

Exercices SDR

Il est demandé de créer un répertoire de travail (Workspace) nommé `WorkspaceSDR`. Tous les projets seront réalisés dans ce répertoire. Le choix d'utiliser l'outil d'automatisation de tâche `Maven` simplifiera la création des projets.

Exercice1

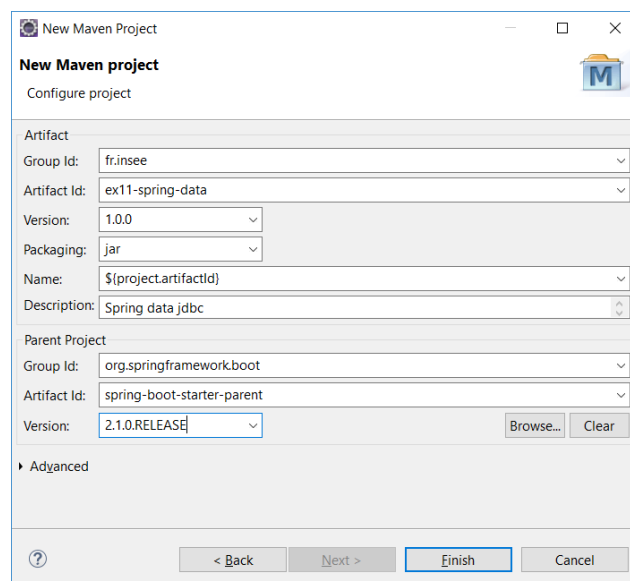
Le but de cet exercice est d'illustrer le module `spring-data-jdbc`. Pour ce faire nous utilisons un support de persistance du type `PostgreSQL` auquel accède un projet `Spring Boot` avec le starter `spring-boot-starter-jdbc`.

Etape 1 : création du projet

A partir d'Eclipse, il faut choisir `File -> New -> Project...` et choisir un `Maven Project`.

Choisir un projet parent Spring de sorte on ait :

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.1.0.RELEASE</version>
</parent>
```



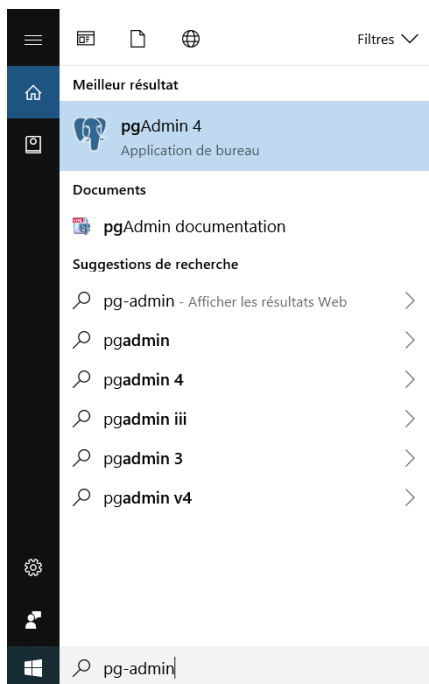
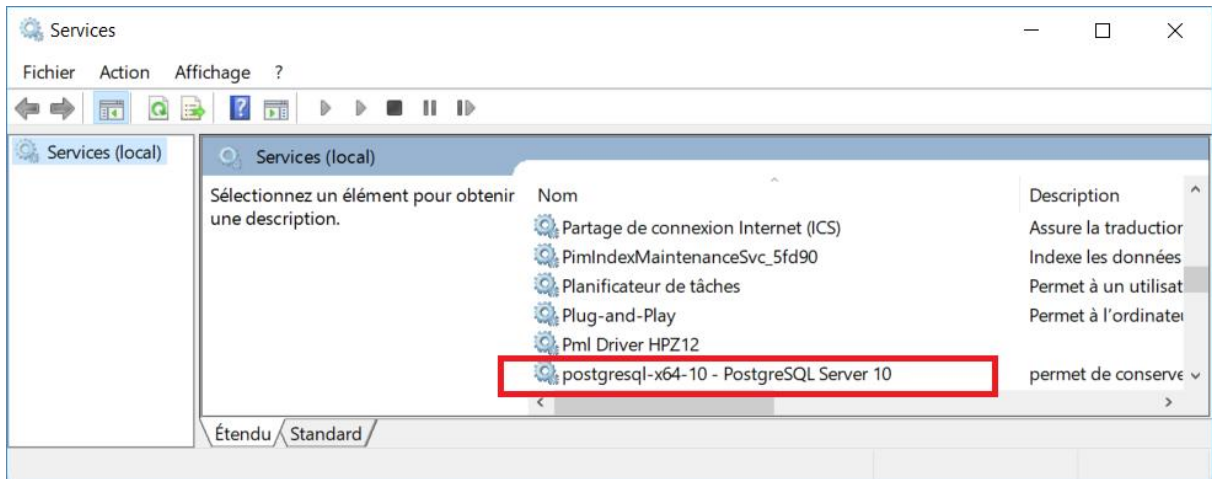
Ajouter les dépendances sur le starter `spring jdbc` et sur le driver `PostgreSQL`.

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

```
</dependencies>
```

Etape 2 : création d'une table pour commencer

Valider que le service PostgreSQL est actif par l'ouverture de la fenêtre de services sous Windows :



Il est alors possible de lancer le client `pg-admin` depuis Cortana.

A partir de ce client, créer une base de données `ex1`. Dans cette base `ex1`, créer une table SQL avec le code suivant :

```
CREATE TABLE IF NOT EXISTS pet (
  id      SERIAL PRIMARY KEY,
  name    VARCHAR(255),
  species VARCHAR(255)
);
```

Assurer vous que la table soit créée corresponde à cette déclaration avec la commande `describe`.

Etape 3 : création des classes pour accéder à la base

Dans un package `ex1.data`, il est demandé de créer :

- Une classe `Pet` qui joue le rôle de l'entité avec une propriété annotée `@Id`.
- Une classe `PetRepository` annotée `@Repository` et possédant un attribut dont la valeur est fournie par Spring :

- o `NamedParameterJdbcTemplate jdbcTemplate`

Et les méthodes suivantes à définir par rapport à l'attribut `jdbcTemplate`:

- o `public Pet findById(Integer id)`
- o `public Iterable<Pet> findAll()`
- o `public int save(Pet pet)`
- o `public void deleteById(Integer id)`

Pour ce faire il pourra être utile d'externaliser les requêtes SQL dans un fichier de propriétés.

- Une classe `PetController` annotée `@Controller` permettant d'exposer ces méthodes sur http. Considérer comme un service REST Spring classique, cette classe est annotée `@RequestMapping(path="/pets")` depuis un client REST. Cette classe possédant un attribut dont la valeur est fournie par Spring :

- o `PetRepository petRepository`

Elle possède autant de méthodes que la précédente soit :

- o `public String createPet(@RequestBody Pet pet)`
- o `public Iterable<Pet> getAllPets()`
- o `public Optional<Pet> getPet(@PathVariable Integer id)`
- o `public String deletePet(@PathVariable Integer id)`

Toutes les données de retour sont annotées `@ResponseBody` et chaque méthode est annotée en fonction du verbe http auquel elle est associée. Leur définition consiste à appeler la bonne méthode de la classe `PetRepository`.

Pour que ce service REST puisse être annotée, il est nécessaire d'ajouter un nouveau starter

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

Etape 4 : création d'une classe `AppMain`

Dans le répertoire `src/main/resources`, il faut définir le fichier `applications.properties` contenant les clés :

```
spring.datasource.initialization-mode=always
spring.datasource.platform=...
spring.datasource.url=...
spring.datasource.username=...
spring.datasource.password=...
```

La classe `AppMain` aura une méthode `main` classique en Spring Boot:

```
public static void main(String[] args) {
    SpringApplication.run(Application.class, args);
}
```

Dans le cas où l'on souhaite gérer des SGBD différents, il est possible d'ajouter des « profiles » au pom.xml.

Enfin il faut fabriquer le livrable en invoquant les phases `clean package`.

Etape 5 : validation par un test depuis un navigateur

Il faut en premier s'assurer que le service PostgreSQL est toujours actif.

Ensuite il faut exécuter le projet qui exposera un service REST :

```
java -jar target/ex1-data-1.0.0.jar
```

Un client REST simple est à utiliser pour effectuer les premières requêtes http. Par exemple dans le navigateur Firefox aller sur le site de google.fr et rechercher « REST Client plugin », le premier lien proposé permet d'ajouter un plugin au navigateur. Il fournit un bouton dans la barre de boutons permettant d'accéder à un panneau pour préparer une requête REST simple.

D'autres outils sont utilisables tels que SOAPUI ou curl ou Postman ; tout dépend de votre expertise.

- 1- Créer de nouveaux animaux sur l'URL <http://localhost:8080/pets> avec le verbe POST et l'entête : `Content-Type : application/json`

```
{
  "name": "Tom",
  "species": "cat"
}
```

Visualiser les conséquences dans la base de données

```
{
  "name": "Jerry",
  "species": "mouse"
}
```

- 2- Accéder à l'ensemble des animaux créés en base, pour ce faire sur l'URL <http://localhost:8080/pets> et le verbe GET, ajouter l'entête `Content-Type : application/json`. Le résultat sera :

```
[
  {
    "id": 1,
    "name": "Tom",
    "species": "cat"
  },
  {
    "id": 2,
    "name": "Jerry",
    "species": "mouse"
  }
]
```

Etape 6 : évolution de la couche de persistance

Il est demandé :

- De remplacer la classe `PetRepository` par une interface `PetRepository` qui hérite de `CrudRepository`.
 - o Ajouter une méthode personnalisée annotée `@Query` dont la valeur est l'ordre SQL paramétré qui retourne une liste de `Pet`.

```
List<Pet> findByName(@Param("name") String name);
```
- De revoir la classe principale `AppMain` pour implémenter l'interface `CommandLineRunner` et invoquer cette méthode de recherche personnalisée.

Refaire les tests précédents afin d'assurer que les services sont toujours fonctionnels.

Exercice2

Le but de cet exercice est d'illustrer le module spring-data-jpa. Pour ce faire nous utilisons un support de persistance du type PostgreSQL auquel accède un projet Spring Boot avec le starter spring-boot-starter-jpa.

Etape 1 : création du projet

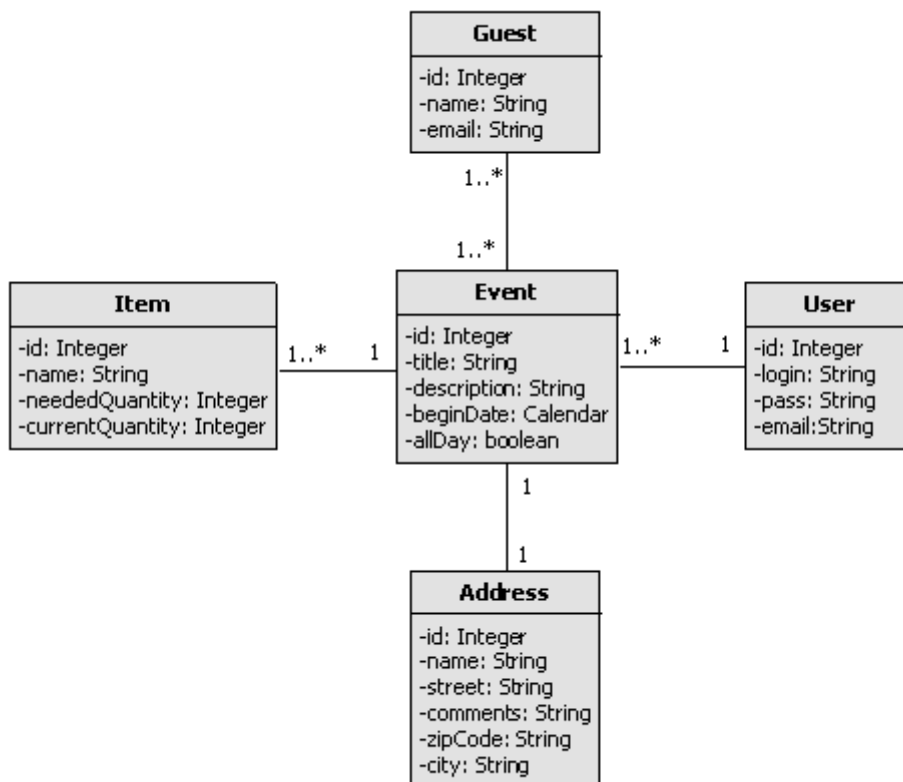
A partir d'Eclipse, créer un projet comme précédemment mais le nommé : ex2-spring-data afin d'illustrer les concepts JPA au sein de spring-data

Ajouter les dépendances sur le starter spring jpa et sur le driver PostgreSQL.

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

Etape 2 : création d'une table pour commencer

Dans le script SQL ex2 . sql, le schéma SQL ci-dessous a été transcrit pour une base PostgreSQL.



5 entités ont été décrites. Pour simplifier l'exercice, il est possible de prendre une partie de des entités si la difficulté paraît trop grande.

Depuis l'outil pg-admin, il est demandé de créer une base de données `ex2` et d'utiliser le script pour créer les tables avec leurs dépendances et autres contraintes auxquelles s'ajoutent les opérations d'ajout d'éléments.

Etape 3 : création des classes pour accéder à la base

Dans un package `ex2.data`, il est demandé de créer :

- Un package `entity` qui contient les classes annotées `@Entity` avec une propriété annotée `@Id`. (comme au slide 24)
- Un package `repository` qui contient les interfaces qui héritent de `CrudRepository` par entité en ajoutant d'éventuelles méthodes personnalisées.
 - o Il est essentiel de prendre soin des associations entre les entités.

Etape 4 : création d'une classe `AppMain`

Dans le répertoire `src/main/resources`, il faut définir le fichier `applications.properties`.

Construire une classe `AppMain` qui implémente `CommandLineRunner` et utilise les méthodes des repositories créés pour :

- Créer un `Event` correspondant à la formation `Dawan Spring Data` (date de la formation)
- Créer une `Address` correspondant à celle du centre de formation
- Créer un `User` Jérôme Merckling comme créateur de la formation
- Créer des `Guest` pour tous les participants à la formation `Spring Data`
- Créer un `Item` `laptop` dont le nombre est égal au nombre de participant

Faire une exécution afin de confirmer les résultats par une requête SQL dans pg-admin.

Etape 5 : création de méthodes de recherche générées à partir de leur nom

Dans cette section, il est demandé de modifier les interfaces du package `repository`.

- Modifier `GuestRepository` pour pouvoir
 - o Rechercher des `Guest` par leur attribut `name`.
 - o Rechercher la liste des `Guest` ayant participé à un événement `Spring Data Formation`
- Modifier `ItemRepository` pour pouvoir rechercher un `Item` ayant une `neededQuantity` et une `currentQuantity` données,
- Modifier `EventRepository` pour
 - o Trouver tous les `Item` associés à un `Event`,
 - o Trouver le seul `User` qui a créé cet `Event`
 - o Trouver les 4 premiers `Guest` associé à cet `Event`
- Modifier `UserRepository` pour trouver un `User` unique à partir de son `email` ou de

son login,

Etape 6 : création d'une classe AppMain

Modifier la classe `AppMain` qui implémente `CommandLineRunner` et effectuer les recherches personnalisées déclarées précédemment.

Comparer avec les requêtes similaires en SQL pour valider les nommages

Etape 7 : création d'interfaces Repository

Il est demandé de créer de nouvelles interfaces dans le package `repository` de sorte qu'elles héritent de `Repository` afin de pouvoir leur associer des requêtes JPA nommée

- `UserSimpleRepository` comprend une méthode annotée `@Query` pour rechercher un `User` à partir de son email unique,
- `EventSimpleRepository` comprend une méthode annotée `@Query` pour chercher un titre d' `Event` à partir de l'id, le type de retour est un `Optional`,
- `GuestSimpleRepository` comprend un méthode annotée `@Query` qui retourne un flux de `Guest` à partir de l'Event où ils ont participé,
- `AddressSimpleRepository` comprend un méthode annotée `@Query` qui retourne une liste d' `Address` connues étant donnée une ville choisie,
- `ItemSimpleRepository` comprend un méthode annotée `@Query` qui retourne un flux d' `Item` dont la `currentQuantity` est strictement inférieur à la `neededQuantity`,

Etape 8 : création d'une classe AppMain

Modifier la classe `AppMain` qui implémente `CommandLineRunner` et effectuer les recherches personnalisées déclarées précédemment.

Comparer avec les requêtes similaires en SQL pour valider les nommages

Etape 9 : création d'interfaces asynchrones

Il est demandé de créer de nouvelles interfaces dans le package `repository` de sorte qu'elles héritent de `Repository` afin de pouvoir leur associer des requêtes JPA nommée

- `UserAsyncRepository` comprend une méthode pour rechercher un `User` à partir de son id unique,
- `EventAsyncRepository` comprend une méthode annotée `@Query` pour chercher une liste d' `Event` créé par un `User`,
- `GuestAsyncRepository` comprend un méthode annotée `@Query` contenant une requête SQL qui retourne un `Future<Optional<Guest>>` à partir de l'id,
- `ItemAsyncRepository` comprend une méthode pour compter le nom d' `Item` et retourne un `Future<Integer>`,
- `AddressAsyncRepository` comprend une méthode de recherche dont le nom génère une requête afin de déterminer que la propriété `comments` contient une chaîne de caractère en paramètre sans que soit pris en compte les majuscule ou minuscule,

Etape 10 : modification d'une classe AppMain

Modifier la classe `AppMain` qui implémente `CommandLineRunner` et effectuer les recherches personnalisées déclarées précédemment.

Comparer avec les requêtes similaires en SQL pour valider les nommages