Programmation Web Serveur

Json Web Token (JWT)

Les services REST sont par nature sans état (stateless). Il n'y a donc pas de session coté serveur. Par conséquent, les informations de sécurité (authentification) doivent être envoyées à chaque requête par le client. Une stratégie courante pour propager les informations de sécurité est le recours aux jetons de sécurité. Il existe différentes solutions basées sur le principe de délégation d'autorisation comme Auth2, OpenID Connect, etc. Toutes ces solutions partagent le principe d'authentification basée sur des jetons. Ces solutions ne garantissent pas la confidentialité des échanges entre le serveur et le client. Il faut donc coupler le mécanisme d'authentification avec le protocole HTTPS.

Json Web Token est un standard (RFC 7519) pour l'échange sécurisé de jetons (tokens). JWT permet de représenter un ensemble d'affirmations (claims) dans un objet JSON et de signer cet objet pour garantir son authenticité.

Le principe est le suivant : Un client s'authentifie auprès d'un serveur (Issuer) par envoi du login et mot de passe par exemple, le serveur vérifie ces informations et génère un jeton JWT qui contient le login du client, ses rôles (par exemple admin) ainsi que diverses autres infos (date d'émission, durée de validité, etc.). Le jeton est signé par le serveur et renvoyé au client.

Le client peut donc maintenant utiliser ce jeton pour « prouver » son identité et ses rôles auprès d'un service REST à chaque requête effectuée. A chaque requête reçue le service vérifie la signature du jeton et ainsi l'intégrité de celui-ci. Bien sûr, les communications entre le serveur et le client doivent être sécurisées (en HTTPS) afin que le jeton ne puisse pas être « volé » par un tiers.

Structure d'un jeton JWT

Un jeton JWT est constitué de 3 parties :

- **L'entête (header)** : un objet JSON avec les propriétés alg et typ. Alg identifie l'algorithme utilisé pour signer le jeton et typ est le type de jeton (JWT). Exemple :

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

- **La charge utile (payload)** : un objet JSON contenant un ensemble d'affirmations comme le login, la date d'expiration du jeton, les rôle, etc. Exemple :

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

- **La signature** : Elle obtenue en signant la concaténation (avec un point entre les deux) des représentations en base 64 des deux objets JSON header et payload.

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

L'algorithme de signature est basé sur l'utilisation d'une paire de clés publique et privée. La clé privée permet de générer la signature et la clé publique permet de vérifier la signature. Avec JWT on utilise généralement la signature RSA avec SHA-256 (on calcule le hashcode du JWT avec SHA-256, puis ce hash est chiffré avec RSA).

Le jeton JWT résultat est la concaténation (séparées par des points) des représentations en base 64 de chacune des parties :

```
base64urlEncoding(header) + '.' + base64urlEncoding(payload) + '.' +
base64urlEncoding(signature)
```

Voici un exemple de résultat :

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4
gRG9lIiwiZ3JvdXBzIjpbImFkbWluIl0sImV4cCI6MTYwNDYwMTYyMCwiaWF0IjoxNTE2MjM5MDIyfQ.
nK8jbdbTCmYSXDrq48cuXk63vtFYJnEBxL7-E5NhugU
```

Les jetons ainsi générés sont en principe envoyés dans une entête des requêtes en suivant le format « Authorization: Bearer MONJETON ». Exemple :

```
GET /resource/1 HTTP/1.1
Host: example.com
Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4
gRG9lIiwiZ3JvdXBzIjpbImFkbWluIl0sImV4cCI6MTYwNDYwMTYyMCwiaWF0IjoxNTE2MjM5MDIyfQ.
nK8jbdbTCmYSXDrq48cuXk63vtFYJnEBxL7-E5NhugU
```

Pour plus d'information voir : https://en.wikipedia.org/wiki/JSON_Web_Token et https://jwt.io/introduction/

Mise en pratique avec Eclipse MicroProfile

Eclipse Microprofile utilise l'algorithme de signature RSA (RS256). Le payload du jetons doit au moins contenir les propriétés "sub", "upn", "exp", "iat", "iss" et "groups".

- **iss** : identifie le fournisseur (issuer) du jeton, i.e. l'URL du serveur d'authentification
- **iat** : date à laquelle le jeton a été émis par l'issuer (au format nombre de sec depuis EPOCH).
- **exp** : date d'expiration du jeton.
- **upn** et **sub** : identifiant (email par exemple) de l'utilisateur (subject). Les deux sont dupliqués car l'utilisation de l'un ou l'autre dépend du serveur d'application utilisé.
- **groups** : les rôles de l'utilisateur (admin, classical user, etc.)

Plus d'informations sur l'utilisation de JWT avec MicroProfile sont dispo dans la spécification :

https://download.eclipse.org/microprofile/microprofile-jwt-auth-1.2/microprofile-jwt-auth-spec-1.2.html

Dans le cadre du projet, la génération des jetons sera faite dans le même service. Cependant, dans des applications « réelles », l'authentification et la génération des jetons sont souvent déléguées à un service spécialisé.

Pour pouvoir signer des jetons il faut une clé privée et pour les vérifier il faut la clé publique correspondante. Seul le serveur d'authentification doit posséder la clé privée (qui devrait être stockée de manière sécurisée), la clé publique est diffusée à tous les services qui veulent vérifier les jetons.

Génération des clés

Pour générer une paire de clés privée et publique, on utilisera OpenSSL (disponible sur MacOs et Linux, pour windows, je ne sais pas).

Tout d'abord, il faut se placer à la racine du projet.

Pour générer la clé de base :

```
openssl genrsa -out baseKey.pem
```

Le fichier baseKey.pem contient la clé privée ainsi que les infos pour générer la clé publique.

Pour extraire la clé privée :

```
openssl pkcs8 -topk8 -inform PEM -in baseKey.pem -out
privateKey.pem -nocrypt
```

Pour extraire la clé publique:

```
openssl rsa -in baseKey.pem -pubout -outform PEM -out
publicKey.pem
```

Il faut ensuite copier les clés dans notre projet. On copie les clés dans le dossier src/main/resources du projet :

```
cp privateKey.pem src/main/resources/
cp publicKey.pem src/main/resources/
```

Configuration de l'application

Ajouter dans le fichier resources/META-INF/microprofile-config.properties le lignes suivantes :

```
mp.jwt.verify.publickey.location=/publicKey.pem
privatekey.location=/privateKey.pem
mp.jwt.verify.issuer=http://miashs.univ-grenoble-alpes.fr
mp.jwt.decrypt.key.location=/privateKey.pem
```

La propriété mp.jwt.verify.issuer permet de s'assurer que le claim iss des jetons est bien celui renseigné. Dans les autres cas, le jeton sera refusé.

Ajouter la dépendance vers la librairie suivante dans le fichier pom.xml du projet :

```
<!-- for generating JWT -->
<dependency>
   <groupId>io.vertx</groupId>
   <artifactId>vertx-auth-jwt</artifactId>
   <version>4.2.1</version>
</dependency>
```

Télécharger et copier la classe `JWTUtil` dans le package `fr.uga.miashs.projetqcm.util`.

Ajouter dans `ProjetQCMApplication.java` :

```
@LoginConfig(authMethod = "MP-JWT")
@DeclareRoles({"user","prof"})
```

Pour générer un jeton suite à l'authentification, vous pouvez vous inspirer de la méthode suivante :

```
@GET
@Path("/login")
public Response authenticate(@QueryParam("email") String email, @QueryParam("passwd")
String password) {
   TypedQuery<Person> q = em.createNamedQuery("Person.login",Person.class);

   q.setParameter("email",email);
   try {
      Person u = q.getSingleResult();
      if (!hashAlgo.verify(password.toCharArray(),u.getPasswordHash())) {
         throw new NotAuthorizedException("unknown user");
      }
      // doc https://vertx.io/docs/vertx-auth-jwt/java/
      List<String> roles = new ArrayList<>();
      roles.add("user");
      String jwt = jwtUtils.generateToken(u.getEmail(),roles,120);
      return Response.ok().entity(jwt).build();
   }
   catch (NoResultException e) {
      throw new NotAuthorizedException("unknown user");
   }
}
```

Il faut penser à avoir les attributs « injectés » suivants :

```
@Inject
private PasswordHash hashAlgo;

@Inject
private JWTUtils jwtUtils;
```

L'appel à la méthode d'authentification, donnera en résultat un jeton. Vous pouvez vérifier le jeton manuellement sur le site : https://jwt.io/ (en donnant la clé publique qu'il y a dans le fichier publicKey.pub).

Ensuite, si vous souhaitez restreindre l'accès à une méthode REST, il suffit d'ajouter l'annotation suivante devant la méthode concernée :

```
@RolesAllowed("user")
```

Ici seulement les requêtes avec un JWT dont l'un des groupes est « user » pourront accéder à cette méthode. Dans les autres cas, une erreur « HTTP Status 401 - Unauthorized » sera envoyée.

Testez ! Avec Postman, c'est assez facile (il faut récupérer le jeton, puis l'ajouter dans l'onglet authorization de la requête avec le type « bearer token »).

Au niveau de l'application, on peut avoir besoin d'accéder aux infos du jeton utilisé par le client.

Il est ainsi possible d'injecter le token :

```
@Inject
private JsonWebToken callerPrincipal;
```

Il est également possible de ne récupérer que certaines parties du jetons (des claims) :

```
@Inject
@Claim(value="upn", standard= Claims.upn)
private String mail;
```

Ce dernier exemple, peut être utile si l’on souhaite savoir quel utilisateur a envoyé la requête.