



Une entreprise ordinaire à l'agilité extraordinaire

FORMATION JAVA SPRING ANGULAR

JWT

mohamed.el-babili@fms-ea.com

33+ 628 111 476

Version : 2.0

DMAJ : 03/07/23

Module : DEV-SPRI-002

Jwt & Spring

DEV-SPRI-002	Api Rest avec Spring	01 Connaitre les web et micro services
		02 Exploiter une Api en ligne à l'aide d'un client Rest
		03 Mise en œuvre d'une api Rest à l'aide de Spring
		04 Réalisation d'une appli cliente avec Angular exploitant une Api Rest
		05 Mise en œuvre de la sécurité d'une Api en utilisant Jwt

SOMMAIRE

- ▶ Introduction à Jwt
- ▶ Définition
- ▶ Jwt.io
- ▶ Mise en œuvre de la sécurité
- ▶ JwtAuthenticationFilter
- ▶ Obtention d'un token via un client Rest
- ▶ JwtAuthorizationFilter
- ▶ Test Complet
- ▶ Access Token et Refresh Token
- ▶ Pour aller plus loin
- ▶ Ressources

Introduction



JWT pourrait s'apparenter à ce badge qu'on vous remet à l'accueil d'une entreprise ultra sécurisée, une fois votre identité confirmé, un badge est configuré avec des droits + ou - restreints, ce badge équivaut au token

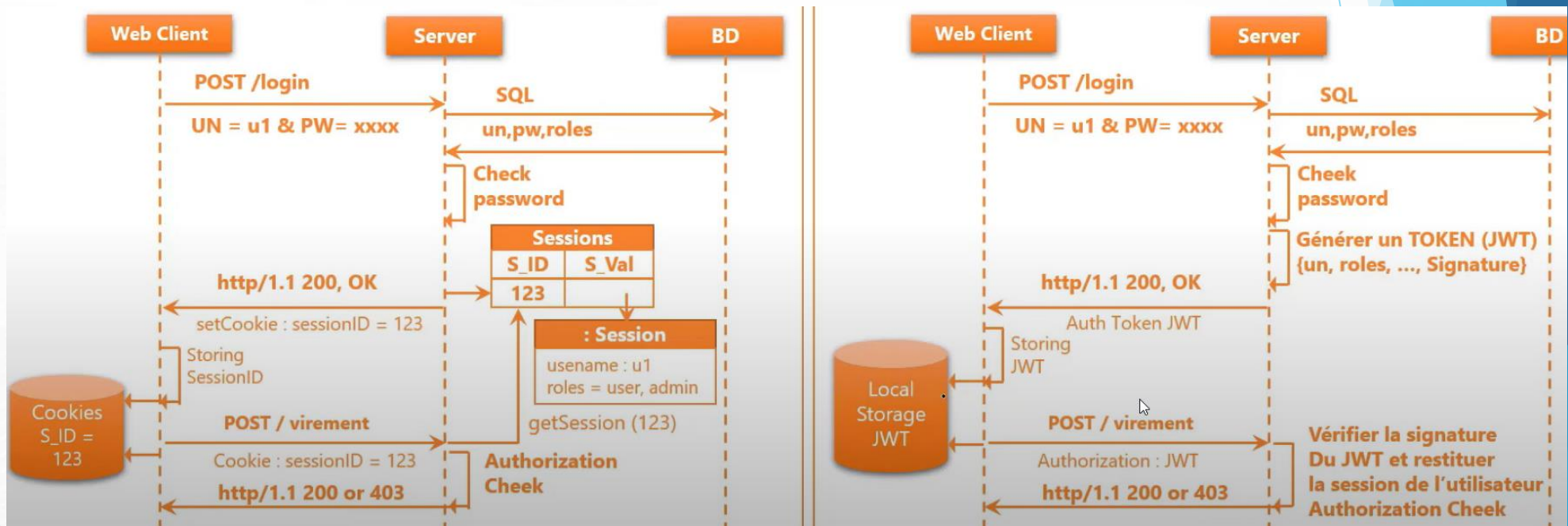
Cette solution est la plus répandue parce qu'elle est notamment exploité/reprise par d'autres applications distribuées comme youtube, gmail qui vont utilisés ce token pour vous permettre l'accès à leurs plateformes...

Sécurité basée sur les cookies et les sessions :
statefull

Les données de la session sont enregistrés côté serveur

Sécurité basée sur les tokens (Jwt) :
stateless

Les données de la session sont enregistrés dans un jeton envoyé au client



Jwt : Definition

JSON Web Token (JWT) est un standard ouvert défini dans la RFC 7519¹. Il permet l'échange sécurisé de jetons (tokens) entre plusieurs parties. Cette sécurité de l'échange se traduit par la vérification de l'intégrité et de *l'authenticité* des données. Elle s'effectue par l'algorithme [HMAC](#) ou [RSA](#).

Un jeton se compose de trois parties :

- ❑ Un *en-tête* (header), utilisé pour décrire le jeton (algo de cryptage). Il s'agit d'un objet JSON encodé.
- ❑ Une *charge utile* (payload) qui représente les informations embarquées dans le jeton sous forme de revendications classées en 3 catégories : Registered, public ou private claims. Il s'agit également d'un objet JSON encodé.
- ❑ Une *signature* numérique combinant le cryptage de [encodage de Header & payload + signature]

Header

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

Encodage Base 64 URL

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9

Payload

```
{
  "sub": "1234567890",
  "iss": "http://localhost:8080/auth",
  "aud": ["Web Front End", "Mobile App"],
  "exp": 54789005,
  "nbf": null,
  "iat": 49865432,
  "jti": "idr56543ftu8909876",
  "name": "med",
  "roles": ["admin", "author"]
}
```

Encodage Base 64 URL

eyJzdWUiOiIxMjM0NTY3ODkwIiwiaXNzIjoiQmFja2VuZCIsImF1dCI6WyJlbnR5cCI6IkpXVCJ9

Signature : RSA (2 clés publiques et privée) ou HMAC (1 clé privée)

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

V4FXAPpIBx1HqONU6qp7ptqz32RroIDlh4cVUQV0

La spécification de JWT propose différents champs (ou paramètres) standards, appelés *Claims*⁵ :

- **iss** : créateur (issuer) du jeton
- **sub** : sujet (subject) du jeton
- **aud** : audience du jeton
- **exp** : date d'expiration du jeton
- **nbf** : date avant laquelle le jeton ne doit pas être considéré comme valide (*not before*)
- **iat** : date à laquelle a été créé le jeton (*issued at*)
- **jti** : identifiant unique du jeton (*JWT ID*)


Tous ces paramètres sont optionnels. Ils permettent simplement de définir plus précisément un jeton et de renforcer sa sécurité (e.g. en limitant la durée de vie d'un jeton).




```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWUiOiIxMjM0NTY3ODkwIiwiaXNzIjoiQmFja2VuZCIsImF1dCI6WyJlbnR5cCI6IkpXVCJ9.eyJzdWUiOiIxMjM0NTY3ODkwIiwiaXNzIjoiQmFja2VuZCIsImF1dCI6WyJlbnR5cCI6IkpXVCJ9
```

NB : le **cryptage** (clé privée) ou chiffage permet de sécuriser l'échange de donnée à l'aide de clé secrète alors que **l'encodage** permet de convertir des données pour faciliter leur envoi ici.

Un token est généré par défaut avec les informations correspondantes

JWT

DebuggerLibrariesIntroductionAsk

Created by auth0
by CHAO

Encoded

PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG91IiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36Pk6yJV_adQssw5c
```

Type of token

Decoded

EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

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

PAYLOAD: DATA

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

VERIFY SIGNATURE

```
HMACSHA256(
  base64urlEncode(header) + ".",
  base64urlEncode(payload),
  your-256-bit-secret
) ☐ secret base64 encoded
```

Signature Verified

SHARE JWT

Il est possible de le personnaliser en modifiant payload + secret :

- La signature a changé
- Donc le token a changé

Algorithm

HS256

Encoded

PASTE A TOKEN HERE

Decoded

EDIT THE PAYLOAD AND SECRET

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJsYW1iZGEiLCJuYW1lIjoiiTW8iLCJpYXQiOiE1OTkyMzkwMjIsInJvbmVzIjpbIlVTRViiLCJBRE1JTjJdfQ.GBOUi0QzkWxc6HAWccRn9IntFIbK9mmDN8NZziYnHNU

HEADER: ALGORITHM & TOKEN TYPE

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

PAYLOAD: DATA

```
{
  "sub": "lambda",
  "name": "Mo",
  "iat": 1599239022,
  "roles": ["USER", "ADMIN"]
}
```

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  Marrakech_2023
)
```

Le génie ici vient de l'idée que si nous envoyons ce token à une personne qui veut le modifier pour des raisons obscures (ex : s'accorder plus de droits sur l'API), une fois qu'il nous le présente, nous pourrions vérifier la signature en la régénérant à partir de : header + payload + notre secret

Mise en œuvre de votre couche sécurité

Phase 1 : tout est permis

Step 1 : Création des utilisateurs et rôles

```
@Entity
@Data @AllArgsConstructor @NoArgsConstructor @ToString
public class AppUser {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(unique = true) // on veut que l'utilisateur soit unique
    private String username;

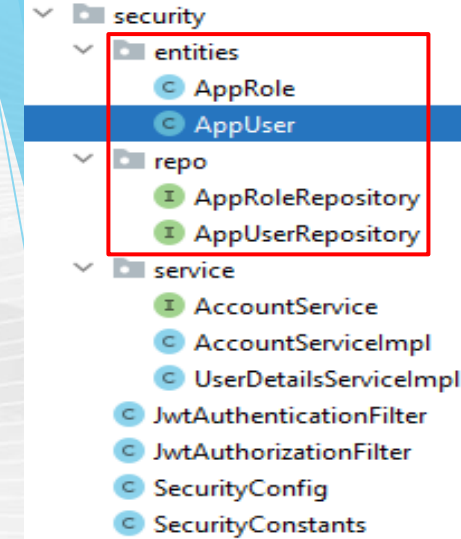
    @JsonProperty(access = JsonProperty.Access.WRITE_ONLY) // équivalent à JsonIgnore afin donc de ne pas renvoyer la valeur du mot de passe
    private String password;

    @ManyToMany(fetch = FetchType.EAGER) //on veut charger tous les roles de l'utilisateur
    private Collection<AppRole> roles = new ArrayList<>(); //dans ce cas, il faut impérativement initialiser notre collection
}
```

```
public interface AppUserRepository extends JpaRepository<AppUser, Long> {
    AppUser findByUsername(String username); //à partir du nom d'utilisateur, on retourne un objet
}
```

```
@Entity
@Data @AllArgsConstructor @NoArgsConstructor @ToString
public class AppRole {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String rolename;
}
```

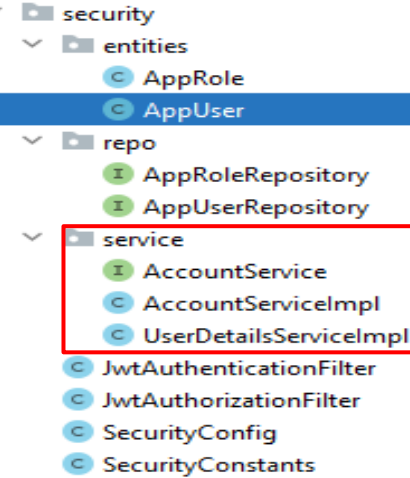
```
public interface AppRoleRepository extends JpaRepository<AppRole, Long> {
    AppRole findByRolename(String rolename); //renvoi à partir de roleName l'objet AppRole correspondant
}
```



Step 2 : les services associés

```
public interface AccountService {  
    public AppUser saveUser(AppUser user);  
    public AppRole saveRole(AppRole role);  
    public void addRoleToUser(String username, String rolename);  
    public AppUser findUserByUsername(String username);  
    ResponseEntity<List<AppUser>> listUsers();  
}
```

```
@Service @Transactional @Slf4j  
public class AccountServiceImpl implements AccountService {  
    @Autowired  
    private AppUserRepository appUserRepository;  
    @Autowired  
    private AppRoleRepository appRoleRepository;  
    @Autowired  
    private BCryptPasswordEncoder bCryptPasswordEncoder;  
  
    @Override  
    public AppUser saveUser(AppUser user) {  
        String hashPW = bCryptPasswordEncoder.encode(user.getPassword());  
        user.setPassword(hashPW);  
        log.info("Sauvegarde d'un nouvel utilisateur {} en base", user);  
        return appUserRepository.save(user);  
    }  
  
    @Override  
    public AppRole saveRole(AppRole role) {  
        log.info("sauvegarde d'un nouveau role en base");  
        return appRoleRepository.save(role);  
    }  
  
    @Override  
    public void addRoleToUser(String userName, String roleName) {  
        AppRole role = appRoleRepository.findByRolename(roleName);  
        AppUser user = appUserRepository.findByUsername(userName);  
        user.getRoles().add(role);  
        log.info("association d'un rôle à un utilisateur");  
        //puisque la méthode est transactionnel, des qu'il y a un commit, il y a ajout en base : role + user + table d'association  
    }  
  
    @Override  
    public AppUser findUserByUsername(String username) { return appUserRepository.findByUsername(username); }  
  
    @Override  
    public ResponseEntity<List<AppUser>> listUsers() { //ResponseEntity permet d'ajouter au corps des entetes et un état  
        return ResponseEntity.ok().body(appUserRepository.findAll());  
    }  
}
```



Step 3 : pour tester il faut configurer la sécurité à minima

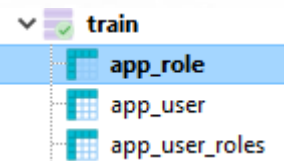
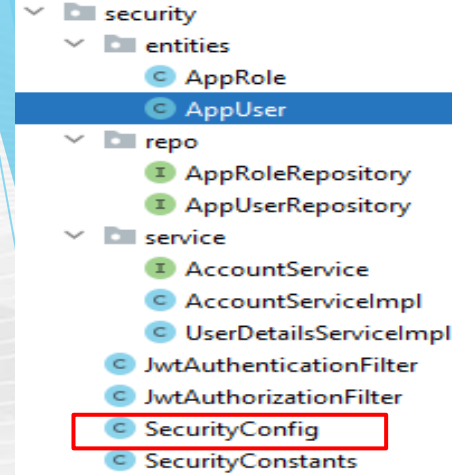
3.1 : Ajouter des utilisateurs, des rôles puis les associer

```
private void generateUsersRoles() {  
    accountService.saveUser(new AppUser( id: null, username: "papa", password: "1234", new ArrayList<>()));  
    accountService.saveUser(new AppUser( id: null, username: "mama", password: "1234", new ArrayList<>()));  
    accountService.saveRole(new AppRole( id: null, rolename: "ADMIN"));   
    accountService.saveRole(new AppRole( id: null, rolename: "USER"));   
    accountService.addRoleToUser( username: "papa", rolename: "ADMIN");   
    accountService.addRoleToUser( username: "papa", rolename: "USER");   
    accountService.addRoleToUser( username: "mama", rolename: "USER");   
}
```

Nb : dans le cas d'une base réelle, il faudra bien basculer le mode en update pour éviter les doublons

De plus, il faut ajouter dans le contexte de l'application l'objet qui servira à crypter les PWD

```
@Bean //sera exécuté par Spring au boot de l'appli delors on pourra l'injecter ailleurs (SecurityConfig par ex)  
public BCryptPasswordEncoder getBCryptPasswordEncoder() { return new BCryptPasswordEncoder(); }
```



id	rolename
1	ADMIN
2	USER

id	password	username
1	\$2a\$10\$qW2IN/AtUZuFLVzOCUm/1ehe8NmLJF75PX...	papa
2	\$2a\$10\$WiSK57ZftD1FIgAwS/CE.u4JqMf8ZI3TQAe1F...	mama

app_user_id	roles_id
1	1
1	2
2	2

3.2 : Configurer la sécurité

```
@Override  
protected void configure(HttpSecurity http) throws Exception {  
    http.authorizeRequests().anyRequest().permitAll();  
}
```

3.3 : Ajout d'un contrôleur puis tester

```
@CrossOrigin("*")
@RestController
public class AccountRestController {
    @Autowired
    AccountServiceImpl accountService;

    @GetMapping("/users")
    ResponseEntity<List<AppUser>> getUsers(){ return this.accountService.listUsers(); }

    @PostMapping("/users")
    public AppUser postUser(@RequestBody AppUser user) { return this.accountService.saveUser(user); }

    @PostMapping("/role")
    public AppRole postRole(@RequestBody AppRole role) { return this.accountService.saveRole(role); }

    @PostMapping("/roleUser")
    public void postRoleToUser(@RequestBody UserRoleForm userRoleForm) {
        accountService.addRoleToUser(userRoleForm.getUsername(),userRoleForm.getRolename());
    }
}

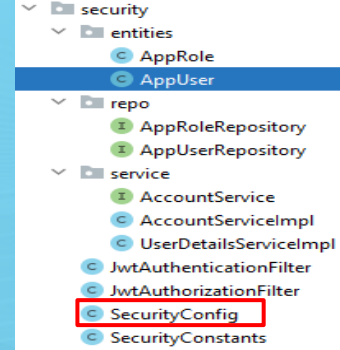
@Data
class UserRoleForm {
    private String username;
    private String rolename;
}
```

```
> C localhost:8080/users
// 20230629132908
// http://localhost:8080/users

[
  {
    "id": 1,
    "username": "papa",
    "roles": [
      {
        "id": 1,
        "rolename": "ADMIN"
      },
      {
        "id": 2,
        "rolename": "USER"
      }
    ]
  },
  {
    "id": 2,
    "username": "mama",
    "roles": [
      {
        "id": 2,
        "rolename": "USER"
      }
    ]
  }
]
```

Mise en œuvre de votre couche sécurité

Phase 2 : tout n'est pas permis



1. Reconfigurer la sécurité pour permettre aux seuls utilisateurs connectés d'avoir accès aux ressources

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests().anyRequest().authenticated(); //toutes les ressources nécessitent dorénavant une authentification
```

2. Récupérer les utilisateurs et leurs rôles

```
@Autowired
private UserDetailsService userDetailsService;

@Autowired
private AccountService accountService;
```

Dans cette autre approche, différente des 2 précédentes (InMemory ou Jdbc), à l'aide de l'objet UserDetailsService, nous indiquons à spring où il va rechercher les users et leurs rôles

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.userDetailsService(new UserDetailsServiceImpl() { //autre moyen pour demander à spring de récupérer les credentials d'un user
        @Override
        public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
            //c'est à nous de lui indiquer comment chercher un user
            AppUser user = accountService.findUserByUsername(username);
            Collection<GrantedAuthority> authorities = new ArrayList<>(); //conversion des rôles d'un user en grantedAuthorities dont spring a besoin
            user.getRoles().forEach( role -> {
                authorities.add(new SimpleGrantedAuthority(role.getRolename()));
            });
            return new User(user.getUsername(),user.getPassword(),authorities);
        }
    });
}
```

3. Il est possible de vérifier ici que cela fonctionne en demandant à spring de générer un formulaire d'authentification puis de tester l'accès aux ressources (mode statefull)

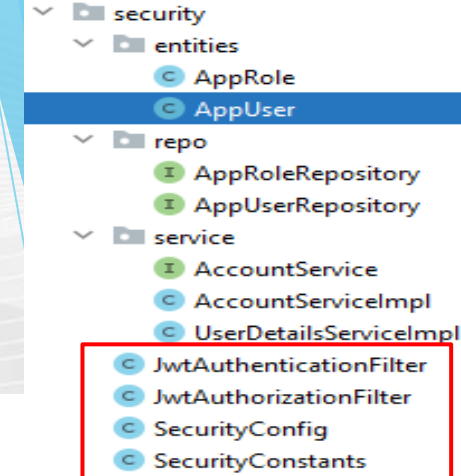
```
http.formLogin();
```



Please sign in

Mise en œuvre de votre couche sécurité

Phase 3 : mode stateless



1. Indiquer à Spring de basculer en mode stateless (le formulaire ne sert plus à rien)

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests().anyRequest().authenticated(); //toutes les ressources nécessitent une authentification
    http.csrf().disable(); //Désactiver la génération automatique du synchronized token pour le placer dans la session / protège des attaques csrf en mode statefull
    //Désactiver l'authentification basée sur les sessions -> demander à Spring d'utiliser le mode stateless
    http.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);
}
```

2. Mise en œuvre de l'authentification basée sur les tokens : Réalisation des 2 filtres dédiés

2.1 Filtre sollicité lors d'une demande d'authentification : générant un token

2.1.1 Tentative d'authentification (si succès alors 2.1.2)

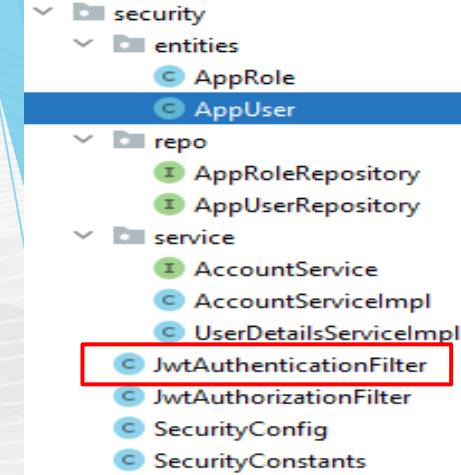
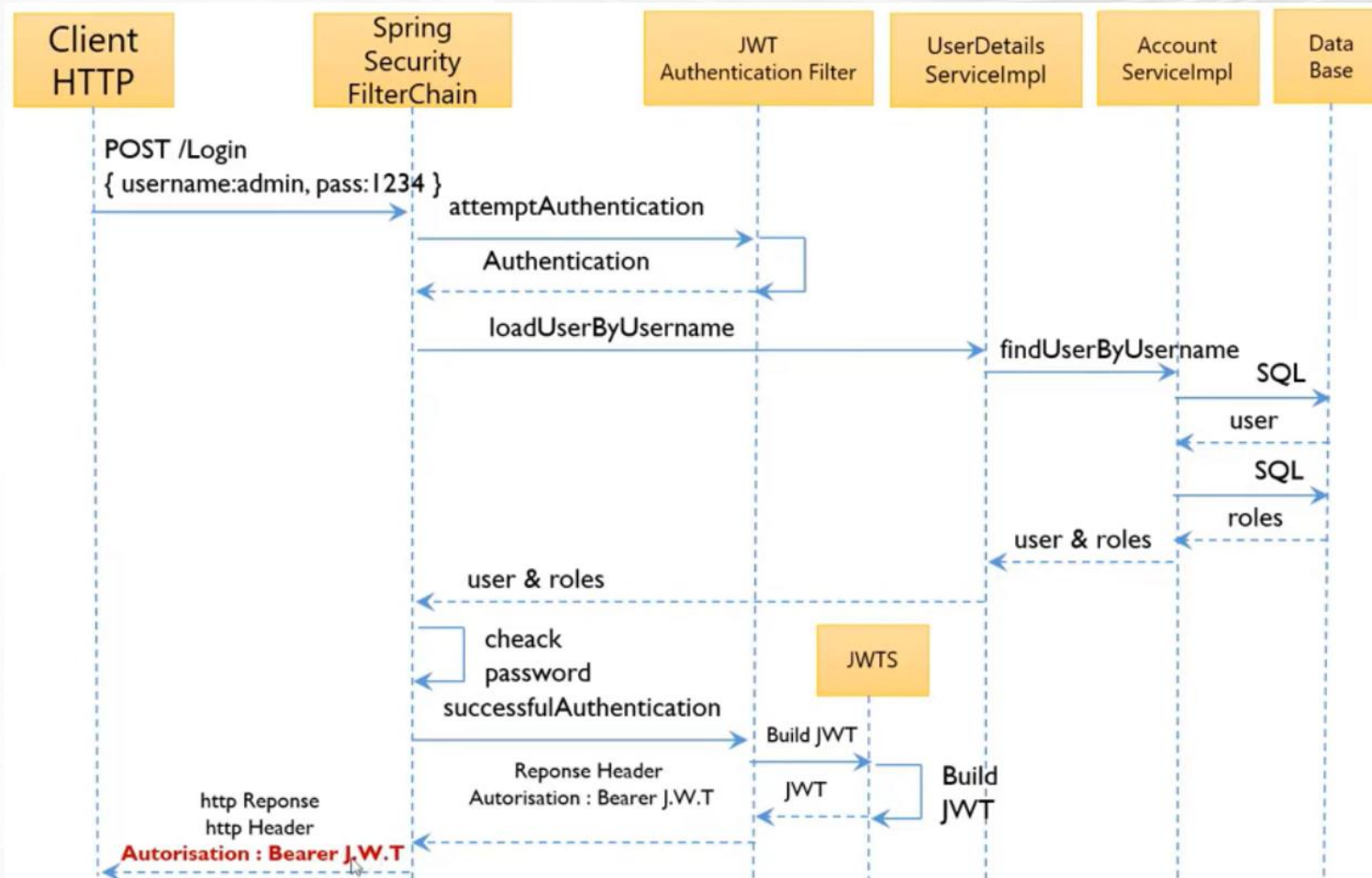
2.1.2 Authentification réussie (alors création du token et renvoi)

2.2 Filtre sollicité lors d'une demande d'autorisation avec un token, ce filtre est appelé en premier, il redéfinit la méthode `doFilterInternal` pour gérer 2 situations

2.2.1 le token existe déjà, on vérifie s'il est toujours bon pour accéder aux ressources

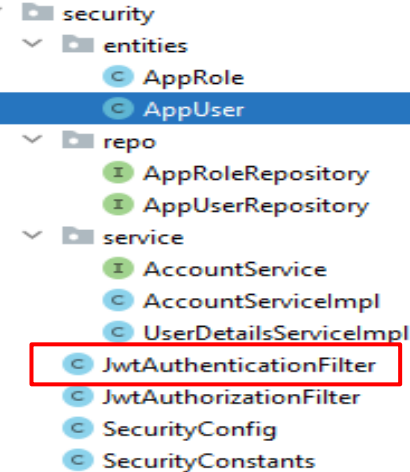
2.2.2 S'il n'est pas bon ou n'existe pas, on renvoi vers la phase d'authentification...

JwtAuthenticationFilter



JwtAuthenticationFilter

Pour ajouter ce filtre à spring security, il faut le créer en lui transmettant le bean de gestion de l'authentification dans SecurityConfig.



```
@Bean
@Override
public AuthenticationManager authenticationManagerBean() throws Exception {
    return super.authenticationManagerBean();
}
```

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests().anyRequest().authenticated(); //toutes les ressources
    http.csrf().disable(); //Désactiver la génération automatique du synchronizer
    //Désactiver l'authentification basée sur les sessions -> demander à Spring d'utiliser
    http.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);
    //ajout de notre filtre d'authentification
    http.addFilter(new JwtAuthenticationFilter(authenticationManagerBean()));
}
```

```
public class JwtAuthenticationFilter extends UsernamePasswordAuthenticationFilter {
    private AuthenticationManager authenticationManager;
```

```
public JwtAuthenticationFilter(AuthenticationManager authenticationManager) {
    this.authenticationManager = authenticationManager;
}
```

```
//méthode qui en charge la demande d'authentification
```

```
@Override
public Authentication attemptAuthentication(HttpServletRequest request, HttpServletResponse response) throws AuthenticationException {
    String username = request.getParameter(name: "username");
    String password = request.getParameter(name: "password");
    UsernamePasswordAuthenticationToken authenticationToken = new UsernamePasswordAuthenticationToken(username, password);
    //après avoir récupérer les credentials, on interroge spring, authentifié ou pas ?
    return authenticationManager.authenticate(authenticationToken);
}
```

```
// une fois authentifié spring security fait appel à cette méthode
```

```
@Override
protected void successfulAuthentication(HttpServletRequest request, HttpServletResponse response, FilterChain chain, Authentication authResult) throws IOException,
    User springUser = (User) authResult.getPrincipal(); //récupération des infos sur l'utilisateur connecté
```

```
//System.out.println(springUser);
```

```
//dès à présent, il faut créer le token Jwt pour l'utilisateur connecté :
```

```
String jwtToken = JWT.create()
    .withSubject(springUser.getUsername())
    .withExpiresAt(new Date(System.currentTimeMillis() + SecurityConstants.EXPIRATION_TIME)) //expiration du token dans 10 minutes
    .withIssuer(request.getRequestURL().toString()) //nom de l'appli qui a généré le token
    .withClaim(name: "roles", springUser.getAuthorities().stream().map(GrantedAuthority::getAuthority).collect(Collectors.toList()))
    //à partir de la liste de GrantedAuthority on récupère une liste de String contenant chaque role
    .sign(Algorithm.HMAC256(SecurityConstants.SECRET));
```

```
response.setHeader(SecurityConstants.HEADER_STRING, value: SecurityConstants.TOKEN_PREFIX + jwtToken); //renvoi une requete contenant : Autorisation + token
```

```
public class SecurityConstants {
    public static final String HEADER_STRING = "Authorization";
    public static final String SECRET = "elbab@gmail.com";
    public static final long EXPIRATION_TIME = 10 * 60 * 1000; //10 mn en ms = 10 * 60sec * 1000ms = 600000
    public static final String TOKEN_PREFIX = "Bearer ";
    public static final String ERROR_MSG = "error-message";
}
```

Ajouter cette dépendance pour JWT

```
<!-- https://mvnrepository.com/artifact/com.auth0/java-jwt -->
<dependency>
    <groupId>com.auth0</groupId>
    <artifactId>java-jwt</artifactId>
    <version>3.19.0</version>
</dependency>
```

- iss : créateur (issuer) du jeton
- sub : sujet (subject) du jeton
- aud : audience du jeton
- exp : date d'expiration du jeton
- nbf : date avant laquelle le jeton ne doit pas être considéré comme valide (not before)
- iat : date à laquelle a été créé le jeton (issued at)
- jti : Identifiant unique du jeton (JWT ID)

Obtention d'un token via un client Rest

POST

http://localhost:8080/login

HEADERS

BODY

AUTHORIZATION 0

ACTIONS 0

CONFIG

CODE SNIPPETS

application/x-www...

Name	Value
username	papa
password	1234

Response

Headers x

Raw

POST http://localhost:8080/login

Response headers

Authorization:

Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJwYXBhIiwicm9sZXMiOiIsiQURNSU4iLCJVV0VSIl0sImZcyI6Imh0dHA6Ly9sb2NhbGhv

TY4ODIxMTA3N30.itiXso01_8VvfhpcmObiNY7J-1NtSZIKAh-XFFdXWKQO

X-Content-Type-Options: nosniff

X-XSS-Protection: 1; mode=block

Cache-Control: no-cache, no-store, max-age=0, must-revalidate

Pragma: no-cache

Expires: 0

X-Frame-Options: DENY

Content-Length: 0

Date: Sat, 01 Jul 2023 11:21:17 GMT

Request headers

Content-Type: application/x-www-form-urlencoded

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJwYXBhIiwicm9sZXMiOiIsiQURNSU4iLCJVV0VSIl0sImZcyI6Imh0dHA6Ly9sb2NhbGhv

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJzZWIiOiJwYXBiIiwicm9sZXMiOiJsIURNSU4iLCJlVU0VSII0sImZcyI6Imh0dHA6Ly9sb2NhbGhvY2N0dA4MC9sb2dpbiIsImV4cCI6MTY0ODIzMtg0Nn0.rT4djocKrI-12Hwu9yD2k4kHE5W-F2SXFQqBar7gQis
```

```
HEADER: ALGORITHM & TOKEN TYPE

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

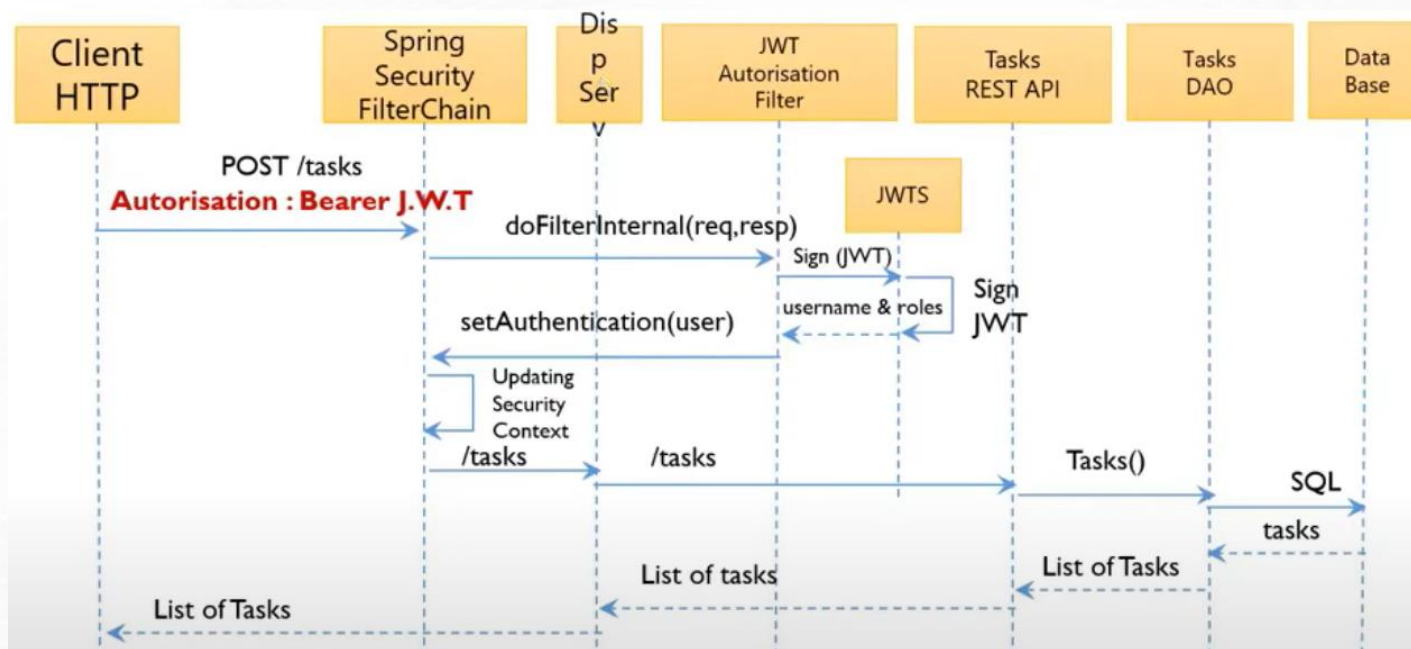
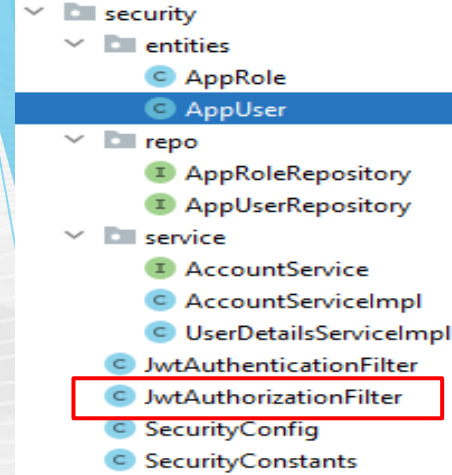
PAYLOAD: DATA

{
  "sub": "papa",
  "roles": [
    "ADMIN",
    "USER"
  ],
  "iss": "http://localhost:8080/login",
  "exp": 1688231846
}
```


JwtAuthorizationFilter

Lorsqu'on tente d'accéder à une ressource de notre Api, il faudra dorénavant présenter un token valide comme vu précédemment.

JwtAuthorizationFilter est chargé de faire cette vérification via la méthode `doFilterInternal` qui contrôle la signature du token présenté



JwtAuthorizationFilter

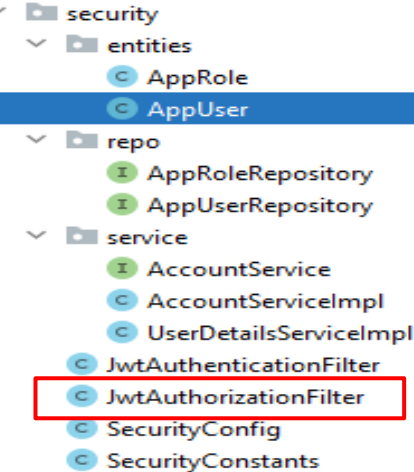
```
//ajout de notre filtre d'authentification
http.addFilter(new JwtAuthenticationFilter(authenticationManagerBean()));
```

```
//ajout de notre filtre d'autorisation qui sera appelé le 1er pour vérifier le token...
http.addFilterBefore(new JwtAuthorizationFilter(), UsernamePasswordAuthenticationFilter.class);
```

Type de filtre

Ce filtre est appelé en premier, il redéfinit la méthode `doFilterInternal` pour gérer 2 situations :

- le token existe déjà, on vérifie s'il est toujours bon pour accéder aux ressources.
- S'il n'est pas bon ou n'existe pas, on renvoi vers la phase d'authentification (filtre suivant)



Nb: cette notion de filtre est retrouvée dans de nombreux frameworks sous l'appellation de middleware

```
public class JwtAuthorizationFilter extends OncePerRequestFilter {
    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain) throws ServletException, IOException {
        String token = request.getHeader(SecurityConstants.HEADER_STRING); // récupération du token à partir de l'entête : Authorization

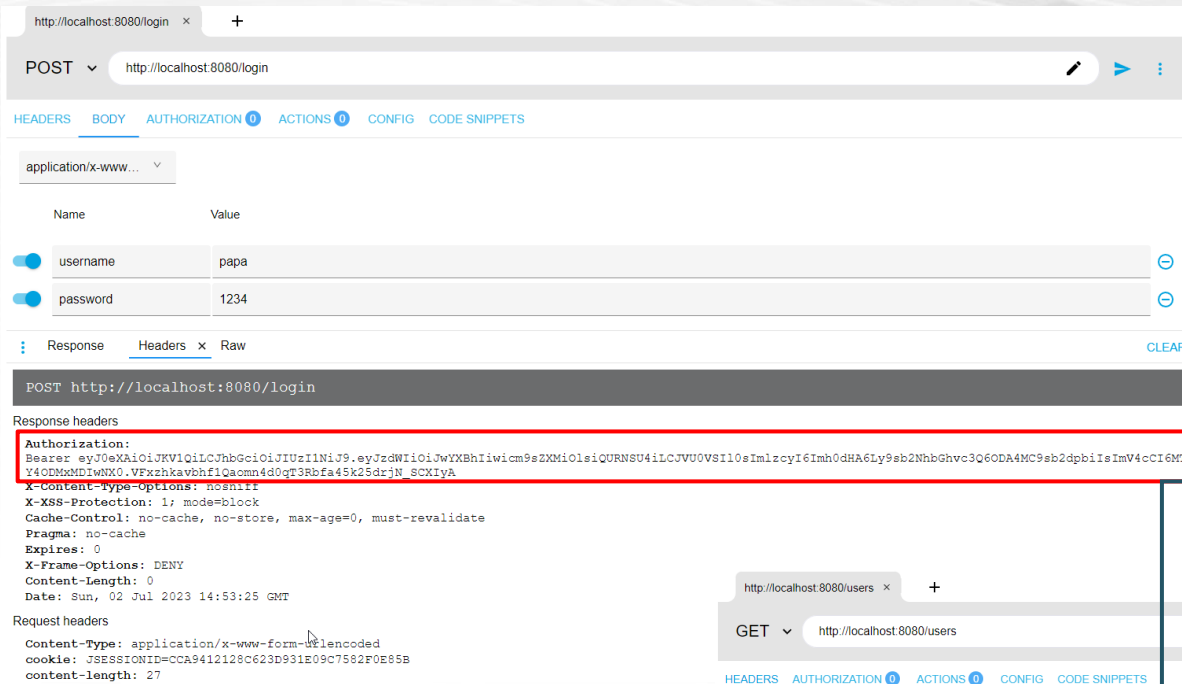
        if(token != null && token.startsWith(SecurityConstants.TOKEN_PREFIX)) {
            try { //génère des exceptions si pb sur le token : expiration / pb signature
                String jwtToken = token.substring(7);
                JWTVerifier jwtVerifier = JWT.require(Algorithm.HMAC256(SecurityConstants.SECRET)).build();
                DecodedJWT decodedJWT = jwtVerifier.verify(jwtToken); //vérification si les clés privées reçues et générées sont identiques

                String username = decodedJWT.getSubject();
                String[] roles = decodedJWT.getClaim("roles").asArray(String.class);
                Collection<GrantedAuthority> authorities = new ArrayList<>();
                for(String role : roles) authorities.add(new SimpleGrantedAuthority(role));

                UsernamePasswordAuthenticationToken authenticationToken = new UsernamePasswordAuthenticationToken(username, credentials: null, authorities);
                SecurityContextHolder.getContext().setAuthentication(authenticationToken);
                //authentification de l'utilisateur dans le contexte de Spring Security
            }
            catch(Exception e) { //en cas de pb, renvoi une requete http avec les messages d'erreurs dont 403
                response.setHeader(SecurityConstants.ERROR_MSG, e.getMessage());
                response.sendError(HttpServletResponse.SC_FORBIDDEN);
            }
        }

        filterChain.doFilter(request, response); //une fois les étapes terminées, renvoi vers le filtre suivant
    }
}
```

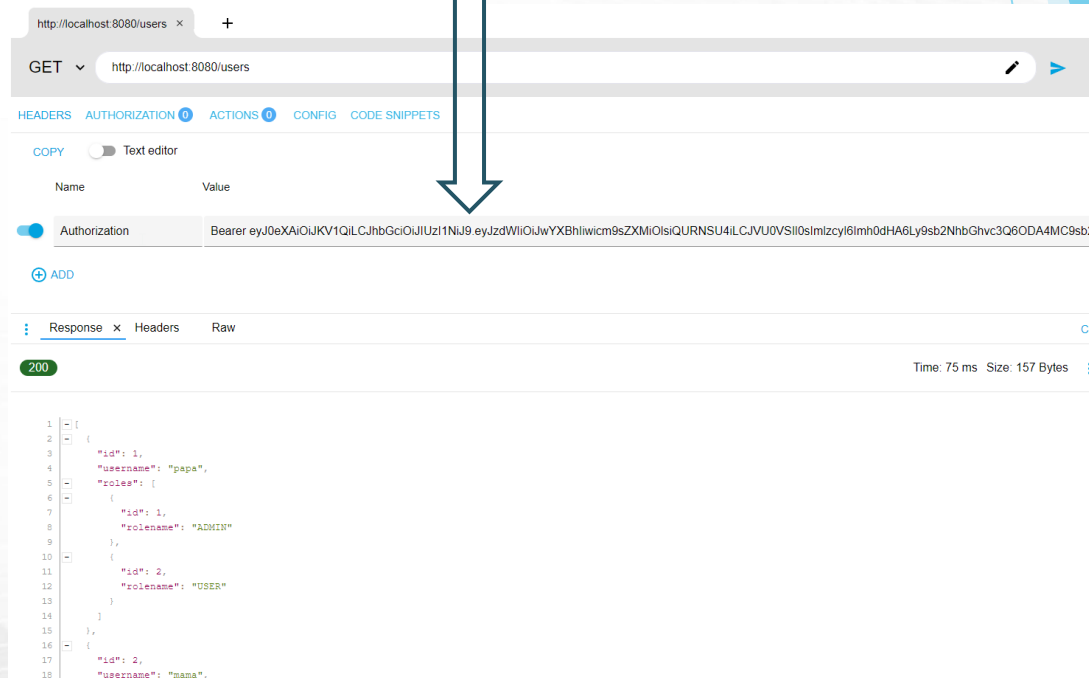
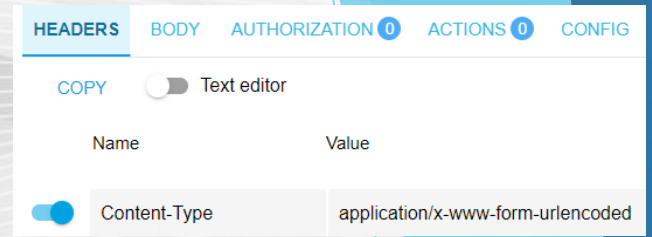
Test complet



La première requête (post) va servir à se connecter et recevoir un token

La seconde requête (get) va demander l'affichage des users en présentant le token qui vient d'être attribué

N'hésitez pas à mettre un point d'arrêt par étape sur votre Api pour suivre pas à pas le déroulé



Access Token et Refresh Token

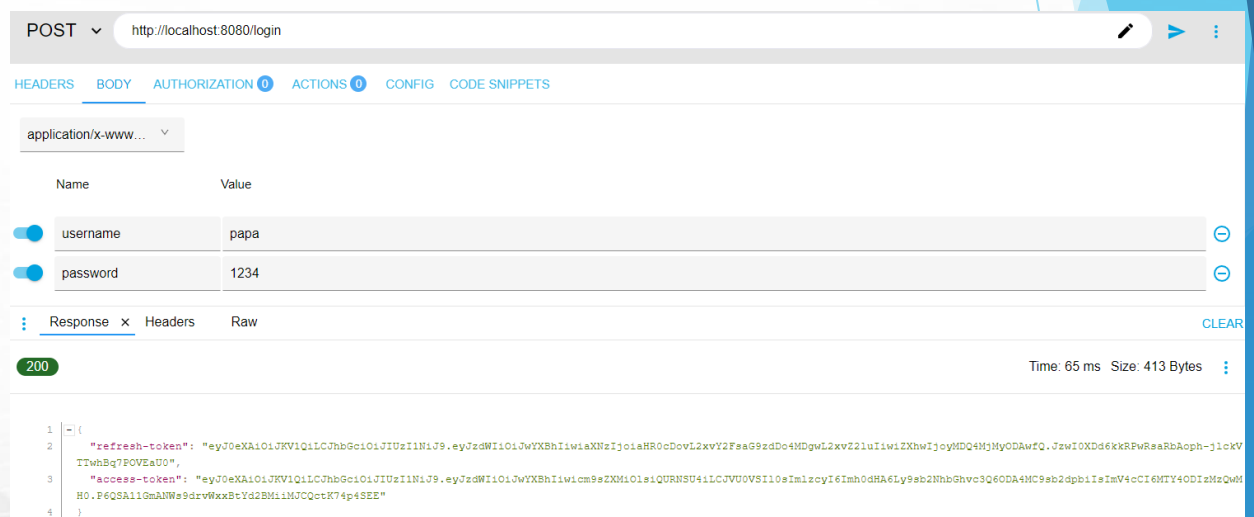
Le token d'accès utilisé jusqu'ici a une durée limitée à 10 minutes dans notre cas et afin de prolonger celle-ci, nous utiliserons le refresh token. Une fois présenté, il permettra d'obtenir un nouveau token d'accès

```
String jwtToken = JWT.create()
    .withSubject(springUser.getUsername())
    .withExpiresAt(new Date(System.currentTimeMillis() + SecurityConstants.EXPIRATION_TIME)) //expiration du token dans 10 minutes
    .withIssuer(request.getRequestURL().toString()) //nom de l'appli qui a généré le token
    .withClaim("roles", springUser.getAuthorities().stream().map(GrantedAuthority::getAuthority).collect(Collectors.toList()))
    //à partir de la liste de GrantedAuthority on récupère une liste de String contenant chaque role
    .sign(Algorithm.HMAC256(SecurityConstants.SECRET));

String refreshToken = JWT.create()
    .withSubject(springUser.getUsername())
    .withExpiresAt(new Date(System.currentTimeMillis() + SecurityConstants.EXPIRATION_TIME * SecurityConstants.EXPIRATION_TIME)) //expiration dans 100 minutes
    .withIssuer(request.getRequestURL().toString())
    .sign(Algorithm.HMAC256(SecurityConstants.SECRET));

Map<String, String> allTokens = new HashMap<>();
allTokens.put("access-token", jwtToken);
allTokens.put("refresh-token", refreshToken);
//injection dans la réponse http des 2 tokens au format Json
response.setContentType("application/json");
new ObjectMapper().writeValue(response.getOutputStream(), allTokens);

//response.setHeader(SecurityConstants.HEADER_STRING, SecurityConstants.TOKEN_PREFIX + jwtToken); //renvoi une requete contenant : Autorisation + token
```



NB : Cette étape n'est pas indispensable, vous pouvez passer directement à la suivante. Si vous souhaitez utiliser cette approche, il faudra demander un nouvel access token à partir du refresh token.

Pour aller plus loin

Différence entre Auth0 (organisation) et Auth2 (protocole)

OpenIDConnect (standard)

Keycloak (serveur de gestion d'authentification)

Cryptage & Encodage

Différence entre Hmac & Rsa

Next step : il n'y a plus qu'à se rendre dans votre application angular et gérer les tokens comme vu ici.

Vous devrez vous logger en présentant : username et password
Si tout va bien, vous obtiendrez un token qu'il faudra stocker et présenter à chaque requête le nécessitant.

En fonction des droits, vous aurez accès à tout ou partie de l'appli.

En cas de pb sur l'Api ou l'Appli, vous devrez informer l'utilisateur par des messages explicites...

Ressources

<https://openclassrooms.com/fr/courses/4668056-construisez-des-microservices>

<https://openclassrooms.com/fr/courses/4668056-construisez-des-microservices/7652911-documentez-votre-microservice-avec-swagger-2>

<https://youtu.be/3q3w-RT1sg0>

<https://youtu.be/VVn9OG9nfH0>

<https://auth0.com/>