esprit Se former autrement	Année Universitaire : 2021-2022 **Ateliers Services Web** **UP_WEB**

Atelier REST n°5
Sécuriser un service web RESTFul

Objectifs

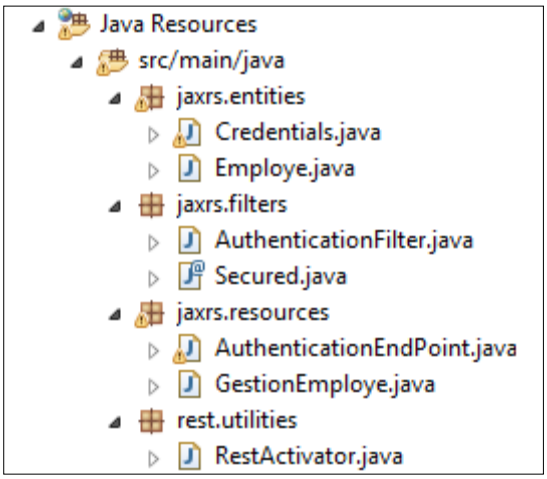
Le but de cet atelier est d'intégrer l'aspect sécurité aux services RESTful avec JAX-RS en utilisant une authentification basée sur les jetons (token-based authentication).Il s'agit d'utiliser les jetons (tokens) pour gérer les autorisations d'accès au service web.

Mise en place de l'environnement

- Notre objectif est de sécuriser le service web RestFul gestion des employés (Atelier 3-CRUD JAX-RS).
- Dans le fichier pom.xml du votre projet gestion des employés, ajoutez les dépendances suivantes:

```
<dependency>
        <groupId>javax.transaction</groupId>
        <artifactId>javax.transaction-api</artifactId>
        <version>1.2</version>
</dependency>
<dependency>
        <groupId>io.jsonwebtoken</groupId>
        <artifactId>jjwt</artifactId>
        <version>0.6.0</version>
</dependency>
<dependency>
        <groupId>javax</groupId>
        <artifactId>javaee-api</artifactId>
        <version>6.0</version>
        <scope>provided</scope>
</dependency>
```

- Le projet peut être structuré comme suit :



Principe

L'objectif est de sécuriser l'accès aux services à travers les jetons. Le processus d'authentification et d'autorisation se déroule ainsi :

1- Le client commence par envoyer son login et son mot de passe au serveur pour l'authentification.

2- Une fois l'authentification est validée, le serveur génère un jeton pour cet utilisateur authentifié.

3- Le serveur stocke ce jeton généré avec l'identifiant de l'utilisateur et sa date d'expiration.

4- Disposant de ce jeton, et durant la validité de ce dernier, le client a maintenant l'autorisation d'envoyer des requêtes au serveur. Pour chaque requête, le client doit envoyer son jeton.

5- En recevant une requête, le serveur récupère le jeton associé à cette dernière. A ce niveau, le serveur vérifie les détails de l'utilisateur et peut :

   a. accepter dans ce cas la requête, si le jeton est valide.

   b. refuser la requête, si le jeton n'est pas valide.

Implémentation

A. On commence par l'étape d'authentification et de génération du jeton.

Créez la ressource RESTful ***AuthenticationEndPoint.java*** comme suit :

```
@Path("authentication")
publicclassAuthenticationEndPoint {

	// ======================================
	// = Injection Points =
	// ======================================

	@Context
	privateUriInfouriInfo;

	@POST
	@Produces(MediaType.TEXT_PLAIN)
	@Consumes(MediaType.APPLICATION_FORM_URLENCODED)

	// @Produces(MediaType.APPLICATION_JSON)
	// @Consumes(MediaType.APPLICATION_JSON)

	//// this is a first alternative (with formParam)
	public Response authenticateUser(@FormParam("username") String username, @FormParam("password") String
password) {
		//// this is a second alternative (with the Credentials class)
		// public Response authenticateUser(Credentials cred) {
		try {

			// Authenticate the user using the credentials provided
			authenticate(username, password);
			// authenticate(cred.getUsername(), cred.getPassword());

			// Issue a token for the user
			String token = issueToken(username);
			// String token = issueToken(cred.getUsername());

			// Return the token on the response
			returnResponse.ok(token).build();

		} catch (Exception e) {
			returnResponse.status(Response.Status.FORBIDDEN).build();
		}
	}

	privatevoid authenticate(String username, String password) {
		// Authenticate against a database, LDAP, file or whatever
		// Throw an Exception if the credentials are invalid
		System.out.println("Authenticating user...");

	}

	private String issueToken(String username) {
		// Issue a token (can be a random String persisted to a database or a JWT token)
		// The issued token must be associated to a user
		// Return the issued token

		String keyString = "simplekey";
		Key key = newSecretKeySpec(keyString.getBytes(), 0, keyString.getBytes().length, "DES");
		System.out.println("the key is : " + key.hashCode());

		System.out.println("uriInfo.getAbsolutePath().toString() : " +
uriInfo.getAbsolutePath().toString());
		System.out.println("Expiration date: " + toDate(LocalDateTime.now().plusMinutes(15L)));

		String jwtToken =
Jwts.builder().setSubject(username).setIssuer(uriInfo.getAbsolutePath().toString())
				.setIssuedAt(new Date()).setExpiration(toDate(LocalDateTime.now().plusMinutes(15L)))
				.signWith(SignatureAlgorithm.HS512, key).compact();
```

```
                System.out.println("the returned token is : " + jwtToken);
                returnjwtToken;
        }

        // =====================================
        // = Private methods =
        // =====================================

        private Date toDate(LocalDateTimelocalDateTime) {
                return Date.from(localDateTime.atZone(ZoneId.systemDefault()).toInstant());
        }
}
```

La méthode **authenticate(username, password)**, prend en paramètre le login et le mot de passe de l'utilisateur pour s'authentifier. Ces paramètres sont récupérés à travers les@FormParam de la requête POST. On peut également envoyer ces paramètres sous la forme d'un objet de type **Crédentials** qu'on définira ainsi:

```
publicclassCredentialsimplements Serializable {

        private String username;
        private String password;

        public String getUsername() {
                returnusername;
        }

        publicvoidsetUsername(String username) {
                this.username = username;
        }

        public String getPassword() {
                returnpassword;
        }

        publicvoidsetPassword(String password) {
                this.password = password;
        }
```

La méthode **authenticateUser** acceptera dans ce cas, au lieu des **formParam** un objet **credentials** :

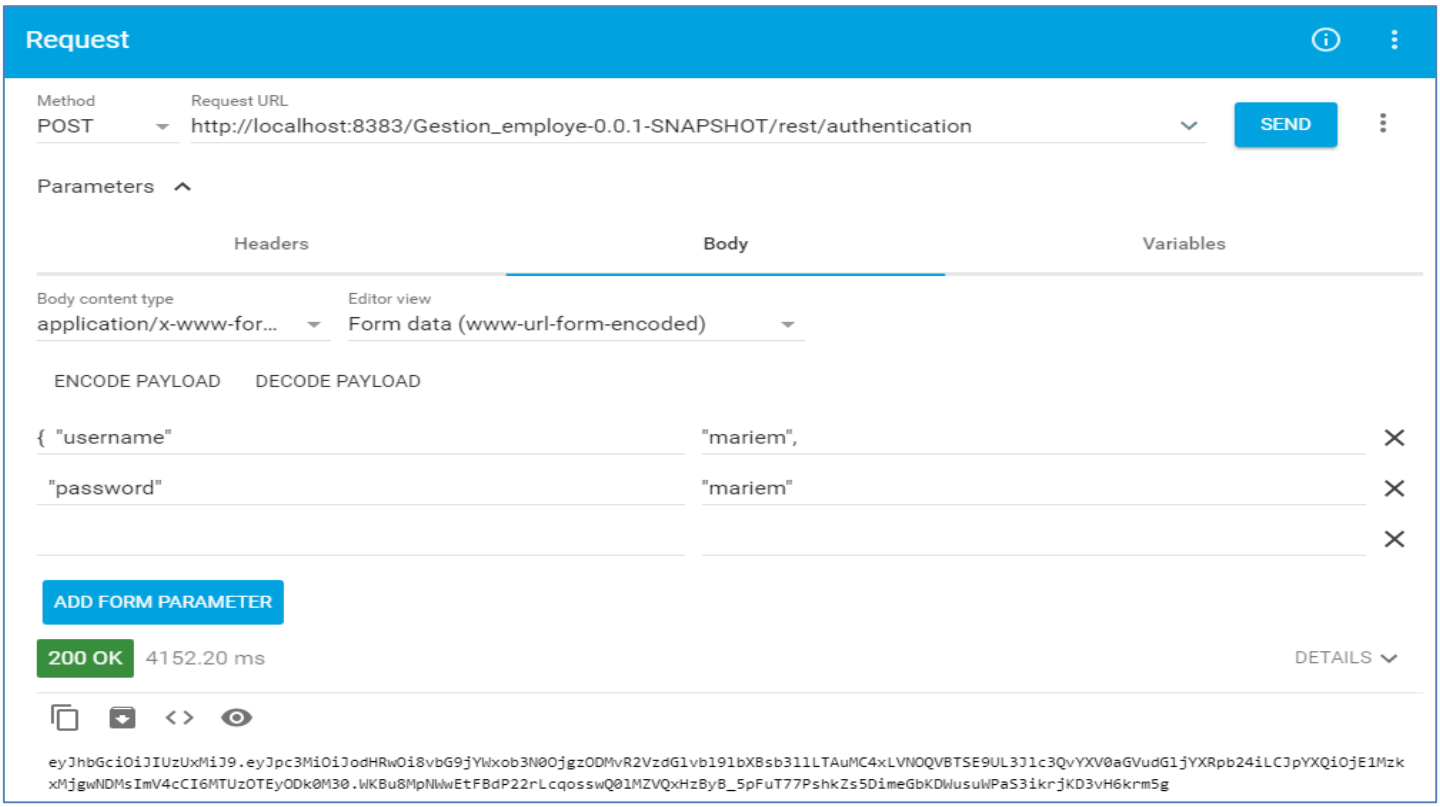
```
        @POST
        // @Produces(MediaType.TEXT_PLAIN)
        // @Consumes(MediaType.APPLICATION_FORM_URLENCODED)

        @Produces(MediaType.APPLICATION_JSON)
        @Consumes(MediaType.APPLICATION_JSON)

        //// this is a first alternative (with formParam)
        // public Response authenticateUser(@FormParam("username") String username,
        //// @FormParam("password") String password) {
        //// this is a second alternative (with the Credentials class)
        public Response authenticateUser(Credentials cred) {
                try {

                        // Authenticate the user using the credentials provided
                        // authenticate(username, password);

                                authenticate(cred.getUsername(), cred.getPassword());
                ...
```



Dans ce cas, le client envoie les données sous la forme JSON suivante :

```
{
        "username" : "Mariem",
        "password" : "mariem"
}
```

Dans le cadre de cet atelier, cette phase d'authentification est laissée au choix de l'étudiant. Il peut s'agir de vérifier les coordonnées envoyées dans une base de données d'utilisateurs, LDAP, ou fichier par exemple.

Une fois le client authentifié, La méthode **issueToken(username)** génère un jeton associé à cet utilisateur. La génération du jeton, effectuée avec le JWT, contient plusieurs arguments (le **subject** ou le **username**, la **date d'expiration**, et une **clé** générée hashée avec l'algorithme HS512. Ce jeton généré est envoyé au client.

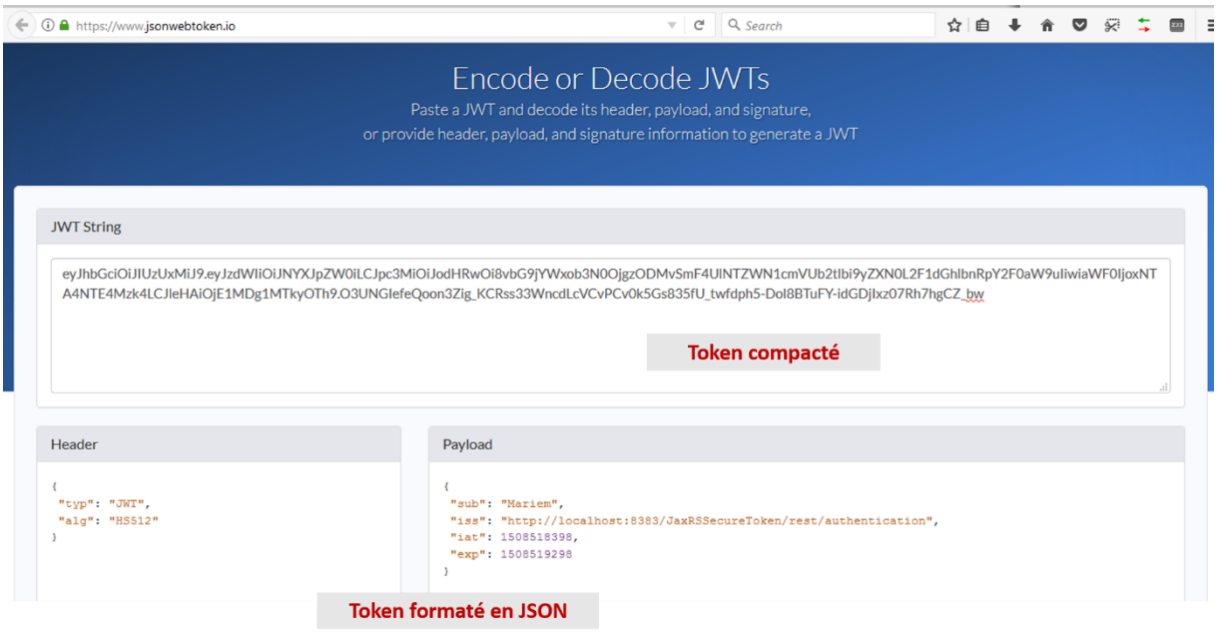
- **SignatureAlgorithm.*HS512:***

Le type "**SignatureAlgorithm**" est une énumération qui définit des représentations des noms des "algorithmes de signature" comme défini dans la spécification "**Json Web Algorithms (JWA)**". ***HS512*** est le nom de l'algorithme HMAC (keyed-Hash Message Authentication Code) utilisant le **SHA-512** (Secure Hash Algorithm - 512 : un standard de traitement de l'information).

(Pour plus de détails sur ces algorithmes, consulter le lien suivant :https://tools.ietf.org/html/rfc7518#section-3.2).

On teste par la suite le service web d'authentification en envoyant la requête POST avec le *path* associé à ce service web comme suit :

Le token généré est affiché avec un format compacté (grâce à la fonction **.compact()** ci-dessus). À la base, le token est un objet Json (d'où le nom JWT(Json Web Token)) comme le montre la figure suivante:



B. Disposant de ce jeton, le client peut maintenant envoyer des requêtes. A ce niveau, il est autorisé à consommer les ressources du serveur, jusqu'à expiration de son jeton.

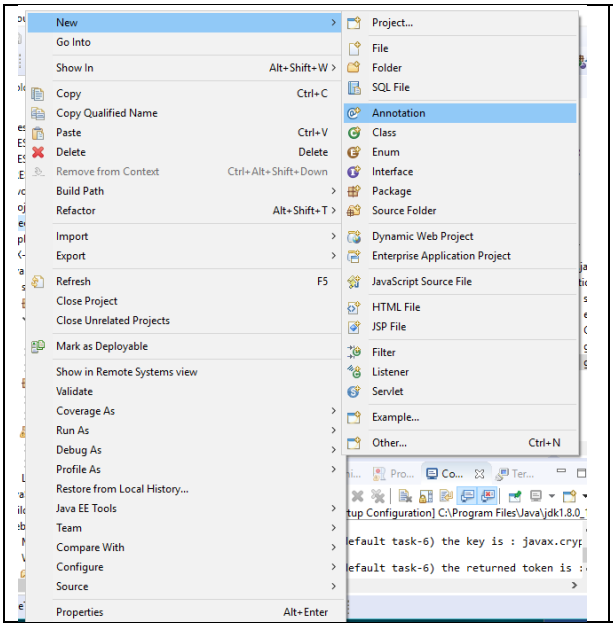
Ce jeton sera envoyé dans la requête dans son champs**« Authorization »** du Header.

Nous avons ici besoin d'un filtre, qui sera appliqué à chaque appel d'une ressource **sécurisée**.

Pour le cas de la ressource **getListeEmployes** de la ressource ***GestionEmployeResource***, ajoutez l'annotation ***@Secured*** pour la sécuriser.

En effet, l'annotation ***@Secured*** n'est pas une annotation prédéfinie. Nous devons ainsi la définir.

Créez l'annotation **« Secured ».** On aurait pu lui attribuer tout autre nom.




```java
@NameBinding
@Retention(RetentionPolicy.RUNTIME)
@Target({ ElementType.TYPE, ElementType.METHOD })

public@interfaceSecured {
}
```

JAX-RS offre unemeta-annotation **@NameBinding**qui permet de créer d'autres annotations pour le binding des filtres et des intercepteurs des ressources (classes et méthodes).

Après la déclaration de cette nouvelle annotation **(@Secured**), cette dernière doit précéder en plus une classe filtre implémentant la classe **ContainerRequestFilter.** En effet, cette classe intercepte la requête avant l'appel de la ressource, et la méthode filtre sera appelée à chaque interception.

Ainsi, on définit la classe **« AuthenticationFilter »,** implémentant l'interface**ContainerRequestFilter**, qui sera appelée lors de l'interception de la requête. Cette classe effectue le traitement suivant :

- Vérifie si la requête est de type **token-based-authentication**, c'est-à-dire que le Header **Authorization** est valide (préfixé par la chaine **"Bearer"**) ;
- extrait le token de la requête qui est la chaine de caractère suivant le « Bearer »
- valide le token avec JWT (la méthode *validateToken(String token))*

  pour ensuite donner l'autorisation au client si le token est valide.

```java
@Secured
@Provider
@Priority(Priorities.AUTHENTICATION)
publicclassAuthenticationFilterimplementsContainerRequestFilter {
        privatestaticfinal String AUTHENTICATION_SCHEME = "Bearer";

        // ======================================
        // = Injection Points =
        // ======================================

        ContainerRequestContextrequestContext;
```

```
        @Override
        publicvoid filter(ContainerRequestContextrequestContext) throwsIOException {

                System.out.println("request filter invoked...");

                // Get the Authorization header from the request
                String authorizationHeader =
requestContext.getHeaderString(HttpHeaders.AUTHORIZATION);

                // Validate the Authorization header
                if (!isTokenBasedAuthentication(authorizationHeader)) {
                        abortWithUnauthorized(requestContext);
                        return;
                }

                // Extract the token from the Authorization header
                String token = authorizationHeader.substring(AUTHENTICATION_SCHEME.length()).trim();

                try {

                        // Validate the token
                        validateToken(token);

                } catch (Exception e) {

        requestContext.abortWith(Response.status(Response.Status.UNAUTHORIZED).build());
                }
        }

        privatebooleanisTokenBasedAuthentication(String authorizationHeader) {

                // Check if the Authorization header is valid
                // It must not be null and must be prefixed with "Bearer" plus a whitespace
                // Authentication scheme comparison must be case-insensitive
                returnauthorizationHeader != null

        &&authorizationHeader.toLowerCase().startsWith(AUTHENTICATION_SCHEME.toLowerCase() + " ");
        }

        privatevoidabortWithUnauthorized(ContainerRequestContextrequestContext) {

                // Abort the filter chain with a 401 status code
                // The "WWW-Authenticate" header is sent along with the response
                requestContext.abortWith(Response.status(Response.Status.UNAUTHORIZED)
                                .header(HttpHeaders.WWW_AUTHENTICATE, AUTHENTICATION_SCHEME).build());
        }

        privatevoidvalidateToken(String token) {
                // Check if it was issued by the server and if it's not expired
                // Throw an Exception if the token is invalid

                try {

                        // Validate the token
                        String keyString = "simplekey";
                        Key key = newSecretKeySpec(keyString.getBytes(), 0,
keyString.getBytes().length, "DES");
                        System.out.println("the key is : " + key);

                        System.out.println("test:" +
Jwts.parser().setSigningKey(key).parseClaimsJws(token));
                        System.out.println("#### valid token : " + token);

                } catch (Exception e) {
                        System.out.println("#### invalid token : " + token);

        (this.requestContext).abortWith(Response.status(Response.Status.UNAUTHORIZED).build());

                }
        }
}
```

Pour tester ce filtre, on envoie la requête GET de la ressource sécurisée qui est, dans notre cas, **getListeEmployes()**. On envoie avec cette requête le token précédemment généré comme le montre la figure suivante :

Si on envoie un token invalide, on aura une **"Response"**de type **« 401 Unauthorized »** :

Par contre, si on invoque une méthode non sécurisée avec un token invalide, la requête est acceptée.