<https://www.codeflow.site/ru/article/hibernate-criteria-queries>

JPA Критерии Запросы

**1. Обзор**

В этом руководстве мы обсудим очень полезную функцию JPA - Criteria Queries.

Это не только позволяет нам писать запросы без использования исходного SQL, но также дает нам некоторый объектно-ориентированный контроль над запросами, который является одной из основных функций Hibernate. API Criteria позволяет нам программно создавать объект запроса критериев, где мы можем применять различные виды правил фильтрации и логических условий.

* Начиная с Hibernate 5.2, API Hibernate Criteria устарел, а новые разработки сфокусированы на API JPA Criteria. \*\* Мы рассмотрим, как использовать Hibernate и JPA для создания Criteria Queries.

**2. Зависимости Maven**

Для иллюстрации API мы будем использовать эталонную реализацию JPA - Hibernate.

Чтобы использовать Hibernate, убедитесь, что вы добавили его последнюю версию в свой файл *pom.xml*:

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-core</artifactId>

<version>5.3.2.Final</version>

</dependency>

Последнюю версию Hibernate можно найти по адресу [here[here .](https://search.maven.org/classic/" \l "search%7Cgav%7C1%7Cg%3A%22org.hibernate%22%20AND%20a%3A%22hibernate-core%22)

**3. Простой пример с использованием критериев**

Давайте начнем с рассмотрения того, как получать данные с помощью запросов Criteria.

Мы посмотрим, как получить все экземпляры определенного класса из базы данных.

У нас есть класс *Item*, который представляет кортеж *«ITEM»*в базе данных:

public class Item implements Serializable {

private Integer itemId;

private String itemName;

private String itemDescription;

private Integer itemPrice;

//standard setters and getters

}

Давайте посмотрим на простой критерий запроса, который извлечет все строки « *ITEM»*из базы данных:

Session session = HibernateUtil.getHibernateSession();

CriteriaBuilder cb = session.getCriteriaBuilder();

CriteriaQuery<Item> cr = cb.createQuery(Item.class);

Root<Item> root = cr.from(Item.class);

cr.select(root);

Query<Item> query = session.createQuery(cr);

List<Item> results = query.getResultList();

Приведенный выше запрос является простой демонстрацией того, как получить все предметы.

Давайте посмотрим, что было сделано, шаг за шагом:

- создайте экземпляр *Session*из объекта *SessionFactory,*

- создайте экземпляр C *\_riteriaBuilder \_*, вызвав *getCriteriaBuilder ()*метод ,

- создайте экземпляр *CriteriaQuery*, вызвав *CriteriaBuilder createQuery () метод*,

- создайте экземпляр *Query*, вызвав *Session createQuery ()* метод ,

- вызовите метод *getResultList ()*объекта *query*, который дает нам результаты, достижения

Теперь, когда мы рассмотрели основы, давайте перейдем к некоторым особенностям запроса критериев:

**3.1. Использование *Expressions***

*CriteriaBuilder*может использоваться для ограничения результатов запроса в зависимости от конкретных условий. Используя метод *CriteriaQuery where ()*и предоставьте *Expressions*, созданный *CriteriaBuilder.*

Вот несколько примеров часто используемых *Expressions*:

Чтобы получить предметы стоимостью более 1000:

cr.select(root).where(cb.gt(root.get("itemPrice"), 1000));

Далее, получение элементов, имеющих *itemPrice*меньше 1000:

cr.select(root).where(cb.lt(root.get("itemPrice"), 1000));

Элементы с *itemNames*содержат *Chair*:

cr.select(root).where(cb.like(root.get("itemName"), "%chair%"));

Записи, имеющие *itemPrice*от 100 до 200:

cr.select(root).where(cb.between(root.get("itemPrice"), 100, 200));

Чтобы проверить, является ли данное свойство нулевым:

cr.select(root).where(cb.isNull(root.get("itemDescription")));

Чтобы проверить, не является ли данное свойство нулевым:

cr.select(root).where(cb.isNotNull(root.get("itemDescription")));

Вы также можете использовать методы *isEmpty ()*и *isNotEmpty ()*, чтобы проверить, является ли *List*в классе пустым или нет.

Теперь неизбежно возникает вопрос, можем ли мы объединить два или более из приведенных выше сравнений или нет. Ответ, конечно, да - **Criteria API позволяет нам легко объединять выражения**:

Predicate[]predicates = new Predicate[2];

predicates[0]= cb.isNull(root.get("itemDescription"));

predicates[1]= cb.like(root.get("itemName"), "chair%");

cr.select(root).where(predicates);

Чтобы добавить два выражения с логическими операциями:

Predicate greaterThanPrice = cb.gt(root.get("itemPrice"), 1000);

Predicate chairItems = cb.like(root.get("itemName"), "Chair%");

элементы с определенными выше условиями объединяются с *Logical OR*:

cr.select(root).where(cb.or(greaterThanPrice, chairItems));

Чтобы получить элементы, соответствующие вышеуказанным условиям, объединенным с *Logical AND*:

cr.select(root).where(cb.and(greaterThanPrice, chairItems));

**3.2. Сортировка**

Теперь, когда мы знаем основное использование *Criteria*, давайте посмотрим на функции сортировки *Criteria*.

В следующем примере мы упорядочиваем список в порядке возрастания имени, а затем в порядке убывания цены:

cr.orderBy(

cb.asc(root.get("itemName")),

cb.desc(root.get("itemPrice")));

**3.3. Проекции, агрегаты и функции группировки**

До сих пор мы рассмотрели большинство основных тем. Теперь давайте посмотрим на различные агрегатные функции:

Получить количество строк:

CriteriaQuery<Long> cr = cb.createQuery(Long.class);

Root<Item> root = cr.from(Item.class);

cr.select(cb.count(root));

Query<Long> query = session.createQuery(cr);

List<Long> itemProjected = query.getResultList();

Ниже приведен пример агрегатных функций:

*Aggregate*функция для *Average*:

CriteriaQuery<Double> cr = cb.createQuery(Double.class);

Root<Item> root = cr.from(Item.class);

cr.select(cb.avg(root.get("itemPrice")));

Query<Double> query = session.createQuery(cr);

List avgItemPriceList = query.getResultList();

Другие полезные методы агрегирования: *sum ()*, *max ()*, *min ()***,***count ()*и т. Д.

**4. Преимущество перед HQL**

В предыдущих разделах мы рассмотрели, как использовать критерии запросов.

Основным и наиболее значительным преимуществом запросов Criteria над HQL является создание динамических запросов, а также красивый, чистый, объектно-ориентированный API.

Мы можем просто написать более гибкие, динамические запросы по сравнению с простым HQL. Логика может быть реорганизована с помощью IDE и обладает всеми преимуществами безопасности типов в самом языке Java.

Конечно, есть и некоторые недостатки, особенно в отношении более сложных соединений.

Так что, вообще говоря, нам придется использовать лучший инструмент для работы - это может быть Criteria API в большинстве случаев, но в определенных случаях нам придется перейти на более низкий уровень.