As a rule of thumb, use bidirectional @OneToMany associations instead of unidirectional ones.

**Don’t Forget to Set mappedBy on the Parent-Side.**

The mappedBy attribute characterizes a bidirectional association and must be set on the parent-side.

In other words, for a bidirectional @OneToMany association, set mappedBy to @OneToMany on the parentside and add @ManyToOne on the child-side referenced by mappedBy . Via mappedBy, the bidirectional @OneToMany association signals that it mirrors the @ManyToOne child-side mapping.

In this case, we add in Author entity to the following:

*@OneToMany(cascade = CascadeType.ALL, mappedBy = "author")*

**Set orphanRemoval on the Parent-Side**

Setting orphanRemoval on the parent-side guarantees the removal of children without references. In other words, orphanRemoval is good for cleaning up dependent objects that should not exist without a reference from an owner object. In this case, we add orphanRemoval to the Author entity:

*@OneToMany(cascade = CascadeType.ALL, mappedBy = "author", orphanRemoval = true)*

**Use @JoinColumn to Specify the Join Column Name**

The join column defined by the owner entity (Book) stores the ID value and has a foreign key to the Author entity. It is advisable to specify the desired name for this column. This way, you avoid potential confusions/mistakes when referring to it (e.g., in native queries). In this case, we add @JoinColumn to the Book entity as follows:

*@JoinColumn(name = "author\_id")*

*@Entity public class Author implements Serializable {*

*…*

*@OneToMany(cascade = CascadeType.ALL, mappedBy = "author", orphanRemoval = true)*

*private List books = new ArrayList<>();*

*public void addBook(Book book) {*

*this.books.add(book);*

*book.setAuthor(this);*

*}*

*public void removeBook(Book book) {*

*book.setAuthor(null);*

*this.books.remove(book);*

*}*

*public void removeBooks() {*

*Iterator iterator = this.books.iterator();*

*while (iterator.hasNext()) {*

*Book book = iterator.next();*

*book.setAuthor(null); iterator.remove();*

*}*

*} // getters and setters omitted for brevity*

*@Override public String toString() {*

*return "Author{" + "id=" + id + ", name=" + name + ", genre=" + genre + ", age=" + age + '}';*

*}*

*}*

*@Entity public class Book implements Serializable {*

*…*

*@ManyToOne(fetch = FetchType.LAZY) @JoinColumn(name = "author\_id")*

*private Author author; // getters and setters omitted for brevity*

*@Override public boolean equals(Object obj) {*

*if(obj == null) { return false; } if (this == obj) { return true; } if (getClass() != obj.getClass()) { return false; } return id != null && id.equals(((Book) obj).id);*

*}*

*@Override public int hashCode() {*

*return 2021;*

*}*

*@Override public String toString() {*

*return "Book{" + "id=" + id + ", title=" + title + ", isbn=" + isbn + '}';*

*}*

*}*

**Why You Should Avoid the Unidirectional @OneToMany Association**

*@OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)*

*private List books = new ArrayList<>();*

Сгенерирует три таблицы для автора, книги и автор\_книга.

Будет много лишних запросов. Так писать не рекомендуется.

**Using @JoinColumn**

*@OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)*

*@JoinColumn(name = "author\_id")*

*private List books = new ArrayList<>();*

Теперь таблицы будет две.

Если мы сохраняем новую книгу нового автора.

1. Insert автора
2. Insert книгу
3. Update книгу, проставить id\_author

Лишний update!

Adding @JoinColumn can provide benefits over the regular unidirectional @OneToMany, but is not better than a bidirectional @OneToMany association. The additional UPDATE statements still cause a performance degradation.

As a rule of thumb, a unidirectional @OneToMany association is less efficient than a bidirectional @OneToMany or unidirectional @ManyToOne associations.

1. **How Efficient Is the Unidirectional @ManyToOne**

*@Entity public class Book implements Serializable {*

*…*

*@ManyToOne(fetch = FetchType.LAZY)*

*@JoinColumn(name = "author\_id")*

*private Author author;*

*... }*

It looks like unidirectional @ManyToOne association is quite efficient and it can be used whenever a bidirectional @OneToMany association is not needed. Again, try to avoid the unidirectional @OneToMany association.

@ManyToOne достаточно эффективна т.к генерирует 2 таблицы.

1. **How to Effectively Shape the @ManyToMany Association**

Always Use Set not List

*private Set books = new HashSet<>(); // in Autho*

*private Set authors = new HashSet<>(); // in Book*

Avoid CascadeType.ALL and CascadeType.REMOVE

*@ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE})*

*private Set books = new HashSet<>();*

Setting Up the Join Table

*@JoinTable(name = "author\_book",*

*joinColumns = @JoinColumn(name = "author\_id"),*

*inverseJoinColumns = @JoinColumn(name = "book\_id")*

Using Lazy Fetching on Both Sides of the Association

By default, the @ManyToMany association is lazy. Keep it this way! Don’t do this: *@ManyToMany(fetch=FetchType.EAGER)*

Choose the Owner of the Relationship

*@ManyToMany(mappedBy = "books")*

*private Set authors = new HashSet<>();*

Example:

*@Entity public class Author implements Serializable {*

*…*

*@ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE})*

*@JoinTable(name = "author\_book",*

*joinColumns = @JoinColumn(name = "author\_id"),*

*inverseJoinColumns = @JoinColumn(name = "book\_id") )*

*private Set books = new HashSet<>();*

*…*

*}*

*@Entity public class Book implements Serializable {*

*…*

*@ManyToMany(mappedBy = "books")*

*private Set authors = new HashSet<>();*

*…*

*}*

**Why Set Is Better than List in @ManyToMany**

Если использовать List при @ManyToMany, то при удалении из соединительной таблицы, будут много лишних insert.

Мы хотим удалить из соединительной таблицы одну связь id\_author+id\_book (книгу написали много авторов).

Из соединительной таблицы выберутся все записи по этой книге все id\_author+id\_book ( по этому автору), из списка удалится нужная id\_author + id\_book, потом остальные записи будут обратно вставляться в соединительную таблицу. Получается много лишних insert

Если использовать Set:

Будет сгенерирован код

*DELETE FROM author\_book\_set WHERE author\_id = ? AND book\_id = ?*

*Binding: [1, 2]*

Нет лишних insert

Using @OrderBy

*@ManyToMany(mappedBy = "books")*

*@OrderBy("name DESC")*

*private Set authors = new HashSet<>();*

если хотим получать авторов, отсортированных по имени

Using @OrderBy with HashSet will preserve the order of the loaded/fetched Set, but this is not consistent across the transient state. If this is an issue, to get consistency across the transient state as well, consider explicitly using LinkedHashSet instead of HashSet. So, for full consistency, use:

*@ManyToMany(mappedBy = "books")*

*@OrderBy("name DESC")*

*private Set authors = new LinkedHashSet<>();*

**Why and When to Avoid Removing Child Entities with CascadeType.Remove and orphanRemoval=true**

Разница между CascadeType.Remove и orphanRemoval

Каскадирование повторяет операцию с родительской сущностью на дочерних сущностях.

CascadeType.Remove(или CascadeType.ALL в которую он входит) удалит все связанные дочерние сущности.

orphanRemoval=true будет физически удалять записи из таблицы бд при удалении сущности

orphanRemoval=false будет делать update, отвязывать дочернюю сущность от родительской, но не будет удалять из бд

Вместе писать CascadeType.Remove(или CascadeType.ALL) и при этом orphanRemoval=true это избыточно

*// in Author.java*

*@OneToMany(cascade = CascadeType.ALL, mappedBy = "author", orphanRemoval = true)*

*private List books = new ArrayList<>();*

1. **Entity Graph**

Для производительности

**The main goal of the JPA Entity Graph is then to improve the runtime performance when loading the entity's related associations and basic fields.**

Briefly put, the JPA provider loads all the graph in one select query and then avoids fetching association with more SELECT queries. This is considered a good approach for improving application performance.

1. **How to Optimize Unidirectional/Bidirectional @OneToOne via @MapsId**

*@Entity public class Book implements Serializable {*

*…*

*@OneToOne(fetch = FetchType.LAZY)*

*@JoinColumn(name = "author\_id")*

*private Author author;*

*}*

*@Entity public class Author implements Serializable {*

*private static final long serialVersionUID = 1L;*

*@Id*

*@GeneratedValue(strategy = GenerationType.IDENTITY)*

*@OneToOne(mappedBy = "author", cascade = CascadeType.ALL, fetch = FetchType.LAZY)*

*private Book book;*

*}*

**@MapsId to the Rescue of @OneToOne**

*@Entity public class Book implements Serializable {*

*…*

*@Id (без генерации!)*

*private Long id;*

*@MapsId*

*@OneToOne(fetch = FetchType.LAZY)*

*@JoinColumn(name = "author\_id")*

*private Author author;*

*}*

В книге id маппится на внешний ключ, id не генерируется. Более чистая понятная структура бд

**Entity**

findBy и getOne

findBy действительно идет в бд и извлекает значение сущности. Жадная загрузка сущности

getOne создает proxy, возвращает ссылку с данным идентификатором. Ленивая загрузка сущности

findOne примерно тоже что getOne

**How to Use Java 8 Optional in Persistence Layer**

Optional способ сообщить – ‘нет результата’

**How to Map a Boolean to a Yes/No**

В таблице бд поле varchar(3), менять его не можем. (yes/no)

Хотим чтобы значение было Boolean

*@Entity*

*public class Author implements Serializable {*

*...*

*@NotNull private Boolean bestSelling;*

*}*

Нужно использовать конвертер

*@Converter(autoApply = true)*

*public class BooleanConverter implements AttributeConverter {*

*@Override*

*public String convertToDatabaseColumn(Boolean attr) {*

*return attr == null ? "No" : "Yes";*

*}*

*@Override*

*public Boolean convertToEntityAttribute(String dbData) {*

*return !"No".equals(dbData);*

*}*

*}*

**Fetching**

findById() method

1 способ:

List<Book>books = bookRepository.findAllById(List.of(1L, 2L, 5L)); - список сущностей по id

2 способ:

*@Query("SELECT b FROM Book b WHERE b.id IN ?1")*

*List<Book> fetchByMultipleIds(List ids);*

3 способ:

*List<Book> books = bookRepository.findAll( new InIdsSpecification(List.of(1L, 2L, 5L)));*

*public class InIdsSpecification implements Specification {*

*private final List ids;*

*public InIdsSpecification(List ids) { this.ids = ids; }*

*@Override*

*public Predicate toPredicate(Root root, CriteriaQuery cquery, CriteriaBuilder cbuilder) {*

*return root.in(ids);*

*// or*

*// Expression expression = root.get("id");*

*// return expression.in(ids); } }*

**How to Lazy Load the Entity Attributes via Hibernate Bytecode Enhancement**

*@Entity*

*public class Author implements Serializable {*

*private static final long serialVersionUID = 1L;*

*@Id private Long id;*

*@Lob private byte[] avatar; …}*

Поля загружаются жадно для каждой сущности. Нужно чтобы картинка аватар загружалась лениво.

**Spring Projection**

Если нужно извлечь данные только для чтения.

DTO и Spring Projection используются для одинаковой цели, но это не одно и тоже.

DTO реализуется через конструктор, и геттеры сеттеры, а Spring Projection через интерфейсы и автоматическую генерацию прокси.

*public interface AuthorNameAge {*

*String getName();*

*int getAge();*

*}*

Это Spting Projection. Определяем интерфейс с геттерами. За сценой Spring сгенерирует прокси интерфейс для каждого Entity объекта.

Такой интерфейс может быть определен внутри репозитория.

*@Repository*

*public interface AuthorRepository extends JpaRepository {*

*@Transactional(readOnly = true)*

*List<AuthorNameAge> findFirst2ByGenre(String genre);*

*public static interface AuthorNameAge {*

*String getName();*

*int getAge();*

*}*

*}*

Это еще называется проекция через интерфейсы.

Проекция через классы, когда вместо интерфейса пишем класс с конструктором, геттерами и сеттерами. Интерфейс может быть вложенным, а класс нет!

*public class AuthorNameAge {*

*private String name;*

*private int age;*

*public AuthorNameAge(String name, int age) {*

*this.name = name;*

*this.age = age;*

*} // getters, setters, equals() and hashCode() omitted for brevity*

*}*

**Как переиспользовать Spring Projection**

Предположим что есть сущность Author, в которой много полей name, age, email, address, genre.

В каком-то запросе нам нужны все поля, в другом только name + age, третьем name + email. Тогда писать проекции для всех запросов слишком расточительно.

Мы определяем спринг проекцию, содержащую максимальное число полей

*@JsonInclude(JsonInclude.Include.NON\_DEFAULT)*

*public interface AuthorDto {*

*public Integer getAge();*

*public String getName();*

*public String getGenre();*

*public String getEmail();*

*public String getAddress();*

*}*

Тут важна аннотация @JsonInclude(JsonInclude.Include.NON\_DEFAULT). Означает что не нужно сериализовать null значения

И затем используем эту проекцию в разных запросах.

*@Query("SELECT a.age AS age, a.name AS name, a.genre AS genre, " + "a.email AS email, a.address AS address FROM Author a")*

*List<AuthorDto> fetchAll();*

Здесь получим все поля.

*@Query("SELECT a.age AS age, a.name AS name, a.genre AS genre FROM Author a")*

*List<AuthorDto> fetchAgeNameGenre();*

А здесь только возраст и имя.

Закрытая проекция – это имена методов точно совпадают с именами полей сущности.

Открытая проекция – которая содержит имена, которые на совпадают с сущностью и которые вычисляются в runtime

**Проекция со связью many-to-one**

1. **Вложенная закрытая проекция**

*public interface BookDto {*

*public String getTitle();*

*public AuthorDto getAuthor();*

*interface AuthorDto {*

*public String getName();*

*public String getGenre();*

*}*

*}*

Полагаемся на Spring Data Query Builder механизм:

*@Repository*

*@Transactional(readOnly=true)*

*public interface BookRepository extends JpaRepository {*

*List<BookDto> findBy();*

*}*

Можно явно через джойн:

*@Query("SELECT b.title AS title, a AS author FROM Book b LEFT JOIN b.author a")*

*// or as a INNER JOIN*

*// @Query("SELECT b.title AS title, b.author AS author FROM Book b")*

*List<BookDto> findByViaQuery();*

Вернет автора в отдельной подсущности в книге:

*[*

*{ "title":"A History of Ancient Prague",*

*"author":{*

*"genre":"History",*

*"name":"Joana Nimar"*

*}*

*},*

*{ "title":"A People's History",*

*"author":{*

*"genre":"History",*

*"name":"Joana Nimar" } },*

*... ]*

1. **Простая закрытая проекция**

*public interface SimpleBookDto {*

*public String getTitle(); // of book*

*public String getName(); // of author*

*public String getGenre(); // of author }*

Здесь Spring Data Query Builder механизм не поможет, пишем явно:

*@Query("SELECT b.title AS title, a.name AS name, a.genre AS genre FROM Book b LEFT JOIN b.author a")*

*List<SimpleBookDto> findByViaQuerySimpleDto();*

Результат:

*[{ "title":"A History of Ancient Prague",*

*"genre":"History",*

*"name":"Joana Nimar" },*

*{ "title":"A People's History",*

*"genre":"History",*

*"name":"Joana Nimar"*

*},…]*

Здесь данные по книге и автору смешаны.

1. **Простая открытая проекция**

Открытая проекция позволяет определять методы, которые не совпадают с доменной моделью и возвращает вычисляет в рантайме. Существенно, что открытые проекции поддерживают виртуальные свойства.

public interface VirtualBookDto {

public String getTitle(); // of book

@Value("#{@authorMapper.buildAuthorDto(target.name, target.genre)}")

AuthorClassDto getAuthor();

}

public class AuthorClassDto {

private String genre;

private String name; // getters, setters, equals() and hashCode() omitted for brevity

}

SpEL expression

*@Component*

*public class AuthorMapper {*

*public AuthorClassDto buildAuthorDto(String genre, String name) {*

*AuthorClassDto authorClassDto = new AuthorClassDto();*

*authorClassDto.setName(name);*

*authorClassDto.setGenre(genre);*

*return authorClassDto; }*

*}*

В репозитории:

@Query("SELECT b.title AS title, a.name AS name, a.genre AS genre FROM Book b LEFT JOIN b.author a")

List<VirtualBookDto> findByViaQueryVirtualDto();

**One-to-many**

1. Закрытая вложенная проекция

*public interface AuthorDto {*

*public String getName();*

*public String getGenre();*

*public List<BookDto> getBooks();*

*interface BookDto {*

*public String getTitle();*

*}*

*}*

Как написать select для этой DTO?

А) *List<AuthorDto> findBy();*

если будем использовать Query Builder Mechanism , то получим много лишних запросов. Будет выбраны авторы, а потом для каждого сделан select по книгам. Это плохо для производительности.

Б) *@Query("SELECT a.name AS name, a.genre AS genre, b AS books FROM Author a INNER JOIN a.books b")*

*List<AuthorDto> findByViaQuery();*

Производительность ок, но не поддерживается структура дерева родитель-дочерние отношения в json. Получаем список, где каждая книга идет с автором

[ { "genre":"History",

"books":[

{ "title":"A History of Ancient Prague" }

],

"name":"Joana Nimar" },

{ "genre":"History",

"books":[

{ "title":"A People's History" }

],

"name":"Joana Nimar" },

{ "genre":"History",

"books":[

{ "title":"History Now" }

],

"name":"Joana Nimar" },

]

В) JPA JOIN FETCH

*@Query("SELECT a FROM Author a JOIN FETCH a.books")*

*Set<AuthorDto> findByJoinFetch();*

Поддерживает родитель-потомок отношения. У одного автора может быть несколько книг

[ { "genre":"Anthology",

"books":[

{ "title":"The Beatles Anthology" }

],

"name":"Mark Janel" },

{ "genre":"Horror",

"books":[

{ "title":"Carrie" },

{ "title":"Nightmare Of A Day" }

],

"name":"Olivia Goy" },

1. Простая закрытая проекция

*public interface SimpleAuthorDto {*

*public String getName(); // of author*

*public String getGenre(); // of author*

*public String getTitle(); // of book*

*}*

*@Query("SELECT a.name AS name, a.genre AS genre, b.title AS title FROM Author a INNER JOIN a.books b")*

*List<SimpleAuthorDto> findByViaQuerySimpleDto();*

Хорошая производительность, но не поддерживает отношения родитель-потомок в json

*[ { "genre":"History",*

*"title":"A History of Ancient Prague",*

*"name":"Joana Nimar"*

*},*

*{ "genre":"History",*

*"title":"A People's History",*

*"name":"Joana Nimar"*

*},*

*{ "genre":"History",*

*"title":"History Now",*

*"name":"Joana Nimar" }, ..*

*. ]*

1. **Трансформация List<Object> в DTO**

Извлечем данные в список и трансформируем в DTO

@Component

*public class AuthorTransformer {*

*public List<AuthorDto> transform(List<Object[]> rs) {*

*final Map<Long, AuthorDto> authorsDtoMap = new HashMap<>();*

*for (Object[] o : rs) {*

*Long authorId = ((Number) o[0]).longValue();*

*AuthorDto authorDto = authorsDtoMap.get(authorId);*

*if (authorDto == null) {*

*authorDto = new AuthorDto();*

*authorDto.setId(((Number) o[0]).longValue());*

*authorDto.setName((String) o[1]);*

*authorDto.setGenre((String) o[2]);*

*}*

*BookDto bookDto = new BookDto();*

*bookDto.setId(((Number) o[3]).longValue());*

*bookDto.setTitle((String) o[4]);*

*authorDto.addBook(bookDto);*

*authorsDtoMap.putIfAbsent(authorDto.getId(), authorDto);*

*}*

*return new ArrayList<>(authorsDtoMap.values());*

*}*

*}*

*public class AuthorDto implements Serializable {*

*private static final long serialVersionUID = 1L;*

*private Long authorId;*

*private String name;*

*private String genre;*

*private List<BookDto> books = new ArrayList<>(); // constructors, getters, setters*

*}*

*public class BookDto implements Serializable {*

*private static final long serialVersionUID = 1L;*

*private Long bookId; private String title; // constructors, getters, setters omitted for*

*}*

*@Query("SELECT a.id AS authorId, a.name AS name, a.genre AS genre, b.id AS bookId, b.title AS title FROM Author a " + "INNER JOIN a.books b")*

*List<Object[]> findByViaArrayOfObjectsWithIds();*

В сервисе трансформируем

*...*

*List<Object[]> authors = authorRepository.findByViaArrayOfObjectsWithIds();*

*List< AuthorDto> authorsDto = authorTransformer.transform(authors);*

*...*

Получаем структуру дерева, у одного автора список книг