EPI-Polytec
Ecole Privée Internationale d'Ingénieurs
المدرسة الدولية العليا الخاصة للمهندسين بسوسة

TP3 Sécuriser des API Rest avec JWT

Json Web Token (JWT)

Un JSON Web Token est un access token (jeton d'accès) qui permet un échange sécurisé de donnée entre deux parties. Il contient toutes les informations nécessaires sur une entité (utilisateur), ce qui rend la consultation d'une base de données inutile.

Il s'agit d'un objet JSON chiffré contenant un ensemble de revendications (claims) et une signature. Il est généralement utilisé dans le contexte d'authentification/autorisation (autre que basique) pour partager des informations relatives aux utilisateurs.

L'authentification avec JWT est un mécanisme d'authentification sans état (STATELESS) basé sur des jetons. Il est couramment utilisé comme session sans état côté client, ce qui signifie que le serveur n'a pas besoin d'enregistrer les informations de session.

Structure de JWT

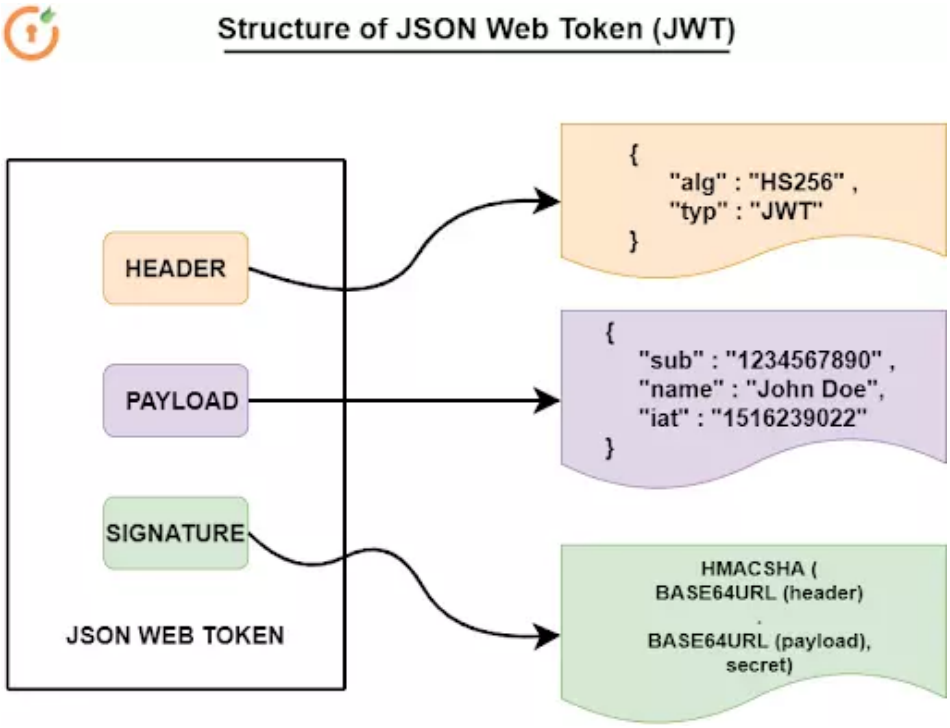
Un JWT chiffré se compose de trois parties codées en base64 et séparées par un point :

```
HEADER.PAYLOAD.SIGNATURE
```

Header ou entête : est en général composé de deux parties et fournit des informations essentielles sur le token. Il contient le type de token et l'algorithme de signature et/ou de chiffrement utilisé.

Payload ou charge utilie : La charge utile du JSON Web Token est la partie qui contient les informations qui doivent être transmises à l'application. C'est là que sont définis certains standards qui déterminent quelles données doivent être transmises. Les informations sont fournies en paire clé/valeur, les clés sont appelées « claims » dans les JWT.

Signature : La signature d'un JSON Web Token est créée grâce au codage base64 de l'en-tête et de la charge utile et la méthode de signature/cryptage spécifiée. Pour que la signature fonctionne, il est nécessaire d'utiliser une clé secrète connue uniquement de l'application source. Cette signature vérifie d'une part que le message ne sera pas modifié pendant le transfert.



Comment fonctionne un JSON Web Token ?

Pour comprendre le fonctionnement d'un JSON Web Token on se base sur l'exemple d'une connexion utilisateur pour s'authentifier. Une clé secrète est déterminée coté serveur avant l'utilisation du JWT. Dès qu'un utilisateur a transmis, avec succès, ses paramètres de connexion (`username` et `password`), le JWT est créée puis renvoyée vers l'utilisateur pour le stocker localement.

Lorsque l'utilisateur veut accéder à une ressource protégée (exemple une API Rest), le JWT sera envoyé par l'agent utilisateur comme en-tête d'autorisation (pour GET, POST, PUT, DELETE, ...) de la requête. Le serveur peut alors déchiffrer la signature du JWT avec sa clé secrète et si la validation a réussie, il exécute la demande.

Une implémentation JWT

À l'aide d'un exemple de JWT, on va découvrir à quoi ressemble le token final. Pour ce faire, nous utilisons l'exemple d'en-tête (Header) que nous avons mentionné au début :

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

Une charge utile (Payload) de JSON Web Token peut ressembler à ceci :

```
{
    "sub": "0123456789",
    "name": "Jean Dupont",
    "admin": true
}
```

Pour obtenir la structure réelle du JWT (trois parties séparées par des points), l'en-tête et la charge utile doivent être codés en `base64`. Pour l'en-tête, cela ressemble à ceci :

```
base64Header = base64Encode(header)
// eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
```

Il en va de même pour la charge utile :

```
base64Payload = base64Encode(payload)
// eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9
```

Il faut encore créer la signature. Dans l'en-tête, nous avons indiqué qu'il doit être signé avec HMAC-SHA256 :

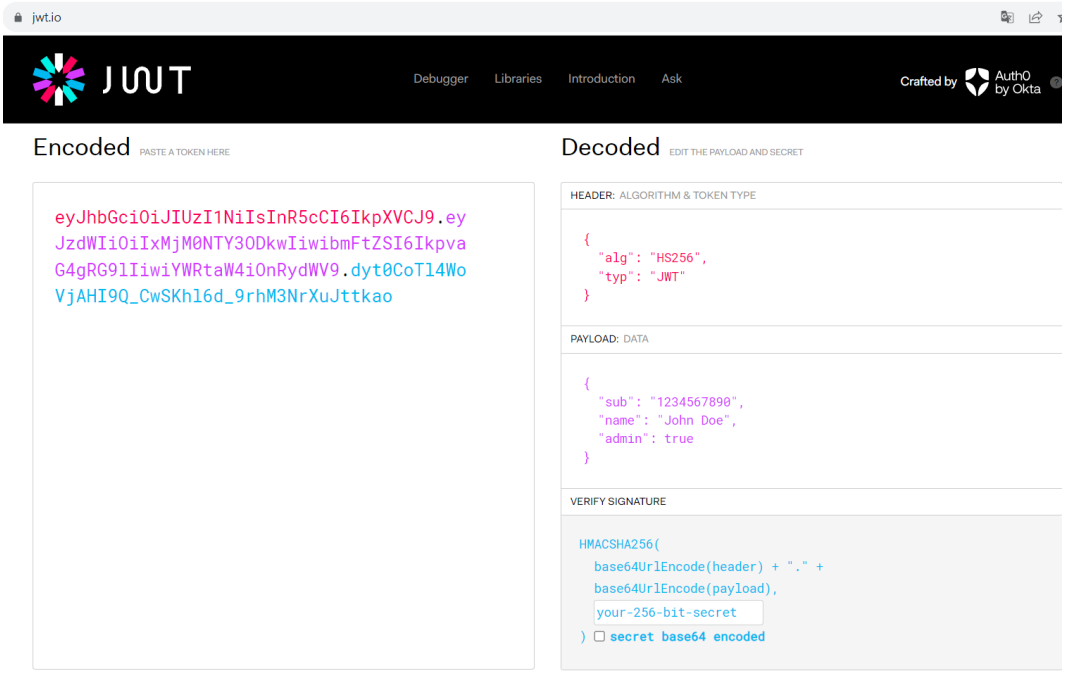
```
signature = HS256(base64Header + '.' + base64Payload, 'secret')
// dyt0CoTl4WoVjAHI9Q_CwSKhl6d_9rhM3NrXuJttkao
```

Pour finir, il faut rassembler ces trois composantes et les séparer par un point.

```
Token = base64Header + '.' + base64Payload + '.' + signature
// eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRt
```

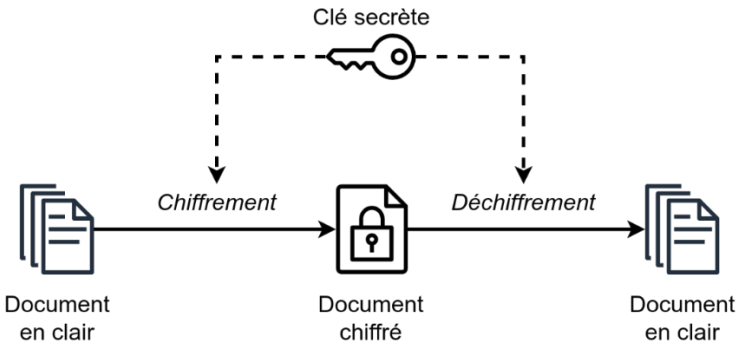
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ik
pvaG4gRG9lIiwiYWRtaW4iOnRydWV9.dyt0CoTl4WoVjAHI9Q_CwSKhl6d_9rhM3NrXuJttkao

Le site jwt.io permet de décoder le format JSON d'un JWT :



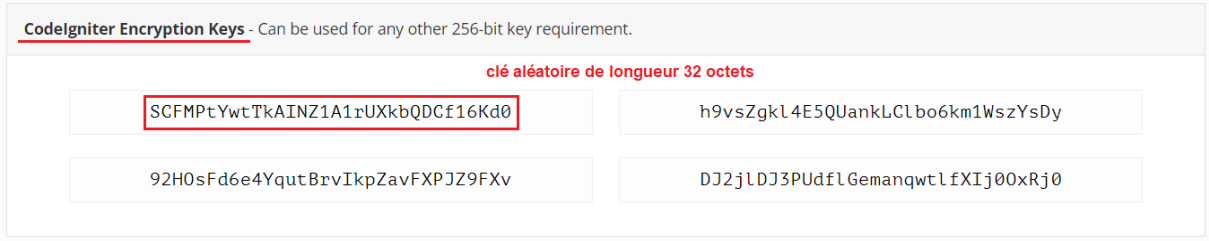
Le chiffrement symétrique

Le chiffrement symétrique (aussi appelé chiffrement à clé privée ou chiffrement à clé secrète) consiste à utiliser la même clé pour le chiffrement et le déchiffrement. L'expéditeur chiffre le message à l'aide de la clé et le destinataire déchiffre le message avec la même clé. L'expéditeur et le destinataire doivent conserver la clé privée pour que leur communication reste privée.



Plusieurs algorithmes sont utilisés pour le cryptage symétrique, dans notre TP nous allons utiliser HS512 (Hmac) à base d'une clé secrète d'une longueur de 64 octets.

Pour obtenir une valeur d'une clé secrète, aller vers le site `https://randomkeygen.com` et concaténer 2 clés de longueur 32 octets pour obtenir une clé de longueur 64 octets.



Enoncé du TP : Authentification avec JWT avec utilisateurs dans la BD

Objectif du TP3 : On va continuer sur le code du projet de TP2 qui consistait à sécuriser les API Rest en utilisant un username et password envoyé avec chaque requête HTTP vers un API, les utilisateurs étant définis dans la mémoire. L'objectif de TP3 est de sécuriser les API Rest avec JWT tout en ayant des utilisateurs enregistrés dans une table dans la BD On commerce par générer les tables `AppUser` et `AppRole` dans la BD, définir les services de registration et d'authentification puis générer le JWT.

Pour avoir un code structuré, on va commencer par créer un package nommé `security` dans lequel sera crée tout les packages invoqués par la suite.

Etape 1 : Création des entités AppUser et AppRole et de leurs Repository

Créer le package `entities` (sous `security`) et ajouter la définition d'une entité nommée `AppUser` qui comporte les attributs suivants : id – username – email – password et roles comme étant une liste des entités `AppRole` annoté par `@ManyToMany(fetch = FetchType.EAGER)`. Utiliser l'annotation `@Column` pour que le `username` soit unique et non null. Annotez la liste des rôles par `@JsonIgnore`. Définissez dans le même package l'entité `AppRole` contenant les attributs id et role qui doit être aussi unique et non null.

Créer un package `repositories` sous `security` et définissez une interface `AppUserRepository` qui comporte la définition d'une requête dérivée qui retourne un `AppUser` par son username. Définissez de même l'interface `AppRoleRepository` qui contient la signature d'une requête dérivée qui retourne un `AppRole` par son role.

Etape 2 : Création des classes Service

Créer le package `service` sous `security`. Définissez l'interface `IServiceAuthentication`

```
public interface IServiceAuthentication {
    1 usage  1 implementation
    public AppUser createAppUser(AppUser appUser);
    1 usage  1 implementation
    public AppRole createAppRole(AppRole appRole);
    1 usage  1 implementation
    public void addRoleToUser(String username, String role);
    1 usage  1 implementation
    public AppUser LoadUserByUserName(String username);
}
```

Définir la classe `ServiceAuthentication` qui implémente cette interface et injecte une dépendance sur les 2 interfaces repository. Dans la méthode `createAppUser` on doit assurer l'unicité du username avant de faire l'ajout et appliquer le cryptage du password avec

une instance de `passwordEncoder`. De même pour la création d'un objet `AppRole` on doit assurer l'unicité de l'attribut `role`.

Dans le même package définissez la classe `UserDetailsServiceImp` qui implémente l'interface prédéfinie `UserDetailsService` et injecte une dépendance sur la classe `ServiceAuthentication`. La classe doit redéfinir la méthode `loadUserByUsername` qui permettra de vérifier l'existence de l'utilisateur dans la BD (par son username) :

```
public class UserDetailsServiceImpl implements UserDetailsService {

    private final IServiceAuthentication iServiceAuthentication;
    no usages
    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        AppUser appUser= iServiceAuthentication.LoadUserByUserName(username);
        if(appUser==null) throw new UsernameNotFoundException("User with " + username + " does not exist");
        String[] roles = appUser.getRoles().stream().map(u -> u.getRole()).toArray(String[]::new);

        return User
                .withUsername(appUser.getUsername())
                .password(appUser.getPassword())
                .roles(roles)
                .build();
    }
}
```

Etape 3 : Création de AuthenticationController

Créer un package `controller` sous `security` dans lequel vous allez créer la classe `AuthenticationController`. Ce contrôleur Rest contiendra 3 actions reliées chacune avec la méthode `post` :

- Une première action qui ajoute un nouveau `AppUser`,
- Une deuxième qui ajoute un nouveau `AppRole`
- Une troisième qui ajoute un rôle existant à un user existant.

Etape 4 : Modification de la classe SecurityConfig

Déplacer la classe `SecurityConfig` vers un package `configuration` à créer sous le package `security`. Dans la classe `SecurityConfig` :

- Pour l'instant on va donner la permission (sans authentification) aux URL des 3 actions précédentes de `AuthenticationController`.
- Enlever le code de la méthode `inMemoryUserDetailsManager` puisque les utilisateurs seront désormais définis dans la BD

Démarrer le projet, remarquez la création de 3 nouvelles tables dans la BD : `appuser`, `approle` et `app_user_roles`.

Tester les API de ces 3 actions dans Swagger. Ajouter deux rôles dans la table `approle` : ADMIN et USER, créer 2 utilisateurs dans la table appuser avec username `admin` et `user` et attribuer à `admin` les 2 roles et à `user` le role USER seulement.

Etape 5 : Sécuriser les API avec JWT

Remplacer dans le fichier `pom.xml` la dépendance de `Spring Security Starter` par `oauth2 resource server` (à rechercher dans mvnrepository.com). Cette nouvelle dépendance contiendra toutes les ressources nécessaires pour la gestion des JWT.

Dans la classe `SecurityConfig`, injecter une dépendance vers la classe `UserDetailsServiceImpl`. Modifier dans la méthode `securityFilterChain` le fournisseur d'authentification comme étant JWT, l'obtention des détails d'un user se fera à travers la méthode `loadUserByUsername` de la classe `UserDetailsServiceImpl`.

```
//.httpBasic(Customizer.withDefaults())
.oauth2ResourceServer(oa -> oa.jwt(Customizer.withDefaults()))
.userDetailsService(userDetailsServiceImpl)
.build();
```

➢ Ajouter deux méthodes `Bean` dans cette classe de configuration :

** une de type `JwtEncoder` qui permet de créer et signer (chiffrer) le JWT au moment de l'authentification.

** une de type `JwtDecoder` qui intercepte une requête HTTP y récupère le JWT pour vérifier la signature (déchiffrement) du token.

Le chiffrement se fait par clé secrète dont la valeur (récupérée de `keygen.io`) est placée dans le fichier `application.propoerties` :

```
jwt-secret=cfcbe09106a3302b71bac5271564810e7c2f217c86218ce8b217006b0312707f
```

```
@Configuration
@EnableWebSecurity
@EnableMethodSecurity(prePostEnabled = true)
public class SecurityConfig {
    @Value("${jwt-secret}")
    private String secretKey;
```

Voici le code des 2 méthodes `jwtEncoder` et `jwtDecoder` :

```
@Bean
JwtEncoder jwtEncoder(){
   return new NimbusJwtEncoder(new ImmutableSecret<>(secretKey.getBytes()));
}

@Bean
JwtDecoder jwtDecoder(){
    SecretKeySpec secretKeySpec = new SecretKeySpec(secretKey.getBytes(), "RSA");
    return  NimbusJwtDecoder.withSecretKey(secretKeySpec).macAlgorithm(MacAlgorithm.HS512).build();
}
```

Dans la même classe de configuration on va définir un nouveau Bean `AuthenticationManager` pour indiquer le fournisseur d'authentification en précisant quel est l'encodage du mot de passe et la source des détails sur les utilisateurs :

```
@Bean
public AuthenticationManager authenticationManager(UserDetailsService userDetailsService){
    DaoAuthenticationProvider daoAuthenticationProvider = new DaoAuthenticationProvider();
    daoAuthenticationProvider.setPasswordEncoder(passwordEncoder());
    daoAuthenticationProvider.setUserDetailsService(userDetailsService);
    return new ProviderManager(daoAuthenticationProvider);
}
```

➢ Créer sous la package `controiller` de `security` une classe `SecurityController` qui contiendra 2 actions :

- `profile` qui affiche le profil de l'utilisateur authentifié (utilisé après une authentification)

- `login` qui traite la requête d'authentification par un `username` et `password` pour générer et retourner le JWT lorsque l'authentification est correcte.

```
@RestController
@RequiredArgsConstructor
@RequestMapping("/auth/")
public class SecurityController {

    private final AuthenticationManager authenticationManager;
    private final JwtEncoder jwtEncoder;
```

Ce contrôleur Rest injecte 2 dépendances : l'une sur un objet de type Bean `AuthenticationManager` et l'autre sur un objet de type `JwTEncoder` définies dans la classe `SecurityConfig`.

➢ Voici le code de l'action profile :

```
@GetMapping("/profile")
public Authentication profile(Authentication authentication){
    return authentication;
}
```

L'interface `Authentication` représente toutes les informations (usernane, autorités, …) sur l'utilisateur connecté.

➢ L'action login prend en arguments les 2 paramètres d'authentification `username` et `password` et retourne un map `clé : valeur` représentant la valeur du JWT.

```
@PostMapping("/login")
public Map<String, String> login(String username, String password){

}
```

Le premier traitement dans cette action consiste à valider le `username` et `password`. Si les paramètres entrés pour l'authentification sont valides et vérifiés, la méthode `authenticate` renvoie une instance d'authentification. Sinon, elle lancera une `AuthenticationException`. Pour ce dernier cas, elle renvoie null :

```
Authentication authentication = authenticationManager.authenticate(
        new UsernamePasswordAuthenticationToken(username,password)
);
```

Si l'authentification est correcte on va préparer le contenu du Payload du Token sous la forme d'un ensemble de revendications (claims), il contiendra :

`issueAt` : date de création du Token

`expiredAt` : date d'expiration du Token

`subject` : information principale sur l'utilisateur (username)

`claim` : les autorités (rôles) de l'utilisateur sous la forme d'une liste de rôles séparés par un espace

```
String scope = authentication.getAuthorities().stream().map(a->a.getAuthority()).collect(Collectors.joining(" "));
Instant instant = Instant.now();
JwtClaimsSet jwtClaimsSet = JwtClaimsSet.builder()
        .issuedAt(instant)
        .expiresAt(instant.plus(10, ChronoUnit.MINUTES))
        .subject(username)
        .claim("scope",scope)
        .build();
```

La dernière étape consiste à préparer les paramètres pour le chiffrement du Token avec l'algorithme HMac512 définit dans le Header du Token :

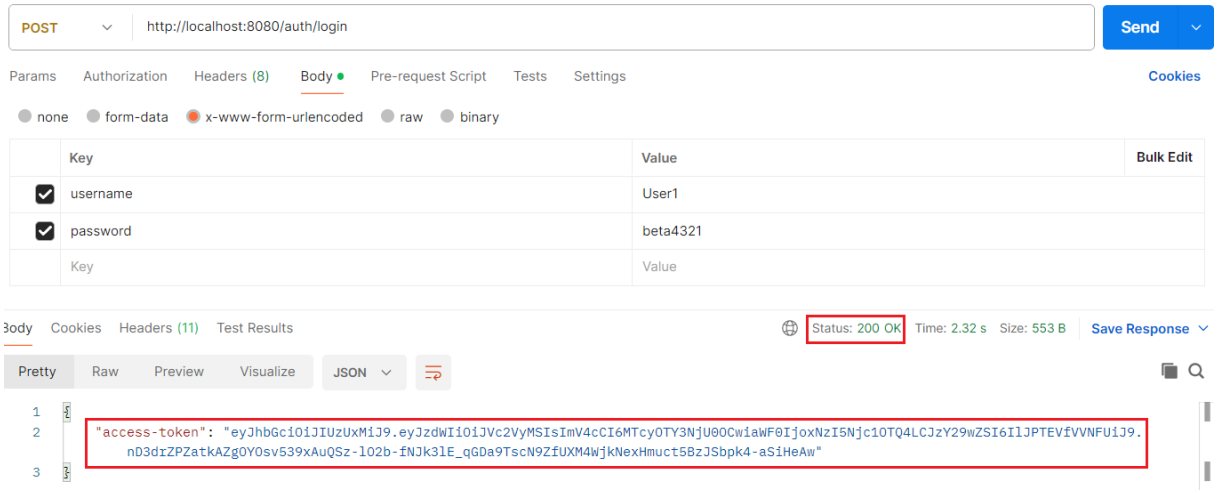
```
JwtEncoderParameters jwtEncoderParameters= JwtEncoderParameters.from(
        JwsHeader.with(MacAlgorithm.HS512).build(),
        jwtClaimsSet
);
```

Finalement avec la méthode `encode` du bean `jwtEncoder` on chiffre le payload du Token pour générer la signature puis on récupère la valeur du Token sous la forme d'une chaine puis on le renvoi comme valeur de retour de l'action `login` :

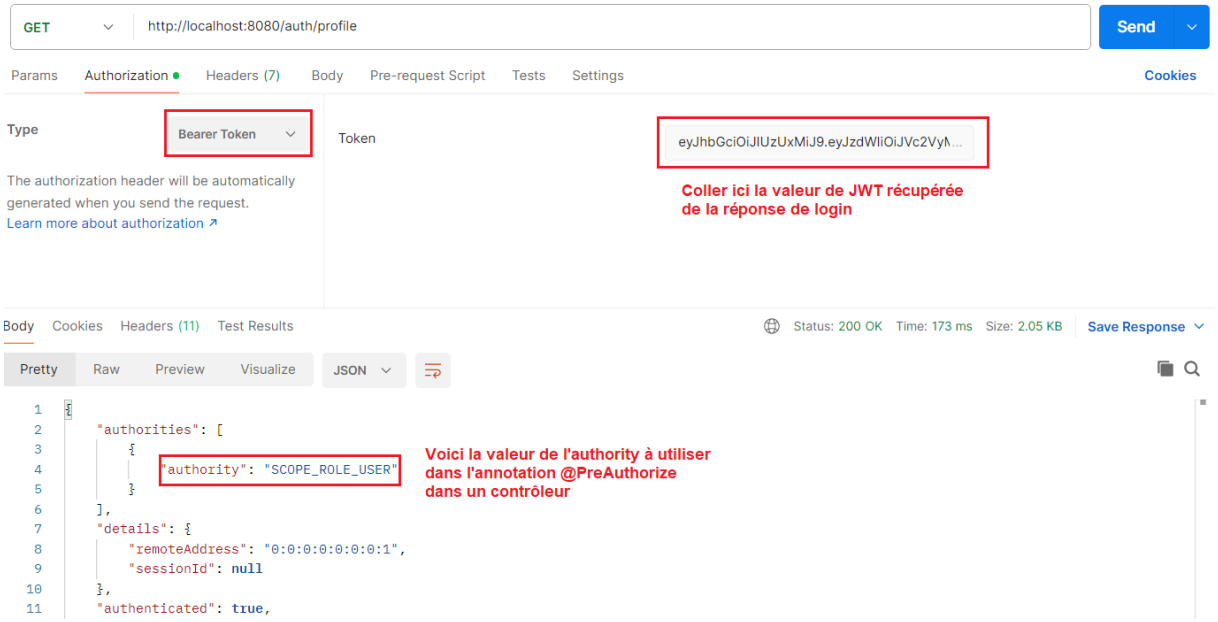
```
String jwt =  jwtEncoder.encode(jwtEncoderParameters).getTokenValue();
return Map.of("access-token", jwt);
```

Il faut revenir maintenant à la classe `SecurityConfig` pour autoriser les urls `/auth/login` et `/auth/register` pour tous.

Redémarrer l'application et exécuter l'URL d'authentification `/auth/login` avec l'utilisateur user, normalement vous devez générer le JWT :



Copier la valeur du JWT puis exécuter maintenant l'URL `/auth/profile` avec la méthode GET pour afficher le profil de l'utilisateur connecté, indiquer la valeur du Token comme Bearer :



Vous remarquez que l'autorité (authority) pour l'utilisateur devient `SCOPE_ROLE_USER` au lieu de `USER` (dans l'authentification Basique). De ce fait, il faut changer dans les `RestController` des patients, médecins et Rdv l'annotation d'autorisation sur les différentes actions. Par exemple :

```
@PostMapping("add")
@PreAuthorize("hasAuthority('SCOPE_ROLE_ADMIN')")
public Patient add(@RequestBody Patient patient){
    return iServicePatient.addPatient(patient);
}
```

Redémarrer l'application, puis faites une authentification avec user, exécuter l'action qui liste les rdvs (il est autorisé). Exécuter l'action d'ajout d'un rdv, il n'est pas autorisé : réponse 403. Faites une authentification avec admin, exécuter l'action qui liste les rdvs (il est autorisé). Exécuter l'action d'ajout d'un rdv, il est aussi autorisé.

JWT et Swagger

Voici comment configurer l'interface utilisateur de Swagger pour inclure JWT lorsqu'on appelle API Rest sécurisé.

Créer sous le package principale, un package nommé `Configuration`. Ajouter sous ce package une classe de configuration :

```
@Configuration
public class SwaggerConfiguration {
```

Définir dans cette classe une méthode Bean de type OpenAPI (une spécification ouverte pour la conception, la documentation et la consommation des API RESTful).

Cette méthode, créée une nouvelle instance de OpenAPI et utilise la méthode `addSecurityItem()` pour ajouter un `SecurityRequirement` qui spécifie que nous avons besoin d'une authentification de type `"Bearer Authentication"`. Cette authentification sera utilisée pour sécuriser les API. Ensuite, nous utilisons la méthode `components()` pour spécifier les composants de notre API. Nous utilisons la méthode `addSecuritySchemes()` pour ajouter un schéma de sécurité nommé `"Bearer Authentication"` en utilisant la méthode `createAPIKeyScheme()`

```
@Bean
public OpenAPI openAPI() {
    return new OpenAPI().addSecurityItem(new SecurityRequirement().
addList("Bearer Authentication"))
            .components(new Components().addSecuritySchemes("Bearer
Authentication", createAPIKeyScheme()));
}
```

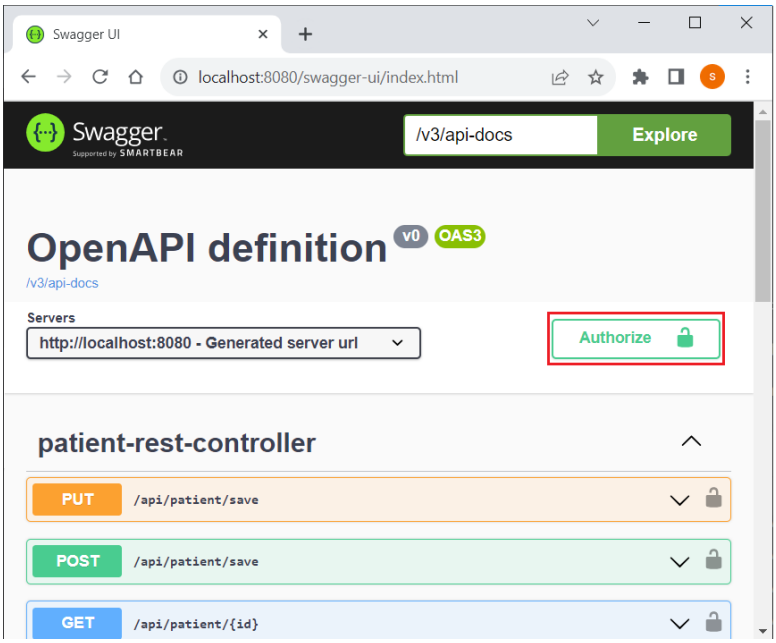
`createAPIKeyScheme` est une méthode privée qui crée un schéma de sécurité de type HTTP. Elle spécifie que le format de l'en-tête d'autorisation est "JWT" et que le schéma d'autorisation est de type "bearer".

```
private SecurityScheme createAPIKeyScheme() {
    return new SecurityScheme().type(SecurityScheme.Type.HTTP)
            .bearerFormat("JWT")
            .scheme("bearer");
}
```

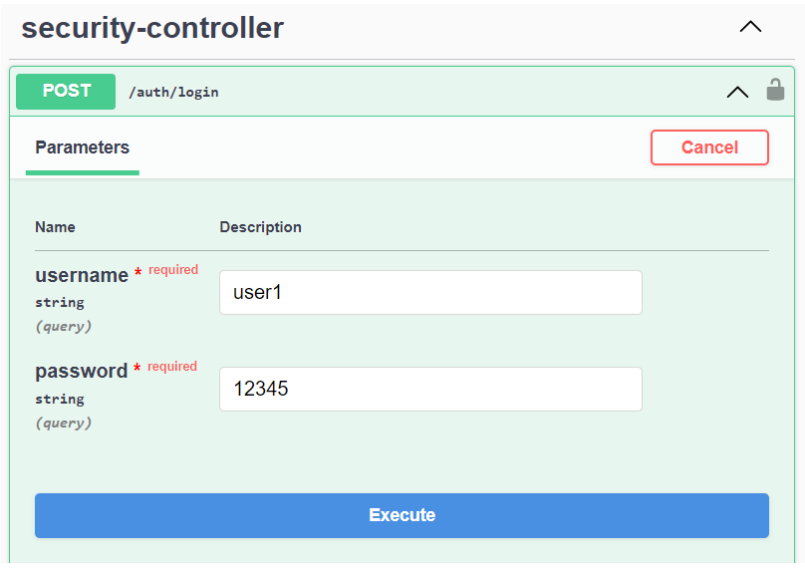
Délarer dans la classe `SecurityConfig` les 2 urls suivants comme étant autorisés sans authentification (permitAll) :

```
"/swagger-ui/**", "/v3/api-docs/**"
```

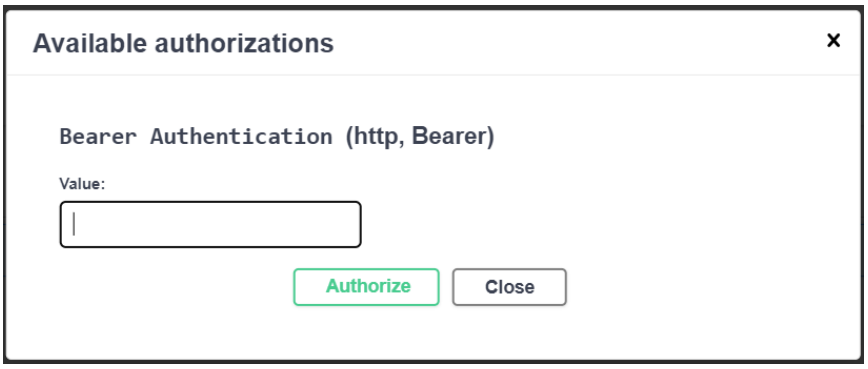
En redémarrant l'application et en accédant à l'interface de swagger, celle-ci fait apparaitre un lien pour saisir le JWT afin d'exploiter les API Rest sécurisé :



Commencer par faire une authentification pour générer un JWT :



Copier le JWT généré et cliquer sur `Authorize` pour saisir le code du JWT reçu de serveur :



La valeur JWT sera valable pour les différentes API. Pour changer de Token, choisir Logout :