More On Hibernate & Relationships

Hibernate

Hibernate is an object-relational mapping (ORM) framework that simplifies database interactions in Java applications. It allows developers to map Java objects to database tables and perform CRUD (Create, Read, Update, Delete) operations without writing complex SQL queries.

Relationships in Hibernate

1. One-to-One Relationship: A one-to-One relationship in database design represents a connection between two tables where one record in the first table is related to exactly one record in the second table, and vice versa. This relationship means that for each record in one table, there is at most one related record in the other table.

Characteristics of a One-to-One Relationship:

- **Single Connection:** Each record in one table is associated with, at most, one record in the other table.
- **Unique Correspondence:** There's a unique correspondence between the records of both tables.
- **Foreign Key:** Typically, one table holds a foreign key that references the primary key of the other table.

Example: Consider two entities: Person and Passport. In a one-to-one relationship, one person has exactly one passport, and each passport belongs to precisely one person. In database terms, it means that a Person entity can be associated with at most one Passport entity, and vice versa.

Example Code (Using Java and JPA/Hibernate Annotations)

Here's an example illustrating a One-to-One relationship between *Person* and *Passport* entities using JPA/Hibernate annotations:

Person entity

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

    private String name;

    @OneToOne(mappedBy = "person")
    private Passport passport;

    // other fields, getters, and setters
}
```

Passport entity

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

    private String passportNumber;

    @OneToOne
    @JoinColumn(name = "person_id")
    private Person person;

    // other fields, getters, and setters
}
```

- In this example, Person and Passport are two entities. The Person entity holds a reference to the Passport entity using the **@OneToOne** annotation, and the Passport entity has a reference to the Person entity using the same annotation.

- In Java Persistence API (JPA), **@Entity** is an annotation that marks a Java class as a persistent entity.

- When you mark a Java class with **@Entity**, you're telling the JPA provider (like Hibernate) that instances of this class should be treated as entities that can be persisted in a database. Each instance of an entity class typically corresponds to a row in a database table.

- The **@Id** annotation marks a field in an entity class as the primary key of the corresponding database table. It signifies that this field uniquely identifies each record in the table.

- **GenerationType.IDENTITY**: Indicates that the database will automatically generate unique primary key values. This strategy is typically used with auto-incremented columns in databases like MySQL, PostgreSQL, etc.

- The **@JoinColumn** annotation is used to specify the foreign key column in the Passport table (person_id) that references the primary key of the Person table.

This relationship setup allows a Person to have exactly one Passport, and a Passport to be associated with exactly one Person, demonstrating a One-to-One relationship between these entities.

2. **One-to-Many Relationship**: A One-to-Many relationship in database design refers to a scenario where one entity instance is associated with multiple instances of another entity. This relationship is a fundamental concept in relational databases and is commonly used in various applications.

Characteristics of One-to-Many Relationship:

One Side

- Refers to the entity that stands alone in the relationship.

- In a One-to-Many relationship, this entity can exist independently, with multiple instances of another entity associated with it.

- For example, the Author entity stands on one side in a scenario involving the Author and Book. An author can exist without any books or have multiple books associated with them.

Many Side

- Refers to the entity with multiple instances associated with a single instance of the other entity.

- This side of the relationship represents the entity dependent on or associated with the entity on the one side.

- In the Author and Book example, the Book entity represents the many side. Multiple books can be associated with a single author.

Example:

Consider entities **Author** and **Book:**

- **Author**: An author can write multiple books.
- **Book**: Each book is written by exactly one author.

Implementation in Hibernate/Spring Boot:

Author class

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

    private String name;

    @OneToMany(mappedBy = "author", cascade = CascadeType.ALL)
    private List<Book> books;

    // Constructors, getters, setters
}
```

Book class

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

    private String title;

    @ManyToOne
    @JoinColumn(name = "author_id")
    private Author author;

    // Constructors, getters, setters
}
```

Author Entity: It has a one-to-many relationship with the Book entity. An Author can have multiple Book instances associated with it.

Book Entity: It has a many-to-one relationship with the Author entity. Each Book belongs to exactly one Author.

Explanation:

- **@OneToMany** annotation in the Author class represents the one-to-many relationship. The mappedBy attribute specifies the field in the Book entity that owns the relationship. In this case, it's the author field in the Book class.

- **@ManyToOne** annotation in the Book class defines the many-to-one side of the relationship. The JoinColumn annotation is used to specify the foreign key column (**author_id**) in the Book table that references the Author table.

Important Points:

- The **cascade = CascadeType.ALL** attributes ensure that operations performed on the Author entity (e.g., saving, updating, deleting) will cascade to associated Book entities.

- In the database schema, the Book table will contain a foreign key column (**author_id**) referencing the id column in the Author table, establishing the relationship.

- This setup allows you to navigate from an Author instance to its associated Book instances and vice versa. It enables effective management of one-to-many relationships in your Spring Boot application using Hibernate.

3. **Many-to-Many**: A Many-to-Many relationship in database design represents a scenario where multiple instances of one entity are associated with multiple instances of another entity. Implementing such relationships requires an intermediary table (a junction or link table) to establish connections between entities in relational databases.

Characteristics of Many-to-Many Relationship:

- Exists between two entities, A and B.

- Multiple instances of entity A can be linked to multiple instances of entity B, and vice versa.

- An intermediary table establishes and manages connections between the two entities.

Example using Students and Courses:

Student Entity:

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

    private String name;
```

```
    @ManyToMany
    @JoinTable(
        name = "student_course",  // Intermediary table name
        joinColumns = @JoinColumn(name = "student_id"),
        inverseJoinColumns = @JoinColumn(name = "course_id")
    )
    private List<Course> courses;

    // Constructors, getters, setters
}
```

Course Entity:

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

    private String name;

    @ManyToMany(mappedBy = "courses")
    private List<Student> students;

    // Constructors, getters, setters
}
```

Explanation:

- **Student Entity**:

@ManyToMany annotation establishes the Many-to-Many relationship with the Course entity.

@JoinTable specifies the name of the intermediary table (student_course) and the columns that link the Student and Course entities.

Course Entity:

@ManyToMany is specified in the Student entity, so in the Course entity, mappedBy = "**courses**" indicates the field (courses) in the Student entity that owns the relationship.

Intermediary Table:

- The student_course table (intermediary table) contains foreign key columns referencing the id columns of both the Student and Course tables.

- This table establishes connections between students and courses, allowing many students to enrol in many courses without any entity owning the relationship directly.

Illustration:

- A Student can be enrolled in multiple Course instances (e.g., John is enrolled in Math, Science, and English courses).

- A Course can have multiple Student instances enrolled (e.g., the Math course has John, Alice, and Sarah enrolled).

This Many-to-Many relationship allows for the flexibility of multiple students being associated with multiple courses and vice versa, managed through an intermediary table that facilitates these connections in a Spring Boot application using Hibernate.

Overview of JPA/Hibernate Cascade Types

Cascade types in Java Persistence API (**JPA**) with Hibernate define how entity state changes should propagate from one entity to other associated entities. These cascade types specify whether operations performed on an entity should be cascaded (applied) to its related entities.

Hibernate provides various cascade types, which can be specified using annotations to manage entity state transitions automatically.

Here's an overview of common cascade types:

CascadeType Overview:

1. **ALL:** All operations are cascaded, including persist, merge, remove, refresh, and detach. This effectively means any operation on the parent entity will propagate to its associated entities.
2. **PERSIST:** When a new entity is persisted (saved), the operation is cascaded to associated entities, making them persist.

3. **MERGE:** When an entity is merged (updated), the operation cascades to associated entities, merging their state with the current persistence context.

4. **REMOVE:** When an entity is removed (deleted), the operation cascades to associated entities, removing them as well.

5. **REFRESH:** Refreshing an entity state from the database will also cascade to associated entities, refreshing their state.

6. **DETACH:** Detaching an entity from the persistence context will also cascade to associated entities, detaching them.

Example Usage:

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

    private String name;

    @OneToMany(mappedBy = "author", cascade = CascadeType.ALL)
    private List<Book> books;

    // Constructors, getters, setters
}
```

In this example:

`cascade = CascadeType.ALL` on the **`books`** field of the **`Author`** entity indicates that any operation (persist, merge, remove, etc.) performed on an **`Author`** entity will cascade to its associated **`Book`** entities**.**

Notes:

- Cascade types are specified at the relationship level in JPA annotations, allowing for fine-grained control over how entity state transitions are managed.

- Care should be taken when using cascade types to avoid unintended side effects, such as inadvertently removing associated entities.

Understanding and appropriately utilising cascade types in JPA/Hibernate can significantly simplify the management of entity state transitions and the associated entity graph. It ensures consistency and helps in reducing boilerplate code for handling related entities' state changes.

Eager/Lazy Loading in Hibernate

Overview

When working with an ORM, data fetching/loading can be classified into eager and lazy. This quick tutorial will point out differences and show how we can use these in Hibernate.

Eager and Lazy Loading

- **Eager Loading** is a design pattern in which data initialisation occurs on the spot.
- **Lazy loading** is a design pattern that we use to defer the initialisation of an object as long as possible.

Eager Loading: Eager Loading is a strategy where the associated objects are fetched from the database along with the main entity. In other words, when you load an entity, Hibernate also loads its related entities immediately.

For example, consider two entities, Author and Book, where an author can have multiple books. With eager loading, Hibernate will automatically fetch all associated Book entities when you fetch an Author entity.

Here's a simple code snippet using Hibernate annotations to demonstrate eager loading:

Author entity

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

    private String name;

    @OneToMany(mappedBy = "author", fetch = FetchType.EAGER)
    private List<Book> books;

    // Getters and setters
}
```

Book entity

```
@Entity
public class Book {
    @Id
```

```
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String title;

    @ManyToOne
    @JoinColumn(name = "author_id")
    private Author author;

    // Getters and setters
}
```

In this example, the **@OneToMany** relationship in the **Author** entity specifies **FetchType.EAGER** indicates that whenever an **Author** object is fetched, its associated **Book** objects will also be fetched eagerly.

Lazy Loading: Lazy Loading is a strategy where associated objects are not fetched from the database immediately when the main entity is loaded. Instead, they are loaded only when accessed or requested.

Using the same Author and Book entities:

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

    private String name;

    @OneToMany(mappedBy = "author", fetch = FetchType.LAZY)
    private List<Book> books;

    // Getters and setters
}
```

Here, the **@OneToMany** relationship in the Author entity specifies **FetchType.LAZY** indicates that the associated Book objects will not be loaded immediately when an **Author** object is fetched. They will only be loaded from the database when the **books** list is accessed.

Lazy Loading can help optimise performance by reducing unnecessary database queries. However, developers need to be cautious about potential **LazyInitializationExceptions** that may occur if the associated objects are accessed outside the session where they were initially loaded. These strategies provide flexibility in fetching associated objects in Hibernate, allowing you to optimise performance based on your application's requirements.

@JoinColumn

In Hibernate, **@JoinColumn** defines the owning side of a relationship between two entities. It specifies the column in the database that maintains the association between these entities.

For example, let's consider an Author entity and a Book entity. An author can have multiple books, creating a one-to-many relationship between them.

Author entity

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

    private String name;

    @OneToMany
     @JoinColumn(name = "author_id") // Defines the foreign key column in
the Book table
    private List<Book> books;

    // Getters and setters
}
```

Book entity

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

    private String title;

    @ManyToOne
      private Author author; // No need for @JoinColumn here, as it's
already defined in Author

    // Getters and setters
}
```

In this example, the **@JoinColumn** annotation is used in the **Author** entity to specify that the **author_id** column in the **Book** table will maintain the relationship with the **Author** table. It defines the foreign key column in the **Book** table that links it to the **Author** table.

mappedBy

The mappedBy attribute is used on the inverse side of a bidirectional relationship to specify the field that owns the relationship. It indicates that the field maps the relationship in the owning entity.

In the same Author and Book entities:

Author entity

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

    private String name;

    @OneToMany(mappedBy = "author")
    private List<Book> books;

    // Getters and setters
}
```

Book entity

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

    private String title;

    @ManyToOne
    @JoinColumn(name = "author_id")
    private Author author;

    // Getters and setters
}
```

Here, the **mappedBy** attribute in the **@OneToMany** annotation of the Author entity refers to the author field in the **Book** entity. This indicates that the **Book** entity's author field maps the relationship, meaning that the **Author** entity is not the owner of the relationship in the database schema.

In summary, **@JoinColumn** is used to define the column that maintains the relationship in the database. In contrast, mappedBy is used to indicate the field in the owning entity that manages the bidirectional relationship. These annotations help Hibernate understand how the entities are related and how they should be mapped to the database tables.

References:

1. JPA Entites
2. JPA Unique Constraints
3. Join Column
4. Jpa JoinColumn VS mappedby
5. JPA and Hibernate