SpaceFox

Zeste de savoir

PostgreSQL + JPA/Hibernate + Kotlin + Spring-boot : trucs en vrac

11 mai 2019

Table des matières

Salut à toutes et à tous,

Récemment, j'ai fait deux projets avec Spring Boot, Hibernate et PostgreSQL, l'un en Java 11, l'autre en Kotlin. Ça m'a donné l'occasion de perdre plein de temps avec des petits détails idiots que je ne connaissais pas ou que j'ai eu du mal à trouver, donc je vous les partage.

C'est vraiment une collection de trucs en vracs, donc je pars du principe que vous savez un minimum de quoi je parle en lisant ça. Il n'y a pas vraiment d'ordre si ce n'est que je regroupe les astuces en fonction du domaine où elles s'appliquent.

L'un des deux projets doit gérer une volumétrie non négligeable (tables de dizaines de millions de lignes), ce qui a mené à la prise en compte de problématiques qui n'existent pas sur des volumétries plus petites.

1. PostgreSQL

La gestion d'une volumétrie importante avec PostgreSQL (version 10 ou plus) m'a fait découvrir quelque trucs que je ne connaissais pas à son sujet.

1.1. Pas d'index sur les clés étrangères par défaut

Par défaut, quand on crée une contrainte de clé étrangère, **PostgreSQL n'ajoute pas l'index correspondant**. Donc quand on va faire une jointure sur cette clé dans les requête (ce qui a de très fortes chances d'arriver par définition), ça va être lent.

Donc si vous avez une table du style :

```
create table if not exists page (
        id          integer not null constraint pk_page
            primary key,
        [...]
        category_id integer not null constraint fk_page__category
            references category
);
```

Pensez à ajouter :

```
create index if not exists i_fk_page__category on page
    (category_id);
```

1.2. Pas de gestion partielle des index multi-colonnes

Cas d'utilisation typique : vous avez une table pour gérer une relation n..m comme celle-ci :

```
create table if not exists page_image (
        page_id  integer not null constraint fk_page_image__page
            references page,
        image_id integer not null constraint fk_page_image__image
            references image,
        constraint pk_page_image primary key (page_id, image_id)
);
```

Il y a un index sur l'intégralité de la table, vu que l'intégralité de la table fait partie de la clé primaire. Sauf que si vous faites une jointure sur cette table, le moteur d'exécution n'aura besoin de filtrer **que sur une colonne**… et ne va pas y arriver, parce qu'il ne peut pas utiliser une seule colonne d'un index multi-colonnes.

La solution est donc d'ajouter un index supplémentaire pour chaque colonne (en admettant que vous suivez la relation dans les deux sens dans le code) :

```
create index if not exists i_fk_page_image__page  on page_image
    (page_id);
create index if not exists i_fk_page_image__image on page_image
    (image_id);
```

Trois index pour une table de liaison, c'est lourd, mais c'est le prix à payer pour que ça fonctionne correctement.

1.3. Compter et trier, c'est lent

1.3.1. Compter

Un `count(*)` (et quel que soit ce qu'il y a entre les parenthèses du `count`, c'est lent. Surtout si d'autres données sont récupérées en même temps. Surtout si c'est un `count(distinct …)`.

Une amélioration possible est de sortir le `count` dans une requête isolée : selon les cas ça peut faire des miracles, ou non, jusqu'à une certaine volumétrie. Dans le cas contraire, où à partir d'un certain nombre de lignes, il n'y aura plus le choix, il faudra dénormaliser[1]

1.3.2. Trier

Trier avec SQL, c'est lent. Dans tous les cas c'est plus rapide de trier en Java quand c'est possible, c'est-à-dire quand on est pas dans un cas du type *« trier et prendre les N premiers résultats »* (ce qui est fréquent avec la pagination).

1.4. Les droits sur un schéma

Au lieu de l'éternel `GRANT ALL` simple mais totalement pas sécurisé, on peut être un peu plus subtil :

```
1  CREATE USER mon_utilisateur WITH PASSWORD
       'un mot de passe compliqué';
2  CREATE ROLE mon_role; -- Permet d'avoir plusieurs utilisateurs avec
       les mêmes droits simplement
3  GRANT mon_role TO mon_utilisateur;
4
5  -- Ne pas oublier ce droit, sinon on a le droit de rien faire sur
       le schéma en question !
6  GRANT USAGE ON SCHEMA mon_schema TO mon_role;
7  -- Ne pas oublier REFERENCES sous peine de surprises…
8  GRANT SELECT, INSERT, UPDATE, DELETE, REFERENCES ON ALL TABLES IN
       SCHEMA mon_schema TO mon_role;
9  -- Indispensable puisqu'on a dit plus haut qu'on devait utiliser
       des sequences
10 GRANT USAGE, SELECT ON ALL SEQUENCES IN SCHEMA mon_schema TO
       mon_role;
11 -- Utile si vous avez des triggers
12 GRANT EXECUTE ON ALL FUNCTIONS IN SCHEMA mon_schema TO mon_role;
```

1. C'est-à-dire enregistrer au fur et à mesure la valeur totale dans une colonne à l'emplacement pertinent. Ça peut se faire avec des triggers en base de donnée, ou avec l'équivalent Hibernate.

2. JPA/Hibernate et PostgreSQL

2.1. Séquence Hibernate et séquences personnalisées

Si vous utilisez une séquence pour générer vos IDs (et vous devriez **toujours** utiliser une telle solution), la séquence par défaut d'Hibernate est déclarée ainsi :

```
1 create sequence if not exists hibernate_sequence increment by 1
      start with 1 no cycle cache 20;
```

Donc si vous voulez utiliser une séquence personnalisée pour cette table, la tentation est de faire ceci (d'après du code facile à trouver sur Internet quand on se renseigne sur ce sujet) :

```
1     @Id
2     @GeneratedValue(strategy = GenerationType.SEQUENCE, generator =
          "project_generator")
3     @SequenceGenerator(name="project_generator", sequenceName =
          "project_seq")
4     @Column(name = "project_id", nullable = false, updatable =
          false)
5     private Integer id;
```

Et de déclarer la séquence équivalente ainsi :

```
1 create sequence if not exists project_seq increment by 1 start with
      1 no cycle cache 20;
```

Ce qui ne fonctionnera pas : Hibernate va essayer d'utiliser des identifiants négatifs ou déjà utilisés.

La raison est technique et fait intervenir des valeurs par défaut incohérentes. En fait, la définition JPA de la séquence fait intervenir un paramètre `allocationSize`, le code précédent est en fait équivalent à celui-ci :

```
1     @Id
2     @GeneratedValue(strategy = GenerationType.SEQUENCE, generator =
          "project_generator")
3     @SequenceGenerator(name="project_generator", sequenceName =
          "project_seq", allocationSize = 50)
```

```
4       @Column(name = "project_id", nullable = false, updatable =
            false)
5       private Integer id;
```

C'est une optimisation qui curieusement n'existe pas sur la séquence Hibernate par défaut : pour éviter d'interroger la base à *chaque* insertion, le moteur va interroger la séquence seulement tous les `allocationSize` éléments, et va utiliser les `allocationSize` IDs, celui renvoyé par la séquence étant le *dernier* de la liste, et seulement ensuite re-demander un nouveau numéro de séquence. Donc, pour que ça fonctionne :

Le paramètre `allocationSize` de l'annotation `@SequenceGenerator` et l'incrément `increment by` de la séquence doivent **obligatoirement** avoir la même valeur (50 par défaut).

3. JPA/Hibernate

3.1. Pour arrêter de se prendre la tête avec les *fetch*

Rappel vite fait : toute relation définie avec JPA/Hibernate peut être paramétrée en *eager* « il faut aller chercher systématiquement les sous-éléments » ou en *lazy* « les sous éléments ne doivent être chargés que lorsque nécessaire ».

La première solution a tendance à aller chercher des éléments en base (via moult jointures ou requêtes supplémentaires) lorsqu'ils sont inutiles, la seconde à provoquer des erreurs lorsqu'on essaie d'accéder à ces sous-éléments hors du contexte Hibernate. Ces deux comportements, souvent mal compris et encore moins bien gérés, sont en grande partie à l'origine de la mauvaise réputation de l'outil.

Une solution simple pour gérer ses relations est de faire ceci :

1. Passer **toutes** les relations en mode lazy (`fetch = FetchType.LAZY`)
2. Utiliser les entity graphs pour indiquer à JPA/Hibernate de charger les sous-éléments dont on a besoin quand on en a besoin

On peut utiliser les `@NamedEntityGraph` comme dans l'exemple ci-dessus et presque tous les autres qu'on trouve sur Internet. Ils sont complets mais assez lourds à écrire. Mais surtout, si on a besoin de ne charger des entités directement enfant de l'entité principale (pas de sous-enfants donc), on peut directement annoter les interfaces du repository (exemples en Kotlin) :

```
1       @EntityGraph(attributePaths = ["indexImage", "gallery"])
2       fun findBySlug(name: String): Gallery
```

On peut aussi annoter une requête manuelle, ce qui évite de se taper les jointures à la main :

```
    @EntityGraph(attributePaths = ["indexImage"])
    @Query("select gallery from Gallery gallery")
    fun findAllForIndex(): List<Gallery>
```

3.2. Récupérer plusieurs entités en même temps

Quand le système des *entity graph* ne suffit plus, on peut aussi directement demander à JPA/Hibernate de nous renvoyer plusieurs entités ou colonnes en même temps :

```
    @Query("select user.id, language, profile " +
            "from User user " +
            "join user.language language " +
            "left join user.profiles profile " +
            "where user.id in :usersIds")
    List<Object[]> findDataByUsersIds(@Param("usersIds")
        Collection<Integer> usersIds);
```

Ici on va se retrouver avec une liste de tableau d'objets, chaque `Object[]` étant une ligne du résultat de la requête. Ici on aura donc dans l'ordre l'ID de l'utilisateur, la langue (sous forme d'entité) et le profil (sous forme d'entité). S'il y a des `null` dans le résultat de la requête, les colonnes correspondantes dans le tableau de résultat sont à `null` aussi.

Deux pièges avec cette technique :

1. On ne peut pas récupérer une seule ligne.
2. Attention aux produits cartésiens.

3.2.1. Récupérer une seule ligne

Il n'y a aucun moyen de récupérer une seule ligne avec plusieurs entités dessus. Les types de retours `List<Object[]>` et `Page<Object[]>` vont produire les résultats attendus, mais `Optional<Object[]>` et `Object[]` vont renvoyer en réalité des `Optional<Object[][]>` et `Object[][]`, donc des tableaux qui eux-même contiennent les lignes de résultats.

3.2.2. Attention aux produits cartésiens

Si l'entité principale a plusieurs relations 1..n ou n..m (celles matérialisés par des `Set` dans votre entité[2], la requête peut aboutir à un produit cartésien, donc un nombre démesuré de lignes et

au final une requête inefficace.

Au lieu de tenter de tout récupérer dans une seul requête, il vaut mieux faire :

— Une requête pour récupérer d'un coup l'entité principale et ses dépendances directes `@OneToOne` et `@ManyToOne`,
— Une requête par relation `@ManyToOne` et `@ManyToMany`

Ça fait plus de requêtes mais garantit de ne récupérer que le nombre strictement nécessaire de lignes ; de plus on peut paralléliser ces requêtes.

3.3. Éviter de récupérer une entité entière avec les projections

Par défaut les requêtes JPA/Hibernate récupèrent *toutes* les colonnes de la table considérée, ce qui peut être long et sous-optimal, d'autant plus que la table a beaucoup de grosses colonnes dont on a pas besoin.

On peut utilise les **projections** pour éviter ça. Une projection, c'est simplement une interface avec des **getters** (pas besoin des setters correspondants ni d'implémentation) qui vont indiquer au système quels éléments doivent être récupérés. L'héritage est géré. Par exemple, si j'ai une entité `User` dont je veux uniquement l'ID et le login, je peux utiliser une projection de ce type :

```
1 public interface Identifiable<T> {
2     T getId();
3 }
4
5 public interface SimpleUserProjection extends Identifiable<Integer>
      {
6     String getLogin();
7 }
```

Et ça peut s'utiliser directement comme type de retour dans les repository :

```
1     List<SimpleUserProjection> findAllSimpleByProfiles_Id(Integer
          id);
```

Ce qui va me générer automatiquement une requête qui ne récupérera dans la base de données que les IDs et logins des utilisateurs dont le `profil_id` est celui passé en paramètre (à travers une jointure n..m ici).

Apparemment on peut faire des choses bien plus compliquées, notamment avec les relations, mais je n'en ai jamais eu besoin.

2. Ou des `List` mais c'est déconseillé parce que ça a des implications potentiellement catastrophiques en terme de performances. Vos `@OneToMany` et `@ManyToMany` devraient systématiquement être matérialisées par des `Set`.

3.3.1. Tester les projections

Une projection étant une interface munie exclusivement de *getters*, elle peut-être casse-pieds à utiliser dans les test unitaires.

Heureusement c'est prévu par le framework, je vous renvoie à cette documentation . En gros il s'agit d'utiliser une *factory* dédiée :

```
1 private ProjectionFactory factory = new
      SpelAwareProxyProjectionFactory();
2 private final SimpleUserProjection userProjection =
      factory.createProjection(
3         SimpleUserProjection.class,
4         Map.of("id", 100, "login", "user100"));
```

3.4. Monitorez vos requêtes SQL !

On peut très facilement garder un œil sur ce que JPA/Hibernate génère comme requêtes vers PostgreSQL en passant les bons paramètres à la configuration Spring (ici en Yaml) :

```
```

À noter qu'afficher les *valeurs* des paramètres des requêtes (à la place des `?`) est beaucoup plus lourd.

4. Kotlin avec Spring Boot et JPA/Hibernate

4.1. Entités avec des relations en *lazy* (Kotlin + JPA/Hibernate)

Le concept de *data class* en Kotlin donne très envie de déclarer ses entités de cette manière :

```
1  @Entity
2  data class Page(
3          @Id
4          @GeneratedValue(strategy = GenerationType.AUTO)
5          val id: Int,
6
7          @Column(nullable = false)
8          val name: String,
9
10         // etc…
11 )
```

Sauf que ça ne fonctionne pas.

Plus exactement, ça fonctionne tant qu'on a pas de relation en *lazy*, et comme on l'a vu tout à l'heure, on aimerait bien mettre *toutes* nos relations en *lazy*. En l'état, le système va silencieusement considérer que toutes les relations sont en *eager*, et probablement vous générer beaucoup trop de requêtes.

La raison est très technique : Hibernate a besoin d'étendre les entités pour sa gestion interne, et a un comportement par défaut pour gérer les cas où il ne peut pas faire d'extension. C'est un problème qui n'arrive presque jamais en Java parce que personne ne mets ses entités en `final`. Or, Kotlin mets tous ses éléments en `final` *par défaut*. Il faut donc autoriser l'extension de notre classe.

La bonne façon de déclarer une entité en Kotlin est la suivante :

```kotlin
@Entity
open class Page(
        @Id
        @GeneratedValue(strategy = GenerationType.AUTO)
        open val id: Int,

        @Column(nullable = false)
        open val name: String,

        // etc…
)
```

4.2. Système de validation (Kotlin + Spring Boot)

Spring (Boot ou non) propose un système de validation des données par annotations. Par exemple ce DTO en Java :

```java
public class UserDto {
    // [...]

    @NotBlank(message = "Name is mandatory")
    private String name;

    @NotBlank(message = "Email is mandatory")
    private String email;
}
```

Ce DTO peut être directement utilisé dans ce contrôleur :

```
@Controller
public class UserController {

    @PostMapping("/users")
    ResponseEntity<String> addUser(@Valid @RequestBody UserDto
        user, BindingResult binding, Model model) {
        if (binding.hasErrors()) {
            return "error";
         }
         return "success";
    }
    // [...]
}
```

Avec ces simples lignes, les contraintes sur le `UserDto` vont être validées et le `BindingResult` sera rempli automatiquement, avec toutes les erreurs le cas échéant.

? Où est le rapport avec Kotlin ?

La tentation est d'implémenter la version Kotlin du DTO de cette façon :

```
data class UserDto(
    @NotBlank(message = "Name is mandatory")
    private val name: String? = null

    @NotBlank(message = "Email is mandatory")
    private val email: String? = null

    // [...]
)
```

Mais de cette façon, les annotations vont porter sur les paramètres du constructeur, et ne seront pas vues par le moteur de validation qui les cherche sur les champs – et évidemment ça va ne rien valider mais ne pas renvoyer d'erreur.

La bonne façon de déclarer ces annotations en Kotlin est de préciser qu'elles doivent s'appliquer sur les champs, ce qui donne :

```
data class UserDto(
    @field:NotBlank(message = "Name is mandatory")
    private val name: String? = null

```

```
5       @field:NotBlank(message = "Email is mandatory")
6       private val email: String? = null
7
8       // [...]
9   )
```

Voilà, y'a déjà pas mal de trucs. Peut-être que j'en rajouterai au fur et à mesure de mes découvertes !

Crédits du logo :

- *Logo PostgreSQL* *autorisé pour ce genre d'usage,*
- *Logo Hibernate* ,
- *Logo Spring* ,
- *Logo Kotlin* *(domaine public).*