Sécurité des Services REST

Emmanuel BRUNO
Université de Toulon, LIS UMR CNRS 7020
emmanuel.bruno@univ-tln.fr

2026-11-01

Table des matières

0.1 TLS avec Grizzly 2
0.2 Authentification 3
0.3 Autorisation 4

```
[INFO] Target directory: /home/jovyan/work/github/ebpro/sample-jakarta-rest-full
[INFO] Updating existing repository in /home/jovyan/work/github/ebpro/sample-jakarta-rest-full
[WARNING] Git operation failed with status code: 1
```

Les exemples suivants sont accessibles dans le dépôt :

✕ : ebpro/sample-jakarta-rest-full

Détails
- **URL**: https://github.com/ebpro/sample-jakarta-rest-full
- **Branch**: develop
- **Local Path**: `/home/jovyan/work/github/ebpro/sample-jakarta-rest-full`
- **Latest Commit**: unknown - No commit info
- **Timestamp**: 2026-11-01T12:24:47Z

Clone Command:

```
git clone -b develop https://github.com/ebpro/sample-jakarta-rest-full
```

Pour assurer la sécurité d'une API REST, la première chose à faire est d'assurer la confidentialité. Pour cela, il faut utiliser HTTPS qui utilise TLS pour permettre de valider l'identité du serveur et pour garantir la confidentialité et l'intégrité des données échangées en utilisant des certificats.

Pour mettre cela en place, il est possible d'utiliser un « reverse proxy » (par exemple nginx qui lui sera sécurisé et servira de facade, le serveur REST n'étant jamais accessible autrement.

L'autre solution est de sécuriser les serveurs web (dans notre exemple Java, Grizzly). Pour cela, il faut idéalement se procurer des certificats pour le serveurs signés par une autorité reconnue. Nous utiliserons ici des certificats auto-signés dans un but de démonstration uniquement.

Le certificat du serveur est habituellement généré avec openssl, ici nous utilisons maven (`keytool-maven-plugin`) pour le générer automatiquement s'il n'existe pas déjà dans le répertoire `/src/jaxrs/sample-jaxrs/src/main/resources/ssl/`. Le certificat est automatiquement ajouté à un keystore Java dans le même répertoire (cert.jks).

Le serveur Grizzly est en écoute avec HTTP sur le port 9998 et en HTTPS sur le port 4443.

Cette méthode ajoute aussi le support de HTTP2 qui améliore grandement les performances.

0.1 TLS avec Grizzly

Error: failed to read file `/home/jovyan/work/materials/github/ebpro/sample-jakarta-rest-full/src/main/java/fr/univtln/bruno/samples/jaxrs/server/BiblioServer.java`: /home/jovyan/work/materials/github/ebpro/sample-jakarta-rest-full/src/main/java/fr/univtln/bruno/samples/jaxrs/server/BiblioServer.java

On commence donc par activer TLS. On en profite pour activer aussi le support de HTTP2.

```
HttpServer httpServer = BiblioServer.startServer();
BiblioServer.addTLSandHTTP2(httpServer);
```

```
CompilationException:
|   HttpServer httpServer = BiblioServer.startServer();
cannot find symbol
  symbol:   class HttpServer

|   HttpServer httpServer = BiblioServer.startServer();
cannot find symbol
  symbol:   variable BiblioServer
```

Pour tester les requêtes sécurisée avec un certificat autosigné il faut d'abord le télécharger (ici avec la commande curl).

```
%%shell
echo quit | \
    openssl s_client -showcerts \
        -servername localhost \
        -connect localhost:4443 >! /tmp/cacert.pem
```

```
40974CD9BB7E0000:error:8000006F:system          library:BIO_connect:Connection
refused:crypto/bio/bio_sock2.c:178:calling connect()
40974CD9BB7E0000:error:10000067:BIO routines:BIO_connect:connect error:crypto/
bio/bio_sock2.c:180:
40974CD9BB7E0000:error:8000006F:system          library:BIO_connect:Connection
refused:crypto/bio/bio_sock2.c:178:calling connect()
40974CD9BB7E0000:error:10000067:BIO routines:BIO_connect:connect error:crypto/
bio/bio_sock2.c:180:
connect:errno=111
```

Il sera ensuite utilisé pour valider l'indentité du serveur web.

```
%%shell
curl --silent \
    --trace-ascii /tmp/trace-secure.txt \
    --http2 \
    --cacert /tmp/cacert.pem \
    https://localhost:4443/mylibrary/library
```

0.2 Authentification

Il faut mettre en place une gestion correcte des utilisateurs (login+mots de passe hashés correctement). Cela pourra être complété/remplacé par des certificats ou une délégation d'authentification.

Dans cet example, nous utilisons une base de données d'utilisateurs en mémoire.

```
fr.univtln.bruno.samples.jaxrs.security.InMemoryLoginModule.USER_DATABASE.getUsers()
```

```
CompilationException:
|  fr.univtln.bruno.samples.jaxrs.security.InMemoryLoginModule.USER_DATABASE.getUsers()
package fr.univtln.bruno.samples.jaxrs.security.InMemoryLoginModule does not
exist
```

0.3 Autorisation

L'Autorisation est cruciale, elle peut s'appuyer un token qui est fourni par le système lors d'un login et a une durée de vie limitée. Ce token est envoyé avec chaque requête et le système lui attribue un ensemble de permission.

Un autre approche est d'utiliser un token cryptographique qui contient ces informations et qui est signée par le serveur.

Par exemple avec les JSON Web Token - JWT qui présente en détail le processus type.

Dans ces exemples, nous utiliserons la librairies Java JJWT.

Voilà des exemples d'utilisations simples.

Accès refusé à une ressource sécurisée.

```
%%shell
curl -s -i --http2 \
    --cacert /tmp/cacert.pem \
    -H "Accept: application/json" \
    https://localhost:4443/mylibrary/setup/secured
```

Utilisation de la « Basic Authentication » pour obtenir un Java Web Token.

```
%%shell
curl -s -s --http2 \
    --cacert /tmp/cacert.pem \
    --user "john.doe@nowhere.com:admin" \
    https://localhost:4443/mylibrary/setup/login
```

Décodage d'un JWT

Il suffit de faire une requête rest et d'en obtenir un.

```
Client client = ClientBuilder.newClient();
WebTarget webResource = client.target("http://localhost:9998/mylibrary");
String email = "john.doe@nowhere.com";
String passwd = "admin";
String token = webResource.path("setup/login")
                .request()
//                .accept(MediaType.TEXT_PLAIN)
                                    .header("Authorization",  "Basic  "  +
java.util.Base64.getEncoder().encodeToString((email + ":" + passwd).getBytes()))
                .get(String.class);
token;
```

```
CompilationException:
|   Client client = ClientBuilder.newClient();
cannot find symbol
  symbol:   class Client

|   Client client = ClientBuilder.newClient();
cannot find symbol
  symbol:   variable ClientBuilder
```

puis en utilisant la clé publique (dans cette exemple simple ont y accède directement côté serveur), il est possible de vérifier les informations. Ici le choix a été fait d'utiliser une approche RBAC (Role Based Access Control) embarquée dans le token qui cumule donc authentification et autorisation. Cela rend le système très simple mais à comme conséquence de faire qu'un change de droit n'est appliqué qu'à la fin de la durée de vie du token.

```
Jws<Claims> jws = Jwts.parser()
                    .verifyWith(InMemoryLoginModule.KEY)
                    .build()
                    .parseSignedClaims(token);

jws;
```

```
CompilationException:
|   Jws<Claims> jws = Jwts.parser()
cannot find symbol
  symbol:   class Jws

|   Jws<Claims> jws = Jwts.parser()
cannot find symbol
  symbol:   class Claims

|                       .parseSignedClaims(token);
cannot find symbol
  symbol:   variable token

|                       .verifyWith(InMemoryLoginModule.KEY)
cannot find symbol
  symbol:   variable InMemoryLoginModule

|   Jws<Claims> jws = Jwts.parser()
cannot find symbol
  symbol:   variable Jwts
```

Utilisation d'un Java Web Token. Le token peut donc être transmis au serveur qui le vérifie et l'utilise pour l'authentification voir l'autorisation. Ici l'accès à une ressource qui demande d'être user ou admin est autorisé à un admin.

```
%%shell
TOKEN=$(curl -s --http2 \
    --cacert /tmp/cacert.pem \
    --user "john.doe@nowhere.com:admin" \
    https://localhost:4443/mylibrary/setup/login)

curl -s -i --http2 \
    --cacert /tmp/cacert.pem \
    -H "Authorization: Bearer ${TOKEN}" \
    https://localhost:4443/mylibrary/setup/secured
```

tout comme l'accès à une ressource qui demande d'être admin est autorisé à un admin.

```
%%shell
TOKEN=$(curl -s --http2 \
    --cacert /tmp/cacert.pem \
    --user "john.doe@nowhere.com:admin" \
    https://localhost:4443/mylibrary/setup/login)

curl -s -i --http2 \
    --cacert /tmp/cacert.pem \
    -H "Authorization: Bearer ${TOKEN}" \
    https://localhost:4443/mylibrary/setup/secured/admin
```

L'accès à une ressource qui demande d'être user ou admin est autorisé à un user.

```
%%shell
TOKEN=$(curl -s --http2 \
    --cacert /tmp/cacert.pem \
    --user "william.smith@here.net:user" \
    https://localhost:4443/mylibrary/setup/login)

curl -i -s --http2 \
    --cacert /tmp/cacert.pem \
    -H "Authorization: Bearer ${TOKEN}" \
    https://localhost:4443/mylibrary/setup/secured
```

mais l'accès à une ressource qui demande d'être admin est refusée à un user.

```
%%shell
TOKEN=$(curl -s \
    --user "william.smith@here.net:user" \
    https://localhost:4443/mylibrary/setup/login)
curl -s -i --http2 \
    --cacert /tmp/cacert.pem \
    -H "Authorization: Bearer ${TOKEN}" \
    https://localhost:4443/mylibrary/setup/secured/admin
```

L'application exemple présente en détail comment un filtre JAX-RS et des annotations peuvent être utilisé pour appliquer une politique de contrôle d'accès.