QueryDSL: Предикаты

QueryDSL Predicate — это мощный и чрезвычайно гибкий инструмент для работы с БД и просто подарок для Java-разработчиков, которые не очень хорошо разбираются в SQL (или совсем не разбираются), поскольку предикаты позволяют работать с БД при помощи привычного объектного представления сущностных зависимостей.

Предикаты позволяют работать с элементами базы данных как с обычными полями класса. При сборке gradle создаёт специальные классы зависимостей, через которые и происходит поиск нужных записей в БД.

Мы создадим сущности, с которыми будем работать. Пусть это будут User с полями name и age и UserGroup, которые будут наследоваться от AbstractEntity. Создадим между ними связь один-ко-многим — в одной группе может находиться много юзеров. Предикаты будем разбирать только на User.  
  
***AbstractEntity:***

# package entity;

# import javax.persistence.\*;

# @MappedSuperclass

# public class AbstractEntity {

# private Long id;

# @Id

# @Column(name = "id")

# @SequenceGenerator(name = "general\_seq", sequenceName = "generalSequenceGenerator")

# @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "general\_seq")

# public Long getId() {

# return id;

# }

# public void setId(Long id) {

# this.id = id;

# }

# @Override

# public boolean equals(Object o) {

# if (this == o) return true;

# if (o == null || getClass() != o.getClass()) return false;

# AbstractEntity that = (AbstractEntity) o;

# return id != null ? id.equals(that.id) : that.id == null;

# }

# @Override

# public int hashCode() {

# return id != null ? id.hashCode() : 0;

# }

# }

***User:***

# package entity;

# import javax.persistence.\*;

# @Entity

# @Table(name = "users")

# public class User extends AbstractEntity {

# private String name;

# private Integer age;

# private UserGroup group;

# @Column(name = "name")

# public String getName() { return name; }

# public void setName(String name) { this.name = name; }

# @Column(name = "age")

# public Integer getAge() { return age; }

# public void setAge(Integer age) { this.age = age; }

# @ManyToOne(fetch = FetchType.LAZY)

# @JoinColumn(name = "category")

# public UserGroup getGroup() { return group; }

# public void setGroup(UserGroup group) { this.group = group; }

# @Override

# public boolean equals(Object o) {

# if (this == o) return true;

# if (o == null || getClass() != o.getClass()) return false;

# if (!super.equals(o)) return false;

# User user = (User) o;

# if (name != null ? !name.equals(user.name) : user.name != null) return false;

# if (age != null ? !age.equals(user.age) : user.age != null) return false;

# return group != null ? group.equals(user.group) : user.group == null;

# }

# @Override

# public int hashCode() {

# int result = super.hashCode();

# result = 31 \* result + (name != null ? name.hashCode() : 0);

# result = 31 \* result + (age != null ? age.hashCode() : 0);

# result = 31 \* result + (group != null ? group.hashCode() : 0);

# return result;

# }

# }

***UserGroup:***

package entity;

import javax.persistence.\*;

import java.util.List;

@Entity

@Table(name = "user\_groups")

public class UserGroup extends AbstractEntity {

private String name;

private List<User> users;

@Column(name = "name")

public String getName() { return name; }

public void setName(String name) { this.name = name; }

@OneToMany(cascade = CascadeType.ALL, fetch = FetchType.LAZY, mappedBy = "group")

public List<User> getUsers() { return users; }

public void setUsers(List<User> users) { this.users = users; }

@Override

public boolean equals(Object o) {

if (this == o) return true;

if (o == null || getClass() != o.getClass()) return false;

if (!super.equals(o)) return false;

UserGroup userGroup = (UserGroup) o;

if (name != null ? !name.equals(userGroup.name) : userGroup.name != null) return false;

return users != null ? users.equals(userGroup.users) : userGroup.users == null;

}

@Override

public int hashCode() {

int result = super.hashCode();

result = 31 \* result + (name != null ? name.hashCode() : 0);

result = 31 \* result + (users != null ? users.hashCode() : 0);

return result;

}

}

Отлично, сущности у нас есть, теперь давайте подумаем, что может потребоваться нам в работе с нашим репозиторием. Мы можем:

* Найти всех юзеров в требуемых пределах возраста, как включая пределы, так и исключая.
* Найти пользователей по ID.
* Найти всех пользователей, состоящих в определённой группе, а также, пользователей, состоящих в нескольких группах.
* Найти пользователей по имени.

… искать, сортировать, фильтровать пользователей как угодно, в зависимости от наших потребностей и количества полей.  
  
Для работы с базой данных через QueryDSL нам потребуется отдельный, кастомизированный, репозиторий. Он расширяется от JpaRepository также, как и в других случаях, когда мы работаем с БД через Spring JPA, но кастомизируется под QueryDSL:

@NoRepositoryBean

public interface ExCustomRepository<T extends AbstractEntity, P extends EntityPathBase<T>, ID extends Serializable>

extends JpaRepository<T, ID>, QuerydslPredicateExecutor<T>, QuerydslBinderCustomizer<P> {

@Override

default void customize(QuerydslBindings bindings, P root) {

}

}

Важное замечание. Если при попытке поднять Spring у Вас выскакивает *org.springframework.data.mapping.PropertyReferenceException: No property customize found for type User*, значит, Вы не реализовали (Implement Methods) метод *customize()*. Просто переопределите его, этого будет достаточно (если Вы не хотите кастомизировать и его).  
  
Итак, для работы с репозиторием нам будет достаточно унаследовать наш интерфейс UserRepository от ExCustomRepository следующим образом, явно указав User, QUser и Long:

@Repository

public interface UserRepository extends ExCustomRepository<User, QUser, Long> {

}

Теперь, наконец, создадим сервисный класс, в котором будем обращаться к базе и искать пользователей (пока методы в нем не заполняем):

@Service

public class UserService {

@Autowired

UserRepository repository;

//ищем по возрасту, исключая границы

public List<User> getByAgeExcluding(Integer minAge, Integer maxAge) { }

//ищем по возрасту, включая границы

public List<User> getByAgeIncluding(Integer minAge, Integer maxAge) { }

//ищем по ID

public User getById(Long id) { }

//ищем по группам

public List<User> getByGroups(List<UserGroup> groups) { }

//ищем по имени

public List<User> get(String name) { }

}

Для того, чтобы Spring мог работать с объектным представлением табличных сущностей, ему необходимо создать связь между ними. Все связи он по умолчанию помещает в папку *build.generated.source.apt.структура\_проекта*, для того, чтобы создать эти связи нужно очистить проект и собрать его классы. В gradle это достигается последовательным исполнением задач clean и classes (gradle -> Tasks -> build -> clean, classes). Если в *build.generated.source.apt* появилась структура проекта, а в ней классы с приставкой Q, значит, Вы всё сделали правильно.

Предположим, что Вы всё сделали правильно и вышеописанные классы появились. Давайте, как мы и хотели, запросим из репозитория всех юзеров, допустим, от 18 до 60 лет. Как я уже упоминал, связь между табличными сущностями в QueryDSL формируется в соответствующем классе с приставкой Q. Для класса User это будет QUser. QUser — это весь репозиторий. В нём есть юзеры: *QUser.user*, у юзеров есть имена: *QUser.user.name*, а также, в нашем случае, возраст: *QUser.user.age*. Таким образом, чтобы получить возраст, мы будем работать с *QUser.user.age*.  
  
В QueryDSL есть 4 основных метода, позволяющих выдавать определённый результат:

* findOne() — позволяет искать какой-то один элемент. Вы должны быть уверены, что искомый элемент в БД только один, иначе дратути исключение.
* findAll() и несколько его перегрузок — возвращает iterable список найденных записей, отвечающих условиям. Обычно этот список потом приходится оборачивать в List (у нас это будет).
* count() — возвращает количество найденных элементов.
* exists() — возвращает булевское значение, существует такой элемент в таблице или нет.

Найти эти методы и более подробно их изучить можно в org.springframework.data.querydsl.QuerydslPredicateExecutor. Как мы видим по именам пакетов, их предоставляет Spring.

Для выполнения условий же по элементам существует несметное количество методов, хранящихся в com.querydsl.core.types.dsl.SimpleExpression. Их мы изучим подробнее.

Итак, в первом методе нам нужно получить всех юзеров в заданном возрастном диапазоне.  
  
В HQL этот запрос выглядел бы так:

SELECT u FROM User u WHERE u.age BETWEEN :minAge AND :maxAge

В реализации же QueryDSL этот метод будет выглядеть так:

public List<User> getByAgeExcluding(Integer minAge, Integer maxAge) {

return Lists.newArrayList(repository.findAll(QUser.user.age.between(minAge, maxAge)));

}

Мы ищем всех пользователей (findAll()), у которых возраст (*QUser.user.age*) в заданном диапазоне (between(minAge, maxAge)). И всё, это весь запрос. По этому запросу мы получаем готовый список пользоватей. Нам не нужно писать запросы на SQL, а при малейшем изменении переписывать заново, иначе всё повалится — QueryDSL обеспечивает максимальную гибкость, которую только может обеспечить объектная связь сущностей, а значит, такой запрос будет, при необходимости, легко рефакториться и никогда не сломается.

Это было небольшое лирическое отступление, а у нас впереди ещё 4 заявленных примера. Перейдём к следующему. Мы нашли всех юзеров в диапазоне, но подборка будет исключать сами значения границ. Для включения значений границ в критерии поиска нам придётся воспользоваться другими методами:

* goe — greater or equal (больше или равно)
* loe — less or equal (меньше или равно)

Запрос у нас получится таким:

public List<User> getByAgeIncluding(Integer minAge, Integer maxAge) {

return Lists.newArrayList(repository.findAll(QUser.user.age.goe(minAge).and(QUser.user.age.loe(maxAge))));

}

Для того, чтобы выполнить этот запрос, нам нужно использовать два условия. Для этого мы применяем метод-связку and(), который фильтрует по всем связанным таким образом условиям. Фреймворк сначала выберет все объекты, которые больше или равны minAge, а потом — все объекты, которые меньше или равны maxAge — и всё в одном запросе. Таких связок может быть большое количество, существует также связка or() и другие, найти их можно в *com.querydsl.core.types.dsl.BooleanExpression*.  
  
Теперь давайте найдём пользователя по его ID. Конечно, это лучше всего делать при помощи соответствующего метода Spring JPA findById(), но, поскольку мы разбираем QueryDSL, составим соответствующий запрос:

public User getById(Long id) {

return repository.findOne(QUser.user.id.eq(id)).orElse(new User());

}

Мы используем оператор eq(), который ищет по полям, равным условию (eq = equals).  
  
Идём дальше. Для того, чтобы найти всех пользователей в группе, нам, в нашем случае, не нужно даже ничего искать — достаточно просто взять нужную UserGroup c её полем List, а если нам требуется найти всех юзеров, состоящих в нескольких группах? И эту задачу можно выполнить при помощи очень простого запроса через QueryDSL:

public List<User> getByGroups(List<UserGroup> groups) {

return Lists.newArrayList(repository.findAll(QUser.user.group.in(groups)));

}

В данном случае, оператор in() позволяет задать условием поиска не одно значение, а несколько.  
  
Ну и напоследок, найдём всех пользователей, чьё имя отлично от запрашиваемого (например, всех не Иванов). Запрос будет выглядеть так:

public List<User> get(String name) {

return Lists.newArrayList(repository.findAll(QUser.user.name.ne(name)));

}