Spring Data JPA - OneToMany

From Config to Relations

1. Database Configuration

H2 (In-Memory) vs. **MySQL/Postgres (Persistent)**

- **H2:** Good for dev/testing. Resets on restart.
- **MySQL:** Production standard. Data survives restart.

Important Setting: `ddl-auto`

```
# Wipes the DB clean on every restart.
# Best for early dev so you always have a fresh slate.
spring.jpa.hibernate.ddl-auto=create-drop

# Updates schema without deleting data.
# Good for keeping data while coding, but can leave messy "ghost" columns.
spring.jpa.hibernate.ddl-auto=update
```

2. The Entity: Identity

Every row needs a unique fingerprint.

```
@Id // Marks this field as the Primary Key
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;
```

Why `IDENTITY`?

We use `IDENTITY` to safely delegate ID generation to the database's auto-increment feature, ensuring every row gets a unique number without us having to manually calculate it or worry about duplicates.

3. The Entity: Structure

```java
@Entity
public class Customer {

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

    private String name;
    private String email;
    private String phone;

    // 1. No-Args Constructor (MANDATORY for JPA)
    public Customer() {}

    // 2. Custom Constructor (For Us)
    // NOTICE: We leave out 'id'. The DB sets that!
    public Customer(String name, String email, String phone) {
        this.name = name;
        this.email = email;
        this.phone = phone;
    }
    // + Getters and Setters
}
```

4. The Repository

The layer that talks to the database.

```
// Why an Interface?
// Spring generates the implementing class at RUNTIME.
// You get CRUD methods (save, findAll, delete) for free.

public interface CustomerRepository
        extends JpaRepository<Customer, Long> {

    // Magic: Spring derives the SQL from the method name
    List<Customer> findByName(String name);
}
```

5. Pre-loading Data: `CommandLineRunner`

`InitData.java`

```java
@Component
public class InitData implements CommandLineRunner {

    @Autowired
    CustomerRepository customerRepository;

    @Override
    public void run(String... args) throws Exception {
        // Runs automatically after the server starts
        customerRepository.save(new Customer("Alice", "a@b.com", "123"));
        customerRepository.save(new Customer("Bob", "b@c.com", "456"));
    }
}
```

6. Relation: One-to-Many

Databases are simple tables. They cannot hold complex objects.

The Solution:

- The Child (Order) holds a Foreign Key pointing back to the Parent.
- The Parent (Customer) table stays simple; it physically contains no reference to orders.

Key Takeaway: The relationship always physically lives on the "Many" side.

7. One-to-Many: The "Many" Side (Child)

This is the **Owning Side**. It physically creates the column in the DB.

```
@Entity
public class Order {
    @Id
    private int id;

    // "Many Orders belong to One Customer"
    @ManyToOne
    @JoinColumn(name = "customer_id") // Creates the FK column
    private Customer customer;
}
```

8. One-to-Many: The "One" Side (Parent)

This is the **Inverse Side**. It maps to a Java List.

```java
@Entity
public class Customer {
    @Id
    private Long id;

    // mappedBy = "Go look at the 'customer' field in Order"
    // Does NOT change the database schema.
    @OneToMany(mappedBy = "customer")
    private List<Order> orders;
}
```

9. Bidirectional vs. Unidirectional

Type	Java Code	Pros/Cons
Unidirectional	`Order` has `Customer` OR `Customer` has `List<Order>`	Simpler code, but strict navigation limits.
Bidirectional	`Order` has `Customer` AND `Customer` has `List<Order>`	You can navigate both ways (`order.getCustomer()` & `customer.getOrders()`). **Standard for JPA.**

10. The Infinite Recursion Trap

If `Customer` has a list of `Orders`, and `Order` has a `Customer`...

JSON Serializer (Jackson) Logic:

1. Print Customer... found list of Orders!
2. Print Orders... found a Customer!
3. Print Customer... found list of Orders!
4. **CRASH: StackOverflowError**

10. (Continued)

The "Band-Aid" Fix:

- `@JsonIgnore` or `@JsonBackReference` on the child side stops the loop.

The "Best Practice" Fix:

- **DTOs (Data Transfer Objects):** Create a separate simple class (POJO/Record) that only contains the data you *want* to send (e.g., `CustomerDTO`). Convert your Entity to DTO before sending.