Stream Exercice

Contexte

Soit le model de donnée suivant:

```
@Data
@Entity
@NoArgsConstructor
public class Customer {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  private String name;
  private Integer tier;
}

@Data
@NoArgsConstructor
@Entity
@Table(name = "product_order")
public class Order {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  private LocalDate orderDate;
  private LocalDate deliveryDate;
  private String status;

  @ManyToOne
  @JoinColumn(name = "customer_id")
  private Customer customer;

  @ManyToMany
  @JoinTable(
```

```
        name = "order_product_relationship",
        joinColumns = { @JoinColumn(name = "order_id") },
        inverseJoinColumns = { @JoinColumn(name = "product_id") }
    )
    @ToString.Exclude
    Set<Product> products;

}



@Data
@NoArgsConstructor
@Entity
public class Product {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    private String category;
    @With private Double price;

    @ManyToMany(mappedBy = "products")
    @ToString.Exclude
    private Set<Order> orders;
}
```

Opération Non terminal

- filter()
- map()
- flatMap
- distinct()
- sorted()
- peek()

Operation terminal

- anyMatch()
- collect()
- count()
- findFirst()

- min()
- max()
- sum()
- average()

Exercice 1 — Obtenir une liste de produits appartenant à la catégorie "Livres" avec un prix > 100

L'idée est d'utiliser un double filter et une collection.

Solution

```
List<Product> result = productRepo.findAll()
  .stream()
  .filter(p -> p.getCategory().equalsIgnoreCase("Livres"))
  .filter(p -> p.getPrice() > 100)
  .collect(Collectors.toList());
```

Exercice 2 — Obtenir une liste de commandes avec des produits appartenant à la catégorie "Bébé"

Vous devez partir du flux de données des entités de la commande, puis vérifier si les produits de la commande appartiennent à la catégorie "Bébé". Par conséquent, la logique de filtrage examine le flux de produits de chaque enregistrement de commande et utilise anyMatch() pour déterminer si un produit remplit les critères.

Solution

```
  List<Order> result = orderRepo.findAll()
        .stream()
        .filter(o ->
          o.getProducts()
```

```
            .stream()
            .anyMatch(p -> p.getCategory().equalsIgnoreCase("Bébé"))
        )
        .collect(Collectors.toList());
```

Exercice 3 - Obtenez une liste de produits avec la catégorie = "Jouets" puis appliquez une remise de 10 %

Dans cet exercice, vous verrez comment transformer des données à l'aide de l'API de flux. Après avoir obtenu une liste de produits avec une catégorie appartenant à "Jouets" à l'aide de filter(), vous pouvez ensuite appliquer une remise de 10 % sur le prix du produit en utilisant map().

Solution

```
List<Product> result = productRepo.findAll()
      .stream()
      .filter(p -> p.getCategory().equalsIgnoreCase("Jouets"))
      .map(p -> p.withPrice(p.getPrice() * 0.9))
      .collect(Collectors.toList());
```

Exercice 4 — Obtenir une liste des produits commandés par le client du niveau 2 entre le 1er février 2021 et le 1er avril 2021

Cet exercice illustre l'utilisation de flatMap(). Vous pouvez d'abord partir d'une liste de commandes, puis filtrer la liste par niveau de client et date de commande. Ensuite, récupérez la liste des produits de chaque enregistrement de commande et utilisez flatMap() pour émettre des enregistrements de produits dans le flux. Par exemple, si nous avons 3 enregistrements de commande et que chaque commande contient 10 produits, alors flatMap() émettra 10 éléments de données pour chaque enregistrement de commande, ce qui entraînera une sortie d'enregistrement

de produit de 30 (3 x 10) dans le flux.

Étant donné que la liste de produits contiendrait des enregistrements de produits en double si plusieurs commandes incluaient le même produit. Afin de générer une liste de produits unique, l'application de l'opération distinct() peut aider à produire la liste unique.

Solution

```
List<Product> result = orderRepo.findAll()
 .stream()
 .filter(o -> o.getCustomer().getTier() == 2)
 .filter(o -> o.getOrderDate().compareTo(LocalDate.of(2021, 2, 1)) >= 0)
 .filter(o -> o.getOrderDate().compareTo(LocalDate.of(2021, 4, 1)) <= 0)
 .flatMap(o -> o.getProducts().stream())
 .distinct()
 .collect(Collectors.toList());
```

Exercice 5 — Obtenir les produits les moins chers de la catégorie « Livres »

L'un des moyens efficaces d'obtenir le produit au prix le plus bas est de trier la liste de produits par prix dans un ordre croissant et d'obtenir le premier élément. L'API Java Stream fournit une opération **sorted ()** pour le tri des données de flux en fonction d'attributs de champ spécifiques. Afin d'obtenir le premier élément du flux, vous pouvez utiliser l'opération de terminal **findFirst().** L'opération renvoie le premier élément de données enveloppé par Facultatif car il est possible que le flux de sortie soit vide.

Cette solution ne peut renvoyer qu'un seul enregistrement de produit avec le prix le plus bas. S'il existe plusieurs enregistrements de produit avec le même prix le plus bas, la solution doit être modifiée de sorte qu'elle recherche d'abord le montant du prix le plus bas, puis filtre les enregistrements de produit par le montant du prix afin d'obtenir une liste de produits avec le même prix le plus bas.

Solution

```
Optional<Product> result = productRepo.findAll()
       .stream()
       .filter(p -> p.getCategory().equalsIgnoreCase("Livres"))
       .sorted(Comparator.comparing(Product::getPrice))
       .findFirst();
```

ou

```
Optional<Product> result = productRepo.findAll()
      .stream()
      .filter(p -> p.getCategory().equalsIgnoreCase("Livres"))
      .min(Comparator.comparing(Product::getPrice));
```

Exercice 6 — Obtenir les 3 dernières commandes passées

Semblable à l'exercice précédent, la solution évidente consiste à trier les enregistrements de commande par champ de date de commande. La partie délicate est que le tri doit cette fois être dans l'ordre décroissant afin que vous puissiez obtenir les enregistrements de commande avec la date de commande la plus récente. Cela peut être réalisé simplement en appelant **Comparator.reversed().**

Solution

```
List<Order> result = orderRepo.findAll()
      .stream()
      .sorted(Comparator.comparing(Order::getOrderDate).reversed())
      .limit(3)
      .collect(Collectors.toList());
```

Exercice 7 - Obtenez une liste des commandes qui ont été commandées le 15 mars 2021, enregistrez les enregistrements de commande sur la console, puis renvoyez sa liste de produits

Vous pouvez voir que cet exercice implique deux actions - (1) écrire des enregistrements de commande sur la console et (2) produire une liste de produits. Générer une sortie différente à partir d'un flux n'est pas possible, comment pouvons-nous répondre à cette exigence ? En plus

d'exécuter le flux de flux deux fois, l'opération **peek**() permet l'exécution de la logique système dans le cadre d'un flux de flux. L'exemple de solution exécute peek() pour écrire les enregistrements de commande dans la console juste après le filtrage des données, puis les opérations suivantes telles que **flatMap**() seront exécutées pour la sortie des enregistrements de produit.

Solution

```
List<Product> result = orderRepo.findAll()
    .stream()
    .filter(o -> o.getOrderDate().isEqual(LocalDate.of(2021, 3, 15)))
    .peek(o -> System.out.println(o.toString()))
    .flatMap(o -> o.getProducts().stream())
    .distinct()
    .collect(Collectors.toList());
```

Exercice 8 — Calculer la somme forfaitaire totale de toutes les commandes passées en février 2021

Tous les exercices précédents consistaient à générer une liste d'enregistrements par une opération de terminal, faisons quelques calculs cette fois. Cet exercice consiste à résumer tous les produits commandés en février 2021. Comme vous avez parcouru les exercices précédents, vous pouvez facilement obtenir la liste des produits à l'aide des opérations filter() et flatMap(). Ensuite, vous pouvez utiliser l'opération mapToDouble() pour convertir le flux en un flux de type de données Double en spécifiant le champ de prix comme valeur de mappage. Enfin, l'opération de terminal sum() vous aidera à additionner toutes les valeurs et à renvoyer la valeur totale.

Solution

```
Double result = orderRepo.findAll()
    .stream()
    .filter(o -> o.getOrderDate().compareTo(LocalDate.of(2021, 2, 1)) >= 0)
    .filter(o -> o.getOrderDate().compareTo(LocalDate.of(2021, 3, 1)) < 0)
    .flatMap(o -> o.getProducts().stream())
    .mapToDouble(p -> p.getPrice())
    .sum();
```

Exercice 9 — Calculer le paiement moyen d'une commande passée le 14 mars 2021

En plus de la somme totale, l'API de flux offre également une opération pour le calcul de la valeur moyenne. Vous constaterez peut-être que le type de données de retour est différent de **sum**() car il s'agit d'un type de données facultatif. La raison en est que le flux de données serait vide et que le calcul ne produira donc pas de valeur moyenne pour un flux de données vide.

Solution

```
    Double result = orderRepo.findAll()
         .stream()
         .filter(o -> o.getOrderDate().isEqual(LocalDate.of(2021, 3, 15)))
         .flatMap(o -> o.getProducts().stream())
         .mapToDouble(p -> p.getPrice())
         .average().getAsDouble();
```

Exercice 10 - Obtenir une collection de chiffres statistiques (c'est-à-dire somme, moyenne, max, min, nombre) pour tous les produits de la catégorie "Livres"

Que faire si vous avez besoin d'obtenir la somme, la moyenne, le max, le min et le compte en même temps ? Devrions-nous exécuter le flux de données 5 fois pour obtenir ces chiffres un par un ? Une telle approche n'est pas tout à fait efficace. Heureusement, l'API de flux fournit un moyen pratique d'obtenir toutes ces valeurs à la fois en utilisant l'opération de terminal **summaryStatistics**(). Il renvoie un type de données **DoubleSummaryStatistics** qui contient tous les chiffres requis.

Solution

```
DoubleSummaryStatistics statistics = productRepo.findAll()
    .stream()
    .filter(p -> p.getCategory().equalsIgnoreCase("Livres"))
    .mapToDouble(p -> p.getPrice())
```

```
    .summaryStatistics();

  System.out.println(String.format("count = %1$d, average = %2$f, max = %3$f, min = %4$f, sum = %5$f",
    statistics.getCount(), statistics.getAverage(), statistics.getMax(), statistics.getMin(), statistics.getSum())));
```

Exercice 11 - Obtenir une carte de données avec l'identifiant de la commande et le nombre de produits de la commande

À l'exception du calcul de valeur, tous les exercices précédents génèrent simplement une liste d'enregistrements. La classe d'assistance Collectors fournit un certain nombre d'opérations utiles pour la consolidation des données et la sortie de la collecte de données. Examinons l'exercice pour créer une carte de données avec l'ID de commande comme clé tandis que la valeur associée est le nombre de produits. L'opération de terminal **Collectors.toMap**() accepte deux arguments pour spécifier respectivement la clé et la valeur.

Solution

```
    Map<Long, Integer>  result = orderRepo.findAll()
        .stream()
        .collect(
            Collectors.toMap(
                order -> order.getId(),
                order -> order.getProducts().size()
                )
            );
```

Exercice 12 — Produire une carte de données avec des enregistrements de commande regroupés par client

Cet exercice consiste à consolider une liste de commandes par client. Collectors.groupingBy() est une fonction pratique, vous pouvez simplement spécifier quel est l'élément de données clé, il regroupera ensuite les données pour vous.

Solution

```
    Map<Customer, List<Order>> result = orderRepo.findAll()
        .stream()
        .collect(
            Collectors.groupingBy(Order::getCustomer)
            );
```

Exercice 13 — Produire une carte de données avec l'enregistrement de la commande et la somme totale des produits

La sortie de la carte de données cette fois n'est pas une simple extraction de champs de données du flux, vous devez créer un sous-flux pour chaque commande afin de calculer la somme totale du produit. Depuis, l'élément clé est l'enregistrement de la commande lui-même au lieu d'un identifiant de commande, donc Function.identity() est utilisé pour indiquer à Collectors.toMap() d'utiliser l'élément de données comme clé.

Solution

```
Map<Order, Double> result = orderRepo.findAll()
       .stream()
       .collect(
         Collectors.toMap(
             Function.identity(),
             order -> order.getProducts().stream()
                    .mapToDouble(p -> p.getPrice()).sum()
         )
         );
```

Exercice 14 — Obtenir une carte de données avec la liste des noms de produits par catégorie

Cet exercice vous aide à vous familiariser avec la manière de transformer la sortie de données des entrées de carte de données. Si vous utilisez uniquement Collectors.groupingBy(Product::getCategory), la sortie sera Map<Category, List of Products> mais la sortie attendue devrait être Map<Category, List of Product Name>. Vous pouvez utiliser Collectors.mapping() pour convertir les objets produit en noms de produit pour la construction de la carte de données.

Solution

```
Map<String, List<String>> result = productRepo.findAll()
        .stream()
        .collect(
            Collectors.groupingBy(
                Product::getCategory,
                Collectors.mapping(product -> product.getName(), Collectors.toList()))
            );
```

Exercice 15 — Obtenez le produit le plus cher par catégorie

Semblable à la transformation de données à l'aide de Collectors.mapping(), Collectors.maxBy() permet d'obtenir l'enregistrement avec la valeur maximale dans le cadre de la construction de la carte de données. En fournissant un comparateur de prix de produit, maxBy() est capable d'obtenir le produit avec la plus grande valeur pour chaque catégorie.

Solution

```
  Map<String, Optional<Product>> result = productRepo.findAll()
        .stream()
        .collect(
            Collectors.groupingBy(
                Product::getCategory,
                Collectors.maxBy(Comparator.comparing(Product::getPrice)))
```

Revision #3
Created 19 June 2022 09:30:32 by ggpilou2
Updated 19 June 2022 10:52:23 by ggpilou2