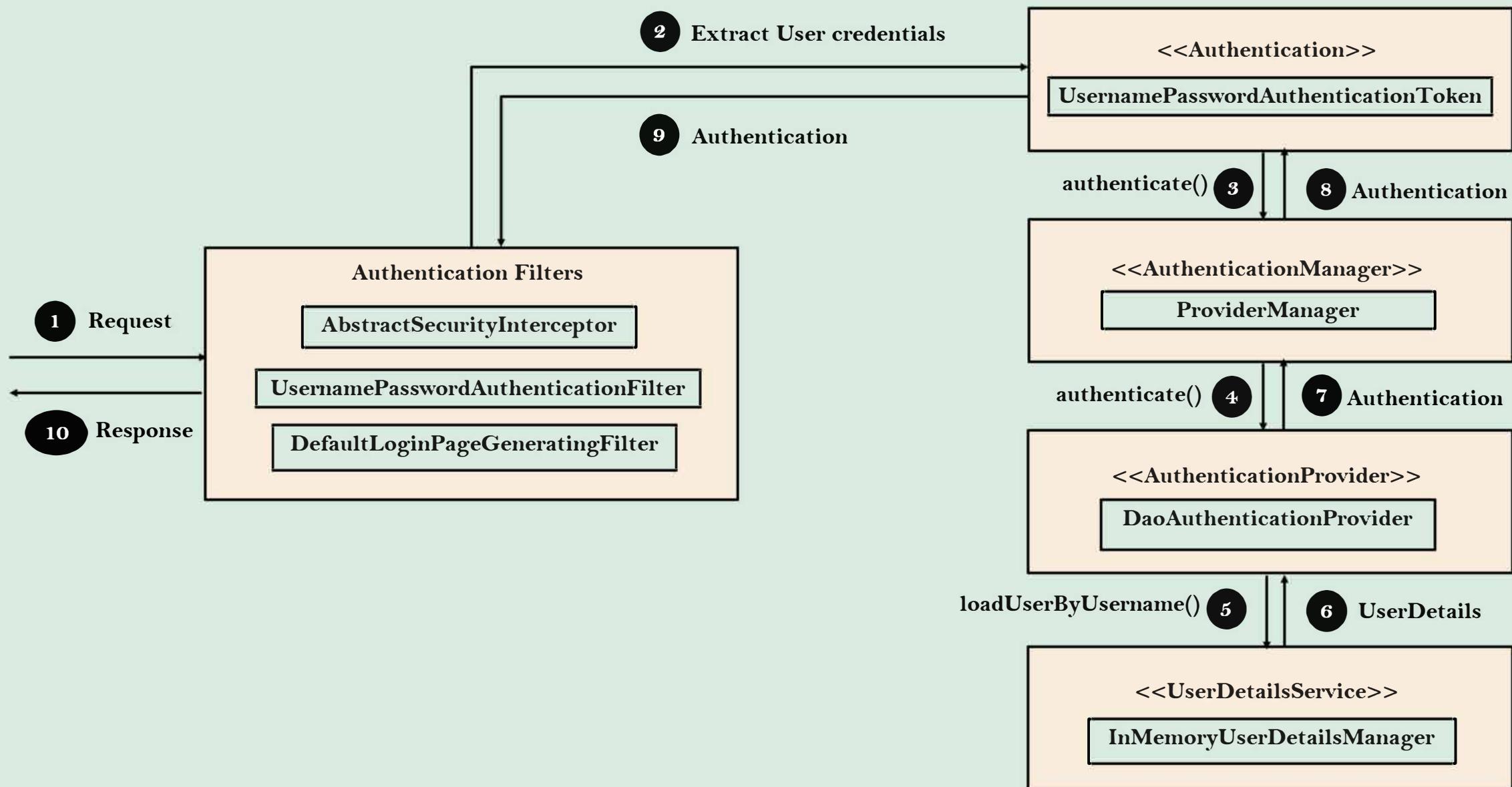
SEQUENCE FLOW

SPRING SECURITY DEFAULT BEHAVIOUR



FLUX DE SÉQUENCE

SPRING SECURITY COMPORTEMENT PAR DÉFAUT



BACKEND REST SERVICES

FOR EAZYBANK APPLICATION



Services sans aucune sécurité

/contact – Ce service doit envoyer les détails du compte de l'utilisateur connecté de la base de données à l'interface utilisateur

/notices – Ce service doit envoyer les détails de l'avis de la base de données à la page "AVIS" de l'interface utilisateur



Services avec sécurité

/myAccount – Ce service doit envoyer les détails du compte de l'utilisateur connecté de la base de données à l'interface utilisateur

/myBalance – Ce service doit envoyer le solde et les détails de la transaction de l'utilisateur connecté de la base de données à l'interface utilisateur

/myLoans – Ce service doit envoyer les détails du prêt de l'utilisateur connecté de la base de données à l'interface utilisateur

/myCards – Ce service doit envoyer les détails de la carte de l'utilisateur connecté de la base de données à l'interface utilisateur

CONFIGURATIONS DE SÉCURITÉ PAR DÉFAUT

A L'INTERIEUR DE SPRING SECURITY FRAMEWORK

*Par défaut, le framework Spring Security protège tous les chemins présents à l'intérieur de l'application web.
Ce comportement est dû au code présent dans la méthode defaultSecurityFilterChain(HttpSecurity http)
de la classe SpringBootWebSecurityConfiguration*

```
@Bean  
@Order(SecurityProperties.BASIC_AUTH_ORDER)  
SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {  
    http.authorizeHttpRequests().anyRequest().authenticated();  
    http.formLogin();  
    http.httpBasic();  
    http.setSharedObject(SecurityContextRepository.class, new DelegatingSecurityContextRepository(  
        new RequestAttributeSecurityContextRepository(), new HttpSessionSecurityContextRepository()));  
    return http.build();  
}  
}
```

CONFIGURATIONS DE SÉCURITÉ PAR DÉFAUT

A L'INTERIEUR DE SPRING SECURITY FRAMEWORK

```
@Configuration
public class ProjectSecurityConfig {

    @Bean
    SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {

        http.authorizeHttpRequests()
            .requestMatchers("/myAccount","/myBalance","/myLoans","/myCards").authenticated()
            .requestMatchers("/notices","/contact").permitAll()
            .and().formLogin()
            .and().httpBasic();

        return http.build();
    }
}
```

REFUSER TOUTES LES CONFIGURATIONS DE SÉCURITÉ

A L'INTERIEUR DE SPRING SECURITY FRAMEWORK

NOT RECOMMENDED
FOR PRODUCTION

Nous pouvons refuser toutes les requêtes provenant de nos API d'application Web, les chemins utilisant le framework Spring Security comme suivra ci-dessous,

```
@Configuration
public class ProjectSecurityConfig {

    @Bean
    SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {
        http.authorizeHttpRequests()
            .anyRequest().denyAll()
            .and().formLogin()
            .and().httpBasic();

        return http.build();
    }
}
```

AUTORISER TOUTES LES CONFIGURATIONS DE SÉCURITÉ

A L'INTERIEUR DE SPRING SECURITY FRAMEWORK

NOT RECOMMENDED
FOR PRODUCTION

Nous pouvons autoriser toutes les requêtes provenant de nos API d'application Web, les chemins utilisant le framework Spring Security comme indiqué ci-dessous,

```
@Configuration
public class ProjectSecurityConfig {

    @Bean
    SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {
        http.authorizeHttpRequests()
            .anyRequest().permitAll()
            .and().formLogin()
            .and().httpBasic();

        return http.build();
    }
}
```

CONFIGURE USERS

Utilisation *InMemoryUserDetailsManager*

NOT RECOMMENDED
FOR PRODUCTION

Au lieu de définir un seul utilisateur dans application.properties, dans une prochaine étape, nous pouvons définir plusieurs utilisateurs ainsi que leurs autorités à l'aide de InMemoryUserDetailsManager & UserDetails

Approche 1 où nous utilisons la méthode withDefaultPasswordEncoder() lors de la création des UserDetails

```
@Bean
public InMemoryUserDetailsManager userDetailsService(){
    UserDetails admin = User.withDefaultPasswordEncoder()
        .username("admin")
        .password("12345")
        .authorities("admin")
        .build();
    UserDetails user = User.withDefaultPasswordEncoder()
        .username("user")
        .password("12345")
        .authorities("read")
        .build();
    return new InMemoryUserDetailsManager(admin, user);
}
```

NOT RECOMMENDED
FOR PRODUCTION

CONFIGURE USERS

USING *InMemoryUserDetailsManager*

Au lieu de définir un seul utilisateur dans application.properties, dans une prochaine étape, nous pouvons définir plusieurs utilisateurs ainsi que leurs autorités à l'aide de InMemoryUserDetailsManager & UserDetails

Approche 2 où nous créons un bean de PasswordEncoder séparément

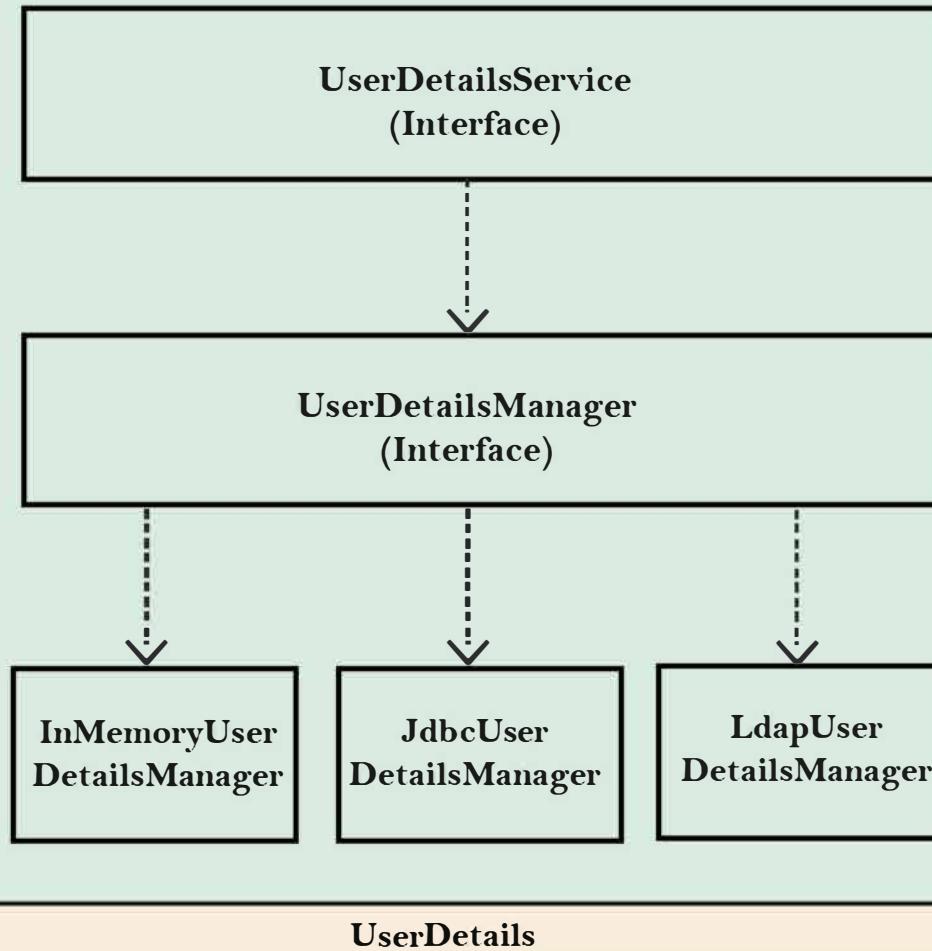
```
@Bean
public InMemoryUserDetailsManager userDetailsService(){
    InMemoryUserDetailsManager inMemoryUserDetailsManager = new InMemoryUserDetailsManager();
    UserDetails admin = User.withUsername("admin").password("12345").authorities("admin").build();
    UserDetails user = User.withUsername("user").password("12345").authorities("read").build();
    inMemoryUserDetailsManager.createUser(admin);
    inMemoryUserDetailsManager.createUser(user);
    return inMemoryUserDetailsManager;
}

/**
 * NoOpPasswordEncoder is not recommended for production usage.
 * Use only for non-prod.
 *
 * @return PasswordEncoder
 */
@Bean
public PasswordEncoder passwordEncoder() {
    return NoOpPasswordEncoder.getInstance();
}
```

GESTION DES UTILISATEURS

CLASSES ET INTERFACES IMPORTANTES

Interface centrale qui charge des données spécifiques à l'utilisateur.



- ✓ `loadUserByUsername(String username)`
- ✓ `createUser(UserDetails user)`
- ✓ `updateUser(UserDetails user)`
- ✓ `deleteUser(String username)`
- ✓ `changePassword(String oldPwd, String newPwd)`
- ✓ `userExists(String username)`

Une extension de **UserDetailsService** qui permet de créer de nouveaux utilisateurs et de mettre à jour ceux qui existent déjà.

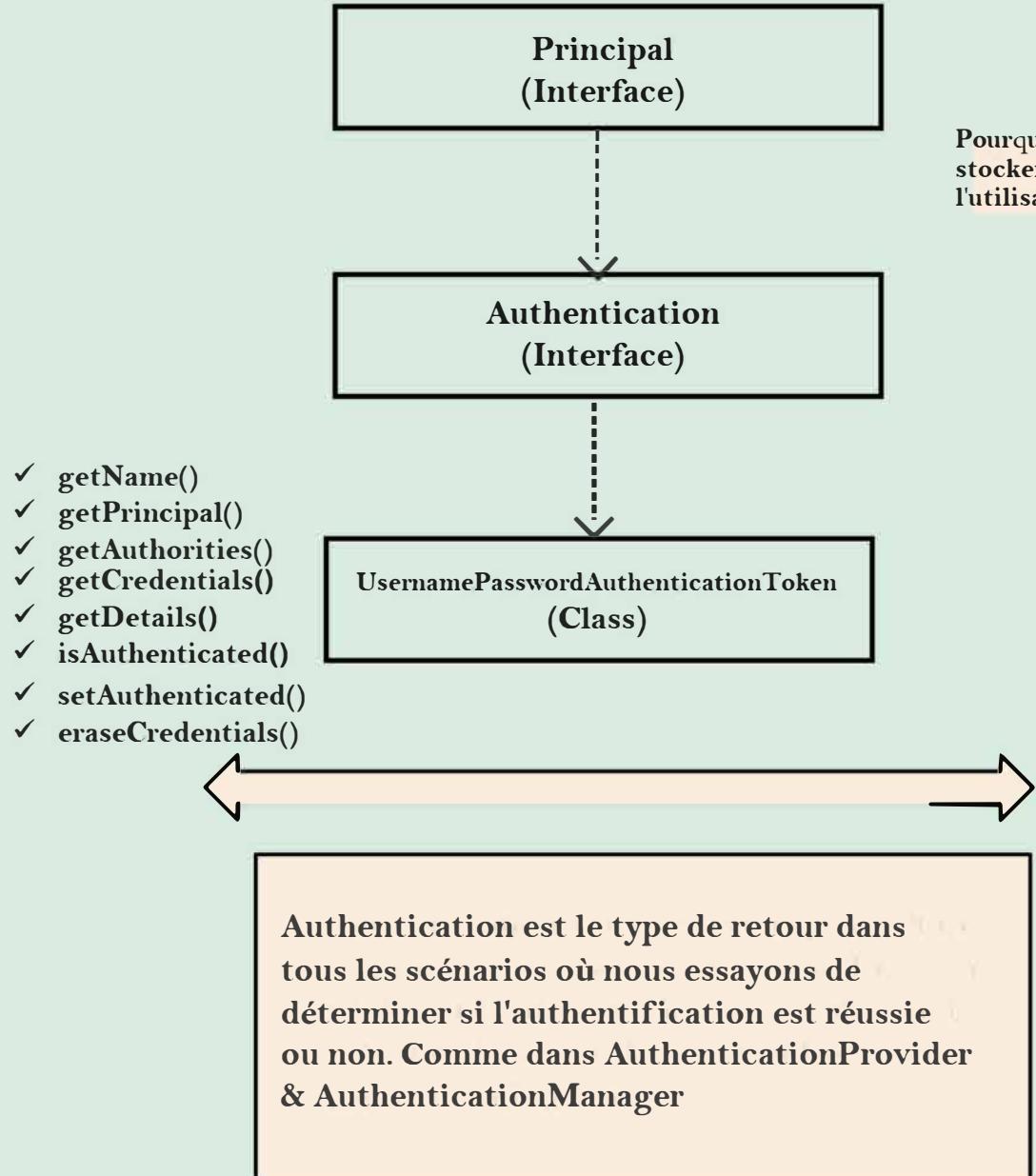
Exemples de cours d'implémentation fournis par l'équipe Spring Security

UserDetails

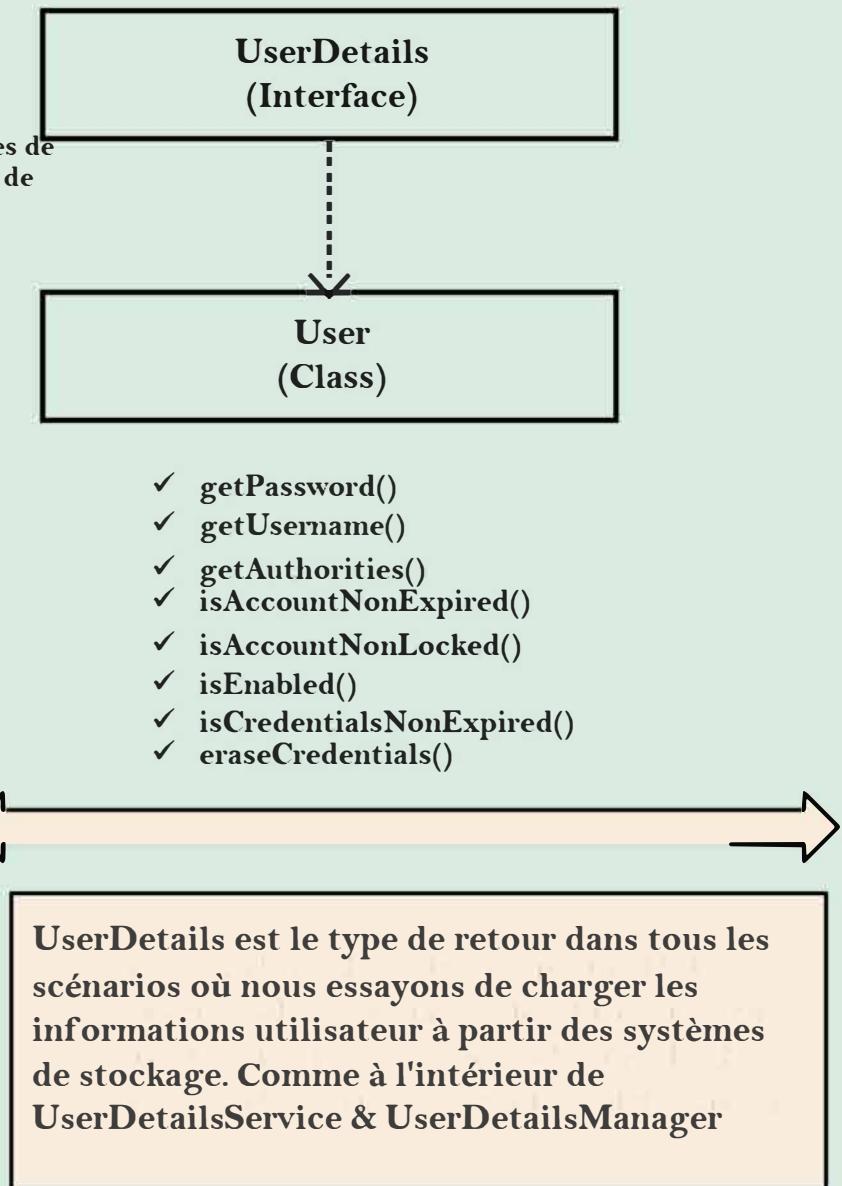
Toutes les interfaces et classes ci-dessus utilisent une interface **UserDetails** et son implémentation qui fournit des informations utilisateur de base.

USERDETAILS & AUTHENTICATION

RELATION ENTRE EUX

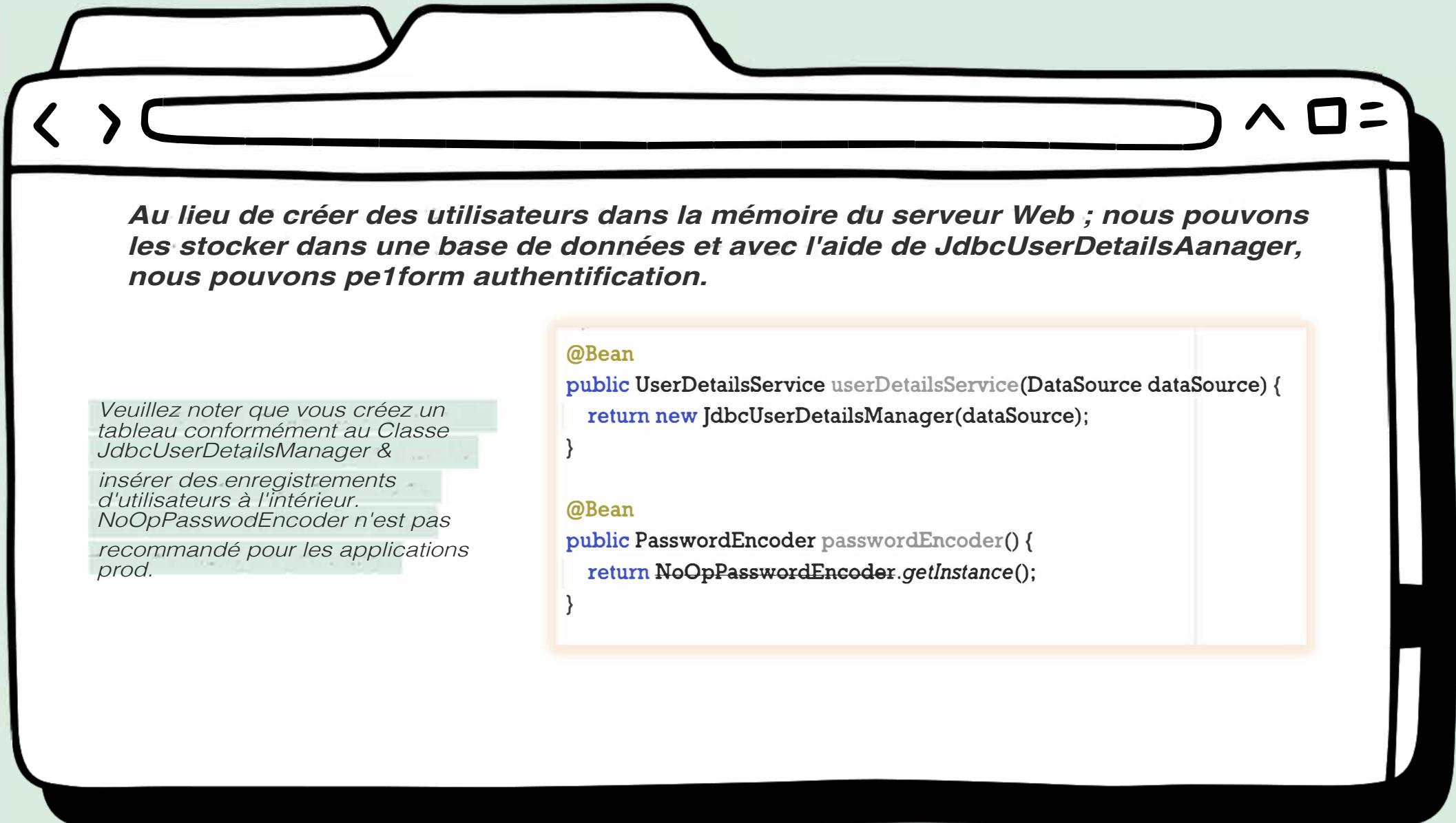


Pourquoi avons-nous 2 façons distinctes de stocker les informations de connexion de l'utilisateur ?



AUTHENTICATION

USING *JdbcUserDetailsManager*



USERDETAILS MISE EN ŒUVRE DU SERVICE

POUR UNE LOGIQUE DE RÉCUPÉRATION UTILISATEUR PERSONNALISÉE

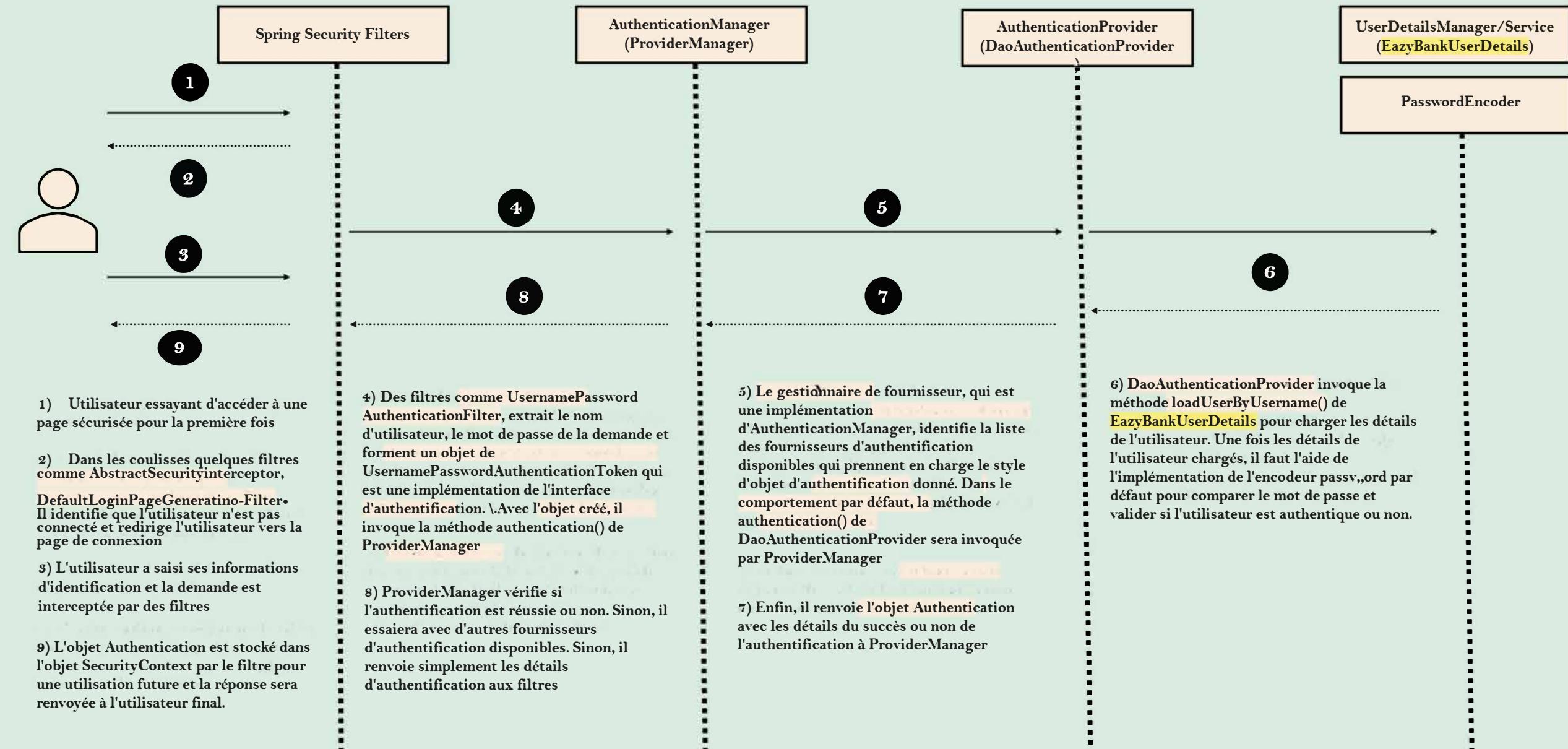
```
@Service
public class EazyBankUserDetails implements UserDetailsService {

    1 usage
    @Autowired
    private CustomerRepository customerRepository;

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        List<Customer> customer = customerRepository.findByEmail(username);
        if (customer.size() == 0) {
            throw new UsernameNotFoundException("User details not found for the user : " + username);
        }
        return new SecurityCustomer(customer.get(0));
    }
}
```

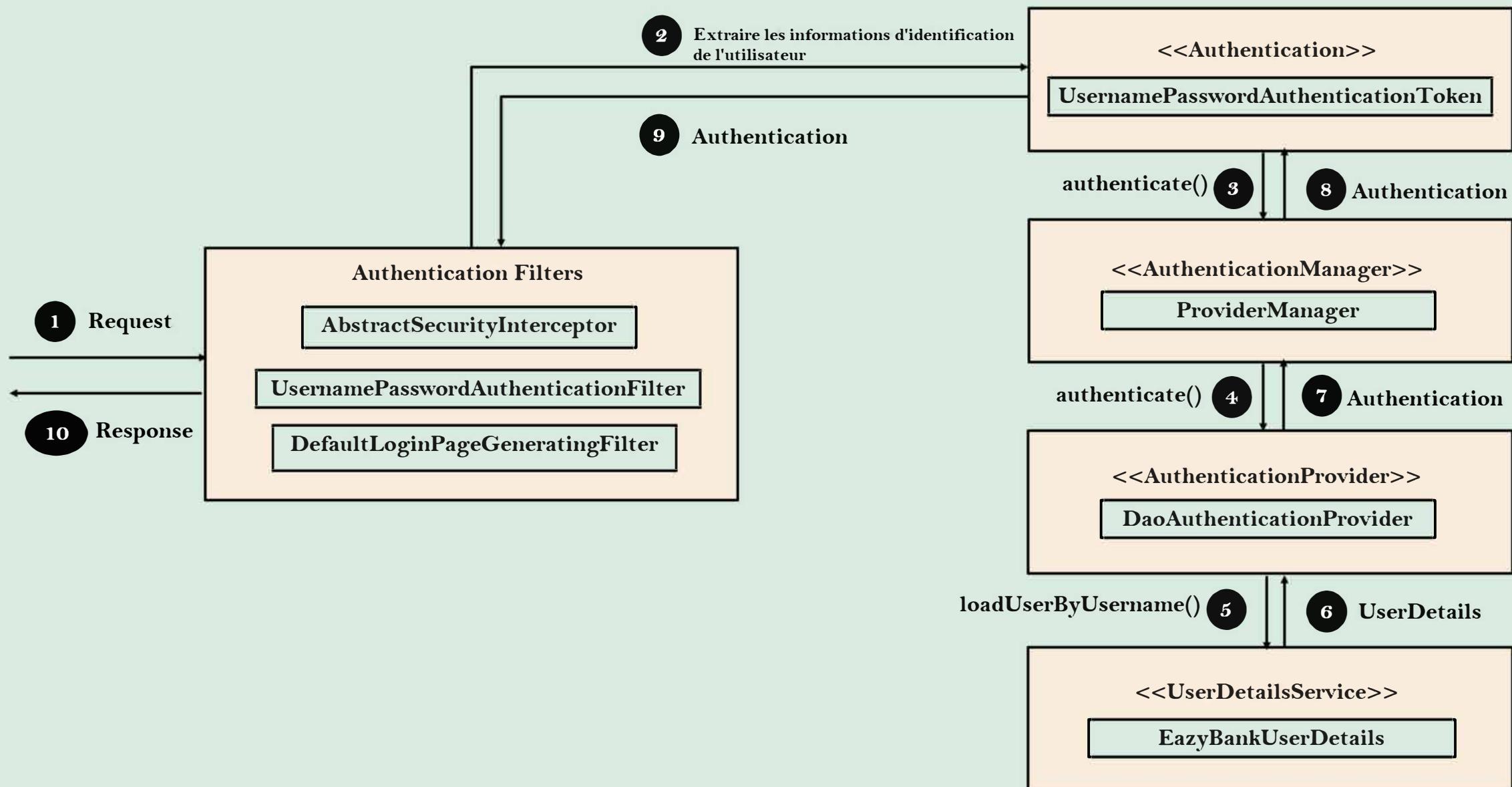
FLUX DE SÉQUENCE

AVEC LA MISE EN ŒUVRE DE NOTRE PROPRE USERDETAILSSERVICE



FLUX DE SÉQUENCE

AVEC L'AMISE EN ŒUVRE DE NOTRE PROPRE USERDETAILSSERVICE



COMMENT LES MOTS DE PASSE SONT VALIDÉS

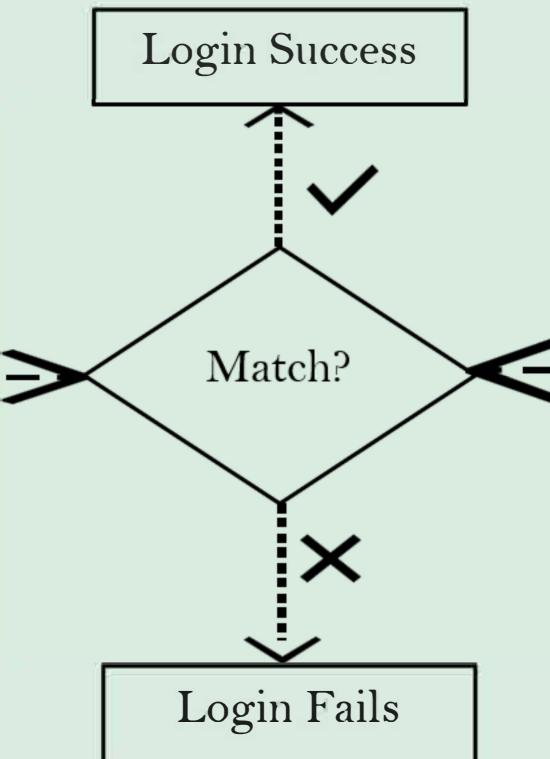
Avec PasswordEncoder par défaut

NOT RECOMMENDED
FOR PRODUCTION

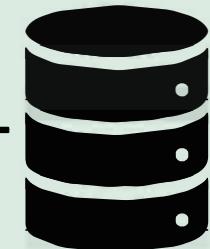
Identifiants saisis par l'utilisateur

Username	Admin
Password	12345

LOGIN



Récupérer les détails du mot de passe à partir de la base de données



Database

Le stockage des mots de passe en texte brut dans un système de stockage tel que DB posera des problèmes d'intégrité et de confidentialité. Ce n'est donc pas une approche recommandée pour les applications de production.

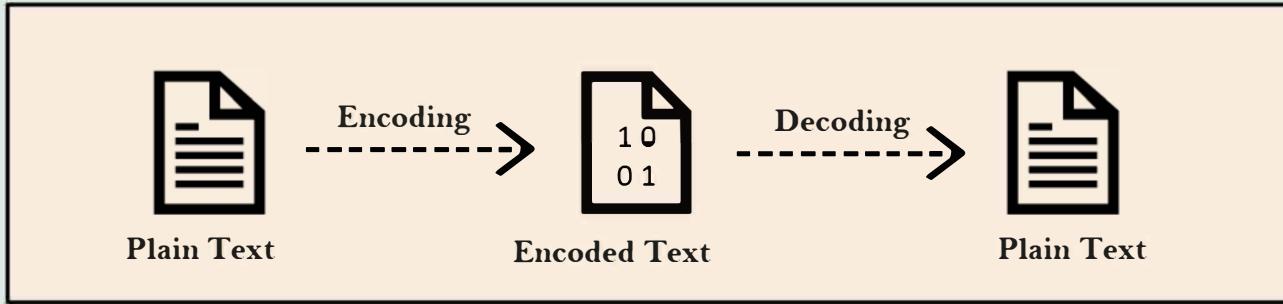
Encoding Vs Encryption Vs Hashing

Différentes manières de gérer les Pwd

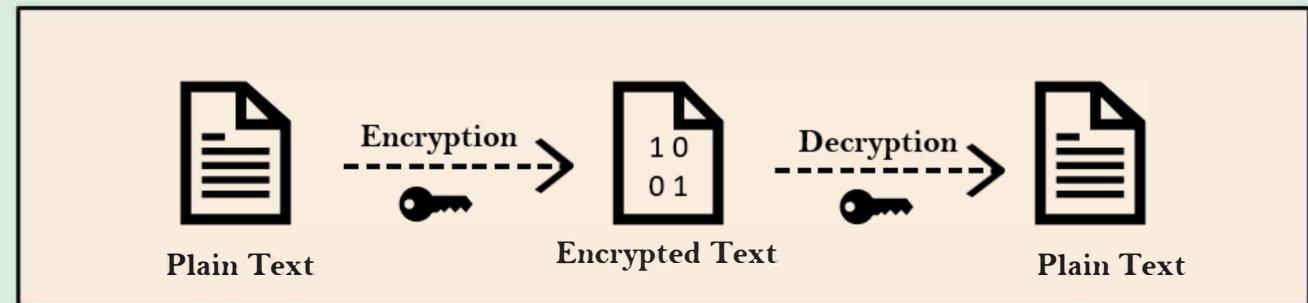
Encoding	Encryption	Hashing
<ul style="list-style-type: none">✓ L'encodage est défini comme le processus de conversion des données d'une forme à une autre et n'a rien à voir avec la cryptographie.✓ Cela n'implique aucun secret et complètement réversible.✓ Le codage ne peut pas être utilisé pour sécuriser les données. Vous trouverez ci-dessous les différents algorithmes accessibles au public utilisés pour l'encodage.	<ul style="list-style-type: none">✓ Le chiffrement est défini comme le processus de transformation des données de manière à garantir la confidentialité.✓ Pour assurer la confidentialité, le chiffrement nécessite l'utilisation d'un secret qui, en termes cryptographiques, on appelle une « clé ».✓ Le chiffrement peut être réversible en utilisant le déchiffrement à l'aide de la "clé". Tant que la "clé" est confidentielle, le chiffrement peut être considéré comme sécurisé.	<ul style="list-style-type: none">✓ Dans le hachage, les données sont converties en valeur de hachage à l'aide d'une fonction de hachage.✓ Les données une fois hachées sont irréversibles. On ne peut pas déterminer les données d'origine à partir d'une valeur de hachage générée.✓ Étant donné certaines données arbitraires ainsi que la sortie d'un algorithme de hachage, on peut vérifier si ces données correspondent aux données d'entrée d'origine sans avoir besoin de voir les données d'origine
Ex: ASCII, BASE64, UNICODE		

Encoding Vs Encryption Vs Hashing

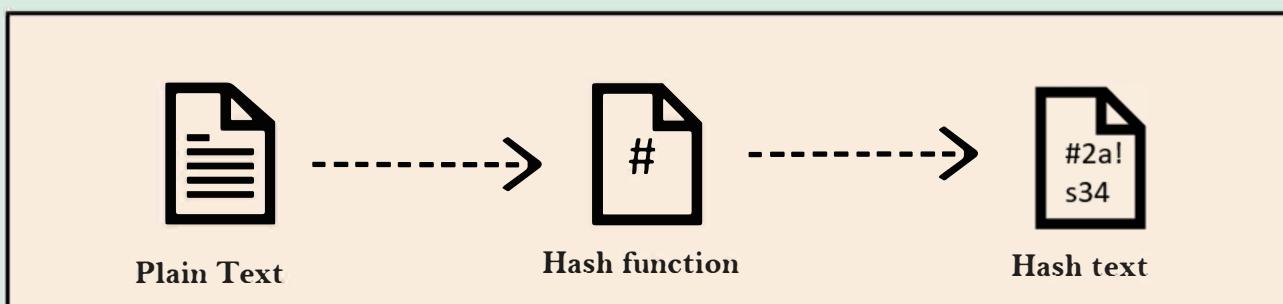
Encoding & Decoding



Encryption & Decryption



Hashing (Only 1-way)



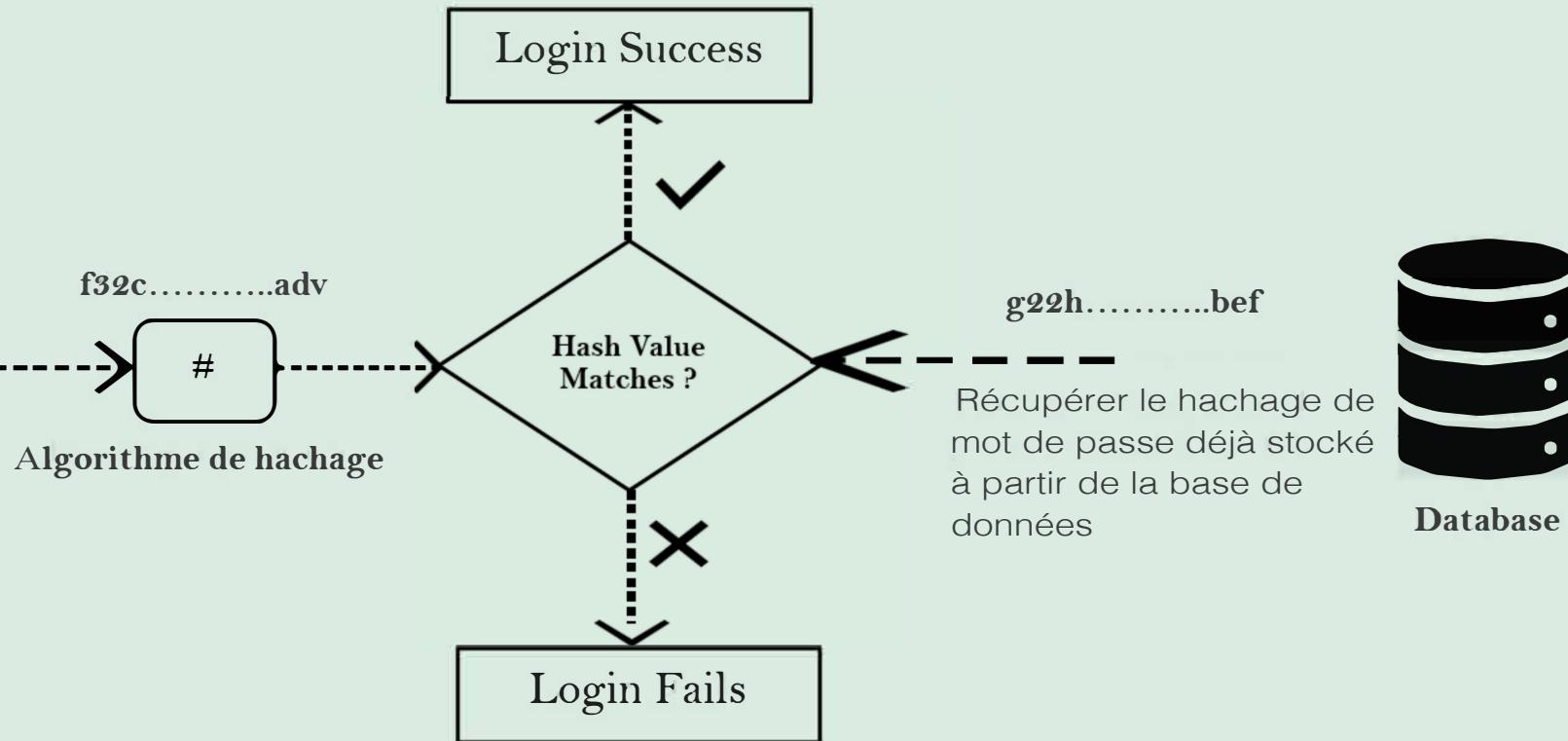
COMMENT LES MOTS DE PASSE SONT VALIDÉS

Avec hachage et codeurs de mot de passe

Identifiants saisis par l'utilisateur

Username	Admin
Password	12345

LOGIN



Le stockage et la gestion des mots de passe avec hachage est l'approche recommandée pour les applications de production.
Avec divers PasswordEncoders disponibles dans Spring Security, cela nous facilite la vie.

DETAILS DU PASSWORDENCODER

Methods inside PasswordEncoder Interface

```
public interface PasswordEncoder {  
  
    String encode(CharSequence rawPassword);  
  
    boolean matches(CharSequence rawPassword, String encodedPassword);  
  
    default boolean upgradeEncoding(String encodedPassword) {  
        return false;  
    }  
}
```

Different implementations of PasswordEncoder inside Spring Security

★ **NoOpPasswordEncoder** (*Not recommended for Prod apps*)

★ **StandardPasswordEncoder** (*Not recommended for Prod apps*)

★ **Pbkdf2PasswordEncoder**

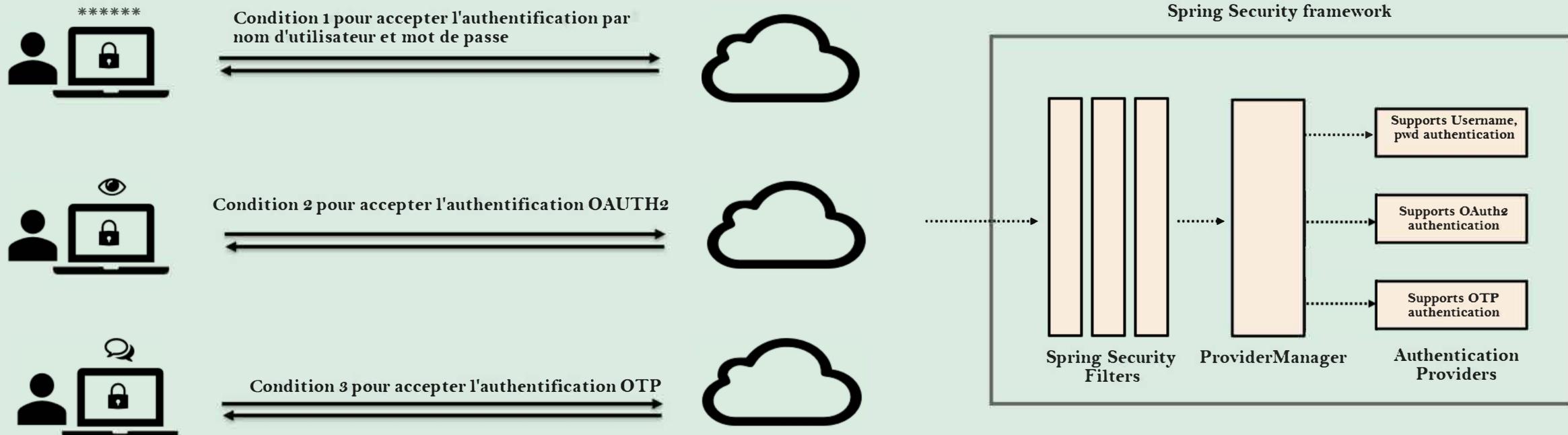
★ **BCryptPasswordEncoder**

★ **SCryptPasswordEncoder**

★ **Argon2PasswordEncoder**

AUTHENTICATION PROVIDER

POURQUOI EN AVONS-NOUS BESOIN ?



- ✓ Le AuthenticationProvider dans Spring Security prend en charge la logique d'authentification. L'implémentation par défaut de AuthenticationProvider consiste à déléguer la responsabilité de trouver l'utilisateur dans le système à une implémentation UserDetailsService & PasswordEncoder pour la validation du mot de passe. Mais si nous avons une exigence d'authentification personnalisée qui n'est pas couverte par le framework Spring Security, nous pouvons créer notre propre logique d'authentification en implémentant l'interface AuthenticationProvider.
- ✓ C'est la responsabilité du ProviderManager qui est une implémentation de AuthenticationManager ; pour vérifier avec toutes les implémentations des fournisseurs d'authentification et essayer d'authentifier l'utilisateur.

DETAILS OF AUTHENTICATION PROVIDER

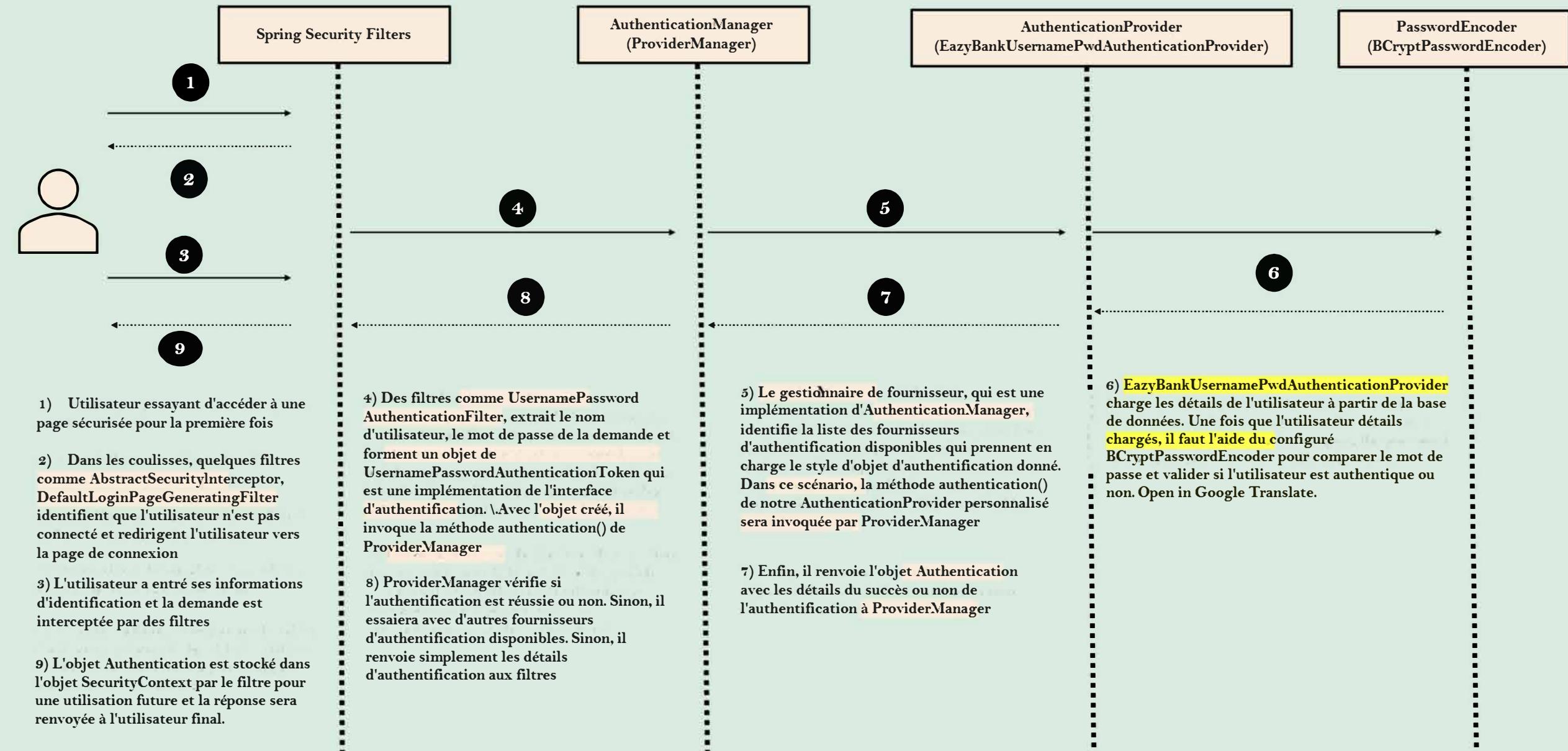
Méthodes à l'intérieur de l'interface AuthenticationProvider

```
public interface AuthenticationProvider {  
  
    Authentication authenticate(Authentication authentication)  
        throws AuthenticationException;  
  
    boolean supports(Class<?> authentication);  
}
```

- ★ La méthode authentication() reçoit et renvoie l'objet d'authentification. Nous pouvons implémenter toute notre logique d'authentification personnalisée dans la méthode authentication().
- ★ La deuxième méthode de l'interface AuthenticationProvider prend en charge (authentification de classe<?>). Vous allez implémenter cette méthode pour retourner true si le AuthenticationProvider actuel prend en charge le type de l'objet Authentication fourni.

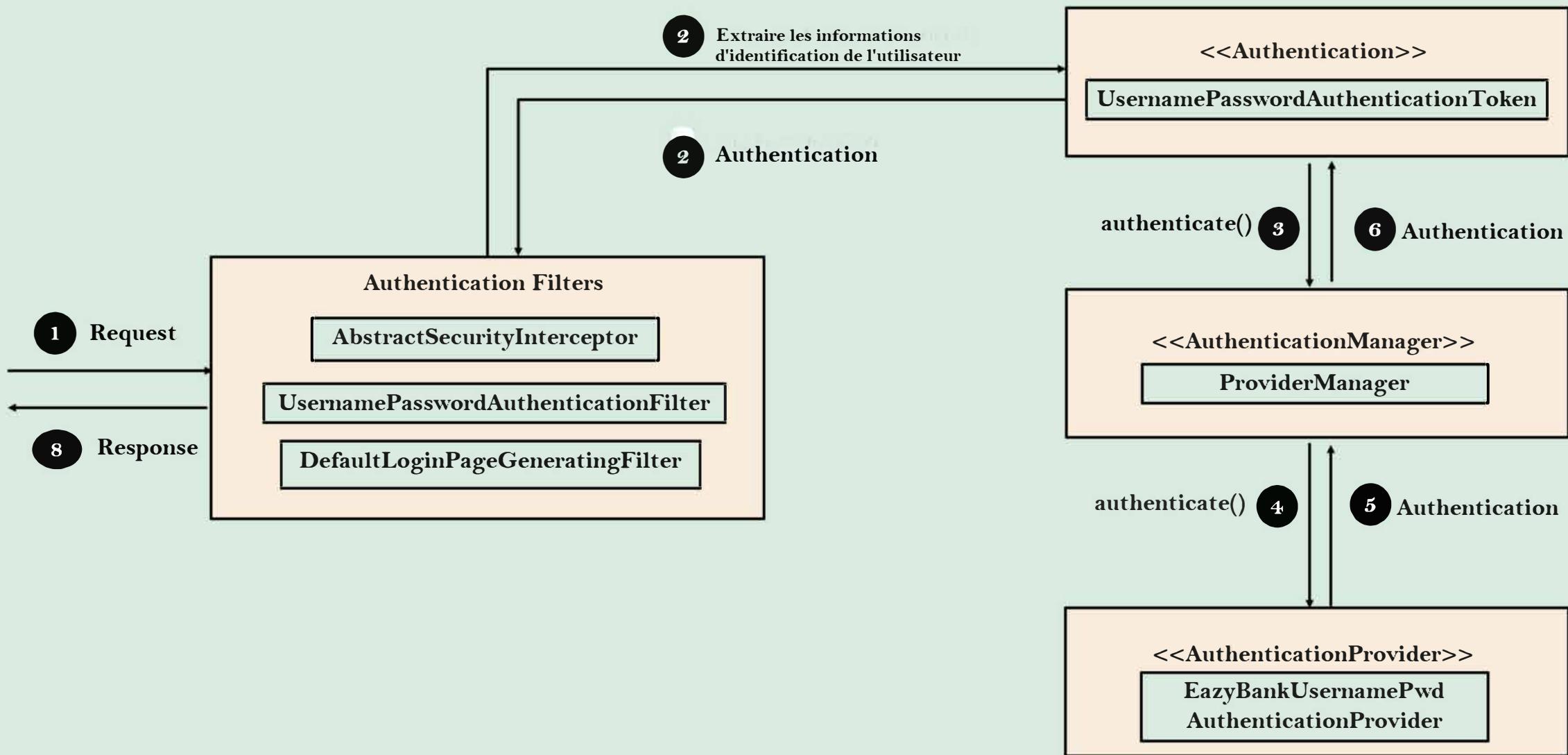
FLUX DE SÉQUENCE

AVEC NOTRE PROPRE IMPLEMENTATION DE FOURNISSEUR D'AUTHENTIFICATION



FLUX DE SÉQUENCE

AVEC NOTRE PROPRE IMPLEMENTATION DE FOURNISSEUR D'AUTHENTIFICATION



CORS & CSRF



CROSS-ORIGIN RESOURCE SHARING (CORS)

CORS est un protocole qui permet aux scripts s'exécutant sur un client de navigateur d'interagir avec des ressources d'une origine différente. Par exemple, si une application d'interface utilisateur souhaite effectuer un appel d'API exécuté sur un domaine différent, elle ne pourra pas le faire par défaut en raison de CORS. Il s'agit d'une spécification de WsC implémentée par la plupart des navigateurs.

CORS n'est donc pas un problème/attaque de sécurité mais la protection par défaut fournie par les navigateurs pour arrêter le partage des données/communications entre différentes origines.

"autres origines" signifie que l'URL consultée diffère de l'emplacement à partir duquel le JavaScript s'exécute, en ayant :

- un schéma différent (HTTP or HTTPS)
- un domaine différent
- un port différent



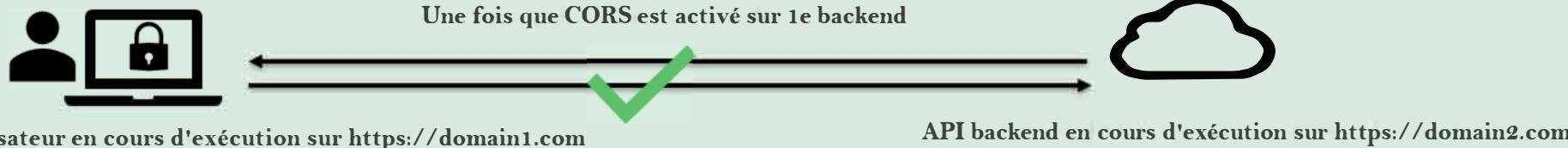
SOLUTION POUR GÉRER CORS

Si nous avons un scénario valide, où une APP UI Web déployée sur un serveur tente de communiquer avec un service REST déployé sur un autre serveur, nous pouvons autoriser ce type de communications à l'aide de l'annotation `@CrossOrigin`.
`@CrossOrigin` permet aux clients de n'importe quel domaine de consommer l'API.

L'annotation `@CrossOrigin` peut être mentionnée au-dessus d'une classe ou d'une méthode comme mentionné ci-dessous,

`@CrossOrigin(origins = "http://localhost:4200") // Autorisera sur le domaine spécifié`

`@CrossOrigin(origins = "*") // Autorisera n'importe quel domaine`



SOLUTION POUR GÉRER CORS

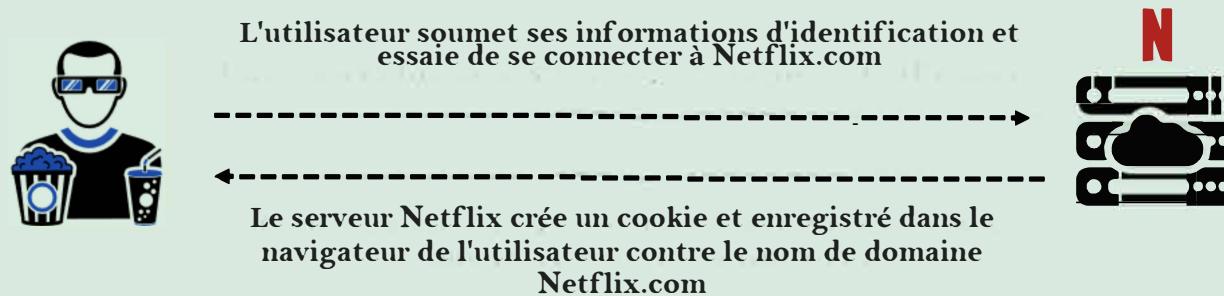
Au lieu de mentionner l'annotation `@CrossOrigin` sur tous les contrôleurs de notre application Web, nous pouvons définir globalement les configurations liées à CORS à l'aide de Spring Security, comme indiqué ci-dessous.

```
@Bean
SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {
    http.securityContext().requireExplicitSave(false)
        .and().cors().configurationSource(new CorsConfigurationSource() {
            @Override
            public CorsConfiguration getCorsConfiguration(HttpServletRequest request) {
                CorsConfiguration config = new CorsConfiguration();
                config.setAllowedOrigins(Collections.singletonList("http://localhost:4200"));
                config.setAllowedMethods(Collections.singletonList("*"));
                config.setAllowCredentials(true);
                config.setAllowedHeaders(Collections.singletonList("*"));
                config.setMaxAge(3600L);
                return config;
            }
        }).and().authorizeHttpRequests()
            .requestMatchers("/myAccount","/myBalance","/myLoans","/myCards", "/user").authenticated()
            .requestMatchers("/notices","/contact","/register").permitAll()
        .and().formLogin()
        .and().httpBasic();
    return http.build();
}
```

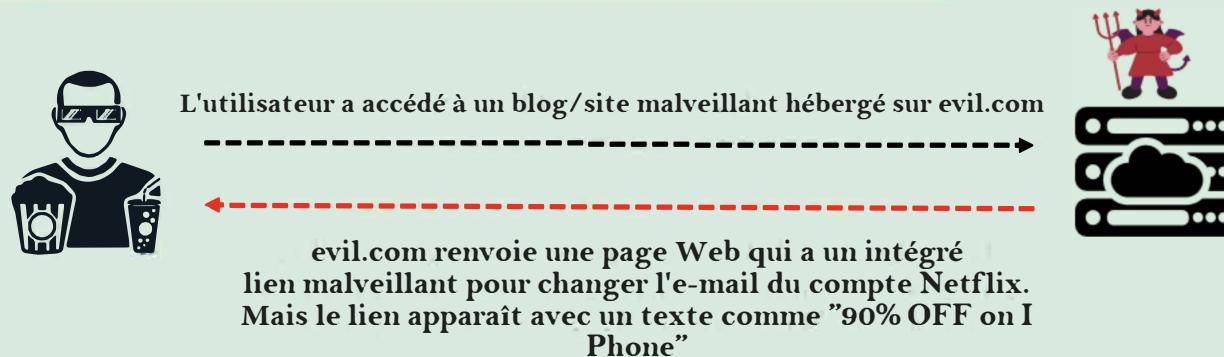
CROSS-SITE REQUEST FORGERY (CSRF)

- Une attaque typique Cross-Site Request Forgery (CSRF ou XSRF) vise à effectuer une opération dans une application Web au nom d'un utilisateur sans son consentement explicite. En général, il ne vole pas directement l'identité de l'utilisateur, mais il exploite l'utilisateur pour effectuer une action sans sa volonté.
- Considérez que vous utilisez un site Web `netflix.com` et le site Web de l'attaquant `evil.com`.

Step 1 : L'utilisateur de Netflix se connectant à Netflix.com et le serveur principal de Netflix fournira un cookie qui sera stocké dans le navigateur contre le nom de domaine Netflix.com

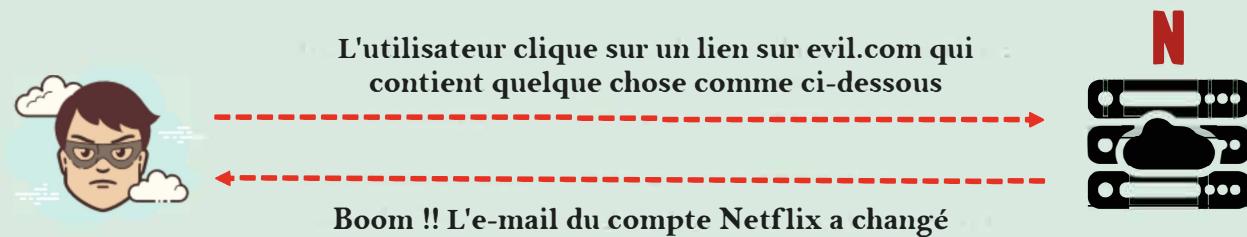


Step 2 : Le même utilisateur de Netflix ouvre un site Web evil.com dans un autre onglet du navigateur :



CROSS-SITE REQUEST FORGERY (CSRF)

Step 3 : Utilisateur tenté et cliqué sur le lien malveillant qui fait un l'equest à Netflix.com. Et puisque le cookie de connexion déjà présent dans la même navigation et la demande de changement d'e-mail est adressée au même domaine Netflix.com, le serveur principal du même site et la demande de changement d'e-mail est adressée au même domaine Netflix.com, le serveur principal de Netflix.com ne peut pas différencier d'où vient la demande. Donc, ici, evil.com a falsifié la demande comme si elle provenait d'une page d'interface utilisateur Netflix.com.



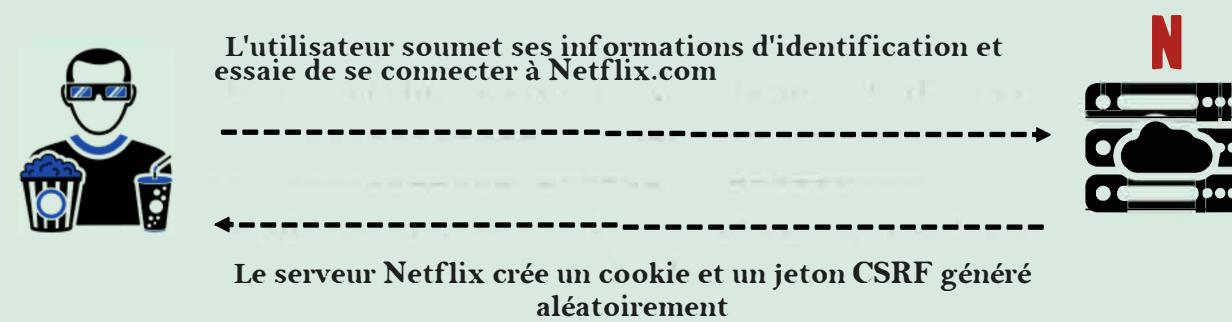
```
<form action="https://netflix.com/changeEmail"
      method="POST" id="form">
    <input type="hidden" name="email" value="user@evil.com">
</form>

<script>
  document.getElementById('form').submit()
</script>
```

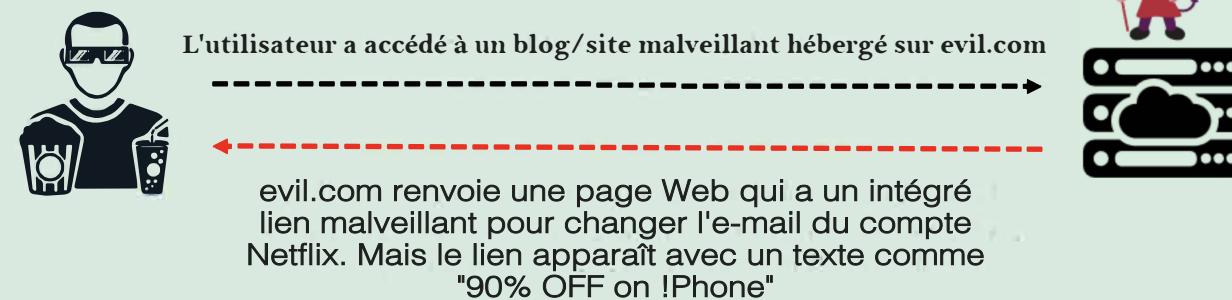
SOLUTION À L'ATTAQUE CSRF

- Pour vaincre une attaque CSRF, les applications ont besoin d'un moyen de déterminer si la requête HTTP est légitimement générée via l'interface utilisateur des applications. La meilleure façon d'y parvenir est d'utiliser un jeton CSRF. Un jeton CSRF est un jeton aléatoire sécurisé utilisé pour empêcher les attaques CSRF. Le jeton doit être unique par session utilisateur et doit avoir une grande valeur aléatoire pour le rendre difficile à deviner.
- Voyons comment cela résout l'attaque CSRF en reprenant l'exemple précédent de Netflix,

Step 1 : L'utilisateur de Netflix se connecte à Netflix.com et le serveur principal de Netflix fournira un cookie qui sera stocké dans le navigateur contre le nom de domaine Netflix.com avec un jeton CSRF unique généré de manière aléatoire pour cette session utilisateur particulière. Le jeton CSRF est inséré dans les paramètres cachés des formulaires HTML pour éviter l'exposition aux cookies de session.

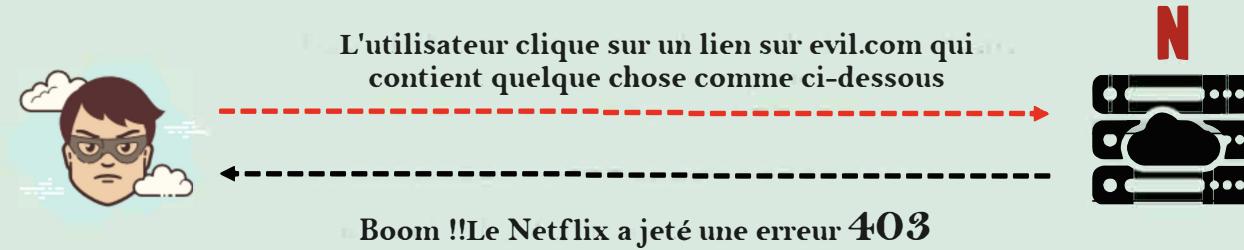


Step 2 : Le même utilisateur de Netflix ouvre un site Web evil.com dans un autre onglet du navigateur



SOLUTION À L'ATTAQUE CSRF

Step 3 : L'utilisateur a tenté et cliqué sur le lien malveillant qui fait une demande à Netflix.com. Et puisque le cookie de connexion déjà présent dans la même navigation et la demande de changement d'adresse e-mail est envoyée au même domaine Netflix.com. Cette fois, le serveur principal de Netflix.com attend Jeton CSRF avec le cookie. Le jeton CSRF doit être identique à la valeur initiale générée lors de l'opération de connexion



Le jeton CSRF sera utilisé par le serveur d'application pour vérifier la légitimité de la demande de l'utilisateur final si elle provient de la même interface utilisateur d'application ou non. Le serveur d'application rejette la demande si le jeton CSRF ne correspond pas au test.

DÉSACTIVER LA PROTECTION CSRF

A L'INTERIEUR DE SPRING SECURITY

NOT RECOMMENDED
FOR PRODUCTION

Par défaut, Spring Security bloque toutes les opérations HTTP POST, PUT, DELETE, PATCH avec une erreur de 403, s'il n'y a pas de solution CSRF implémentée dans une application Web. Nous pouvons modifier ce comportement par défaut en désactivant la protection CSRF fournie par Spring Security.

```
@Bean
SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {
    http.csrf().disable()
        .securityContext().requireExplicitSave(false).and()
        .authorizeHttpRequests()
            .requestMatchers("/myAccount","/myBalance","/myLoans","/myCards", "/user").authenticated()
            .requestMatchers("/notices","/contact","/register").permitAll()
        .and().formLogin()
        .and().httpBasic();
    return http.build();
}
```

SOLUTION D'ATTAQUE CSRF

A L'INTERIEUR DE SPRING SECURITY

Avec la configuration ci-dessous de Spring Security, nous pouvons laisser le cadre générer un jeton aléatoire CSRF qui peut être envoyé à l'interface utilisateur après une connexion réussie. Le même jeton doit être envoyée par l'interface utilisateur pour toutes les demandes ultérieures qu'elle adresse au backend. Pour certains chemins, nous pouvons désactiver CSRF avec l'aide d'ignoringAntMatchers.

```
@Bean  
SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {  
    http.securityContext().requireExplicitSave(false).and()  
        .csrf().ignoringRequestMatchers("/contact", "/register")  
        .csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse())  
        .and().authorizeHttpRequests()  
        .requestMatchers("/myAccount", "/myBalance", "/myLoans", "/myCards", "/user").authenticated()  
            .requestMatchers("/notices", "/contact", "/register").permitAll()  
        .and().formLogin()  
        .and().httpBasic();  
    return http.build();  
}
```

SOLUTION D'ATTAQUE CSRF

A L'INTERIEUR DE SPRING SECURITY

Avec la configuration ci-dessous de Spring Security, nous pouvons laisser le travail de base de génération d'un jeton CSRF aléatoire qui peut être envoyé à l'interface utilisateur après une connexion réussie. La même prise doit être envoyée par l'interface utilisateur pour toutes les demandes ultérieures qu'elle adresse au backend. Pour certains chemins, nous pouvons désactiver CSRF avec l'aide d'ignoringAntMatchers.

```
@Bean  
SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {  
    http.securityContext().requireExplicitSave(false).and()  
        .csrf().ignoringRequestMatchers("/contact", "/register")  
        .csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse())  
        .and().authorizeHttpRequests()  
        .requestMatchers("/myAccount", "/myBalance", "/myLoans", "/myCards", "/user").authenticated()  
            .requestMatchers("/notices", "/contact", "/register").permitAll()  
        .and().formLogin()  
        .and().httpBasic();  
    return http.build();  
}
```

COMMENT LES AUTORITÉS STOCKÉES ?

A L'INTERIEUR DE SPRING SECURITY

- Les informations sur les autorités/rôles dans Spring Security sont stockées dans GrantedAuthority. Il n'y a qu'une seule méthode dans GrantedAuthority qui renvoie le nom de l'autorité ou du rôle.
- SimpleGrantedAuthority est la classe d'implémentation par défaut de l'interface GrantedAuthority dans le framework Spring Security.

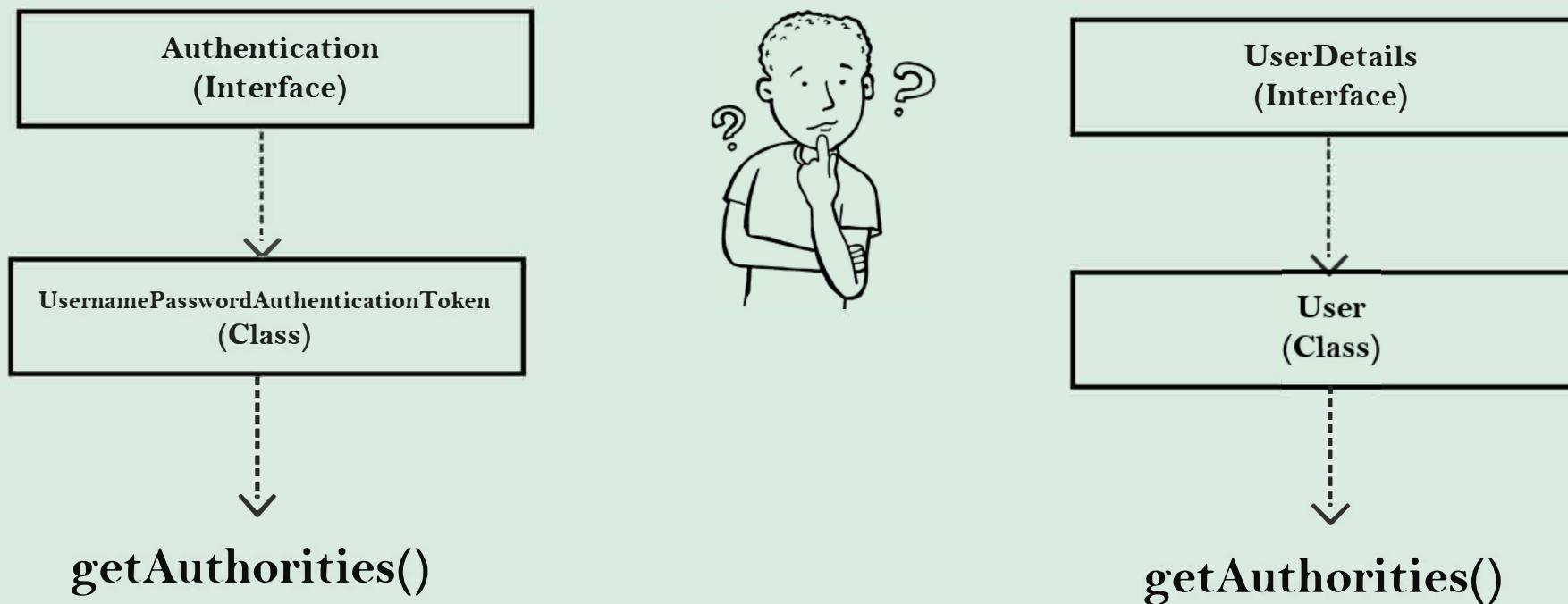
```
public interface GrantedAuthority {  
  
    String getAuthority();  
  
}
```

```
public final class SimpleGrantedAuthority implements GrantedAuthority {  
  
    2 usages  
    private final String role;  
  
    public SimpleGrantedAuthority(String role) {  
        this.role = role;  
    }  
  
    @Override  
    public String getAuthority() {  
        return this.role;  
    }  
}
```

COMMENT LES AUTORITÉS STOCKÉES ?

A L'INTERIEUR DE SPRING SECURITY

Comment les informations des Autorités sont-elles stockées dans les objets des interfaces UserDetails & Authentication qui jouent un rôle vital dans l'authentification de l'utilisateur ?



CONFIGURATION DES AUTORITÉS

A L'INTERIEUR DE SPRING SECURITY



Dans Spring Security, les exigences des autorités peuvent être configurées de la manière suivante,

hasAuthority() — accepte une seule autorité pour laquelle le terminal sera configuré et l'utilisateur sera validé par rapport à l'autorité unique mentionnée. Seuls les utilisateurs ayant la même autorité configurée peuvent appeler le point de terminaison.

hasAnyAuthority() — accepte plusieurs autorités pour lesquelles le point de terminaison sera configuré et l'utilisateur sera validé par rapport aux autorités mentionnées. Seuls les utilisateurs disposant de l'une des autorisations configurées peuvent appeler le point de terminaison.

access() — En utilisant Spring Expression Language (**SpEL**), il vous offre des possibilités illimitées pour configurer les autorités qui ne sont pas possibles avec les méthodes ci-dessus. Nous pouvons utiliser des opérateurs comme "OR", "AND" dans la méthode `access()`.

CONFIGURATION DES AUTORITÉS

A L'INTERIEUR DE SPRING SECURITY

Comme indiqué ci-dessous, nous pouvons configurer les exigences d'autorité pour les API/chemins.

```
@Bean  
SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {  
    http.securityContext().requireExplicitSave(false)  
        .and().authorizeHttpRequests()  
            .requestMatchers("/myAccount").hasAuthority("VIEWACCOUNT")  
            .requestMatchers("/myBalance").hasAnyAuthority("VIEWACCOUNT", "VIEWBALANCE")  
            .requestMatchers("/myLoans").hasAuthority("VIEWLOANS")  
            .requestMatchers("/myCards").hasAuthority("VIEWCARDS")  
            .requestMatchers("/user").authenticated()  
            .requestMatchers("/notices", "/contact", "/register").permitAll()  
        .and().formLogin()  
        .and().httpBasic();  
    return http.build();  
}
```

AUTHORITY vs ROLE

A L'INTERIEUR DE SPRING SECURITY



- Les noms des autorités/rôles sont de nature arbitraire et ces noms peuvent être personnalisés selon les besoins de l'entreprise
- Les rôles sont également représentés à l'aide du même contrat GrantedAuthority dans Spring Security.
- Lors de la définition d'un rôle, son nom doit commencer par le préfixe ROLE_. Ce préfixe précise la différence entre un rôle et une autorité.

CONFIGURATION DES AUTORITÉS

A L'INTERIEUR DE SPRING SECURITY



Dans Spring Security, les exigences ROLES peuvent être configurées de la manière suivante,

hasRole() — Accepte un nom de rôle unique pour lequel le point de terminaison sera configuré et l'utilisateur sera validé par rapport au rôle unique mentionné. Seuls les utilisateurs ayant le même rôle configuré peuvent appeler le point de terminaison.

hasAnyRole() — Accepte plusieurs rôles pour lesquels le point de terminaison sera configuré et l'utilisateur sera validé par rapport aux rôles mentionnés. Seuls les utilisateurs ayant l'un des rôles configurés peuvent appeler le point de terminaison.

access() — En utilisant Spring Expression Language (SpEL), il vous offre des possibilités illimitées pour configurer des rôles qui ne sont pas possibles avec les méthodes ci-dessus. Nous pouvons utiliser des opérateurs comme OR, AND dans la méthode access().

Note :

- Préfixe ROLE_ uniquement à utiliser lors de la configuration du rôle dans DB. Mais lorsque nous configurons les rôles, nous le faisons uniquement par son nom.
- La méthode access () peut être utilisée non seulement pour configurer l'autorisation en fonction de l'autorité ou du rôle, mais également avec toutes les exigences particulières que nous avons. Par exemple, nous pouvons configurer l'accès en fonction du pays de l'utilisateur ou de l'heure/date actuelle.

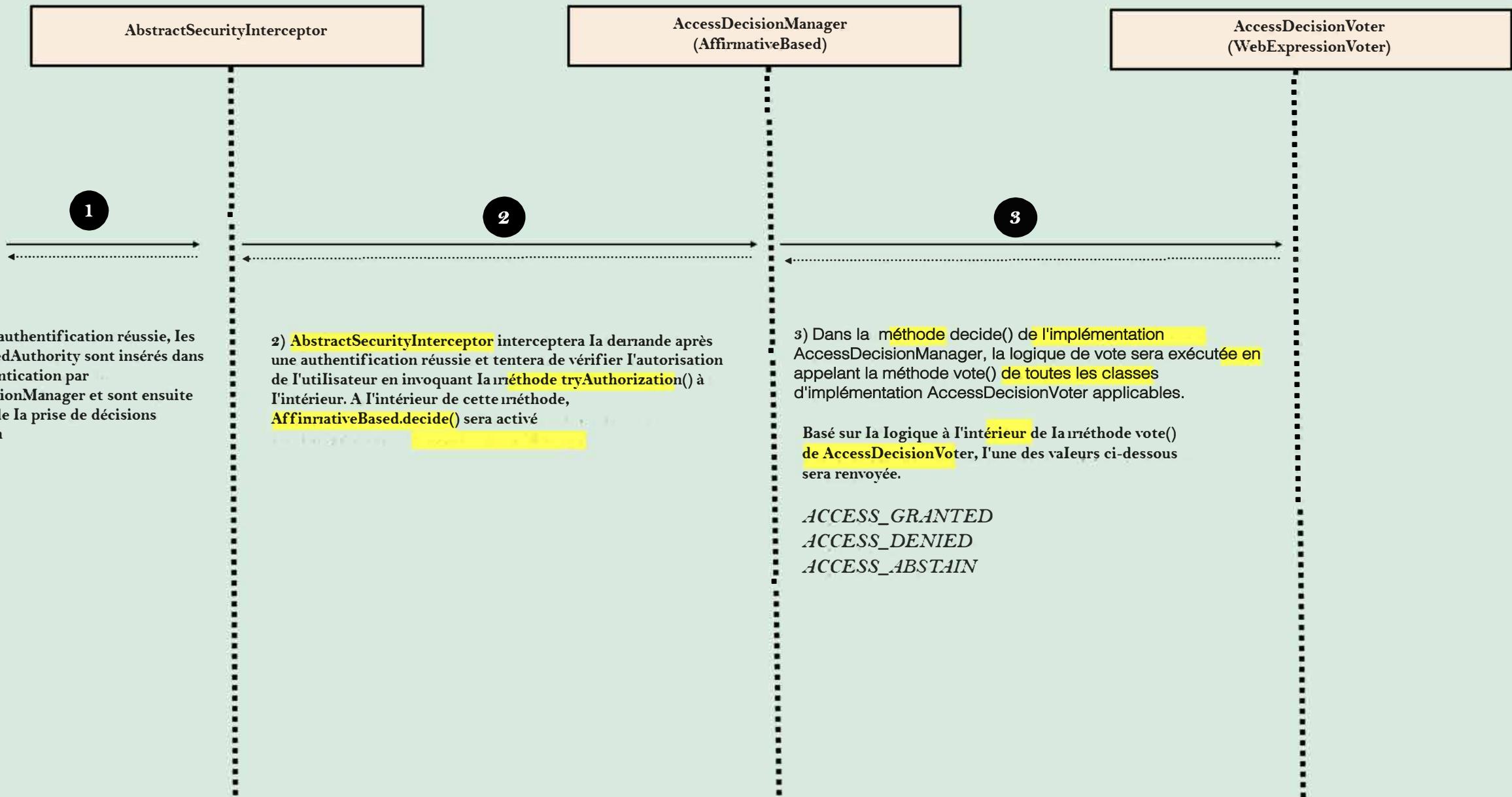
CONFIGURATION DES RÔLES

AL'INTERIEUR DE SPRING SECURITY

Comme indiqué ci-dessous, nous pouvons configurer les exigences ROLES pour les API/Paths.

```
@Bean  
SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {  
    http.securityContext().requireExplicitSave(false)  
        .and().authorizeHttpRequests()  
            .requestMatchers("/myAccount").hasRole("USER")  
            .requestMatchers("/myBalance").hasAnyRole("USER", "ADMIN")  
            .requestMatchers("/myLoans").hasRole("USER")  
            .requestMatchers("/myCards").hasRole("USER")  
            .requestMatchers("/user").authenticated()  
            .requestMatchers("/notices", "/contact", "/register").permitAll()  
        .and().formLogin()  
        .and().httpBasic();  
    return http.build();  
}
```

FLUX DE SÉQUENCE DE L'AUTORISATION AVEC DES ÉTAPES IMPORTANTES



LES FILTRES AVEC SPRING SECURITY

- ✓ Souvent, nous aurons des situations où nous devrons effectuer des activités de maintenance interne pendant le flux d'authentification et d'autorisation. Peu d'exemples de ce genre sont,
 - Validation des entrées
 - Traçage, audit et reporting
 - Journalisation des entrées comme l'adresse IP, etc.
 - Chiffrement et déchiffrement
 - Authentification multifacteur à l'aide d'OTP
- ✓ Toutes ces exigences peuvent être gérées à l'aide de filtres HTTP dans Spring Security. Les filtres sont des concepts de servlet qui sont également exploités dans Spring Security.

- ✓ Nous avons déjà vu certains filtres intégrés du framework de sécurité Spring, comme UsernamePasswordAuthenticationFilter, BasicAuthenticationFilter, DefaultLoginPageGeneratingFilter etc. dans les sections précédentes.
- ✓ Un filtre est un composant qui reçoit les requêtes, traite sa logique et passe au filtre suivant dans la chaîne
- ✓ Spring Security est basé sur une chaîne de filtres de servlets. Chaque filtre a une responsabilité spécifique et selon la configuration, des filtres sont ajoutés ou supprimés. Nous pouvons également ajouter nos filtres personnalisés en fonction des besoins.

NOT RECOMMENDED
FOR PRODUCTION

LES FILTRES AVEC SPRING SECURITY

- ✓ Nous pouvons toujours vérifier les filtres enregistrés dans Spring Security avec les configurations ci-dessous,
 1. `@EnableWebSecurity(debug = true)` – Nous devons activer le débogage des détails de sécurité
 2. Activez la journalisation des détails en ajoutant la propriété ci-dessous dans application.properties
 - `logging.level.org.springframework.security.web.FilterChainProxy=DEBUG`

Vous trouverez ci-joint certains des filtres internes de Spring Security qui sont exécutés dans le flux d'authentification,

Security filter chain: [
`DisableEncodeUrlFilter`
`WebAsyncManagerIntegrationFilter`
`SecurityContextHolderFilter`
`HeaderWriterFilter`
`CorsFilter`
`CsrfFilter`
`LogoutFilter`
`UsernamePasswordAuthenticationFilter`
`DefaultLoginPageGeneratingFilter`
`DefaultLogoutPageGeneratingFilter`
`BasicAuthenticationFilter`
`RequestCacheAwareFilter`
`SecurityContextHolderAwareRequestFilter`
`AnonymousAuthenticationFilter`
`SessionManagementFilter`
`ExceptionTranslationFilter`
`FilterSecurityInterceptor`
]

MISE EN ŒUVRE DE FILTRES PERSONNALISÉS

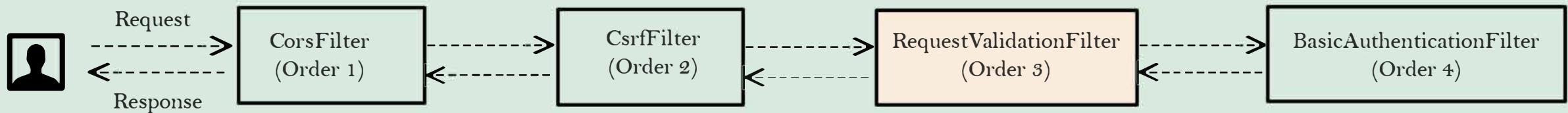
A L'INTERIEUR DE SPRING SECURITY

- ✓ Nous pouvons créer nos propres filtres en implémentant l'interface Filter du package jakarta.servlet. Postez que nous devons remplacer la méthode doFilter() pour avoir notre propre logique personnalisée. Cette méthode accepte 3 paramètres le ServletRequest, ServletResponse et FilterChain.
 - **ServletRequest**— Il représente la requête HTTP. Nous utilisons l'objet ServletRequest pour récupérer les détails de la requête du client
 - **ServletResponse**— Il représente la réponse HTTP. Nous utilisons l'objet ServletResponse pour modifier la réponse avant de la renvoyer au client ou plus loin dans la chaîne de filtrage.
 - **FilterChain**— La chaîne de filtres représente une collection de filtres avec un ordre défini dans lequel ils agissent. Nous utilisons l'objet FilterChain pour transmettre la requête au filtre suivant dans la chaîne.

- ✓ Vous pouvez ajouter un nouveau filtre à la chaîne de sécurité à ressort avant, après ou à la position d'un filtre connu. Chaque position du filtre est un index (un nombre), et vous pouvez le trouver également appelé "l'ordre".
- ✓ Vous trouverez ci-dessous les méthodes disponibles pour configurer un filtre personnalisé dans le flux de sécurité Spring.
 - **addFilterBefore(filter, class)** – ajoute un filtre avant la position de la classe de filtre spécifiée
 - **addFilterAfter(filter, class)** – ajoute un filtre après la position de la classe de filtre spécifiée
 - **addFilterAt(filter, class)** – ajoute un filtre à l'emplacement de la classe de filtre spécifiée

ADD FILTER BEFORE DANS SPRING SECURITY

`addFilterBefore(filter, class)` – Il ajoutera un filtre avant la position de la classe de filtre spécifiée.

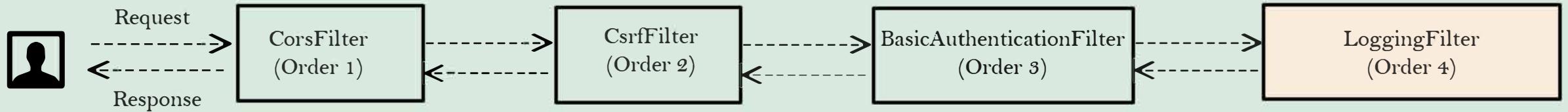


Ici, nous ajoutons un filtre juste avant l'authentification pour écrire notre propre validation personnalisée où l'e-mail d'entrée fourni ne doit pas contenir la chaîne 'test'.

ADD FILTER AFTER

DANS SPRING SECURITY

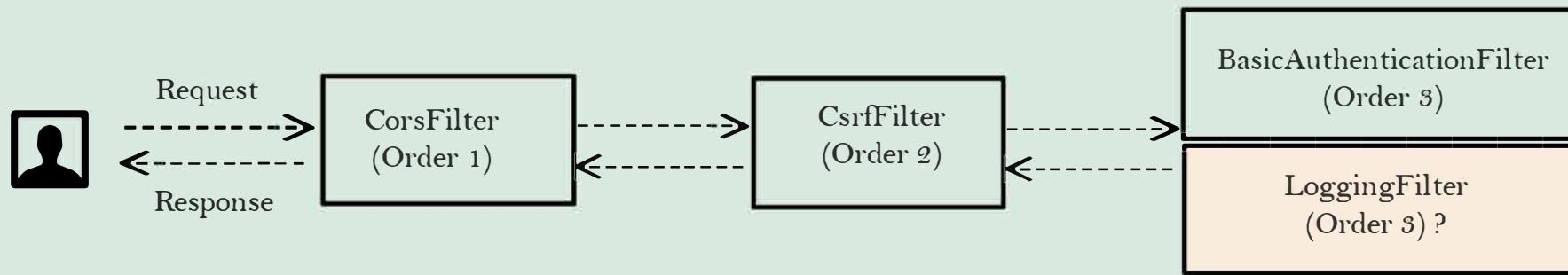
`addFilterAfter(filter, class)` – Il ajoutera un filtre après la position de la classe de filtre spécifiée



Ici, nous ajoutons un filtre juste après l'authentification pour écrire un enregistreur sur l'authentification réussie et les détails des autorités des utilisateurs connectés.

ADD FILTER AT DANS SPRING SECURITY

addFilterAt(filter, class) – Ajoute un filtre à l'emplacement de la classe de filtre spécifiée. Mais l'ordre d'exécution ne peut être garanti. Cela ne remplacera pas les filtres déjà présents à la même commande.



Étant donné que nous n'aurons aucun contrôle sur l'ordre des filtres et qu'il est de nature aléatoire, nous devrions éviter de fournir les filtres dans le même ordre.

D'AUTRES FILTRES IMPORTANTS



GenericFilterBean

Il s'agit d'un bean de filtre de classe abstrait qui vous permet d'utiliser les paramètres d'initialisation et les configurations définis dans les descripteurs de déploiement



OncePerRequestFilter

Spring ne garantit pas que votre filtre ne sera appelé qu'une seule fois. Mais si nous avons un scénario où nous devons nous assurer d'exécuter notre filtre une seule fois, nous pouvons l'utiliser.

MATCHER METHODS

DANS SPRING SECURITY

Spring Security propose trois types de méthodes de correspondance pour configurer la sécurité des terminaux,

- 1) MVC matchers, 2) Ant matchers, 3) Regex matchers



MVC matchers

mvcMatcher() utilise le HandlerMappingIntrospector de Spring MVC pour faire correspondre le chemin et extraire les variables.

mvcMatchers(HttpServletRequest method, String... patterns) - Nous pouvons spécifier à la fois la méthode HTTP et le modèle de chemin pour configurer les restrictions

```
mvcMatchers(HttpServletRequest.POST, "/example").authenticated()  
mvcMatchers(HttpServletRequest.GET, "/example").permitAll()
```

mvcMatchers(String... patterns) - Nous ne pouvons spécifier que le modèle de chemin pour configurer les restrictions et toutes les méthodes HTTP seront autorisées.

```
mvcMatchers( "/example/edit/**").authenticated()
```

Note :

- ** indique un nombre quelconque de chemins. Par exemple, /x/**/z will match both /x/y/z and /x/y/abc/z
- Simple * indique un chemin unique. Par example/x/*/z will match /x/y/z, /x/abc/z but not /x/y/abc/z

MATCHER METHODS

AVEC SPRING SECURITY



ANT matchers

`antMatchers()` est une implémentation pour les modèles de chemin de style Ant. Une partie de ce code de mappage a été aimablement empruntée à Apache Ant.

antMatchers(HttpMethod method, String... patterns) - Nous pouvons spécifier à la fois la méthode HTTP et le modèle de chemin pour configurer les restrictions

```
antMatchers(HttpMethod.POST, "/example").authenticated()
```

antMatchers(String... patterns) - nous ne pouvons spécifier que le modèle de chemin pour configurer les restrictions et toutes les méthodes HTTP seront autorisées.

```
antMatchers( "/example/edit/**").authenticated()
```

antMatchers(HttpMethod method) - Nous ne pouvons spécifier que la méthode HTTP en ignorant le modèle de chemin pour configurer les restrictions. C'est la même chose que `antMatchers(httpMethod, "/*")`

```
antMatchers(HttpMethod.POST).authenticated()
```

Note : Généralement, `mvcMatcher` est plus convivial pour les développeurs qu'un `antMatcher`. Par exemple

- `antMatchers("/secured")` correspond uniquement à l'URL exacte / sécurisée
- `mvcMatchers("/secured")` correspond à / secured ainsi qu'à / secured/, / secured.html, / secured.xyz

MATCHER METHODS

DANS SPRING SECURITY



REGEX matchers

Les Regex peuvent être utilisés pour représenter n'importe quel format d'une chaîne, ils offrent donc des possibilités illimitées.

regexMatchers(HttpMethod method, String regex) - Nous pouvons spécifier à la fois la méthode HTTP et l'expression régulière du chemin pour configurer les restrictions

```
regexMatchers(HttpMethod.GET, ".*/(en|es|zh)").authenticated()
```

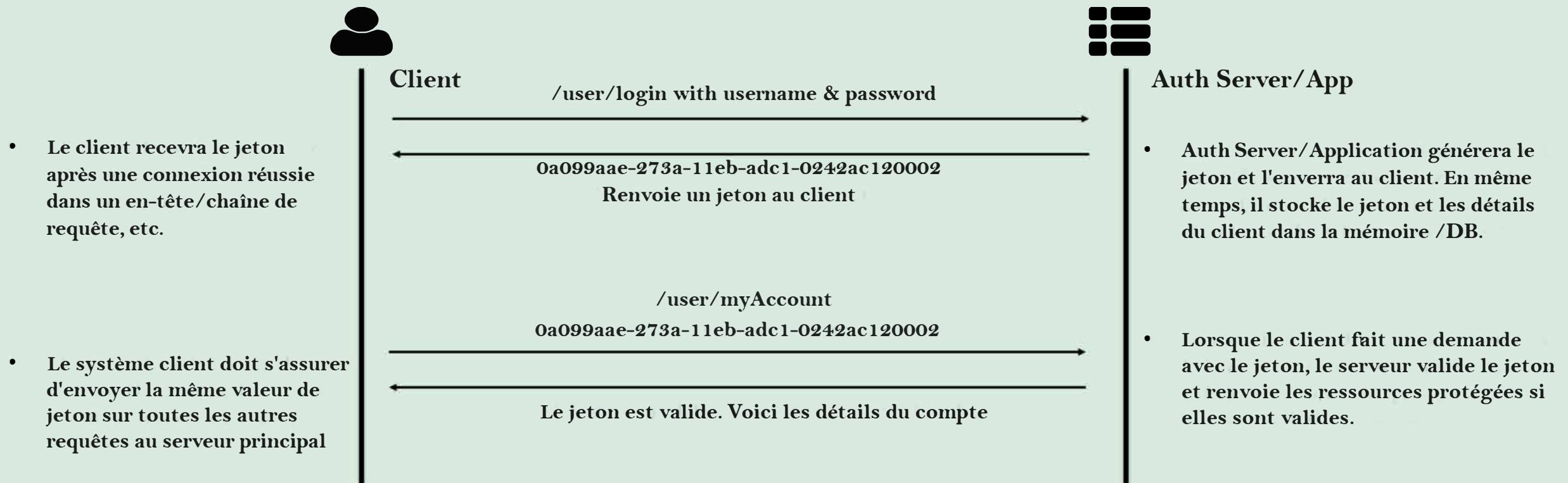
regexMatchers(String regex) - Nous ne pouvons spécifier que l'expression régulière du chemin pour configurer les restrictions et toutes les méthodes HTTP seront autorisées.

```
regexMatchers(".*/(en|es|zh)").authenticated()
```

RÔLE DES JETONS (TOKEN)

DANS AUTHN & AUTHZ

- ✓ Un jeton peut être une chaîne ordinaire au format d'identificateur unique universel (UUID) ou il peut être de type JSON Web Token (JWT), généralement généré lorsque l'utilisateur s'est authentifié pour la première fois lors de la connexion.
- ✓ À chaque demande adressée à une ressource restreinte, le client envoie le jeton d'accès dans la chaîne de requête ou l'en-tête d'autorisation. Le serveur valide ensuite le jeton et, s'il est valide, renvoie la ressource sécurisée au client.



LES AVANTAGES DES JETONS

- ★ Token nous aide à ne pas partager les informations d'identification pour chaque demande. L'envoi fréquent d'informations d'identification sur le réseau présente un risque pour la sécurité.
- ★ Les jetons peuvent être invalidés lors de toute activité suspecte sans invalider les informations d'identification de l'utilisateur.
- ★ Les jetons peuvent être créés avec une courte durée de vie.
- ★ Les jetons peuvent être utilisés pour stocker les informations relatives à l'utilisateur telles que les rôles/autorités, etc.
- ★ Réutilisabilité - Nous pouvons avoir de nombreux serveurs distincts, fonctionnant sur plusieurs plates-formes et domaines, réutilisant le même jeton pour authentifier l'utilisateur.
- ★ Sans état, plus facile à mettre à l'échelle. Le jeton contient toutes les informations permettant d'identifier l'utilisateur, éliminant ainsi le besoin de connaître l'état de la session. Si nous utilisons un équilibrEUR de charge ; nous pouvons transmettre l'utilisateur à n'importe quel serveur, au lieu d'être lié au même serveur sur lequel nous nous sommes connectés.
- ★ Nous avons déjà utilisé des jetons dans les sections précédentes sous la forme de jetons **CSRF** et **JSESSIONID**.
 - CSRF Token a protégé notre application des attaques CSRF.
 - JSESSIONID est le jeton par défaut généré par Spring Security qui nous a aidés à ne pas partager les informations d'identification avec le backend à chaque fois.

JWT TOKENS

- ✓ JWT signifie JSON Web Token. Il s'agit d'une implémentation de jeton qui sera au format JSON et conçue pour être utilisée pour les requêtes Web.
- ✓ JWT est le type de jeton le plus courant et le plus apprécié que de nombreux systèmes utilisent de nos jours en raison de ses caractéristiques et avantages particuliers.
- ✓ Les jetons JWT peuvent être utilisés à la fois dans les scénarios d'autorisation/d'authentification et d'échange d'informations, ce qui signifie que vous pouvez partager certaines données relatives à l'utilisateur dans le jeton lui-même, ce qui réduira le fardeau de la maintenance de ces détails dans les sessions sur le cycle du serveur.

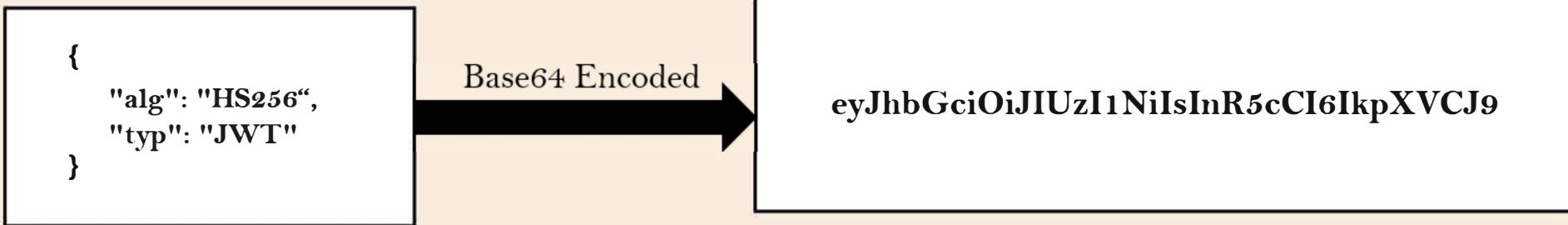
Un jeton JWT comporte 3 parties séparées chacune par un point (.). Vous trouverez ci-dessous un exemple de jeton JWT,

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

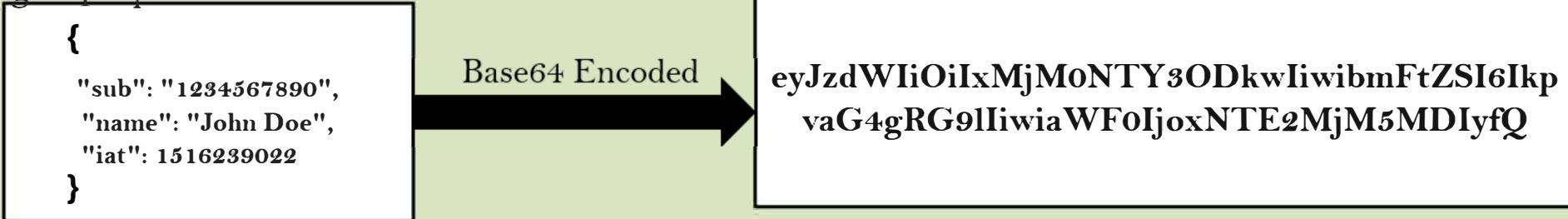
1. Header
2. Payload
3. Signature (Optional)

JWT TOKENS

- ✓ Dans l'en-tête JWT, nous stockons les métadonnées/informations liées au jeton. Si j'ai choisi de signer le jeton, l'en-tête contient le nom de l'algorithme qui génère la signature.



- ✓ Dans le corps, nous pouvons stocker des détails liés à l'utilisateur, aux rôles, etc. qui peuvent être utilisés ultérieurement pour AuthN et AuthZ. Bien qu'il n'y ait pas une telle limitation de ce que nous pouvons envoyer et de la quantité que nous pouvons envoyer dans le corps, nous devons faire de notre mieux pour le garder aussi léger que possible.



JWT TOKENS

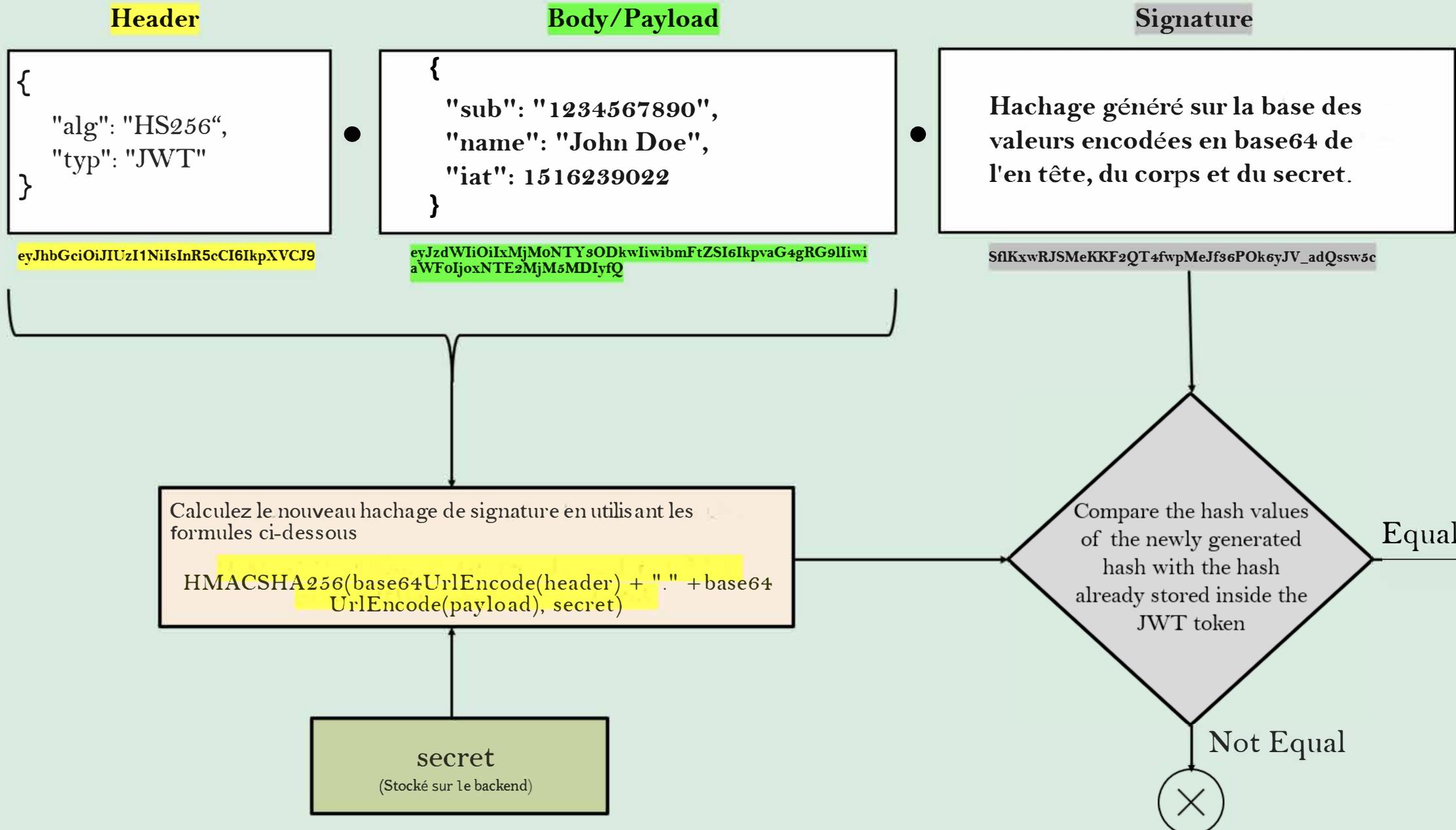
- ✓ La dernière partie du jeton est la signature numérique. Cette partie peut être facultative si la partie avec laquelle vous partagez le jeton JWT est interne et qu'il s'agit d'une personne de confiance mais pas ouverte sur le Web.
- ✓ Mais si vous partagez ce jeton avec les applications clientes qui seront utilisées par tous les utilisateurs du Web ouvert, nous devons nous assurer que personne n'a modifié les valeurs d'en-tête et de corps telles que les autorités, le nom d'utilisateur, etc.
- ✓ Pour s'assurer que personne n'a altéré les données sur le réseau, nous pouvons envoyer la signature du contenu lors de la génération initiale du jeton. Pour créer la partie signature, vous devez prendre l'en-tête encodé, la charge utile encodée, un secret, l'algorithme spécifié dans l'en-tête et le signer.

- ✓ Par exemple si vous souhaitez utiliser l'algorithme HMAC SHA256, la signature sera créée de la manière suivante :

```
HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), secret)
```

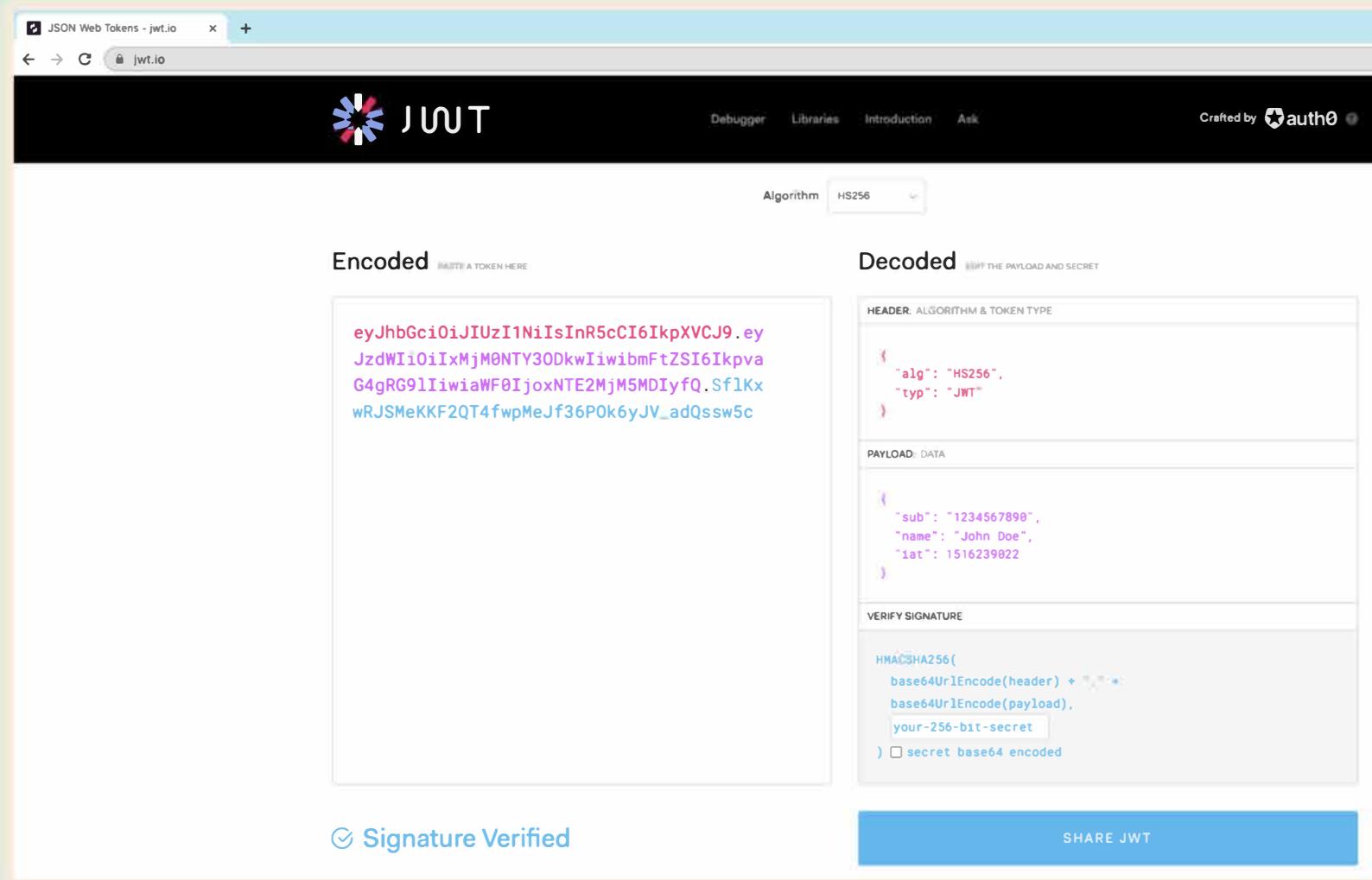
- ✓ La signature est utilisée pour vérifier que le message n'a pas été modifié en cours de route et, dans le cas de jetons signés avec une clé privée, elle peut également vérifier que l'expéditeur du JWT est celui qu'il prétend être.

VALIDATION DES JETONS JWT



JWT TOKENS

- ✓ Si vous voulez jouer avec les jetons JWT et mettre ces concepts en pratique, vous pouvez utiliser le débogueur jwt.io pour décoder, vérifier et générer des JWT.



SÉCURITÉ AU NIVEAU DE LA MÉTHODE

- ✓ À partir de maintenant, nous avons appliqué des règles d'autorisation sur les chemins d'API/URL en utilisant la sécurité Spring, mais la sécurité au niveau de la méthode permet d'appliquer les règles d'autorisation à n'importe quelle couche d'une application, comme la couche de service ou la couche de référentiel, etc. La sécurité au niveau de la méthode peut être activée à l'aide de la annotation `@EnableGlobalMethodSecurity` sur la classe de configuration.
- ✓ La sécurité au niveau de la méthode aidera également les règles d'autorisation, même dans les applications non Web où nous n'aurons aucun point de terminaison.

- ✓ La sécurité au niveau de la méthode fournit les approches ci-dessous pour appliquer les règles d'autorisation et exécuter votre logique métier,
 - **Invocation authorization** – Valide si quelqu'un peut invoquer une méthode ou non en fonction de son rôles/autorités.
 - **Filtering authorization** – Valide ce qu'une méthode peut recevoir via ses paramètres et ce que l'invocateur peut recevoir en retour de la méthode après l'exécution de la logique métier.

SÉCURITÉ AU NIVEAU DE LA MÉTHODE

- ✓ La sécurité Spring utilisera les aspects du module AOP et aura les intercepteurs entre l'invocation de la méthode pour appliquer les règles d'autorisation configurées.
- ✓ La sécurité au niveau de la méthode offre ci-dessous 3 styles différents pour configurer les règles d'autorisation en plus des méthodes,
 - La propriété **prePostEnabled** active les annotations Spring Security **@PreAuthorize & @PostAuthorize**
 - La propriété **secureEnabled** active l'annotation **@Secured**
 - La propriété **jsr250Enabled** active l'annotation **@RoleAllowed**

```
@Configuration  
@EnableGlobalMethodSecurity(prePostEnabled = true, securedEnabled = true, jsr250Enabled = true)  
public class ProjectSecurityConfig {  
    ...  
}
```

- ✓ **@Secured** et **@RoleAllowed** sont moins puissants que **@PreAuthorize** et **@PostAuthorize**

SÉCURITÉ AU NIVEAU DE LA MÉTHODE

- En utilisant l'autorisation d'invocation, nous pouvons décider si un utilisateur est autorisé à invoquer une méthode avant que la méthode ne s'exécute (préautorisation) ou après que l'exécution de la méthode soit terminée (postautorisation). Pour filtrer les paramètres avant d'appeler la méthode, nous pouvons utiliser le préfiltrage,

```
@Service  
public class LoansService {  
  
    @PreAuthorize("hasAuthority('VIEWLOANS')")  
    @PreAuthorize("hasRole('ADMIN')")  
    @PreAuthorize("hasAnyRole('ADMIN','USER')")  
    @PreAuthorize("# username == authentication.principal.username")  
    public Loan getLoanDetails(String username) {  
        return loansRepository.loadLoanDetailsByUserName(username);  
    }  
}
```

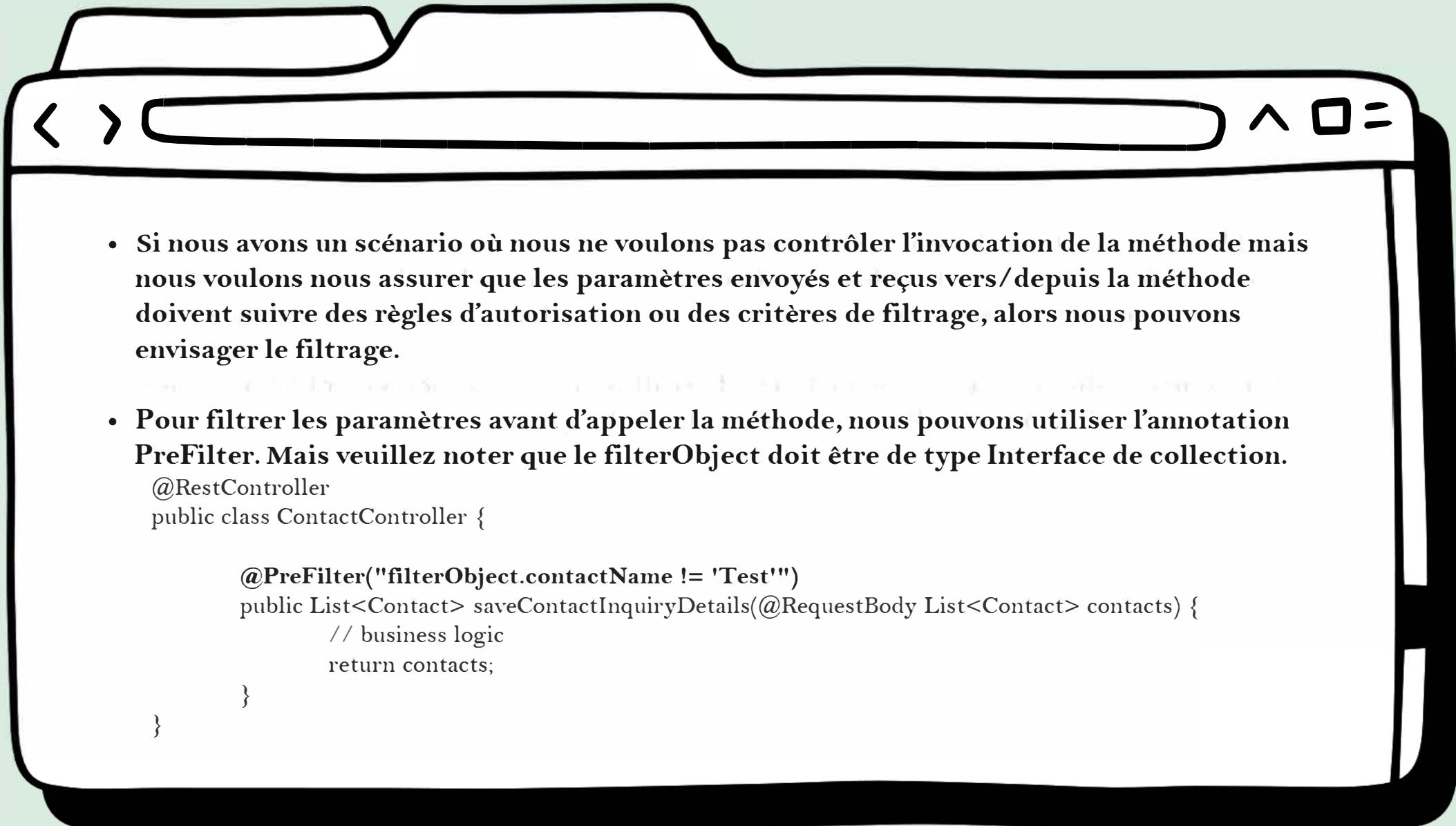
SÉCURITÉ AU NIVEAU DE LA MÉTHODE

- Pour appliquer les règles de post-autorisation ci-dessous est l'exemple de configuration,

```
@Service  
public class LoanService {  
  
    @PostAuthorize ("returnObject.username == authentication.principal.username")  
    @PostAuthorize("hasPermission(returnObject, 'ADMIN')")  
    public Loan getLoanDetails(String username) {  
        return loanRepository.loadLoanByUserName(username);  
    }  
}
```

- Lors de l'implémentation d'une logique d'autorisation complexe, nous pouvons séparer la logique à l'aide d'une classe distincte qui implémente **PermissionEvaluator** et écraser la méthode **hasPermission()** à l'intérieur qui peut être exploitée dans les configurations **hasPermission**.

SÉCURITÉ AU NIVEAU DE LA MÉTHODE



SÉCURITÉ AU NIVEAU DE LA MÉTHODE

- Pour filtrer les paramètres après l'exécution de la méthode, nous pouvons utiliser l'annotation `PostFilter`. Mais veuillez noter que le `filterObject` doit être de type Interface de collection.

```
@RestController  
public class ContactController {  
  
    @PostFilter("filterObject.contactName != 'Test'"')  
    public List<Contact> saveContactInquiryDetails(@RequestBody List<Contact> contacts) {  
        // business logic  
        return contacts;  
    }  
}
```

- Nous pouvons également utiliser `@PostFilter` sur les méthodes du référentiel Spring Data pour filtrer toutes les données indésirables provenant de la base de données.