

Exercice5

Le but de cet exercice est d'illustrer le module spring-data-redis, une base de données orientée clé/valeur.

Etape 1 : création du projet

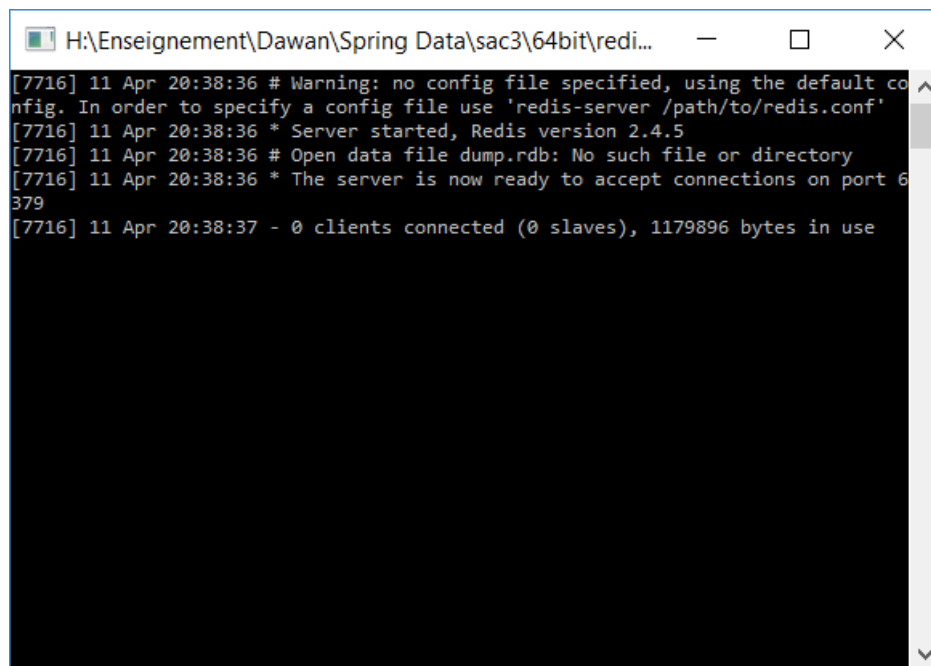
A partir d'Eclipse, créer un projet comme précédemment mais le nommé : ex5-spring-data afin d'illustrer les concepts Redis au sein de spring-data

Ajouter les dépendances sur le starter spring redis et les dépendances pour faire des services REST.

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
</dependencies>
```

Etape 2 : Configuration de Redis

Redis est un datastore NoSQL clé / valeur, c'est-à-dire un grand hashmap persistant très rapide. Si Redis n'est pas déjà installé utiliser l'installation pour Windows. Démarrer le serveur avec redis-server.exe :



```
H:\Enseignement\Dawan\Spring Data\sac3\64bit\redi...
[7716] 11 Apr 20:38:36 # Warning: no config file specified, using the default co
nfig. In order to specify a config file use 'redis-server /path/to/redis.conf'
[7716] 11 Apr 20:38:36 * Server started, Redis version 2.4.5
[7716] 11 Apr 20:38:36 # Open data file dump.rdb: No such file or directory
[7716] 11 Apr 20:38:36 * The server is now ready to accept connections on port 6
379
[7716] 11 Apr 20:38:37 - 0 clients connected (0 slaves), 1179896 bytes in use
```

Il est possible de connecter un client avec le shell redis-cli.exe : puis d'exécuter une commande pour lire toutes les clés.

```
H:\Enseignement\Dawan\Spring Data\sac3\64bit\redis-cli.exe
redis 127.0.0.1:6379> keys *
(empty list or set)
redis 127.0.0.1:6379> ping
PONG
redis 127.0.0.1:6379> set formation Spring Data
(error) ERR wrong number of arguments for 'set' command
redis 127.0.0.1:6379> set formation "Spring Data"
OK
redis 127.0.0.1:6379> get formation
"Spring Data"
redis 127.0.0.1:6379>
```

Il est demandé de créer un fichier `application.properties` dans `src/main/resources` afin de configurer la connexion à la base Redis :

```
spring.session.store-type=redis # Session store type.
spring.redis.host=localhost # Redis server host.
spring.redis.password= # Login password of the redis server.
spring.redis.port=6379 # Redis server port.
```

Etape 3 : Création du modèle de données

Dans un package `ex5.data`, il est demandé de créer une classe principale `Customer` pour représenter les clients de la formation. Cette classe est annotée `@RedisHash`, cette classe dispose :

- `Long id`
- `String externalId` annotée `Indexed`
- `String name`
- `List<Account> accounts`

Pour les méthodes : constructeur, getters et setters, `toString`,

La classe `Account` est définie dans le même package, elle n'est pas annotée `@RedisHash`, et dispose des attributs suivants :

- `Long id` annotée `Indexed`
- `String number`
- `Integer balance`

Pour les méthodes : constructeur, getters et setters, `toString`,

Dans le même package, il est demandé de créer une interface `CustomerRepository` qui hérite de `CrudRepository<Customer, String>`,

En étendant `CrudRepository` dans `CustomerRepository`, nous obtenons automatiquement un ensemble complet de méthodes de persistance exécutant la fonctionnalité CRUD.

Etape 4 : Définition d'un contrôleur Web

Dans un package `ex5.web`, il est demandé de créer une classe principale `CustomerController` pour accéder à `CustomerRepository`. Cette classe est annotée `@RestController` et `@RequestMapping` pour l'associer à l'URL `/customers`. Elle possède

- Un attribut injecté par Spring pour accéder au `CustomerRepository`.

- Une méthode `add` associé au POST qui prend un `customer` en JSON dans le body de la requête,
- Une méthode `findById` associé à GET qui prend en paramètre une id en tant que PathVariable afin que l'identificateur soit dans le path.

Etape 5 : Création d'une classe principale.

Il est demandé de recopier la classe `AppMain` d'un exercice passé et de faire un test avec un navigateur est son plugin REST.

Il est demandé de contrôler les résultats dans le shell client.

Exercice6

Le but de cet exercice est d'illustrer le module spring-data-rest, et la gestion des échanges REST.

Etape 1 : création du projet

A partir d'Eclipse, créer un projet comme précédemment mais le nommé : ex6-spring-data afin d'illustrer les concepts Restful au sein de spring-data

Ajouter les dépendances sur le starter spring rest et les dépendances pour disposer d'un moteur d'application Wb embarqué.

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-rest</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jpa</artifactId>
  </dependency>
  <dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
  </dependency>
</dependencies>
```

Etape 2 : Configuration JPA

Créer une base ex6 dans PostgreSQL et réutiliser la configuration de l'exercice 2 pour simplifier cette tâche.

Etape 3 : Utilisation de Spring Data Rest

Spring Data Rest est construit sur Spring Data, Spring Web MVC et Spring Hatos. Il analyse tous les modèles de domaine et expose automatiquement les endpoints Hypermedia Driven REST. Toutes les fonctionnalités des référentiels de données Spring, telles que le tri, la pagination, etc., sont disponibles dans ces endpoints.

Il est demandé de construire dans un package `ex6.data` 2 classe `City` et `Country`, entités JPA pour créer pays en relation avec des villes. La classe `Country` annotée `@Entity` et `@RestResource` a 2 attributs :

- Long id annotée `@Id` et `@GeneratedValue`
- String name

Auquel s'ajoutent les méthodes classiques : constructeur, getters et setters et `toString`

La classe `City` annotée `@Entity` et `@RestResource` a 3 attributs

- Long id annotée `@Id` et `@GeneratedValue`
- String name
- Country country annotée `@ManyToOne`

Auquel s'ajoutent les méthodes classiques : constructeur, getters et setters et toString

En utilisant JPA dans notre projet, nous créons une association entre une ville et un pays. De nombreuses villes peuvent être associées à un pays. Il est demandé de créer les 2 référentiels pour eux, soit :

- CountryRepository qui hérite de JpaRepository,
- CityRepository qui hérite de JpaRepository et est annoté @RepositoryRestResource auquel on associe le chemin /metropolises afin de pouvoir y accéder par REST

Cela créera un référentiel pour Country et exposera également les endpoints REST (GET, POST, PUT, DELETE, PATCH). Comme JpaRepository étend PagingAndSortingRepository, des fonctionnalités de pagination et de tri seront automatiquement ajoutées pour le endpoint GET. Par défaut, le chemin est dérivé du nom de classe simple, non capitalisé, et pluralisé de la classe de domaine en cours de gestion. Dans notre cas, le chemin sera /countries.

Etape 4: Requetage Rest

Faire une classe ex6.AppMain pour définir la méthode principale de notre application SpringBoot. Vérifions les API après avoir exécuté notre projet. Accès à 'http://localhost:8080' depuis un navigateur :

```
{
  "_links": {
    "countries": {
      "href": "http://localhost:8080/countries{?page,size,sort}",
      "templated": true
    },
    "cities": {
      "href": "http://localhost:8080/metropolises{?page,size,sort}",
      "templated": true
    },
    "profile": {
      "href": "http://localhost:8080/profile"
    }
  }
}
```

Nous obtenons des informations sur les API disponibles. Nous pouvons explorer davantage les métadonnées en tapant sur l'API de profil.

Il est demandé d'ajouter plusieurs pays avec l'url exemple

'http://localhost:8080/countries' en POST avec dans le body de la requête

'{"name": "Japan"}' et l'entête 'Content-Type: application/json'

Récupérons un résultat paginé de countries avec les résultats triés par nom de pays, le format de page 2 et la 1ère page, soit la requête GET suivante

'http://localhost:8080/countries/?sort=name,asc&page=1&size=2'

```
{
  "_embedded": {
    "countries": [
      {
        "name": "Germany",
        "_links": {
```

```

        "self": {
            "href": "http://localhost:8080/countries/3"
        },
        "country": {
            "href": "http://localhost:8080/countries/3"
        }
    },
    {
        "name": "India",
        "_links": {
            "self": {
                "href": "http://localhost:8080/countries/2"
            },
            "country": {
                "href": "http://localhost:8080/countries/2"
            }
        }
    }
]
},
"_links": {
    "first": {
        "href": "http://localhost:8080/countries?page=0&size=2&sort=name,asc"
    },
    "prev": {
        "href": "http://localhost:8080/countries?page=0&size=2&sort=name,asc"
    },
    "self": {
        "href": "http://localhost:8080/countries"
    },
    "next": {
        "href": "http://localhost:8080/countries?page=2&size=2&sort=name,asc"
    },
    "last": {
        "href": "http://localhost:8080/countries?page=2&size=2&sort=name,asc"
    },
    "profile": {
        "href": "http://localhost:8080/profile/countries"
    }
},
"page": {
    "size": 2,
    "totalElements": 5,
    "totalPages": 3,
    "number": 1
}
}

```

Outre les pays attendus, nous recevons également des liens vers différentes pages et des informations complémentaires susceptibles d'aider à mieux gérer la pagination. Les liens vers les première, précédente, même, précédente et dernière pages peuvent être directement utilisés.

Ajoutons une ville et associons-la à un pays via la requête :

'http://localhost:8080/metropolises' en POST avec le contenu de la requête
'{"name": "Osaka", "country": "http://localhost:8080/countries/1"}' en ajoutant l'entête 'Content-Type: application/json'

Nous devons passer l'URL du pays et cette ville sera placé au Japon. Nous avons d'abord sauvé le Japon, son identifiant est donc 1. Exécuter une requête pour chercher cette ville : soit la requête
'http://localhost:8080/metropolises/1'

```

{
    "name": "Osaka",

```

```
{
  "_links": {
    "self": {
      "href": "http://localhost:8080/metropolises/1"
    },
    "city": {
      "href": "http://localhost:8080/metropolises/1"
    },
    "country": {
      "href": "http://localhost:8080/metropolises/1/country"
    }
  }
}
```

Nous obtenons un lien vers le pays qui lui est associé. Voyons ce que nous obtenons en réponse de ce lien :

```
{
  "name": "Japan",
  "_links": {
    "self": {
      "href": "http://localhost:8080/countries/1"
    },
    "country": {
      "href": "http://localhost:8080/countries/1"
    }
  }
}
```