Master MIASHS
Université Grenoble Alpes

JavaEE – Cours 4

1- Usage un peu plus avancé de JPA

Une des principales difficultés de JPA est le chargement des entités dépendantes (i.e. relations). Il existe différentes stratégies de chargement : le lazy loading (chargement à la demande) et le eager loading (chargement à l'avance).

Le « eager loading » charge tous les éléments dépendants. Si on applique cela à toute les relations, on risque de charger une grosse partie de la base de données à chaque requête. Ce n'est pas souhaitable.

Le « lazy loading » ne charge rien par défaut et à chaque accès à un attribut, une requête est envoyée pour récupérer le ou les objets dépendants. Cela permet d'optimiser la mémoire mais cela peut engendrer un grand nombre de requêtes et affecter la performance de l'application. De plus, lorsque l'objet n'est plus attaché à l'entity manager (i.e. quand on est sorti de la transaction par exemple), le chargement en mode lazy n'est plus possible et une exception de type `LazyIntitializationException` est levée.

Par défaut, avec JPA, les attributs "simples", i.e. les types simples, les chaines de caractères, les dates, etc. ainsi que les relations « `@ManyToOne` » et « `@OnetoOne` » sont chargés en mode « Eager ». Les autres relations sont chargées en mode « Lazy ».

Pour pallier ces problèmes de chargement, je vous conseille de contrôler à chaque requête ce dont vous avez besoin de récupérer. Pour cela, il existe deux stratégies : les EntityGraphs et les requêtes avec le mot clé « `JOIN FETCH` » de JPQL.

1.1- Les entity graphs

Les EntityGraphs permettent de spécifier les attributs (et le sous graphe d'attributs en cas de dépendance) à charger lors d'une requête ou d'un find. Un exemple est donné pour la classe `AppGroup` qui permet de ne charger que l'identifiant et le nom d'un groupe (et pas la dépendance `owner` normalement chargée):

```
@NamedEntityGraph(name = "AppGroup.groupOnly",
        attributeNodes ={@NamedAttributeNode("id"),@NamedAttributeNode("name")})
```

Plus d'infos sur les entity graphs sont disponibles sur le tutoriel suivant :

https://www.baeldung.com/jpa-entity-graph

1.2- Les requêtes JOIN FETCH

Le mot clé JOIN FETCH permet dans une requête JPQL de charger une relation. Par exemple, si l'on veut charger à la fois un groupe et la liste des membres, on peut écrire la requête suivante :

```
"SELECT g FROM AppGroup g LEFT JOIN FETCH g.members"
```

Le mot clé LEFT permet de charger également les groupes qui n'ont pas de membre. Le mot clé FETCH indique de charger les liste de membres pour chaque groupe.

Une fois les objets récupérés, il se peut que certains champs ne soient toujours pas chargé par ce que vous n'en avez pas besoin. Cependant, lors de la transformation d'un objet « java » vers un objet Json toutes les méthodes de l'objets java sont appelée et on obtient alors des pour les attributs non chargés. Pour résoudre ce problème, j'ai ajouté dans la classe `GenericJpaRestDao` la méthode `protected void setNullForAllLazyLoadEntities(Object source)` qui permet de (1) détacher l'entité source, (2) de mettre à `null` tous les attributs non chargés, et ce que manière récursive dans les entités dépendantes.

1.3- Architecture d'une application REST avec JPA

Pour réaliser un service REST avec JPA, je vous propose architecture suivante.

Couche modèle (package model) : Contient toutes les classes du modèle de données. Ce sont généralement des entités.

Couche d'accès aux données – Data Access Object – DAO (package dao) : cette couche permet de fournir les moyens pour interagir avec la base de données. Nous avons une classe « DAO » par entité avec les méthodes de bases suivantes : `create`, `read`, `update`, `delete`, `listAll` et `count`. En plus de ces méthodes, on rajoute les méthode spécifiques (requêtes) retournant des instances de l'entités.

Généralement les méthodes des DAO retournent des instances des classes du modèle. Ici, nous retournons des « `Response` » qui contiennent ces instances mais aussi le code de retour et éventuellement d'autres informations.

Pour faciliter la tâche et factoriser le code redondant, la classe abstraite `GenericJpaRestDao` fourni les méthodes de bases CRUD + `listAll` + `count` ainsi que quelques méthodes utilitaire.

Pour implémenter un DAO spécifique, il suffit d'étendre cette classe, de définir un constructeur sans paramètre qui appelle le constructeur de la super classe avec la classe des objets gérés par le DAO (c.f. `UsersDao` ou `GroupsDao`).

Couche Contrôleur (package controllers) : Cette couche contient toute les classes « ressource web » de l'application. Généralement, on a une classe par entité. Ces classes s'occupent de faire le lien entre la requête HTTP et l'appel au DAO. Dans ces classes les DAO sont des attributs injectés via (l'annotation `@Inject`)

2- Sécurisation JWT

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 (envoi du login + mot de passe par exemple), le serveur vérifie les informations , génère un jeton JWT qui contient le login du client et ses rôles (par exemple admin). 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 du serveur à chaque requête effectuée. A chaque requête reçue le serveur vérifie la signature du jeton et ainsi l'intégrité de celui-ci. Bien sûr, les communications entre le serveur et le client doivent être sécurisés (en HTTPS) afin que le jeton ne puisse pas être « volé » par un tiers.

2.1- 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'algoritme 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.

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

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/

2.2- Mise en pratique avec Eclipse MicroProfile

On s'intéresse ici qu'à l'authentification via un jeton. La génération des jetons par l'application n'est pas présentée dans ce document. Pour générer les jetons on utilisera le site officiel : https://jwt.io/

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

La référence sur laquelle s'appuie cet exemple est :

https://www.eclipse.org/community/eclipse_newsletter/2017/september/article2.php

2.2.1- Génération des clés et configuration de l'application

Générer une paire de clé :

```
ssh-keygen -m PEM -t rsa -b 2048 -f jwtRS256.key
# Ne pas donner de mot de passe
openssl rsa -in jwtRS256.key -pubout -outform PEM -out jwtRS256.key.pub
```

Le fichier `jwtRS256.key` contient la clé privée et `jwtRS256.key.pub` la clé publique.

Déposer le fichier `jwtRS256.key.pub` dans le dossier resources :

```
cp jwtRS256.key.pub src/main/resources/
```

Dé-commenter la ligne :
`<mp.jwt.verify.publickey.location>/jwtRS256.key.pub</mp.jwt.verify.publickey.location>` du fichier `pom.xml` (vers la fin).

Dé-commenter dans `Cours2RestApplication.java` :

```
@LoginConfig(authMethod = "MP-JWT", realmName = "jwt-jaspi")
@DeclareRoles({"user","admin"})
```

Dé-commenter dans `GroupControllers.java` :

```
@RolesAllowed("admin")
```

Relancer le serveur :

```
mvn package tomee:run
```

Vérifier que la requête suivante reourne une erreur HTTP 401 :

```
curl -i -X GET http://localhost:8080/data/groups
```

Ca veut dire que l'on n'est pas autorisé à accéder à cette ressource.

2.2.2- Génération du token

Aller sur le site :
https://jwt.io/#debugger-io

Renseignez le formulaire en utilisant les données suivantes.

Dans le header :

```
{
  "alg": "RS256",
  "typ": "JWT"
}
```

Dans le payload :

```
{
  "sub": "toto@toto.org",
  "groups": ["admin"],
  "exp" : 1608850800
}
```

exp est la date d'expiration au format nombre de secondes depuis l'EPOCH
Pour récupérer une date au format « EPOCH » :

LINUX : `date -d "Dec 25 2020" +%s`
MAC : `date -j -f "%b %d %Y %T" "Dec 25 2020 00:00:00" "+%s"`
Ou sur le web : https://www.epochconverter.com/

Mettre les clés publique et privée dans le champs prévus.

Récupérer le token encodé et exécuter la requête suivante :

```
curl –i –X GET –H "Authorization: Bearer VOTRE_TOKEN"
http://localhost:8080/data/groups
```

On devrait être autorisé maintenant à récupérer la liste des groupes.