JWT et Spring Boot

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en programmation par contrainte (IA)
Ingénieur en génie logiciel

elmouelhi.achref@gmail.com

Plan

Spring Boot

JWT : **J**SON **W**eb **T**oken

- Librairie d'échange sécurisé d'informations
- Utilisant des algorithmes de cryptage comme **HMAC SHA256** ou **RSA**
- Utilisant les jetons (tokens)
- Un jeton est composé de trois parties séparées par un point :
  - entête (**header**) : objet **JSON** décrivant le jeton encodé en base 64
  - charge utile (**payload**) : objet **JSON** contenant les informations du jeton encodé en base 64
  - Une signature numérique = concaténation de deux éléments précédents séparés par un point + une clé secrète (le tout crypté par l'algorithme spécifié dans l'entête)
- Documentation officielle : `https://jwt.io/introduction/`

Spring Boot

Entête : exemple

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

Spring Boot

Entête : exemple

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

Charge utile : exemple

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

Spring Boot

Exemple de construction de signature en utilisant l'algorithme précisé dans l'entête

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

Spring Boot

Exemple de construction de signature en utilisant l'algorithme précisé dans l'entête

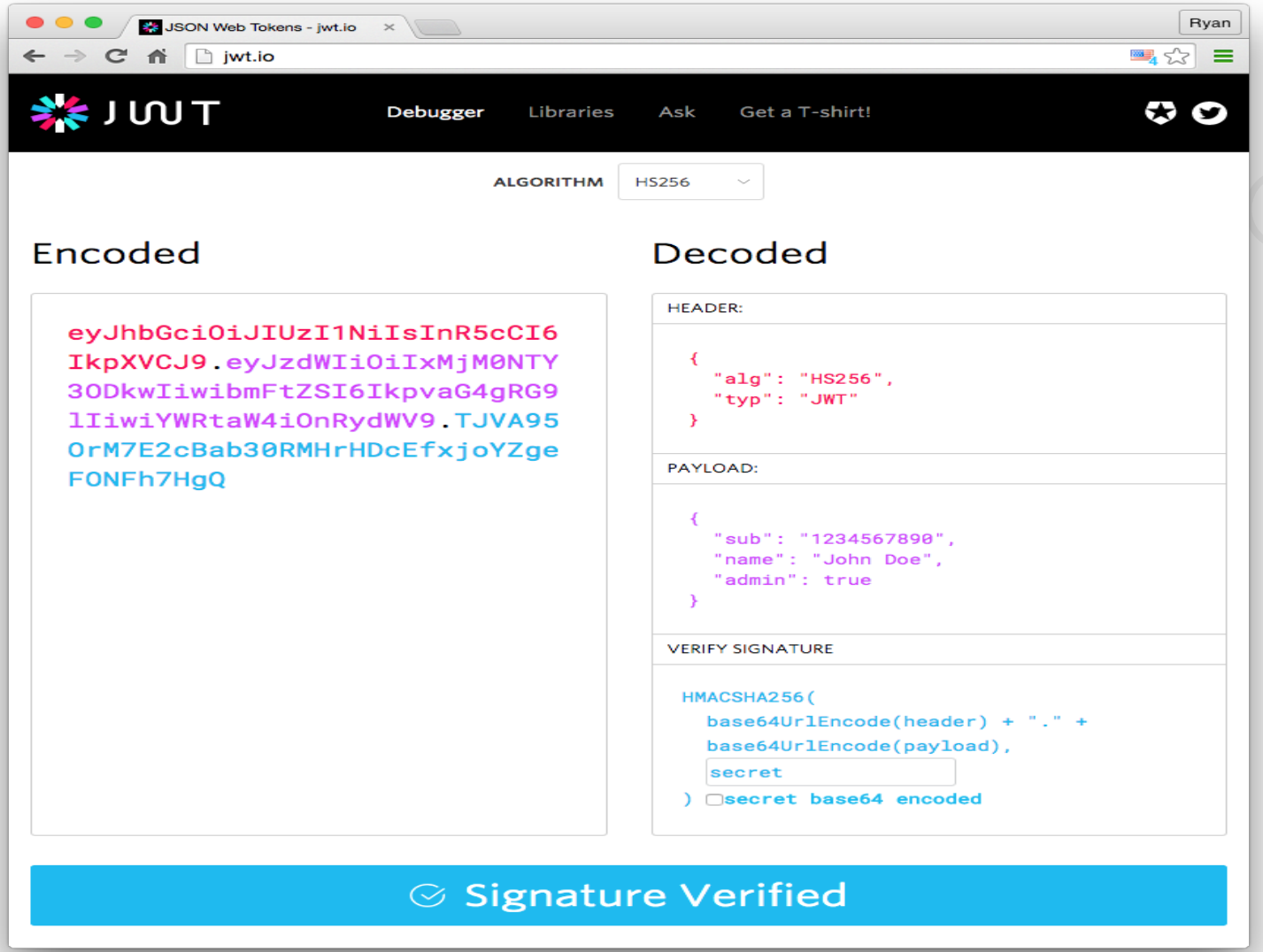
```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret)
```

Résultat

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4
gRG9lIiwiaXNTb2NpYWwiOnRydWV9.
4pcPyMD09olPSyXnrXCjTwXyr4BsezdI1AVTmud2fU4

Spring Boot

Exemple complet (pour tester `https://jwt.io/#debugger-io`)



Spring Boot

Pour décoder les deux premières parties d'un jeton

`http://calebb.net/`

Spring Boot

Objectif

Sécuriser l'accès à nos ressources **REST** en minimisant l'accès à la base de données pour vérifier l'identité de l'utilisateur.

Spring Boot

Objectif

Sécuriser l'accès à nos ressources **REST** en minimisant l'accès à la base de données pour vérifier l'identité de l'utilisateur.

Ce qu'il faut

- **Spring Security** pour la gestion des utilisateurs
- **JWT** pour les jetons

Spring Boot

Création de projet Spring Boot

- Aller dans `File > New > Other`
- Chercher `Spring`, dans `Spring Boot` sélectionner `Spring Starter Project` et cliquer sur `Next >`
- Saisir
  - `spring-boot-jwt` dans `Name`,
  - `com.example` dans `Group`,
  - `spring-boot-jwt` dans `Artifact`
  - `com.example.demo` dans `Package`
- Cliquer sur `Next`
- Chercher et cocher les cases correspondantes aux `Spring Data JPA`, `MySQL Driver`, `Spring Web`, `Spring Boot DevTools`, `Lombok` et `Spring Security`
- Cliquer sur `Next` puis sur `Finish`

Spring Boot

Explication

- Le package contenant le point d'entrée de notre application (la classe contenant le `puclic static void main`) est `com.example.demo`
- Tous les autres packages `dao`, `model`... doivent être dans le package `demo`.

Spring Boot

Explication

- Le package contenant le point d'entrée de notre application (la classe contenant le `puclic static void main`) est `com.example.demo`
- Tous les autres packages `dao`, `model`... doivent être dans le package `demo`.

Pour la suite, nous considérons

- une entité `Personne` à définir dans `com.example.demo.model`
- une interface DAO `PersonneRepository` à définir dans `com.example.demo.dao`
- un contrôleur REST `PersonneController` à définir dans `com.example.demo.dao`

Spring Boot

Dans `application.properties`, ajoutons les données permettant la connexion à la base de données et la configuration de `Hibernate`

```
spring.datasource.url = jdbc:mysql://localhost:3306/cours_boot?createDatabaseIfNotExist
  =true
spring.datasource.username = root
spring.datasource.password = root
spring.jpa.hibernate.ddl-auto = update
spring.jpa.show-sql = true
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQLDialect
spring.jpa.hibernate.naming.physical-strategy = org.hibernate.boot.model.naming.
  PhysicalNamingStrategyStandardImpl
```


L'ajout de la propriété `spring.jpa.hibernate.naming.physical-strategy` permet de forcer **Hibernate** à utiliser les mêmes noms pour les tables et les colonnes que les entités et les attributs.

Spring Boot

Lancez l'application et vérifiez la génération d'un mot de passe dans la console

Using generated security password : 895a8a45-d296-4ee6-a246-421d6dd6e64e

Spring Boot

Lancez l'application et vérifiez la génération d'un mot de passe dans la console

Using generated security password : 895a8a45-d296-4ee6-a246-421d6dd6e64e

Remarque

Ce mot de passe est à utiliser pour accéder aux ressources.

Spring Boot

Pour tester

- Il faut aller à `http://localhost:8080/` et vérifiez l'affichage de l'interface d'authentification
- Utilisez `user` comme nom d'utilisateur et le mot de passé généré et affiché dans la console pour la connexion

Créons une entité `Personne` dans `com.example.demo.model`

```
package com.example.demo.model;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@NoArgsConstructor
@AllArgsConstructor
@Data
@Entity
public class Personne {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long num;
  private String nom;
  private String prenom;
}
```

Spring Boot & REST

Préparons notre interface DAO `PersonneRepository`

```
package com.example.demo.dao;

import org.springframework.data.jpa.repository.JpaRepository;

import com.example.demo.model.Personne;

public interface PersonneRepository extends JpaRepository<Personne, Long> {

}
```

Spring Boot

Créons le contrôleur REST suivant dans `com.example.demo.controller`

```
@RestController
public class PersonneRestController {

  @Autowired
  private PersonneRepository personneRepository;

  @GetMapping("/personnes")
  public List<Personne> getPersonnes() {
    return personneRepository.findAll();
  }

  @GetMapping("/personnes/{id}")
  public Personne getPersonne(@PathVariable("id") long id) {
    return personneRepository.findById(id).orElse(null);
  }

  @PostMapping("/personnes")
  public Personne addPersonne(@RequestBody Personne personne) {
    return personneRepository.save(personne);
  }
}
```

Spring Boot

Pour tester

Il faut aller à `http://localhost:8080/personnes` ou sur `http://localhost:8080/personnes/1`.

Spring Boot

Dans `com.example.demo.configuration`**, définissons la classe** `SecurityConfig`

```
package com.example.demo.configuration;

import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.configuration
   .EnableWebSecurity;

@Configuration
@EnableWebSecurity
public class SecurityConfig {

}
```

Spring Boot

Contenu de `SecurityConfig`

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {
    @Autowired
    private UserDetailsService userDetailsService;

    @Bean
    public static NoOpPasswordEncoder passwordEncoder() {
        return (NoOpPasswordEncoder) NoOpPasswordEncoder.getInstance();
    }

    @Bean
    public AuthenticationManager authenticationManager(HttpSecurity http,
      NoOpPasswordEncoder noOpPasswordEncoder, UserDetailsService userDetailService)
      throws Exception {
        return http.getSharedObject(AuthenticationManagerBuilder.class)
                .userDetailsService(userDetailsService)
                .passwordEncoder(noOpPasswordEncoder)
                .and().build();
    }

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        return http.csrf().disable().authorizeHttpRequests((authz) -> authz.anyRequest
          ().authenticated())
                .httpBasic(Customizer.withDefaults()).build();
    }
}
```

Spring Boot

Contenu de l'entité `Role`

```
@NoArgsConstructor
@AllArgsConstructor
@Data
@Entity
public class Role {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  Long id;
  String titre;
}
```

Spring Boot

Contenu de l'entité `User`

```
@NoArgsConstructor
@AllArgsConstructor
@Data
@Entity
public class User {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  Long num;
  String username;
  String password;
  @ManyToMany(cascade = { CascadeType.PERSIST, CascadeType.REMOVE },
     fetch = FetchType.EAGER)
  List<Role> roles;
}
```

`fetch = FetchType.EAGER` : les `Role` seront chargés au même temps que les `User`.

Spring Boot

Contenu de `UserRepository`

```
public interface UserRepository extends JpaRepository<User, Long> {
  public User findByUsername(String username);
}
```

La méthode `findByUsername` sera utilisée plus tard.

Spring Boot

Contenu de `UserRepository`

```
public interface UserRepository extends JpaRepository<User, Long> {
  public User findByUsername(String username);
}
```

La méthode `findByUsername` sera utilisée plus tard.

Contenu de `RoleRepository`

```
public interface RoleRepository extends JpaRepository<Role, Long> {

}
```

Spring Boot

Créons une classe qui implémente l'interface `UserDetailsService`

```
package com.example.demo.security;

import org.springframework.stereotype.Service;

@Service
public class UserDetailsServiceImpl implements UserDetailsService {

}
```


- Cette classe implémente l'interface `UserDetailsService` et doit donc implémenter la méthode `loadUserByUserName()` qui retourne un objet de type `UserDetails` (interface).
- L'annotation Service nous permettra d'utiliser cette classe en faisant une injection de dépendance.

Spring Boot

Contenu de la classe qui implémente `UserDetailsService`

```
package com.example.demo.security;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;
import com.example.demo.dao.UserRepository;
import com.example.demo.model.User;

@Service
public class UserDetailsServiceImpl implements UserDetailsService {

  @Autowired UserRepository userRepository;
  @Override

  public UserDetailsImpl loadUserByUsername(String username) throws
     UsernameNotFoundException {
    User user = userRepository.findByUsername(username);
    if (null == user){
      throw new UsernameNotFoundException("No user named " + username);
    } else {
      return new UserDetailsImpl(user);
    }
  }
}
```

Spring Boot

Créons une classe qui implémente l'interface `UserDetails`

```
package com.example.demo.security;

public class UserDetailsImpl  implements UserDetails {

}
```


Cette classe implémente l'interface `UserDetails` et doit donc implémenter toutes ses méthodes abstraites

Spring Boot

Créer une classe qui implémente l'interface `UserDetails`

```
package com.example.demo.security;

import java.util.ArrayList;
import java.util.Collection;
import java.util.List;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;
import com.example.demo.model.Role;
import com.example.demo.model.User;

public class UserDetailsImpl  implements UserDetails {

  private User user;

  public  UserDetailsImpl(User user){
    this.user = user;
  }

  @Override
  public Collection<? extends GrantedAuthority> getAuthorities() {
    final List<GrantedAuthority> authorities = new ArrayList<GrantedAuthority>();
    for (final Role role: user.getRoles())
      authorities.add(new SimpleGrantedAuthority(role.getTitre()));
    return authorities;
  }
```

Spring Boot

`UserDetailsImpl` **(la suite)**

```
  @Override
  public boolean isAccountNonExpired() {
    return true;
  }
  @Override
  public boolean isAccountNonLocked() {
    return true;
  }
  @Override
  public boolean isCredentialsNonExpired() {
    return true;
  }
  @Override
  public boolean isEnabled() {
    return true;
  }
  @Override
  public String getUsername() {
    return user.getUsername();
  }
  @Override
  public String getPassword() {
    return user.getPassword();
  }
}
```

Spring Boot

Avant de tester, lancez le projet pour que Spring crée les tables ensuite créez trois utilisateurs avec des rôles différents

```
INSERT INTO role VALUES (1, "ROLE_ADMIN"),
(2, "ROLE_USER");

INSERT INTO user VALUES (1, "wick", "wick"),
(2, "john", "john"),
(3, "alan", "alan");

INSERT INTO user_roles VALUES (1, 1),
(2, 2),
(1, 2),
(3, 1);
```

Spring Boot

Pour tester

Il faut aller à `http://localhost:8080/personnes`, s'authentifier avec les identifiants d'un utilisateur de la table `user`

Spring Boot

Ajoutons les dépendances pour JWT

```
<!-- https://mvnrepository.com/artifact/io.jsonwebtoken/jjwt-api -->
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-api</artifactId>
    <version>0.11.5</version>
</dependency>
<dependency>
        <groupId>io.jsonwebtoken</groupId>
        <artifactId>jjwt-impl</artifactId>
        <version>0.11.5</version>
        <scope>runtime</scope>
</dependency>
<dependency>
        <groupId>io.jsonwebtoken</groupId>
        <artifactId>jjwt-jackson</artifactId>
        <version>0.11.5</version>
        <scope>runtime</scope>
</dependency>
```

Spring Boot

Ajoutons aussi la propriété suivante dans `application.properties`

```
jwt.secret = uneclesur32octetsuneclesur32octetsuneclesur32octets
```

Spring Boot

Commençons par définir une classe `JwtTokenUtil` pour la génération et la validation du `token`

```
package com.example.demo.configuration;

@Component
public class JwtTokenUtil {

}
```

Spring Boot

Définissons la durée d'un jeton (5 heures par exemple)

```
package com.example.demo.configuration;

@Component
public class JwtTokenUtil {

    public static final long JWT_TOKEN_VALIDITY = 60 * 60;

}
```

Spring Boot

Définissons la durée d'un jeton (5 heures par exemple)

```
package com.example.demo.configuration;

@Component
public class JwtTokenUtil {

    public static final long JWT_TOKEN_VALIDITY = 60 * 60;

}
```

Spring Boot

Récupérons la propriété `jwt.secret` **d'**`application.properties`

```
package com.example.demo.configuration;

@Component
public class JwtTokenUtil {

    public static final long JWT_TOKEN_VALIDITY = 60 * 60;

    @Value("${jwt.secret}")
    private String secret;

}
```

Spring Boot

Décodons et créons une clé avec le secret

```
private Key getSigningKey() {
    byte[] keyBytes = Base64.getDecoder().decode(this.secret);
    return Keys.hmacShaKeyFor(keyBytes);
}
```

Spring Boot

Décodons et créons une clé avec le secret

```
private Key getSigningKey() {
    byte[] keyBytes = Base64.getDecoder().decode(this.secret);
    return Keys.hmacShaKeyFor(keyBytes);
}
```

Définissons les méthodes qui permettent la récupération de données depuis un jeton

```
public String getUsernameFromToken(String token) {
    return getClaimFromToken(token, Claims::getSubject);
}

public Date getExpirationDateFromToken(String token) {
    return getClaimFromToken(token, Claims::getExpiration);
}

public <T> T getClaimFromToken(String token, Function<Claims, T> claimsResolver) {
    final Claims claims = getAllClaimsFromToken(token);
    return claimsResolver.apply(claims);
}

private Claims getAllClaimsFromToken(String token) {
    return Jwts.parserBuilder().setSigningKey(getSigningKey()).build().parseClaimsJws(token).
      getBody();
}
```

Spring Boot

Définissons les méthodes qui permettent la génération et la validation du token

```
public String generateToken(UserDetails userDetails) {
    Map<String, Object> claims = new HashMap<>();
    claims.put("roles", userDetails.getAuthorities());
    return Jwts.builder().setClaims(claims).setSubject(userDetails.getUsername()).setIssuedAt(
      new Date(System.currentTimeMillis()))
            .setExpiration(new Date(System.currentTimeMillis() + JWT_TOKEN_VALIDITY * 1000))
            .signWith(getSigningKey(), SignatureAlgorithm.HS256).compact();
}
```

Spring Boot

Définissons les méthodes qui permettent la génération et la validation du token

```
public String generateToken(UserDetails userDetails) {
    Map<String, Object> claims = new HashMap<>();
    claims.put("roles", userDetails.getAuthorities());
    return Jwts.builder().setClaims(claims).setSubject(userDetails.getUsername()).setIssuedAt(
      new Date(System.currentTimeMillis()))
            .setExpiration(new Date(System.currentTimeMillis() + JWT_TOKEN_VALIDITY * 1000))
            .signWith(getSigningKey(), SignatureAlgorithm.HS256).compact();
}
```

Et aussi celles qui permettent la génération d'un jeton

```
private Boolean isTokenExpired(String token) {
    final Date expiration = getExpirationDateFromToken(token);
    return expiration.before(new Date());
}

public Boolean validateToken(String token, UserDetails userDetails) {
    final String username = getUsernameFromToken(token);
    return (username.equals(userDetails.getUsername()) && !isTokenExpired(token));
}
```

Spring Boot

Définissons une classe filtre qui intercepte l'accès et qui permet d'autoriser ou interdire l'accès à un utilisateur

```
package com.example.demo.configuration;

import org.springframework.stereotype.Component;

@Component
public class JwtFilter {

}
```

Spring Boot

Cette classe doit hériter de `OncePerRequestFilter` et implémenter sa méthode abstraite `doFilterInternal`

```
package com.example.demo.configuration;

import java.io.IOException;

import org.springframework.stereotype.Component;
import org.springframework.web.filter.OncePerRequestFilter;

import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;

@Component
public class JwtFilter extends OncePerRequestFilter {

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse
       response, FilterChain filterChain)
            throws ServletException, IOException {

        filterChain.doFilter(request, response);
    }

}
```

Spring Boot

Injectons les dépendances nécessaires pour le filtre

```
@Component
public class JwtFilter extends OncePerRequestFilter {
    @Autowired
    private JwtTokenUtil jwtTokenUtil;
    @Autowired
    private UserDetailsService userDetailsService;

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse
       response, FilterChain filterChain)
            throws ServletException, IOException {

        filterChain.doFilter(request, response);
    }

}
```

Implémentons maintenant `doFilterInternal`

```
@Override
protected void doFilterInternal(HttpServletRequest request, HttpServletResponse
   response, FilterChain filterChain) throws ServletException, IOException {

    final String requestTokenHeader = request.getHeader("Authorization");
    String username = null;
    String jwtToken = null;
    if (requestTokenHeader != null && requestTokenHeader.startsWith("Bearer ")) {
        jwtToken = requestTokenHeader.substring(7);
        try {
            username = jwtTokenUtil.getUsernameFromToken(jwtToken);
        } catch (IllegalArgumentException e) {
            System.out.println("Impossible de récupérer le jeton JWT");
        } catch (ExpiredJwtException e) {
            System.out.println("Jeton JWT expiré");
        }
    } else {
        logger.warn("Il ne s'agit pas d'une authentification Bearer");
    }
    if (username != null && SecurityContextHolder.getContext().getAuthentication() ==
       null) {
        UserDetails userDetails = userDetailsService.loadUserByUsername(username);
        if (jwtTokenUtil.validateToken(jwtToken, userDetails)) {
            var principalUser = new     UsernamePasswordAuthenticationToken(userDetails
               , null, userDetails.getAuthorities());
            principalUser.setDetails(new WebAuthenticationDetailsSource().buildDetails(
               request));
            SecurityContextHolder.getContext().setAuthentication(principalUser);
        }
    }
    filterChain.doFilter(request, response);
}
```

Spring Boot

Mettons à jour `SecurityConfig` pour charger toutes les classes précédentes

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Autowired
    private UserDetailsService userDetailsService;

    @Autowired
    private JwtFilter jwtFilter;

}
```

Spring Boot

Chargeons les deux beans responsables du chargement de données de l'utilisateur

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Autowired
    private UserDetailsService userDetailsService;

    @Autowired
    private JwtFilter jwtFilter;

    @Bean
    public static NoOpPasswordEncoder passwordEncoder() {
        return (NoOpPasswordEncoder) NoOpPasswordEncoder.getInstance();
    }

    @Bean
    public AuthenticationManager authenticationManager(HttpSecurity http, NoOpPasswordEncoder
       noOpPasswordEncoder, UserDetailsService userDetailService) throws Exception {
        return http.getSharedObject(AuthenticationManagerBuilder.class)
                .userDetailsService(userDetailsService)
                .passwordEncoder(noOpPasswordEncoder).and().build();
    }

}
```

Spring Boot

Et ensuite le bean responsable de charger le filtre

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

    http.addFilterBefore(jwtFilter, UsernamePasswordAuthenticationFilter.class);
    http.cors().and().csrf().disable();
    http.authorizeHttpRequests((authz) -> authz.requestMatchers("/authenticate").permitAll());
    http.authorizeHttpRequests((authz) -> authz.anyRequest().fullyAuthenticated());
    http.httpBasic();
    return http.build();
}
```

Et enfin le contrôleur

```
@RestController
@CrossOrigin
public class JwtAuthenticationController {

    @Autowired
    private AuthenticationManager authenticationManager;

    @Autowired
    private JwtTokenUtil jwtTokenUtil;

    @Autowired
    private UserDetailsService userDetailsService;

    @PostMapping(value = "/authenticate")
    public ResponseEntity<?> createAuthenticationToken(@RequestBody User user) {
        try {
            authenticationManager.authenticate(new UsernamePasswordAuthenticationToken(user.
              getUsername(), user.getPassword()));
            final UserDetails userDetails = userDetailsService.loadUserByUsername(user.
              getUsername());
            final String token = jwtTokenUtil.generateToken(userDetails);

            return ResponseEntity.ok(token);
        } catch (Exception e) {
            e.printStackTrace();
        }
        return new ResponseEntity<>(null, HttpStatus.NOT_FOUND);
    }
}
```

Spring Boot

Pour obtenir le jeton avec **Postman**

- Dans la liste déroulante, choisir `POST` puis saisir l'URL vers notre web service `http://localhost:8080/authenticate`
- Dans le `Headers`, saisir `Content-Type` comme `Key` et `application/json` comme `Value`
- Ensuite cliquer sur `Body`, cocher `raw`, choisir `JSON (application/json)` et saisir des données sous format `JSON` correspondant à l'objet personne à ajouter

```
{
    "username": "wick",
    "password": "wick"
}
```

- Cliquer sur `Send` puis copier le jeton

Spring Boot

Pour obtenir la liste des personnes avec **Postman**

- Dans la liste déroulante, choisir `GET` puis saisir l'URL vers notre web service `http://localhost:8080:personnes`
- Dans le `Headers`, saisir `Content-Type` comme `Key` et `application/json` comme `Value`
- Dans `Authorization`, cliquer sur `Type`, choisir `Bearer Token` et coller le token
- Cliquer sur `Send`