🥦 Vegetable Grocery Store – JPA Mapping Assignment (Enhanced Edition)

Welcome to the **Spring Boot JPA Mapping Assignment** for the **Vegetable Grocery Store System**. This assignment is designed to help you understand and implement real-world **entity relationships** using **Spring Data JPA**.

🎯 Learning Objectives

By the end of this assignment, you will understand:

- **What is JPA?** (Java Persistence API)
- **Entity Relationships** and how they work in databases
- **Mapping annotations** and their purposes
- **Bidirectional vs Unidirectional** relationships
- **Foreign keys** and **Join tables**
- **Database normalization** principles

📖 Background Knowledge

What is JPA?

JPA (Java Persistence API) is a specification that allows Java applications to manage relational data. Think of it as a bridge between your Java objects and database tables.

Key Concepts:

- **Entity**: A Java class that represents a database table
- **Mapping**: Connecting Java class fields to database columns
- **Relationship**: How tables/entities are connected to each other
- **Primary Key**: Unique identifier for each row in a table
- **Foreign Key**: A column that references the primary key of another table

Types of Relationships

1. **One-to-One (1:1)**: One record relates to exactly one other record
2. **One-to-Many (1:N)**: One record can relate to multiple other records
3. **Many-to-One (N:1)**: Multiple records relate to one other record
4. **Many-to-Many (N:N)**: Multiple records relate to multiple other records

---

🗂️ Database Schema Overview

📍 Real-World Scenario

Imagine you're building a system for a vegetable grocery store:

- **Customers** place **orders**
- Each **order** contains multiple **vegetables** with specific **quantities**

- **Vegetables** are supplied by different **suppliers**
- One **supplier** can provide multiple **vegetables**
- One **vegetable** can be provided by multiple **suppliers**

🔗 Relationship Analysis

Customer ↔ Order (One-to-Many)

- **Business Rule**: One customer can place multiple orders, but each order belongs to only one customer
- **Example**: John (customer) can place Order#1 (vegetables), Order#2 (fruits), etc.
- **Database Implementation**: `orders` table has a `customer_id` foreign key

Order ↔ OrderItem (One-to-Many)

- **Business Rule**: One order can contain multiple items, but each item belongs to only one order
- **Example**: Order#1 contains 2kg tomatoes, 1kg onions, 3kg potatoes
- **Database Implementation**: `order_items` table has an `order_id` foreign key

OrderItem ↔ Vegetable (Many-to-One)

- **Business Rule**: Multiple order items can reference the same vegetable, but each order item references only one vegetable
- **Example**: Many customers can order tomatoes, but each order line is for one specific vegetable
- **Database Implementation**: `order_items` table has a `vegetable_id` foreign key

Vegetable ↔ Supplier (Many-to-Many)

- **Business Rule**: One vegetable can be supplied by multiple suppliers, and one supplier can supply multiple vegetables
- **Example**: Tomatoes can come from Supplier A, B, and C. Supplier A can provide tomatoes, onions, and carrots
- **Database Implementation**: A separate `vegetable_suppliers` join table

🧱 Detailed Schema Tables

1. `customers` Table

Purpose: Store customer information

Column	Type	Constraints	Description
id	BIGINT	PRIMARY KEY, AUTO_INCREMENT	Unique customer identifier
name	VARCHAR(100)	NOT NULL	Customer's full name
email	VARCHAR(150)	UNIQUE, NOT NULL	Customer's email address
address	VARCHAR(255)	NULL	Customer's delivery address

Example Data:

```
id | name          | email                   | address
1  | John Smith    | john.smith@email.com    | 123 Main St, City
2  | Sarah Johnson | sarah.johnson@email.com | 456 Oak Ave, Town
```

2. `orders` Table

Purpose: Store order metadata

Column	Type	Constraints	Description
id	BIGINT	PRIMARY KEY, AUTO_INCREMENT	Unique order identifier
order_date	DATETIME	NOT NULL	When the order was placed
customer_id	BIGINT	FOREIGN KEY → customers(id)	Which customer placed the order

Example Data:

```
id | order_date          | customer_id
1  | 2024-01-15 10:30:00 | 1
2  | 2024-01-15 14:20:00 | 2
3  | 2024-01-16 09:15:00 | 1
```

3. `vegetables` Table

Purpose: Store vegetable inventory information

Column	Type	Constraints	Description
id	BIGINT	PRIMARY KEY, AUTO_INCREMENT	Unique vegetable identifier
name	VARCHAR(100)	NOT NULL	Vegetable name
price	DECIMAL(10,2)	NOT NULL	Price per unit (kg/piece)
stock_qty	INT	DEFAULT 0	Available quantity in stock

Example Data:

```
id | name     | price | stock_qty
1  | Tomato   | 3.50  | 100
2  | Onion    | 2.00  | 200
3  | Carrot   | 2.75  | 150
```

4. `order_items` Table

Purpose: Store what vegetables and quantities are in each order

Column	Type	Constraints	Description
id	BIGINT	PRIMARY KEY, AUTO_INCREMENT	Unique order item identifier
order_id	BIGINT	FOREIGN KEY → orders(id)	Which order this item belongs to
vegetable_id	BIGINT	FOREIGN KEY → vegetables(id)	Which vegetable is being ordered
quantity	INT	NOT NULL, CHECK (quantity > 0)	How many units (kg/pieces)
total_price	DECIMAL(10,2)	NOT NULL	quantity × vegetable.price

Example Data:

```
id | order_id | vegetable_id | quantity | total_price
1  | 1        | 1            | 2        | 7.00
2  | 1        | 2            | 1        | 2.00
3  | 2        | 1            | 3        | 10.50
```

5. `suppliers` Table

Purpose: Store supplier information

Column	Type	Constraints	Description
id	BIGINT	PRIMARY KEY, AUTO_INCREMENT	Unique supplier identifier
name	VARCHAR(100)	NOT NULL	Supplier company name
contact	VARCHAR(150)	NULL	Phone/email contact information

Example Data:

```
id | name              | contact
1  | Fresh Farm Co.    | contact@freshfarm.com
2  | Green Valley Ltd. | +1-555-0123
3  | Organic Harvest   | organic@harvest.com
```

6. `vegetable_suppliers` Table (Join Table)

Purpose: Link vegetables with their suppliers (Many-to-Many relationship)

Column	Type	Constraints	Description
vegetable_id	BIGINT	FOREIGN KEY → vegetables(id)	Which vegetable
supplier_id	BIGINT	FOREIGN KEY → suppliers(id)	Which supplier provides it

Primary Key: Combination of (vegetable_id, supplier_id)

Example Data:

```
vegetable_id | supplier_id
1            | 1          # Tomato supplied by Fresh Farm Co.
1            | 3          # Tomato also supplied by Organic Harvest
2            | 1          # Onion supplied by Fresh Farm Co.
2            | 2          # Onion also supplied by Green Valley Ltd.
```

🔄 Relationship Mapping Details

Understanding Directionality

Bidirectional Relationships

- **Definition**: Both entities can navigate to each other
- **Example**: Customer can access their orders, and Order can access its customer
- **Code**: Both entities have references to each other

Unidirectional Relationships

- **Definition**: Only one entity can navigate to the other
- **Example**: OrderItem can access Vegetable, but Vegetable doesn't track which OrderItems reference it
- **Code**: Only one entity has a reference to the other

Detailed Relationship Breakdown

From Entity	To Entity	Relationship Type	Direction	Reason for Direction Choice
Customer	Order	One-to-Many	Bidirectional	Customers need to see their orders; Orders need customer info
Order	OrderItem	One-to-Many	Bidirectional	Orders need to show items; Items need order context
OrderItem	Vegetable	Many-to-One	Unidirectional	Items need vegetable info; Vegetables don't need to track every order
Vegetable	Supplier	Many-to-Many	Bidirectional	Vegetables need supplier info; Suppliers need to see their vegetables

---

🧠 Your Implementation Tasks

Task 1: Create Entity Classes

Step 1: Customer Entity

Create a `Customer.java` class with:

- Fields: id, name, email, address
- One-to-Many relationship with Orders
- Proper JPA annotations

Key Annotations to Use:

```
java

@Entity
@Table(name = "customers")
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
@Column(nullable = false, unique = true)
@OneToMany(mappedBy = "customer", cascade = CascadeType.ALL, fetch = FetchType.LAZY)
@JsonManagedReference
```

Step 2: Order Entity

Create an `Order.java` class with:

- Fields: id, orderDate, customer
- Many-to-One relationship with Customer
- One-to-Many relationship with OrderItems

Key Annotations to Use:

```java
@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "customer_id", nullable = false)
@JsonBackReference
```

Step 3: Vegetable Entity

Create a `Vegetable.java` class with:

- Fields: id, name, price, stockQuantity
- Many-to-Many relationship with Suppliers

Step 4: Supplier Entity

Create a `Supplier.java` class with:

- Fields: id, name, contact
- Many-to-Many relationship with Vegetables

Key Annotations for Many-to-Many:

```java
@ManyToMany
@JoinTable(
    name = "vegetable_suppliers",
    joinColumns = @JoinColumn(name = "vegetable_id"),
    inverseJoinColumns = @JoinColumn(name = "supplier_id")
)
```

Step 5: OrderItem Entity

Create an `OrderItem.java` class with:

- Fields: id, order, vegetable, quantity, totalPrice
- Many-to-One relationship with Order
- Many-to-One relationship with Vegetable

Task 2: Database Creation Process

Step 1: Add Dependencies

Add these to your `pom.xml`:

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

Step 2: Configure Application Properties

Add to `application.yml`:

```yaml
spring:
  datasource:
    url: jdbc:h2:mem:testdb
    driver-class-name: org.h2.Driver
    username: sa
    password: password
  jpa:
    database-platform: org.hibernate.dialect.H2Dialect
    hibernate:
      ddl-auto: create-drop
    show-sql: true
  h2:
    console:
      enabled: true
```

Step 3: Automatic Table Creation

When you run your Spring Boot application:

1. Hibernate will read your entity annotations
2. It will automatically create tables based on your entities
3. Foreign key constraints will be added based on relationships
4. Join tables will be created for Many-to-Many relationships

---

🎓 Advanced Concepts Explained

Cascade Types

- **CascadeType.ALL**: All operations cascade to related entities
- **CascadeType.PERSIST**: Only save operations cascade
- **CascadeType.REMOVE**: Only delete operations cascade

Fetch Types

- **FetchType.LAZY**: Related data is loaded only when accessed (recommended for performance)

- **FetchType.EAGER**: Related data is loaded immediately (can cause performance issues)

JSON Serialization

- **@JsonManagedReference**: Used on the "parent" side of bidirectional relationship
- **@JsonBackReference**: Used on the "child" side to prevent infinite loops

---

🧪 Testing Your Implementation

Sample Data Script

Create a `data.sql` file in `src/main/resources`:

```sql
-- Insert customers
INSERT INTO customers (name, email, address) VALUES
('John Smith', 'john@email.com', '123 Main St'),
('Sarah Johnson', 'sarah@email.com', '456 Oak Ave');

-- Insert vegetables
INSERT INTO vegetables (name, price, stock_qty) VALUES
('Tomato', 3.50, 100),
('Onion', 2.00, 200),
('Carrot', 2.75, 150);

-- Insert suppliers
INSERT INTO suppliers (name, contact) VALUES
('Fresh Farm Co.', 'contact@freshfarm.com'),
('Green Valley Ltd.', '+1-555-0123');
```

Testing Checklist

- [ ] All tables are created correctly
- [ ] Foreign key constraints are in place
- [ ] Sample data loads without errors
- [ ] You can access H2 console at `http://localhost:8080/h2-console`
- [ ] Relationships work when querying data

🎁 Bonus Challenges

Bonus 1: Service Layer Implementation

Create a `OrderService` class that:

- Accepts customer ID and list of vegetables with quantities
- Calculates total prices automatically
- Validates stock availability
- Persists the complete order

Bonus 2: REST API Endpoints

Create controllers for:

- `GET /customers/{id}/orders` - Get all orders for a customer
- `GET /vegetables/suppliers` - Get all vegetables with their suppliers
- `POST /orders` - Place a new order

Bonus 3: Custom Queries

Add repository methods with:

- Find vegetables by supplier
- Find orders by date range
- Calculate total sales by vegetable

📚 Essential JPA Annotations Reference

Annotation	Purpose	Example Usage
`@Entity`	Marks class as JPA entity	`@Entity public class Customer`
`@Table(name = "...")`	Specifies table name	`@Table(name = "customers")`
`@Id`	Marks primary key field	`@Id private Long id;`
`@GeneratedValue`	Auto-generate primary key	`@GeneratedValue(strategy = GenerationType.IDENTITY)`
`@Column`	Configure column properties	`@Column(nullable = false, unique = true)`
`@OneToMany`	One-to-Many relationship	`@OneToMany(mappedBy = "customer")`
`@ManyToOne`	Many-to-One relationship	`@ManyToOne @JoinColumn(name = "customer_id")`
`@ManyToMany`	Many-to-Many relationship	`@ManyToMany @JoinTable(...)`
`@JoinColumn`	Specifies foreign key column	`@JoinColumn(name = "customer_id")`
`@JoinTable`	Configures join table	`@JoinTable(name = "vegetable_suppliers")`

🚨 Common Pitfalls to Avoid

1. Circular References in JSON

Problem: Bidirectional relationships cause infinite loops during JSON serialization **Solution**: Use `@JsonManagedReference` and `@JsonBackReference`

2. N+1 Query Problem

Problem: Lazy loading can cause multiple database queries **Solution**: Use `@Query` with JOIN FETCH or appropriate fetch strategies

3. Cascade Operations

Problem: Accidentally deleting related data **Solution**: Carefully choose cascade types based on business requirements

4. Table Naming Conflicts

Problem: JPA default naming might conflict with database reserved words **Solution**: Explicitly specify table and column names using `@Table` and `@Column`

✅ Submission Requirements

Code Requirements

- [ ] All 5 entity classes implemented with proper annotations
- [ ] Relationships correctly mapped (4 different relationship types)
- [ ] Code compiles and runs without errors
- [ ] Tables are created automatically by Hibernate
- [ ] Sample data can be inserted successfully

Documentation Requirements

- [ ] Comments explaining each relationship choice
- [ ] Brief explanation of why certain relationships are bidirectional/unidirectional
- [ ] Description of any challenges faced and how you solved them
- [ ] README file with instructions to run the project

File Structure

```
src/
├── main/
│   ├── java/
│   │   └── com/example/grocery/
│   │       ├── entity/
│   │       │   ├── Customer.java
│   │       │   ├── Order.java
│   │       │   ├── OrderItem.java
│   │       │   ├── Vegetable.java
│   │       │   └── Supplier.java
│   │       └── GroceryApplication.java
│   └── resources/
│       ├── application.yml
│       └── data.sql
```

🎯 Grading Criteria

Criteria	Points	Description
Entity Classes	25	All entities properly created with correct fields
Relationship Mapping	30	Correct use of JPA relationship annotations
Database Schema	20	Tables created correctly with proper constraints
Code Quality	15	Clean code, proper naming, good structure
Documentation	10	Clear comments and explanations

Total: 100 Points

🆘 Getting Help

Common Questions

1. **Q**: "My application won't start" **A**: Check your `application.yml` configuration and ensure all dependencies are in `pom.xml`
2. **Q**: "Tables are not being created" **A**: Verify `hibernate.ddl-auto: create-drop` is set and your entities have `@Entity` annotation
3. **Q**: "Getting circular reference errors" **A**: Add `@JsonManagedReference` and `@JsonBackReference` to bidirectional relationships
4. **Q**: "Foreign key constraint violations" **A**: Ensure you're saving entities in the correct order (parent entities before child entities)

Resources

- Spring Data JPA Documentation
- Hibernate User Guide
- JPA Relationship Mapping Guide

🏆 Success Tips

1. **Start Simple**: Begin with basic entities, then add relationships gradually
2. **Test Frequently**: Run your application after each entity to catch errors early
3. **Use H2 Console**: Access `http://localhost:8080/h2-console` to verify table creation
4. **Read Error Messages**: Hibernate provides detailed error messages - read them carefully
5. **Draw Diagrams**: Sketch out your entity relationships before coding
6. **Follow Naming Conventions**: Use consistent naming for tables, columns, and Java fields

---

🙇 Final Words

This assignment simulates real-world backend development for e-commerce applications. The skills you learn here - entity modeling, relationship mapping, and database design - are fundamental to building robust, scalable applications.

Take your time to understand each concept rather than rushing through the implementation. The relationships you model here will form the foundation of how data flows through your application.

Remember: Good database design is the backbone of any successful application!

Happy Coding! 💻 🥕

Last Updated: June 2025