

Understanding JPA, Part 2: Relationships the JPA way

Handle data relationships with object-oriented grace

By Aditi Das

JavaWorld |

Your Java applications are dependent on a web of data relationships, which can become a tangled mess if improperly handled. In this second half of her introduction to the Java Persistence API, Aditi Das shows you how JPA uses annotations to create a more transparent interface between object-oriented code and relational data. The resulting data relationships are easier to manage and more compatible with the object-oriented programming paradigm.

Data is an integral part of any application; equally important are the relationships between different pieces of data. Relational databases support a number of different types of relationships between tables, all designed to enforce referential integrity.

In this second half of Understanding JPA, you will learn how to use the Java Persistence API and Java 5 annotations to handle data relationships in an object-oriented manner. This article is meant for readers who understand basic JPA concepts and the issues involved in relational database programming in general, and who would like to further explore the object-oriented world of JPA relationships. For an introduction to JPA, see "Understanding JPA, Part 1: The object-oriented paradigm of data persistence."

A real-life scenario

Imagine a company called XYZ that offers five subscription products to its customers: A, B, C, D, and E. Customers are free to order products in combination (at a reduced price) or they can order individual products. A customer need not pay anything at the time of ordering; at the end of the month, if the customer is satisfied with the product, an invoice is generated and sent to the customer for billing. The data model

for this company is shown in Figure 1. A customer can have zero or more orders, and each order can be associated with one or more products. For each order, an invoice is generated for billing.


 Diagram of a data model

Figure 1. A sample data model (click to enlarge)

Now XYZ wants to survey its customers to see how satisfied they are with its products, and hence needs to find out how many products each customer has. In order to figure out how to improve the quality of its products, the company also wants to conduct a special survey of those customers who cancelled their subscriptions within the first month.

Traditionally, you might tackle this problem by building a data access object (DAO) layer where you would write complex joins between CUSTOMER, ORDERS, ORDER_DETAIL, ORDER_INVOICE, and PRODUCT tables. Such a design would look good on the surface, but it might be hard to maintain and debug as the application grew in complexity.

JPA offers another, more elegant way to address this problem. The solution I present in this article takes an object-oriented approach and, thanks to JPA, doesn't involve creating any SQL queries. Persistence providers are left with the responsibility of doing the job transparently to the developers.

Before continuing, you should download the sample code package from the Resources section below. This includes sample code for the one-to-one, many-to-one, one-to-many, and many-to-many relationships explained in this article, in the context of the example application.

One-to-one relationships

First off, the example application will need to address the order-invoice relationship. For each order, there will be an invoice; and, similarly, each invoice is associated with an order. These two tables are related with one-to-one mapping as shown in Figure 2, joined with the help of the foreign key ORDER_ID. JPA facilitates one-to-one mapping with the help of the @OneToOne annotation.

Figure 2. A one-to-one relationship (click to enlarge)

The sample application will fetch the order data for a particular invoice ID. The Invoice entity shown in Listing 1 maps all the fields of the INVOICE table as attributes and has an Order object joined with the ORDER_ID foreign key.

Listing 1. A sample entity depicting a one-to-one relationship

```
@Entity(name = "ORDER_INVOICE")
public class Invoice {

    @Id
    @Column(name = "INVOICE_ID", nullable = false)
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long invoiceId;

    @Column(name = "ORDER_ID")
    private long orderId;

    @Column(name = "AMOUNT_DUE", precision = 2)
    private double amountDue;

    @Column(name = "DATE_RAISED")
    private Date orderRaisedDt;

    @Column(name = "DATE_SETTLED")
    private Date orderSettledDt;

    @Column(name = "DATE_CANCELLED")
    private Date orderCancelledDt;

    @Version
    @Column(name = "LAST_UPDATED_TIME")
    private Date updateTime;
    @OneToOne(optional=false)
    @JoinColumn(name = "ORDER_ID")
    private Order order;
    ...
    //getters and setters goes here
}
```

The `@OneToOne` and the `@JoinColumn` annotations in Listing 1 are internally resolved by the persistence provider, as illustrated in Listing 2.

Listing 2. SQL query resolving a one-to-one relationship

```
SELECT t0.LAST_UPDATED_TIME, t0.AMOUNT_PAID, t0.ORDER_ID,
t0.DATE_RAISED ,t1.ORDER_ID, t1.LAST_UPDATED_TIME, t1.CUST_ID,
t1.OREDER_DESC, t1.ORDER_DATE, t1.TOTAL_PRICE
FROM ORDER_INVOICE t0
INNER JOIN ORDERS t1 ON t0.ORDER_ID = t1.ORDER_ID
WHERE t0.INVOICE_ID = ?
```

The query in Listing 2 shows an inner join between the `ORDERS` and `INVOICE` tables. But what happens if you need an outer join relationship? You can control the join type very easily by setting the optional attribute of `@OneToOne` to either `true` or `false` to indicate whether or not the association is optional. The default value is `true`, which signifies that the related object may or may not exist and that the join will be an outer join in that case. Since each order must have an invoice and vice versa, in this case the optional attribute has been set to `false`.

Listing 3 demonstrates how to fetch an order for a particular invoice you write.

Listing 3. Fetching objects involved in a one-to-one relationship

```
....
EntityManager em = entityManagerFactory.createEntityManager();
Invoice invoice = em.find(Invoice.class, 1);
System.out.println("Order for invoice 1 : " + invoice.getOrder());
em.close();
entityManagerFactory.close();
....
```

But what happens if you want to fetch the invoice for a particular order?

Bidirectional one-to-one relationships

Every relationship has two sides:

- The *owning* side is responsible for propagating the update of the relationship to the database. Usually this is the side with the foreign key.
- The *inverse* side maps to the owning side.

In the one-to-one mapping in the example application, the Invoice object is the owning side. Listing 4 demonstrates what the inverse side -- the Order -- looks like.

Listing 4. An entity in the sample bidirectional one-to-one relationship

```
@Entity(name = "ORDERS")
public class Order {

    @Id
    @Column(name = "ORDER_ID", nullable = false)
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long orderId;

    @Column(name = "CUST_ID")
    private long custId;

    @Column(name = "TOTAL_PRICE", precision = 2)
    private double totPrice;

    @Column(name = "ORDER_DESC")
    private String orderDesc;

    @Column(name = "ORDER_DATE")
    private Date orderDt;

    @OneToOne(optional=false,cascade=CascadeType.ALL,
    mappedBy="order",targetEntity=Invoice.class)
    private Invoice invoice;
    @Version
    @Column(name = "LAST_UPDATED_TIME")
    private Date updateTime;

    ....
    //setters and getters goes here
}
```

Listing 4 maps to the field (order) that owns the relationship by mappedBy="order". targetEntity specifies the owning class name. Another attribute that has been introduced here is cascade. If you are performing insert, update, or delete operations on the Order entity and you want to propagate the same operations to the child object (Invoice, in this case), use the cascade option; you might want to only propagate PERSIST, REFRESH, REMOVE, or MERGE operations, or propagate all of them.

Listing 5 demonstrates how to fetch the invoice details for a particular Order you write.

Listing 5. Fetching objects involved in a bidirectional one-to-one relationship

```
....
EntityManager em = entityManagerFactory.createEntityManager();
Order order = em.find(Order.class, 111);
System.out.println("Invoice details for order 111 : " + order.getInvoice());
em.close();
entityManagerFactory.close();
....
```

Many-to-one relationships

In the previous section, you saw how to successfully retrieve invoice details for a particular order. Now you'll change your focus to see how to get order details for a particular customer, and vice versa. A customer can have zero or more orders, whereas an order is mapped to one customer. Thus, a Customer enjoys a one-to-many relationship with an Order, whereas an Order has a many-to-one relationship with the Customer. This is illustrated in Figure 3.


 Diagram of a many-to-one/one-to-many relationship.

Figure 3. A many-to-one/one-to-many relationship (click to enlarge)

Here, the Order entity is the owning side, mapped to Customer by the CUST_ID foreign key. Listing 6 illustrates how a many-to-one relationship can be specified in the Order entity.

Listing 6. A sample entity illustrating a bidirectional many-to-one relationship

```
@Entity(name = "ORDERS")
public class Order {

    @Id //signifies the primary key
    @Column(name = "ORDER_ID", nullable = false)
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long orderId;

    @Column(name = "CUST_ID")
    private long custId;

    @OneToOne(optional=false,cascade=CascadeType.ALL,
    mappedBy="order",targetEntity=Invoice.class)
    private Invoice invoice;

    @ManyToOne(optional=false)
    @JoinColumn(name="CUST_ID",referencedColumnName="CUST_ID")
    private Customer customer;

    .....
    The other attributes and getters and setters goes here
}
```

In Listing 6, the Order entity is joined with the Customer entity with the help of the CUST_ID foreign key column. Here also the code specifies optional=false, as each order should have a customer associated to it. The Order entity now has a one-to-one relationship with Invoice and a many-to-one relationship with Customer.

Listing 7 illustrates how to fetch the customer details for a particular Order.

Listing 7. Fetching objects involved in a many-to-one relationship

```
.....
EntityManager em = entityManagerFactory.createEntityManager();
Order order = em.find(Order.class, 111);
System.out.println("Customer details for order 111 : " + order.getCustomer());
em.close();
entityManagerFactory.close();
.....
```

But what happens if you want to find out how many orders have been placed by a customer?

One-to-many relationships

Fetching order details for a customer is pretty easy once the owning side has been designed. In the previous section, you saw that the `Order` entity was designed as the owning side, with a many-to-one relationship. The inverse of many-to-one is a one-to-many relationship. The `Customer` entity in Listing 8 encapsulates the one-to-many relationship by being mapped to the owning side attribute `customer`.

Listing 8. A sample entity illustrating a one-to-many relationship


```

@Entity(name = "CUSTOMER")
public class Customer {
    @Id //signifies the primary key
    @Column(name = "CUST_ID", nullable = false)
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long custId;

    @Column(name = "FIRST_NAME", length = 50)
    private String firstName;

    @Column(name = "LAST_NAME", nullable = false,length = 50)
    private String lastName;

    @Column(name = "STREET")
    private String street;
    @OneToMany(mappedBy="customer",targetEntity=Order.class,
    fetch=FetchType.EAGER)
    private Collection orders;
    .....
    // The other attributes and getters and setters goes here
}

```

The `@OneToMany` annotation in Listing 8 introduces a new attribute: `fetch`. The default fetch type for one-to-many relationship is `LAZY`. `FetchType.LAZY` is a hint to the JPA runtime, indicating that you want to defer loading of the field until you access it. This is called *lazy loading*. Lazy loading is completely transparent; data is loaded from the database in objects silently when you attempt to read the field for the first time. The other possible fetch type is `FetchType.EAGER`. Whenever you retrieve an entity from a query or from the `EntityManager`, you are guaranteed that all of its eager fields are populated with data store data. In order to override the default fetch type, `EAGER` fetching has been specified with `fetch=FetchType.EAGER`. The code in Listing 9 fetches the order details for a particular Customer.

Listing 9. Fetching objects involved in a one-to-many relationship


.....

```
EntityManager em = entityManagerFactory.createEntityManager();
Customer customer = em.find(Customer.class, 100);
System.out.println("Order details for customer 100 : " + customer.getOrders());
em.close();
entityManagerFactory.close();
```

.....

Many-to-many relationships

There is one last leg of relationship mapping left to consider. An order can consist of one or more products, whereas a product can be associated with zero or more orders. This is a many-to-many relationship, as illustrated in Figure 4.

 Diagram of a many-to-many relationship.

*Figure 4. A many-to-many relationship
(click to enlarge)*

