Spring Security, OAuth et JWT

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

OAuth

- Protocole créé en 2006
- Utilisé pour l'autorisation (pas pour l'authentification)
- Permettant de vérifier si une application cliente a le droit d'accéder à un Service Web
- Projet appartenant initialement à **Twitter** et ensuite utilisé par **Google**, **Facebook**...
- Fonctionnant avec un jeton d'accès (access-token)
- Documentation officielle : `https://www.oauth.com/`

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

```
{
  "iss": "spring-boot",
  "sub": "admin",
  "exp": 1273038184,
  "iat": 1273037884,
  "roles": "ROLE_ADMIN"
}
```

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

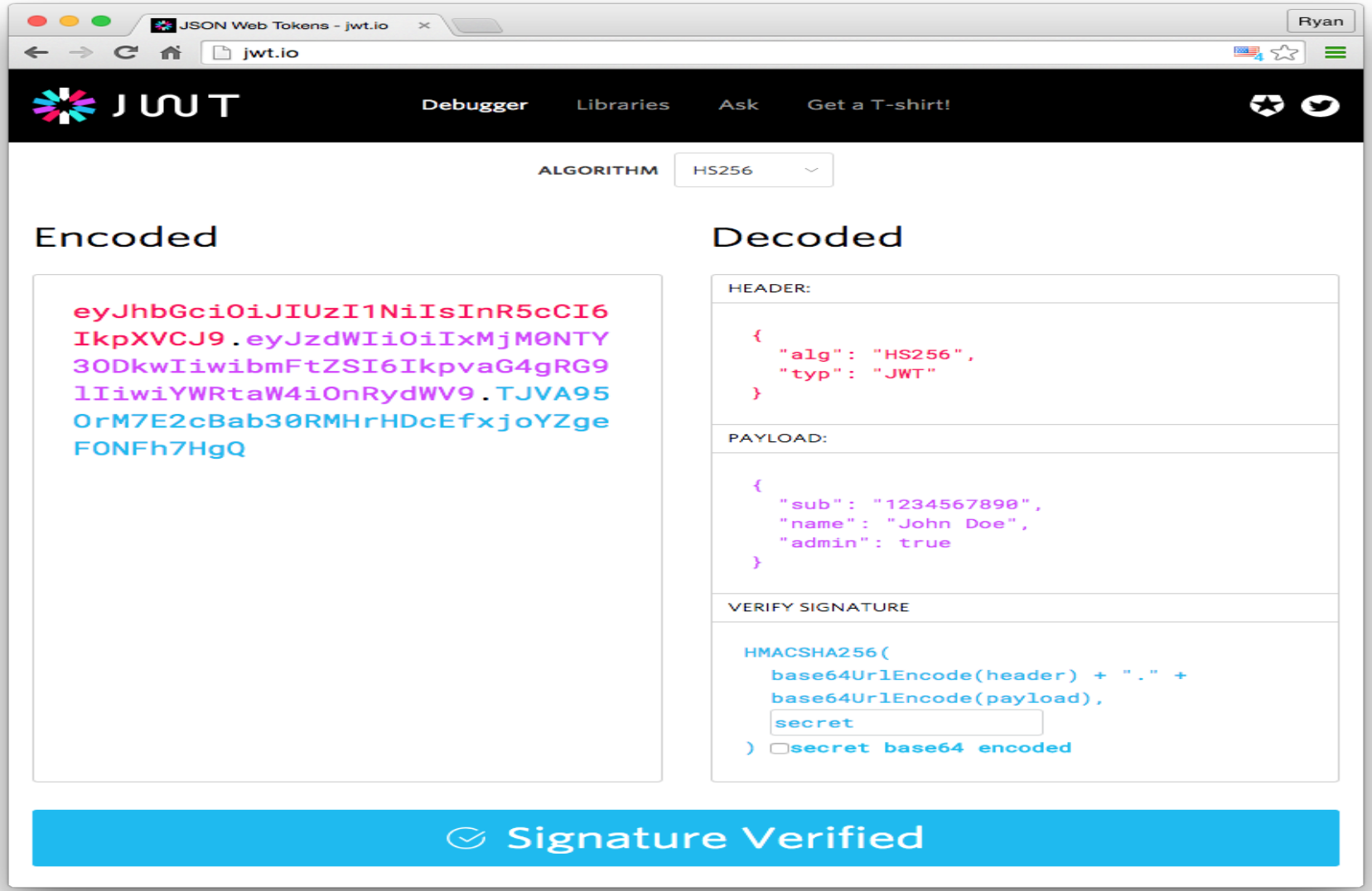
```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret)
```

Résultat

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4
gRG9lIiwiaXNTb2NpYWwiOnRydWV9.
4pcPyMD09olPSyXnrXCjTwXyr4BsezdI1AVTmud2fU4

Spring Boot

Exemple



Spring Boot

Pour décoder les deux premières parties d'un jeton

- `https://jwt.io/#debugger-io`
- `http://calebb.net/`

Spring Boot

Terminologies autour de **JWT**

- **JWK** (**J**SON **W**eb **K**ey) : format standard de représentation de clés (privé et public)
- **JWS** (**J**SON **W**eb **S**ignature) : format standard de vérification de contenu d'un jeton **JWT**
- **JWE** (**J**SON **W**eb **E**ncryption) : format standard de représentation de contenu chiffré d'un jeton **JWT**

Spring Boot

Librairies **Java** pour la génération et la validation de **JWT**

- **Nimbus JOSE** (**J**SON **O**bject **S**igning and **E**ncryption)
- **JJWT** (**J**ava **JWT**)
- ...

Spring Boot

JJWT : avantages

- Open-source
- Prise en charge de plusieurs algorithmes de chiffrement : **HMAC**, **RSA**, **ECDSA**...
- Ne Supporte pas **JWK**, **JWS**...
- Rapide

Nimbus : avantages

- Open-source
- Prise en charge de plusieurs algorithmes de chiffrement : **HMAC**, **RSA**, **ECDSA**, **PBES2**...
- Supporte **JWK**, **JWS**...
- Facile à mettre en place

JJWT : inconvénients

- Jeton volumineux
- Moins sécurisé que **Nimbus**

Nimbus : inconvénients

- Librairie volumineuse
- Courbe d'apprentissage plus importante

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-oauth-jwt` dans `Name`,
  - `com.example` dans `Group`,
  - `spring-boot-oauth-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`, `Spring Security`, `Spring Configuration Processor` et `OAuth2 Resource Server`
- 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 `PersonneController` à définir dans `com.example.demo.controller`

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/jwt_oauth?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

Spring Boot

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;
import lombok.NonNull;
import lombok.RequiredArgsConstructor;

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

Spring Boot

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

Aller à `http://localhost:8080/personnes` ou `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
    AuthenticationManager authenticationManager() {
        DaoAuthenticationProvider authProvider = new DaoAuthenticationProvider();
        authProvider.setUserDetailsService(userDetailsService);
        authProvider.setPasswordEncoder(encoder());
        return new ProviderManager(authProvider);
    }

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

Spring Boot

Contenu de l'entité `Role`

```
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;
import lombok.NonNull;
import lombok.RequiredArgsConstructor;

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

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

Spring Boot

Contenu de l'entité `User`

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

Spring Boot

Contenu de `UserRepository`

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

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

Modifions la classe de démarrage afin d'alimenter la base de données avec des données de test

```
@SpringBootApplication
public class SpringBootOauthJwtApplication implements ApplicationRunner {

    @Autowired
    private PersonneRepository personneRepository;

    @Autowired
    private UserRepository userRepository;

    public static void main(String[] args) {
        SpringApplication.run(SpringBootOauthJwtApplication.class, args);
    }

    @Override
    public void run(ApplicationArguments args) throws Exception {
      personneRepository.save(new Personne("Wick", "John"));
      personneRepository.save(new Personne("Dalton", "Jack"));
      userRepository.save(new User("admin", "admin", List.of(new Role("ADMIN"))));
      userRepository.save(new User("user", "user", List.of(new Role("USER"))));
    }

}
```

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

Avant de générer les clés publique et privé

- Créer un dossier `jwt` dans `src/main/resources`
- Faire clic-droit sur `jwt` et aller à `Show In > Terminal`

Spring Boot

Pour créer un répertoire `jwt` dans `src/main/resources` pour les clés, exécutez

```
openssl genrsa -out keypair.pem 2048
```

Spring Boot

Pour créer un répertoire `jwt` dans `src/main/resources` pour les clés, exécutez

```
openssl genrsa -out keypair.pem 2048
```

Pour générer une clé public, exécutez

```
openssl rsa -in keypair.pem -pubout -out public.pem
```

Spring Boot

Pour créer un répertoire `jwt` dans `src/main/resources` pour les clés, exécutez

```
openssl genrsa -out keypair.pem 2048
```

Pour générer une clé public, exécutez

```
openssl rsa -in keypair.pem -pubout -out public.pem
```

Pour pour générer une clé privé, exécutez

```
openssl pkcs8 -topk8 -inform PEM -outform PEM -nocrypt -in keypair.pem -out private.pem
```

Spring Boot

Remarque

Vérifiez que trois fichiers `keypair.pem`, `private.pem` et `public.pem` ont été générés dans `src/main/resources/jwt`.

Spring Boot

Remarque

Vérifiez que trois fichiers `keypair.pem`, `private.pem` et `public.pem` ont été générés dans `src/main/resources/jwt`.

Dans `application.properties`, ajoutons deux propriétés qui font référence aux deux clés générées

```
rsa.public-key=classpath:jwt/public.pem
rsa.private-key=classpath:jwt/private.pem
```

Spring Boot

Remarques

- La clé publique est utilisée pour décoder le jeton
- La clé privé est
  - générée à partir de la clé public
  - utilisée pour encoder le jeton

Spring Boot

Définissons un record `RsaKeyProperties` pour importer les clés public et privé

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

import java.security.interfaces.RSAPrivateKey;
import java.security.interfaces.RSAPublicKey;

import org.springframework.boot.context.properties.ConfigurationProperties;

@ConfigurationProperties(prefix = "rsa")
public record RsaKeyProperties(RSAPublicKey publicKey, RSAPrivateKey privateKey) {

}
```

Spring Boot

Définissons un record `RsaKeyProperties` pour importer les clés public et privé

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

import java.security.interfaces.RSAPrivateKey;
import java.security.interfaces.RSAPublicKey;

import org.springframework.boot.context.properties.ConfigurationProperties;

@ConfigurationProperties(prefix = "rsa")
public record RsaKeyProperties(RSAPublicKey publicKey, RSAPrivateKey privateKey) {

}
```


Activons la configuration de propriétés en ajoutant `@EnableConfigurationProperties` dans la classe de démarrage

```
@SpringBootApplication
@EnableConfigurationProperties(RsaKeyProperties.class)
public class SpringBootOauthJwtApplication implements ApplicationRunner {

    // + le contenu précédent

}
```

Spring Boot

Injectons `RsaKeyProperties` **dans** `SecurityConfig`

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Autowired
    private UserDetailsService userDetailsService;

    @Autowired
    private RsaKeyProperties keyProperties;

    // + les beans précédents

}
```

Spring Boot

Définissons ensuite les deux beans suivants pour encoder et décoder les jetons

```
@Bean
JwtDecoder jwtDecoder() {
    return NimbusJwtDecoder.withPublicKey(keyProperties.publicKey()).build();

}

@Bean
JwtEncoder jwtEncoder() {
    JWK jwk = new RSAKey.Builder(keyProperties.publicKey()).privateKey(keyProperties.
      privateKey()).build();
    JWKSource<SecurityContext> jwks = new ImmutableJWKSet<>(new JWKSet(jwk));
    return new NimbusJwtEncoder(jwks);
}
```

Spring Boot

Définissons ensuite les deux beans suivants pour encoder et décoder les jetons

```
@Bean
JwtDecoder jwtDecoder() {
    return NimbusJwtDecoder.withPublicKey(keyProperties.publicKey()).build();

}

@Bean
JwtEncoder jwtEncoder() {
    JWK jwk = new RSAKey.Builder(keyProperties.publicKey()).privateKey(keyProperties.
      privateKey()).build();
    JWKSource<SecurityContext> jwks = new ImmutableJWKSet<>(new JWKSet(jwk));
    return new NimbusJwtEncoder(jwks);
}
```


Les imports manquants

```
import com.nimbusds.jose.jwk.JWK;
import com.nimbusds.jose.jwk.JWKSet;
import com.nimbusds.jose.jwk.RSAKey;
import com.nimbusds.jose.jwk.source.ImmutableJWKSet;
import com.nimbusds.jose.jwk.source.JWKSource;
import com.nimbusds.jose.proc.SecurityContext;
```

Spring Boot

Modifions également le bean `SecurityFilterChain` pour pouvoir utiliser les jetons

```
@Bean
SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    return http
        .csrf(csrf -> csrf.disable())
        .cors(Customizer.withDefaults())
        .authorizeHttpRequests((authz) -> authz.requestMatchers(HttpMethod.POST, "/
          authenticate").permitAll());
        .authorizeHttpRequests((authz) -> authz.anyRequest().fullyAuthenticated());
        .sessionManagement(session -> session.sessionCreationPolicy(
          SessionCreationPolicy.STATELESS));
        .oauth2ResourceServer(oa -> oa.jwt(Customizer.withDefaults()));
        .httpBasic(Customizer.withDefaults());
        .build();
}
```

Spring Boot

Dans `com.example.demo.service`, définissons un service dont le rôle est de gérer les jetons

```
@Service
public class TokenService {

    @Autowired
    private JwtEncoder jwtEncoder;

    public Map<String, String> generateToken(String username, String roles) {
        Map<String, String> idToken = new HashMap<>();

        JwtClaimsSet jwtClaimsSet = JwtClaimsSet.builder()
        .issuedAt(Instant.now())
        .issuer("spring-boot")
        .expiresAt(Instant.now().plusSeconds(5 * 60))
        .claim("scope", roles)
        .subject(username)
        .build();
        var token = jwtEncoder.encode(JwtEncoderParameters.from(jwtClaimsSet)).getTokenValue();
        idToken.put("accessToken", token);
        return idToken;
    }
}
```

Spring Boot

Créons un contrôleur `JwtAuthenticationController` **dans lequel nous injectons** `TokenService`

```
package com.example.demo.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RestController;

import com.example.demo.service.TokenService;

@RestController
public class JwtAuthenticationController {

    @Autowired
    private  TokenService tokenService;

}
```

Spring Boot

Utilisons le service dans le contrôleur pour retourner le jeton

```
package com.example.demo.controller;

import java.util.Map;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.Authentication;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RestController;

import com.example.demo.service.TokenService;

@RestController
@Slf4j
public class JwtAuthenticationController {

    @Autowired
    private  TokenService tokenService;

    @PostMapping(value = "/authenticate")
    public Map<String, String> createAuthenticationToken(Authentication authentication) {
        String roles = authentication.getAuthorities().stream()
                .map(GrantedAuthority::getAuthority)
                .collect(Collectors.joining(" "));
        return tokenService.generateToken(authentication.getName(), roles);

    }
}
```

Et enfin nous ajoutons la journalisation

```
package com.example.demo.controller;

import java.util.Map;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.Authentication;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RestController;

import com.example.demo.service.TokenService;

import lombok.extern.slf4j.Slf4j;

@RestController
@Slf4j
public class JwtAuthenticationController {

    @Autowired
    private  TokenService tokenService;

    @PostMapping(value = "/authenticate")
    public Map<String, String> createAuthenticationToken(Authentication authentication) {
        String roles = authentication.getAuthorities().stream()
                .map(GrantedAuthority::getAuthority)
                .collect(Collectors.joining(" "));
        log.info("Utilisateur demandant le token : '{}'", authentication.getName());
        var token = tokenService.generateToken(authentication.getName(), roles);;
        log.info("Token généré : {}", token.get("accessToken"));
        return token;
    }
}
```

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 `Authorization`, choisir `Basic Auth`
- Saisir `admin` dans `Username`
- Saisir `admin` dans `Password`
- 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`

Spring Boot

Refresh Token

- Permet de régénérer un **Access Token**
- Ne doit contenir le mot de passe utilisateur ni ses rôles
- Doit contenir le strict minimum pour générer un **Access Token** : `subject`, `issuer`, `issuedAt` et `expiresAt`

Spring Boot

Modifions `TokenService` en ajoutant une méthode qui permet de retourner un access et un refresh tokens

```
public Map<String, String> generateTokens(String username, String roles) {
    var idToken = generateToken(username, roles);

    JwtClaimsSet jwtClaimsSet = JwtClaimsSet.builder()
    .issuedAt(Instant.now())
    .issuer("spring-boot")
    .expiresAt(Instant.now().plus(1, ChronoUnit.DAYS))
    .subject(username)
    .build();
    var token = jwtEncoder.encode(JwtEncoderParameters.from(jwtClaimsSet)).getTokenValue();
    idToken.put("refreshToken", token);
    return idToken;
}
```

Mettons à jour l'action du contrôleur

```
package com.example.demo.controller;

import java.util.Map;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.Authentication;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RestController;

import com.example.demo.service.TokenService;

import lombok.extern.slf4j.Slf4j;

@RestController
@Slf4j
public class JwtAuthenticationController {

    @Autowired
    private  TokenService tokenService;

    @PostMapping(value = "/authenticate")
    public Map<String, String> createAuthenticationToken(Authentication authentication) {

        String roles = authentication.getAuthorities().stream()
                .map(GrantedAuthority::getAuthority)
                .collect(Collectors.joining(" "));
        log.info("Utilisateur demandant le token : '{}'", user.getUsername());
        var token = tokenService.generateTokens(user.getUsername(), roles);
        log.info("Token généré : {}", token.get("accessToken"));
        return token;
    }
}
```

Spring Boot

Pour obtenir les jetons avec **Postman**

- Dans la liste déroulante, choisir `POST` puis saisir l'URL vers notre web service `http://localhost:8080/authenticate`
- Dans `Authorization`, choisir `Basic Auth`
- Saisir `admin` dans `Username`
- Saisir `admin` dans `Password`
- Cliquer sur `Send` puis vérifier la réception de deux jetons

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`

Spring Boot

Question

Comment savoir si la méthode `createAuthenticationToken` de `JwtAuthenticationController` doit générer ou rafraichir un jeton ?

Spring Boot

Question

Comment savoir si la méthode `createAuthenticationToken` de `JwtAuthenticationController` doit générer ou rafraichir un jeton ?

Réponse

Oauth recommande d'envoyer une clé `grant_type` dans la requête **HTTP** avec la valeur

- `password` lorsqu'on souhaite créer un **Access Token** à partir d'une connexion avec un `username` et un `password`
- `refresh_token` lorsqu'on souhaite regénérer un **Access Token** à partir d'un **Refresh Token**.

Spring Boot

Démarche

- Créer une classe `UserDto` : **D**ata **T**ransfer **O**bject
- Définir les attributs `username`, `password`, `grantType` et `refreshToken`
- Utiliser cette classe pour récupérer les données envoyées dans la requête utilisateur et vérifier si l'utilisateur souhaite avoir les jetons ou les rafraichir.

Spring Boot

Contenu de la classe `UserDto`

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

import lombok.Data;

@Data
public class UserDto {
    private String username;
    private String password;
    private String refreshToken;
    private String grantType;
}
```

Spring Boot

Commençons par intégrer la classe DTO dans notre contrôleur précédent

```
@RestController
@Slf4j
public class JwtAuthenticationController {

    @Autowired
    private TokenService tokenService;

    @Autowired
    private AuthenticationManager authenticationManager;

    @PostMapping(value = "/authenticate")
    public Map<String, String> createAuthenticationToken(@RequestBody UserDto user) {

        Authentication authentication = authenticationManager.authenticate(
            new UsernamePasswordAuthenticationToken(user.getUsername(), user.getPassword())
        );

        log.info("Utilisateur demandant le token : '{}'", authentication.getName());
        var token = tokenService.generateTokens(authentication);
        log.info("Token généré : {}", token.get("accessToken"));

        return token;
    }
}
```

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": "admin",
    "password": "admin"
}
```

- 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`

Spring Boot

Modifions l'action du contrôleur précédent pour permettre la génération d'un Access Token à partir d'un Refresh Token

```
@PostMapping(value = "/authenticate")
public Map<String, String> createAuthenticationToken(@RequestBody UserDto user) {

    if (!user.getGrantType().equals("refreshToken")) {
        Authentication authentication = authenticationManager.authenticate(
            new UsernamePasswordAuthenticationToken(user.getUsername(), user.getPassword())
        );
        String roles = authentication.getAuthorities().stream()
                .map(GrantedAuthority::getAuthority)
                .collect(Collectors.joining(" "));
        log.info("Utilisateur demandant le token : '{}'", user.getUsername());

        var token = tokenService.generateTokens(user.getUsername(), roles);
        log.info("Token généré : {}", token.get("accessToken"));
        return token;
    } else {
        log.info("refreshToken : '{}'", user.getRefreshToken());
        var token = tokenService.generateFromRefreshToken(user);
        log.info("Nouveau token généré : {}", token.get("accessToken"));
        return token;
    }

}
```

Spring Boot

Dans `TokenService`, injectons le `bean` permettant de décoder le Refresh Token

```
@Service
public class TokenService {

    @Autowired
    private JwtEncoder jwtEncoder;

    @Autowired
    private JwtDecoder jwtDecoder;

    @Autowired
    private UserDetailsService userDetailsService;

    // + les deux méthodes précédentes

}
```

Spring Boot

Ajoutons la méthode `generateFromRefreshToken` qui permet de régénérer les deux jetons à partir d'un Refresh Token

```
public Map<String, String> generateFromRefreshToken(UserDto user) {
    var decodedJwt = jwtDecoder.decode(user.getRefreshToken());
    var username = decodedJwt.getSubject();
    var connectedUser = userDetailsService.loadUserByUsername(username);
    String roles = connectedUser.getAuthorities().stream()
            .map(GrantedAuthority::getAuthority)
            .collect(Collectors.joining(" "));
    return generateTokens(username, roles);
}
```

Spring Boot

Pour obtenir le **Access Token** 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": "admin",
    "password": "admin",
    "grantType": "password"
}
```

- 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`

Spring Boot

Pour obtenir le **Access Token** depuis le **Refresh Token** 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

```
{
    "grantType": "refreshToken",
    "refreshToken": "coller votre refresh token ici"
}
```

- Cliquer sur `Send` puis copier le jeton

Spring Boot

Dans `application.properties`, ajoutons la propriété suivante pour activer la journalisation automatique pour Spring Security

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

Spring Boot

Ajoutons l'annotation `@Secured` à la méthode `GET` du contrôleur `PersonneRestController`

```
@RestController
@AllArgsConstructor
public class PersonneRestController {

        private PersonneRepository personneRepository;

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

        // + le reste du code

}
```

Spring Boot

Ajoutons l'annotation `@Secured` à la méthode `GET` du contrôleur `PersonneRestController`

```
@RestController
@AllArgsConstructor
public class PersonneRestController {

        private PersonneRepository personneRepository;

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

        // + le reste du code

}
```


Explication

- `@Secured("SCOPE_USER")` : rend l'accès à la route `/personnes` unique aux utilisateurs ayant le rôle `SCOPE_USER`
- **OAuth 2** utilise le terme `scope` pour désigner les rôles.

Spring Boot

Pour activer les annotations de sécurité, il faut annoter `SecurityConfig` **par** `EnableMethodSecurity`

```
@Configuration
@EnableWebSecurity
@EnableMethodSecurity(securedEnabled = true)
public class SecurityConfig {

    // contenu précédent

}
```

Spring Boot

Pour activer les annotations de sécurité, il faut annoter `SecurityConfig` par `EnableMethodSecurity`

```
@Configuration
@EnableWebSecurity
@EnableMethodSecurity(securedEnabled = true)
public class SecurityConfig {

    // contenu précédent

}
```


Explication

`securedEnabled = true` : pour activer l'annotation Spring `@Secured`

Spring Boot

Pour tester

- Connectez-vous avec `(admin, admin)` et vérifiez que vous avez accès aux `GET` et `POST`
- Connectez-vous avec `(user, user)` et vérifiez qu'un message d'erreur `403 Forbidden` lorsque vous essayez d'accéder aux `GET` et `POST`